ADRV9008-1/ADRV9008-2/ADRV9009
Hardware Reference Manual
UG-1295

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • www.analog.com

Hardware Reference Manual for the ADRV9008-1, ADRV9008-2, and ADRV9009

INTRODUCTION

This hardware reference manual serves as the main source of information for system engineers and software developers using the ADRV9008-1 receiver (Rx), ADRV9008-2 transmitter (Tx) and observation receiver (ORx), and the ADRV9009 transceiver.

TABLE OF CONTENTS

REVISION HISTORY

6/2018—Revision 0: Initial Version

SYSTEM OVERVIEW

The device consist of three highly integrated, radio frequency (RF) devices.

The ADRV9008-1 is a dual-channel RF receiver that operates over a frequency range of 75 MHz to 6 GHz. The ADRV9008-1 supports a receiver bandwidth of 200 MHz and supports multicarrier, global systems for mobile communications (GSM) cellular phone standards. Data transfer is accomplished using four JESD204B dedicated, high speed, serial interface lanes. See the ADRV9008-1 data sheet for a block diagram of the device.

The ADRV9008-2 is a dual-channel RF transmitter and observation receiver that operates over a frequency range of 75 MHz to 6 GHz. The ADRV9008-2 supports up to 450 MHz of transmitter bandwidth and supports multicarrier GSM. The device includes a pair of observation receivers. Only one channel can be operational at any given time. This pair of channels supports the same frequency range and bandwidth of the transmitter channels. Data transfer is accomplished through eight JESD204B dedicated, high speed interface lanes. There are four lanes each for the transfer of transmitter data and observation receiver data. See the ADRV9008-2 data sheet for a block diagram of the device.

The ADRV9009 is a dual-channel RF transmitter and receiver with an observation receiver. The ADRV9009 operates over a frequency range of 75 MHz to 6 GHz. The device supports up to 450 MHz of synthesis bandwidth on its transmitters. The receiver portion of the device can operate either as a dual-channel receiver that supports bandwidths up to 200 MHz or as a single-channel observation receiver that supports bandwidths up to 450 MHz. The receiver portion is switched between single-channel and dual-channel mode in time division duplex (TDD) operation, utilizing the dual-channel receiver mode during receiver slots and utilizing the single-channel observation receiver in transceiver slots. The single-channel observation receiver can select either of the ORX1_IN+/ORX1_IN− or ORX2_IN+/ORX2_IN− RF inputs. Data transfer is accomplished through eight JESD204B dedicated, high speed interface lanes. There are four interface lanes each for the transfer of transmitter data and main receiver/observation receiver data. See the ADRV9009 data sheet for a block diagram of the device.

A serial peripheral interface (SPI) transmits and receives control information between the devices and a baseband processor (BBP). All software control is communicated via the SPI. The devices include a control interface that utilizes GPIO lines to provide hardware control to and from the devices. These GPIOs can be configured to provide dedicated sets of functions for different application scenarios. Some GPIO are intended for digital control and others are supplied by a 3.3 V analog supply to control external analog components. The devices also include a set of four low speed auxiliary ADCs that can be used to monitor external voltages of interest to system operation.

The AD9528 is a good choice for systems and is used on the Analog Devices, Inc., evaluation platform because the AD9528 provides a two-stage PLL with multiple outputs, along with an integrated JESD204B SYSREF generator for multiple device synchronization.

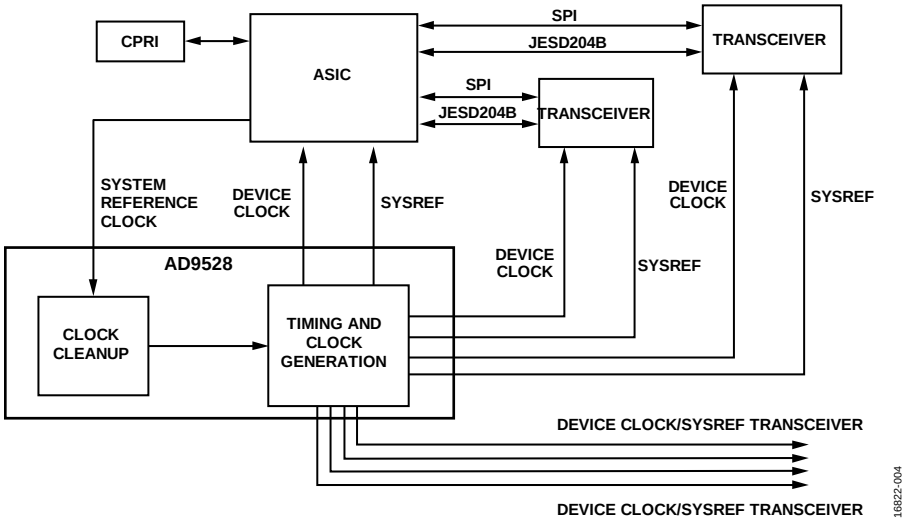

Figure 1. Mutiple Device System Example

SYSTEM ARCHITECTURE DESCRIPTION

This reference manual provides information about the application programming interface (API) software, developed by Analog Devices for the ADRV9008-1, ADRV9008-2, and ADRV9009. This document outlines the overall architecture, folder structure, and methods for using API software on any platform. This reference manual does not explain the API library functions. Detailed information regarding the API functions is in the device API doxygen document (**Talise.chm**) located at **/src/doc.** This file can also be viewed in the **Help** tab on the Talise transceiver evaluation software (TTES) that controls the evaluation platform.

SOFTWARE ARCHITECTURE

Figure 2 and Figure 3 illustrate the software architecture for a generic system and for the Analog Devices evaluation platform, respectively.

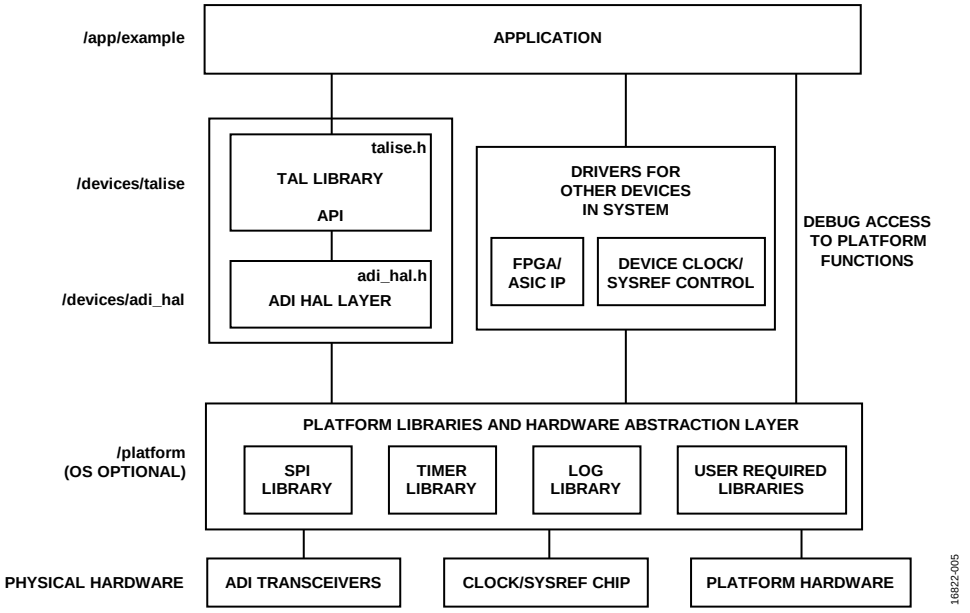

Figure 2. API Software Architecture (Generic System)

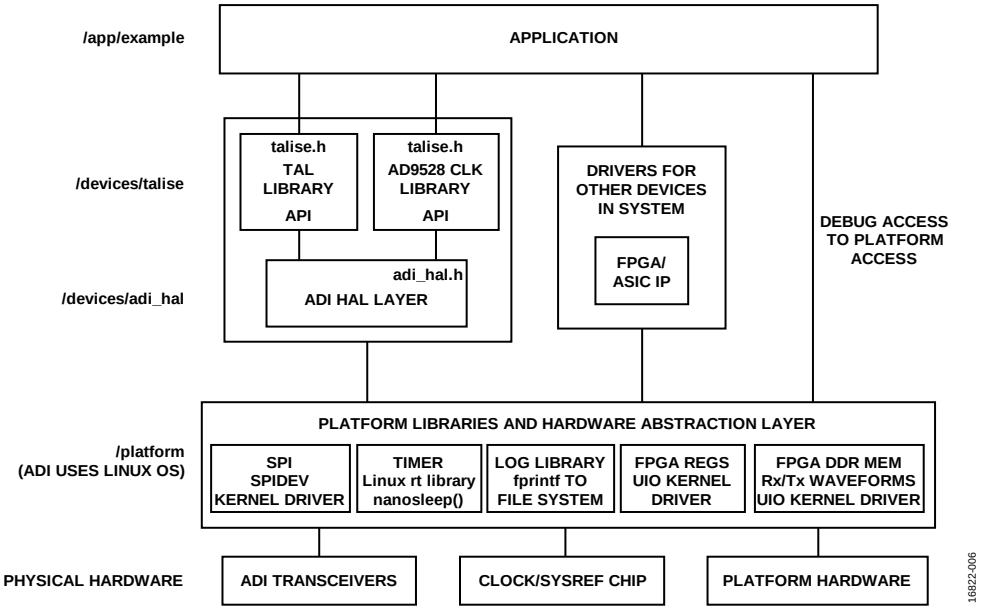

Figure 3. API Software Architecture (Analog Devices Evaluation Platform)

FOLDER STRUCTURE

Source files are provided by Analog Devices in the folder structure shown in Figure 4. Analog Devices understands that the developer may desire to use a different folder structure. Analog Devices provides the API source code releases in the folder structure shown in Figure 4, and the developer may organize the API into a custom folder organization if required. The developer is not permitted to to modify the content of the API source code. Modifying the content of each API source file prevents updates to future API code releases.

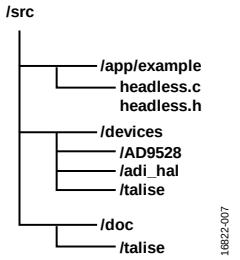

Figure 4. API Folder Structure

/src/app/example Folder

The **/src/app/example** folder contains a simple example application layer program. The **headless.c** file has the top level main function that demonstrates the sequence of the API function calls to initialize the device. Software developers can use this example code as a starting point to begin development on a custom application layer program. The **/src/app/example** folder also contains the **talise_config.c** file. The **talise_config.c** file holds the initialization and run-time data structures used by the API. The TTES can generate the initialization and run-time data structures for the API. Generating the **talise_config.c** file from the evaluation software allows the user to create the structures with custom values selected by the graphical user interface (GUI) configuration pages.

/src/devices Folder

The **/src/devices** folder includes the main API code for the transmitter (**/src/devices/talise**). The **/talise** folder contains the high level function prototypes, data types, macros, and source code that are used to build the final user software system. The user is strictly forbidden from modifying the files contained in the **/talise** folder because the code is maintained by Analog Devices. The only exception to this restriction is that the developer can modify the **/talise/talise_user.c** file, which contains receiver gain tables and user selectable define macros, for example, `TALISE_VERBOSE` mode, which enables and disables certain API messages to the log.

The **/adi_hal** folder provides the means for a developer to insert custom platform hardware driver code for system integration with the API. It is important that the function prototypes in the **adi_hal.c** file do not change. The developer is responsible for implementing the code inside each **adi_hal.c** function to ensure that the correct hardware drivers are called for the platform hardware. In the example code provided in the **adi_hal.c** file, the functions are generic wrappers that call hardware layer functions, devices, and resources for the Xilinx® Zynq®-7000 SoC ZC706 platform. Analog Devices API implementation attempts to keep the **adi_hal.c** implementation generic to allow simplified platform swapping.

/src/doc Folder

The **/src/doc** folder contains the device API doxygen (**Talise.chm**) file for user reference. This file is in compressed HTML format. For security reasons, **.chm** files can only be opened from a local drive. Attempting to open these files from a network drive can result in a file that appears empty.

PRIVATE vs. PUBLIC API FUNCTIONS

The API is made up of multiple **.c** and **.h** files. The functionality of the API is broken into modular pieces to help organize the API functions. Because the API is written in C, there are no language modifiers to identify a function as private or public, as commonly done in object oriented languages. Public API functions are denoted by the function name prepended with `TALISE_functionName()`. The application layer is free to use any API function that is prepended with the `TALISE_` naming. Private helper functions lack the `TALISE_` prefix. It is not intended that the private helper functions add any value to the application layer.

Most functions in the API are prefixed with `TALISE_` and are for public use. Many of these functions are never called directly from the application layer. For this reason, the majority of the initialization and other helper functions are separated from the top level **talise.c**/**talise.h** files to help the developer focus on the functions that are most commonly and widely used by the application layer program.

HARDWARE ABSTRACTION LAYER (HAL)

The HAL interface is a library of functions that the transceiver APIs uses when the API must access the target platform hardware. The HAL is defined by **adi_hal.h**; however, the implementation of this interface is platform dependent and is implemented by the end user in the **adi_hal.c** file. This architecture is depicted in Figure 2.

The HAL is a collection of APIs, macros, and defines that are designed to make the upper layers (libraries and application) as platform independent as possible. This reference manual describes those HAL components.

The Analog Devices source code has a subfolder under the **/adi_hal** folder in the **/device** folder. The **adi_hal.h** header file details the HAL interface and functions. The **adi_hal.c** provides details of the Analog Devices platform specific implementation of the Analog Devices HAL (ADIHAL) interface. The **adi_hal.c** file can be used as an example by the end user when developing the HAL function implementation for a custom platform.

Hardware Functions

The transceiver API requires a library of functions that facilitate access to the hardware interfaces on the target platform (see Figure 5).

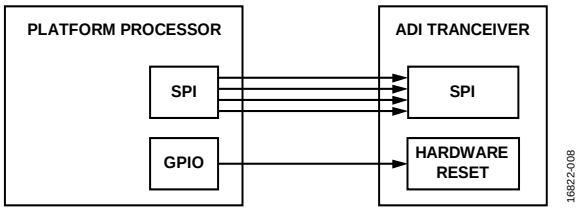

Figure 5. Hardware Controls Required by HAL Interface Functions

Access to the SPI controller that communicates with the transmitter is required. The SPI Interface details are illustrated in the SPI section. In addition, control of the hardware reset signal that controls the $\overline{\text{RESET}}$ pin of the Analog Devices transceiver is required. This hardware reset signal is usually implemented by a platform processor GPIO. Refer to the ADRV9008-1, ADRV9008-2, and ADRV9009 data sheets for full details on the $\overline{\text{RESET}}$ pin. Figure 5 shows a short overview of the required HAL functions. Full details are provided in the Hardware Abstraction Layer (HAL) section of this document.

Logging Functions

The API provides a simple logging feature function that can be enabled for debugging purposes. This feature requires an implementation for the `ADIHAL_writeToLog` function. The APIs optionally call to send debug information to the system via the HAL. The `ADIHAL_setLogLevel` function can be used to configure HAL flags, and the HAL flags configure how the HAL processes the various message types from the API layer. The Analog Devices transceiver open hardware function `TALISE_openHw` calls this function to set the desired logging operation.

Multidevice Support

For applications with multiple transceivers, the HAL layer requires a reference to the targeted device and its hardware particulars, for example, the SPI chip select and reset signal. The first parameter of the HAL function prototype, `void* devHalInfo`, acts as the device reference for the HAL functions.

Note that for Analog Devices transceiver APIs, it is required that only one thread can control and configure a particular device at any given time.

devHalInfo Pointer Parameter

To transfer the target device information from the application to the HAL functions, the API layer transfers a void pointer parameter, `devHalInfo`, from the application to all HAL functions. This void pointer acts as a state container for the relevant hardware information for a particular device.

The API user must define this state container as per system HAL implementation requirements. The user can implement any structure to transfer any hardware configuration information that the hardware requires between the application layer and the platform layer. This state container can be used to transfer device reference information in multithreaded and multitransceiver systems.

The application transfers the device state container, devHalInfo, via the API transceiver device structure, for example, `taliseDevice_t`. The API function does not read or write the `(void *) devHalInfo` but transfers the state container as a parameter to all HAL function calls.

Pseudo Code Example Use of devHalInfo

The device structure is as follows:

```
typedef struct
{
    void   *devHalInfo;          /*HAL layer State Container*/
   void   *devStateInfo;         /*API internal State Container*/
} taliseDevice_t;
```

The example target system HAL device state container is as follows:

```
typedef struct
{
    uint8_t devIndex                    /*Device Reference*/
    zynqSpiSettings_t *spiSettings;     /*Platform SPI controller settings*/
    uint16_t wait_timeout;                          /*Timeout limit for HAL function
Completion*/
} zynqcAdiDev_t;
```

The example API call on the target system application is as follows:

```
       zynqAdiDev_tTaliseDev1 {…};
       taliseDevice_t device = {&TaslieDev1…..} ;
Talise_initialize(&device) ;
retval = TALISE_openHw (device-> devHalInfo);
```

The example target system HAL function implementation is as follows:

```
ADIHAL_openHw(void * device) {
       zynqAdiDev_t *talDev = device->devHalInfo;
}
```

HAL Functions Timeout

For systems where HAL resources are shared, it is possible that HAL functions block and must wait for resources to complete the hardware operation. The HAL timeout variable, set by the API, instructs the HAL implementation with how long each function can take to try to complete the operation. If the operation fails to complete within this time interval, the HAL function must return indicating that the function timed out. See the adiHalErr_t section for details.

This timeout value is set by the API at HAL initialization via the `ADIHAL_openHW` function. Additionally, the API can update the timeout requirement by calling the HAL function `ADIHAL_setTimeout` from the API layer for operations that may have different time constraints.

It is expected that the HAL implementation of the end user captures and maintains the timeout value to implement the timeout feature for all subsequent HAL function calls. The HAL implementation can use the HAL device state container, `devHalInfo`, to maintain this timeout value.

HAL INTERFACE DEFINITION

This section describes the Hardware Application Layer (HAL) Interface.

DATA TYPES AND ENUMERATIONS

The following data types and enumerations are used by the HAL.

adiHalErr_t

`adiHalErr_t` is an enumerated data type that lists the error types returned by the Analog Devices HAL interface function. These error types include errors that the HAL implementation may detect and return to the transceiver API. The transceiver API reports any HAL layer error detected to the application. The transceiver API can fail due to a HAL error and then recommend a recovery action to the application. The recovery action recommended depends on the stage of API execution during which the error occurs, as well as the function of the API. Refer to the API documentation for more details.

adiHalErr_t Synopsis

The synopsis for `adiHalErr_t` is as follows:

```
typedef enum
{
    ADIHAL_OK=0,
    ADIHAL_SPI_FAIL,
    ADIHAL_GPIO_FAIL,
    ADIHAL_TIMER_FAIL,
    ADIHAL_WAIT_TIMEOUT,
    ADIHAL_GEN_SW,
    ADIHAL_WARNING
} adiHalErr_t;
```

adiHalErr_t Enumerators

The enumerators for `adiHalErr_t` are as follows:

- `ADIHAL_OK`. This enumerator indicates that the HAL function completed successfully and that no errors are detected.
- `ADIHAL_SPI_FAIL`. This enumerator indicates that HAL SPI operation is unable to complete. The SPI controller may not be accessible due to a fatal error. The API fails and directly passes the error to the application.
- `ADIHAL_GPIO_FAIL`. This enumerator indicates that HAL operations that require a system GPIO, such as `ADIHAL_resetHW`, are unable to complete. The system GPIO may not be accessible due to a fatal error. The API fails and passes the error to the application.
- `ADIHAL_TIMER_FAIL`. This enumerator indicates that HAL operations that require delays and sleeps, such as `ADIHAL_wait_us`, are unable to complete. System timers or time functions may not be accessible due to a fatal error. The API fails and passes the error to the application.
- `ADIHAL_WAIT_TIMEOUT`. This enumerator indicates that HAL operations that cannot complete within the interval set by the API return this error. This timeout may be due to the required hardware (SPI controller, GPIO, and so on) being temporarily unavailable. The timeout value is passed to the HAL layer by the API with the `ADIHAL_openHW` HAL initialization function and/or at any other time with the `ADIHAL_setTimeout` function. The API can fail depending on the function of the API and at what stage of execution of the API the error is detected. The error is passed directly to the API.
- `ADIHAL_GEN_SW`. This enumerator indicates that the HAL functions detect a general software error during execution. This error type includes, but is not limited to, open hardware or closed hardware failures, the device state container being a NULL pointer, an unknown chip select, or the device being already open. The API fails and passes the error to the application.
- `ADIHAL_WARNING`. This enumerator indicates that the HAL functions detect an error that does not affect operation. This error acts as a warning that such an error occurred. The API reports this error to the application, but does not take any action or indicate a failure.

adiLogLevel_t

adiLogLevel_t is an enumerated data type in bit mask format that lists the log message type received by the HAL layer from the API via the ADIHAL_writeToLog function.

adiLogLevel_t Synopsis

The synopsis for adiLogLevel_t is as follows:

```
typedef enum
{
    ADIHAL_LOG_NONE  = 0x0,
    ADIHAL_LOG_MSG   = 0x1,
    ADIHAL_LOG_WARN  = 0x2,
    ADIHAL_LOG_ERR   = 0x4,
    ADIHAL_LOG_SPI   = 0x8,
    ADIHAL_LOG_ALL   = 0xF

} adiLogLevel_t;
```

adiLogLevel_t Enumerators

The enumerators for adiLogLevel_t are as follows:

- ADIHAL_LOG_NONE. This enumerator indicates that the mask for no logging is enabled. This function can be optionally used by the ADIHAL_writeToLog implementation to ignore all error types. This function is not used by the API as an error category, but can be used by ADIHAL_setLogLevel.
- ADIHAL_LOG_MSG. This enumerator indicates the mask for a log message or warning. This enumerator does not indicate an error.
- ADIHAL_LOG_ERR. This enumerator indicates the mask for error message.
- ADIHAL_LOG_SPI. This enumerator indicates the mask error message related to SPI operation.
- ADIHAL_LOG_ALL. This enumerator indicates that the mask for all logging is enabled. This function can be optionally used by the ADIHAL_writeToLog implementation to represent all error types. This function is not used by the API as an error category, but can be used by ADIHAL_setLogLevel.

DETAILED HAL FUNCTION DEFINITIONS

The following sections describe the detailed HAL function definitions.

ADIHAL_openHW

The ADIHAL_openHW HAL function performs a platform hardware initialization for device. This function initializes all external hardware components required by the device and the HAL functions for correct functionality, such as SPI drivers, GPIOs, clocks (if necessary), as per the target platform and the target device requirements. At a minimum, any SPI driver for the device is initialized within this function for the SPI writes within the API to function.

The hardware initialize API calls the ADIHAL_openHW HAL function. For example, for the ADRV9008-1, ADRV9008-2, and ADRV9009 devices, the API calls the TALISE_openHw API. This API is called before any other API.

Based on the required operation, the API sets the value of the timeout parameter. After this value is set, the value serves as the timeout requirement for all HAL operations. It is the responsibility of the HAL implementation to maintain this timeout value as a reference. The HAL implementation must use this value to ensure the HAL function does not block longer than this time interval. Refer to the HAL Functions Timeout section for details.

ADIHAL_openHW must be called before calling any other HAL functions.

ADIHAL_openHW Synopsis

The synopsis for the ADIHAL_openHW function is as follows:

```
adiHalErr_t ADIHIAL_openHw(void *devHalInfo, uint32_t timeout_ms)
```

Parameters include the following:

- void*devHalInfo. This parameter is a void pointer to the targeted device state container. Refer to the devHalInfo Pointer Parameter section for details.
- uint32_t timeout_ms. This parameter is a positive integer representing the time interval (in ms) in which the HAL functions can block before returning. The HAL functions may not block indefinitely. The API provides this time value to the HAL.

Refer to the HAL Functions Timeout section for details.

Return value: an error of type adiHalErr_t, ADIHAL_OK indicates successful operation. Any other value may represent an error code to be returned to the application. Refer to the adiHalErr_t section for details.

Dependencies: the ADIHAL_openHW HAL function is dependent on application and platform specific modules.

ADIHAL_openHW Remarks

Although ADIHIAL_openHw initially sets the timeout value for HAL functions, the API may modify the timeout value by calling ADIHAL_setTimeout for any time sensitive operations.

For Analog Devices transceiver APIs, there is a requirement that only one thread can control and configure a particular device at any given time. For this reason, ADIHIAL_openHw may flag an ADIHAL_GEN_SW error to the API and application if the device targeted by the *devHalInfo parameter is already initialized.

ADIHAL_closeHW

The ADIHAL_closeHW HAL function performs a platform hardware shutdown for the device. This function shuts down all external hardware resources and peripherals required by the device for correct functionality, such as SPI drivers, GPIOs, clocks (as per the targeted platform), and the target device. The ADIHAL_closeHW HAL function closes and frees any resources assigned in the ADIHAL_openHw API.

The hardware shutdown API calls the ADIHAL_closeHW HAL function. For example, for the ADRV9008-1, ADRV9008-2, and ADRV9009 devices, the TALISE_closeHw API calls this function.

ADIHAL_closeHW Synopsis

The synopsis for the ADIHAL_closeHW function is as follows:

```
adiHalErr_t ADIHAL_closeHw(void *devHalInfo)
```

Parameter: void*devHalInfo. This parameter is a void pointer to the targeted device state container. Refer to the devHalInfo Pointer Parameter section for details.

Return value: an error of type adiHalErr_t, ADIHAL_OK indicates successful operation. Any other value may represent an error code to be returned to the application. Refer to the adiHalErr_t section for details.

Dependencies: the ADIHAL_closeHW HAL function is dependent on application and platform specific modules, as well as ADIHAL_openHW.

ADIHAL_closeHW Remarks

For Analog Devices transceiver APIs, there is a requirement that only one thread can control and configure a particular device at any given time. For this reason, ADIHAL_closeHw may flag an ADIHAL_GEN_SW error to the API and application if the device targeted by the *devHalInfo parameter is already closed.

ADIHAL_resetHW

Toggle the active low hardware reset pin, $\overline{\text{RESET}}$, of the device. To ensure a successful hardware reset of the device, pull the targeted device $\overline{\text{RESET}}$ pin low for period of at least 1 ms and then pull $\overline{\text{RESET}}$ high again. In general, for Analog Devices transceiver devices, low = 0 V and high = the VDD_INTERFACE value. The exact reset procedure is described in the data sheet of the targeted Analog Devices device data sheet.

ADIHAL_resetHW Synopsis

The synopsis for the ADIHAL_resetHW function is as follows:

```
adiHalErr_t ADIHAL_resetHw(void *devHalInfo)
```

Parameter: void*devHalInfo. This parameter is a void pointer to the targeted device state container. Refer to the devHalInfo Pointer Parameter section for details.

Return value: an error of type adiHalErr_t, ADIHAL_OK indicates successful operation. Any other value may represent an error code to be returned to the application. Refer to the adiHalErr_t section for details

Dependencies: the ADIHAL_resetHW HAL function is dependent on application and platform specific modules, as well as ADIHAL_openHW and ADIHAL_closeHW. In general, it is expected that ADIHAL_openHW initializes all hardware resources for the transceiver. This function depends on the GPIO. Therefore, it is expected that the required GPIO is initialized in ADIHAL_openHW. Do not call ADIHAL_resetHW before ADIHAL_openHW and not call ADIHAL_resetHW after ADIHAL_closeHW.

ADIHAL_resetHW Remarks

For the ADIHAL_resetHW function, devHalInfo contains information to determine which reference to a specific device on the platform is to be reset and any additional platform information required to toggle the $\overline{\text{RESET}}$ pin.

Do not make any communication attempts to the device for 100 µs following the toggle of the $\overline{\text{RESET}}$ pin. In general, a call to this function is followed by call to ADIHAL_wait_us by the API.

ADIHAL_setTimeout

The ADIHAL_setTimeout HAL function sets the timeout duration for the HAL functions. This function sets the timeout duration for the HAL functions. If the HAL operation exceeds this time, the function returns with the ADIHAL_WAIT_TIMEOUT error.

Based on the required operation, the API sets this timeout value. After this value is set, it serves as the timeout requirement for all HAL operations. The HAL implementation must maintain this timeout value as a reference and ensure that the HAL function does not block longer than this time interval.

ADIHAL_setTimeout Synopsis

The synopsis for the ADIHAL_setTimeout function is as follows:

```
adiHalErr_t ADIHAL_setTimeout(void *devHalInfo, uint32_t timeout_ms);
```

Parameters:

- void*devHalInfo. This parameter is a void pointer to the targeted device state container. Refer to the devHalInfo Pointer Parameter section for details.
- uint32_t timeout_ms. This parameter is a positive integer representing the time interval (in ms) in which the HAL functions can block before returning. The HAL functions cannot block indefinitely. The API provides this value to the HAL.

Refer to the HAL Functions Timeout section for details.

Return value: an error of type adiHalErr_t, ADIHAL_OK indicates successful operation. Any other value may represent an error code to be returned to the application. Refer to the adiHalErr_t section for details.

Dependencies: the ADIHAL_setTimeout HAL function is dependent on application and platform specific modules.

ADIHAL_setTimeout Remarks

The ADIHAL_setTimeout function is required for system environments in which HAL resources are shared and in cases where resources, such as the SPI, are functional but not available due to these resources being used by another device.

ADIHAL_spiWriteByte

The ADIHAL_spiWriteByte function performs a single SPI write to a device. Using the SPI interface protocol, write a single byte to a specified address within a specific device. The state container maintains the SPI chip select value to specify the target device and any other platform specific settings required for the platforms SPI controller, such as write bit polarity and a long instruction word.

The SPI write implementation must support 15-bit addressing and 8-bit data bytes. Full details of the SPI protocol required for SPI communication with the transceiver is described in the SPI section.

The ADIHAL_spiWriteByte HAL function is used by transceiver APIs. Therefore, any necessary SPI drivers or resources are expected to be already opened by the ADIHAL_openHw() function call.

ADIHAL_spiWriteByte Synopsis

The synopsis for the ADIHAL_spiWriteByte function is as follows:

```
adiHalErr_t ADIHAL_spiWriteByte(void *devHalInfo, uint16_t addr, uint8_t data);
```

Parameters inlcude the following:

- void*devHalInfo. This parameter is a void pointer to the targeted device state container. The device state provides the SPI device chip select information.
- addr. This parameter provides an SPI register address in which to perform an SPI write. The address value must not exceed 15-bits because the MSB bit is used for a read/write bit in the SPI implementation.
- data. This parameter provides an 8-bit data value to write to the specified SPI address.

Return value: an error of type adiHalErr_t, ADIHAL_OK indicates successful operation. Any other value may represent an error code to be returned to the application. Refer to the adiHalErr_t section for details.

Dependencies: the ADIHAL_spiWriteByte HAL function is dependent on application and platform specific modules, as well as ADIHAL_openHW and ADIHAL_closeHW. In general, it is expected that ADIHAL_openHW initializes all hardware resources for the transceiver. This function depends on the SPI. Therefore, it is expected that the required SPI is initialized in ADIHAL_openHW. Do not call ADIHAL_spiWriteByte before ADIHAL_openHW and not call ADIHAL_spiWriteByte after ADIHAL_closeHW.

ADIHAL_spiWriteByte Remarks

Analog Devices devices support various modes of the SPI protocol, such as 3-wire or 4-wire mode (see the SPI Data Transfer Protocol section). The HAL can implement any of the supported modes. However, the transceiver must be configured to the same mode as the HAL implementation. The SPI mode configuration of the transceiver is set by the initialization API and the desired SPI mode defined by taliseSpiSettings_t of the initialization parameter.

ADIHAL_spiReadByte

The ADIHAL_spiReadByte HAL function performs a single SPI read to a device. Using the SPI interface protocol, read a single byte from a specified address within a specific device. The state container maintains the SPI chip select value to specify the target device and any other platform specific settings required for the SPI controller, such as writ bit polarity and a long instruction word.

The SPI read implementation must support 15-bit addressing and 8-bit data bytes. Full details of the SPI protocol required for SPI communication with the transceiver is described in the SPI Data Transfer Protocol section.

This HAL function is used by most transceiver APIs. Therefore, any necessary SPI drivers or resources are expected to be already opened by the ADIHAL_openHw() function call.

ADIHAL_spiReadByte Synopsis

The synopsis for the ADIHAL_spiReadByte function is as follows:

```
adiHalErr_t ADIHAL_spiReadByte(void *devHalInfo, uint16_t addr, uint8_t *readdata);
```

Parameters include the following:

- void*devHalInfo. This parameter is a void pointer to the targeted device state container. The device state provides the SPI device chip select information.
- addr. This parameter provides an SPI register address on which to perform an SPI read. The address value must not exceed 15-bits because the MSB is used for a read/write bit in the SPI implementation.
- *readdata. This parameter is a pointer to the 8-bit variable to be updated with the value read from the SPI register address. The API layer allocates the memory for this pointer.

Return value: an error of type adiHalErr_t, ADIHAL_OK indicates successful operation. Any other value may represent an error code to be returned to the application. Refer to the adiHalErr_t section for details.

Dependencies: the ADIHAL_spiReadByte HAL function is dependent on application and platform specific modules, as well asADIHAL_openHW and ADIHAL_closeHW. In general, it is expected that ADIHAL_openHW initializes all hardware resources for the transceiver. This function depends on the SPI. Therefore, it is expected that the required SPI is initialized in ADIHAL_openHW. Do not call ADIHAL_spiReadByte before ADIHAL_openHW and do not call ADIHAL_spiReadByte after ADIHAL_closeHW.

ADIHAL_spiReadByte Remarks

Analog Devices devices support various modes of the SPI protocol, such as 3-wire or 4-wire mode (see the SPI Data Transfer Protocol section). The HAL can implement any of the supported modes. However, the transceiver must be configured to the same mode as the HAL implementation. The SPI mode configuration of the transceiver is set by the initialization API and the desired SPI mode defined by `taliseSpiSettings_t` of the initialization parameter.

ADIHAL_spiWriteBytes

The `ADIHAL_spiWriteBytes` HAL function performs a set of SPI writes to a device. A list of SPI addresses is passed to this function with a corresponding list of values to be written to these addresses. The `ADIHAL_spiWriteBytes` function performs an SPI write to the targeted device for each member in the arrays. Each address element corresponds to the same index element in the data array.

For example, Address 0 is the SPI address for Data 0, Address 1 is the SPI add for Data 1, and so on.

If the platform layer SPI driver has no way to write an array to the SPI driver, set this function to call `ADIHAL_spiWriteByte` in for a loop.

The `HAL_SPIWRITEARRAY_BUFFERSIZE` macro in **adi_hal.h** must be set to the maximum number of SPI transactions supported by the HAL implementation of `ADIHAL_spiWriteBytes`. The API layer references this macro when creating buffers of data to write with `ADIHAL_spiWriteBytes`.

ADIHAL_spiWriteBytes Synopsis

The synopsis for the `ADIHAL_spiWriteBytes` function is as follows:

```
adiHalErr_t  ADIHAL_spiWriteBytes(void *devHalInfo, uint16_t *addr, uint8_t *data, uint32_t
count);
```

Parameters include the following:

- `void*devHalInfo`. This parameter is a void pointer to the targeted device state container. The device state provides the SPI device chip select information.
- `*addr.` This parameter provides an array of addresses of SPI registers to write (15-bit SPI register addresses, the MSB in the SPI implementation sets the read/write bit).
- `*readdata.` This parameter provides an array of 8-bit data values to write to the SPI addresses listed in the address array.
- `count`. This parameter provides the number of registers on which to perform SPI writes (the size of the address and read data arrays).

Return value: an error of type `adiHalErr_t, ADIHAL_OK` indicates successful operation. Any other value represents an error code to be returned to the application. Refer to the adiHalErr_t section for details.

Dependencies: the `ADIHAL_spiWriteBytes` HAL function is dependent on application and platform specific modules, as well as `ADIHAL_openHW` and `ADIHAL_closeHW`. In general, it is expected that `ADIHAL_openHW` initializes all hardware resources for the transceiver. This function depends on the SPI. Therefore, it is expected that the required SPI is initialized in `ADIHAL_openHW`. Do not call `ADIHAL_spiWriteBytes` before `ADIHAL_openHW` and do not call `ADIHAL_spiWriteBytes` after `ADIHAL_closeHW`.

ADIHAL_spiWriteBytes Remarks

Analog Devices devices support various modes of the SPI protocol, such as 3-wire or 4-wire mode (see the SPI Data Transfer Protocol section). The HAL can implement any of the supported modes. However, the transceiver must be configured to the same mode as the HAL implementation. The SPI mode configuration of the transceiver is set by the initialization API and the desired SPI mode defined by `taliseSpiSettings_t` of the initialization parameter.

ADIHAL_spiReadBytes

The `ADIHAL_spiReadBytes` function performs a set of SPI reads from a device. A list of SPI addresses is passed to this function with a data array pointer to store the data read back from the list of SPI addresses. This function performs an SPI read from the list of SPI addresses in the targeted device. Each address element has a corresponding index in the data array.

For example, readdata[0] stores the data from the SPI address from dataarray[0].

If the platform layer SPI driver has no way to read an array from the SPI driver, set this function to call `ADIHAL_spiReadByte` in for a loop.

ADIHAL_spiReadBytes Synopsis

The synopsis for the ADIHAL_spiReadBytes function is as follows:

```
adiHalErr_t  ADIHAL_spiReadBytes(void *devHalInfo, uint16_t *addr, uint8_t *data, uint32_t
count);
```

Parameters include the following:

- void*devHalInfo. This parameter is a void pointer to the targeted device state container. The device state provides the SPI device chip select information.
- *addr. This parameter provides an array of addresses of SPI register to read (15-bit SPI register addresses, the MSB in SPI implementation sets the read/write bit).
- *readdata. This parameter provides an array of 8-bit data values to store the data read from the SPI addresses listed in the address array. The API layer allocates the memory for this pointer.
- count. This parameter provides the number of registers on which to perform the SPI reads (the size of the address and read data arrays).

Return value: an error of type adiHalErr_t, ADIHAL_OK indicates successful operation. Any other value may represent an error code to be returned to the application. Refer to the adiHalErr_t section for details.

Dependencies: the ADIHAL_spiReadBytes HAL function is dependent on application and platform specific modules, as well as ADIHAL_openHW and ADIHAL_closeHW.

In general, it is expected that ADIHAL_openHW initializes all hardware resources for the transceiver. This function depends on the SPI. Therefore, it is expected that the required SPI is initialized in ADIHAL_openHW. Do not call ADIHAL_spiReadBytes before ADIHAL_openHW and not call ADIHAL_spiReadBytes after ADIHAL_closeHW.

ADIHAL_spiReadBytes Remarks

Analog Devices devices support various modes of the SPI protocol, such as 3-wire or 4-wire mode (see the SPI Data Transfer Protocol section). The HAL can implement any of the supported modes. However, the transceiver must be configured to the same mode as the HAL implementation. The SPI mode configuration of the transceiver is set by the initialization API and the desired SPI mode defined by taliseSpiSettings_t of the initialization parameter.

ADIHAL_spiWriteField

The ADIHAL_spiWriteField HAL function performs a write to a specific field within an SPI register. This function performs a read/modify/write operation on a single SPI register. This function implements an SPI read of all 8-bit bits in the register, modifies the bits of a specific field, and writes the new value back to the same SPI register in the device.

The mask and startBit parameters specify the field of bits within the register to be modified. The mask is expected to be applied to the 8-bit value read back from register. The startBit parameter specifies the start bit of targeted field within the register. The value of startBit is used to shift the fieldVal parameter to the correct starting bit in the SPI register.

An example of pseudo code is as follows:

```
 |___****_|  Mask = 0x1E     startBit = 1

      /*Read a specific register*/
spiRead(regAddr, &regVal);

/* Modify specific field*/
regVal = regVal & ~mask) | ((fieldVal << startBit) & mask);

/*Write the modified value back to the SPI register*/
spiWrite(regAddr, regVal);
```

The SPI write/read implementations must support 15-bit addressing and 8-bit data bytes. Full details of the SPI protocol required for SPI communication with the transceiver is described in the SPI Data Transfer Protocol section.

This HAL function is used by most transceiver APIs. Therefore, any necessary SPI drivers or resources are expected to be already opened by the ADIHAL_openHw() function call.

ADIHAL_spiWriteField Synopsis

The synopsis for the `ADIHAL_spiWriteField HAL` is as follows:

```
adiHalErr_t ADIHAL_spiWriteField(void *devHalInfo, uint16_t addr, uint8_t  fieldVal, uint8_t
mask, uint8_t startBit);
```

Parameters include the following

- `void*devHalInfo`. This parameter is a void pointer to the targeted device state container. The device state provides the SPI device chip select information.
- `addr`. This parameter provides the SPI register address on which to perform an SPI read. The address value must not exceed 15 bits because the MSB bit is used for the read/write bit in the SPI implementation.
- `fieldVal`. This parameter provides the desired new value for the targeted field within the register specified by `addr` parameter.
- `mask`. Field mask, describing the targeted bits within the register specified by `addr` parameter.
- `startBit`. This parameter provides the field LSB position in the register (0 to 7).

Return value: an error of type `adiHalErr_t, ADIHAL_OK` indicates a successful operation. Any other value may represent an error code to be returned to the application. Refer to the adiHalErr_t section for details.

Dependencies: the `ADIHAL_spiWriteField` HAL function is dependent on application and platform specific modules, as well as `ADIHAL_openHW` and `ADIHAL_closeHW`. In general, it is expected that `ADIHAL_openHW` initializes all hardware resources for the transceiver. This function depends on the SPI. Therefore, it is expected that the required SPI is initialized in `ADIHAL_openHW.` Do not call `ADIHAL_spiWriteField` before `ADIHAL_openHW` and not call `ADIHAL_spiWriteField` after `ADIHAL_closeHW`.

ADIHAL_spiWriteField Remarks

Analog Devices devices support various modes of the SPI protocol, such as 3-wire or 4-wire mode (see the SPI Data Transfer Protocol section). The HAL can implement any of the supported modes. However, the transceiver must be configured to the same mode as the HAL implementation. The SPI mode configuration of the transceiver is set by the initialization API and the desired SPI mode defined by `taliseSpiSettings_t` of the initialization parameter.

ADIHAL_spiReadField

The `ADIHAL_spiReadField` HAL function performs a read to a specific field within an SPI register. This function performs a read operation on a single SPI register.

The `mask` and `startBit` parameters specify the field of bits within the register to be read. The mask is expected to be applied to the 8-bit value read back from register. The startBit `startBit` specifies the LSB of the targeted field within the register, which is used to shift the desired field value down to the Bit 0 position.

An example of pseudo code is as follows:

```
 /*Read a specific register*/
 spiRead(regAddr, &regVal);
*fieldVal = ((regVal & mask) >> startBit);
```

The SPI read/write implementations must support 15-bit addressing and 8-bit data bytes. Full details of the SPI protocol required for SPI communication with the transceiver is described in the SPI section and the HAL Interface Definition section.

This HAL function is used by most transceiver APIs. Therefore, any necessary SPI drivers or resources are expected to be already opened by the `ADIHAL_openHw()` function call.

ADIHAL_spiReadField Synopsis

The synopsis for `ADIHAL_spiReadField` is as follows:

```
adiHalErr_t ADIHAL_spiReadField(void *devHalInfo, uint16_t addr, uint8_t *fieldVal, uint8_t
mask, uint8_t startBit);
```

Parameters include the following:

- `void*devHalInfo`. This parameter is a void pointer to the targeted device state container. The device state provides the SPI device chip select information.
- `addr`. This parameter provides an SPI register address on which to perform an SPI read. The address value must not exceed 15 bits because the MSB is used for the read/write bit in the SPI implementation.
- `fieldVal`. This parameter is a pointer variable to return the value of the desired field specified by the mask parameter.

- `mask`. Field mask, describing the targeted bits within the register specified by `addr` parameter.
- `startBit`. Field LSB bit position in the register (0 to 7).

Return value: an error of type `adiHalErr_t, ADIHAL_OK` indicates successful operation. Any other value may represent an error code to be returned to the application. Refer to the adiHalErr_t section for details.

Dependencies: the `ADIHAL_spiReadField` HAL function is dependent on application and platform specific modules, as well as `ADIHAL_openHW` and `ADIHAL_closeHW`. In general, it is expected that `ADIHAL_openHW` initializes all hardware resources for the transceiver. This function depends on the SPI. Therefore, it is expected that the required SPI is initialized in `ADIHAL_openHW.` Do not call `ADIHAL_spiReadField` before `ADIHAL_openHW` and not call `ADIHAL_spiReadField` after `ADIHAL_closeHW`.

ADIHAL_spiReadField Remarks

Analog Devices devices support various modes of the SPI protocol, such as 3-wire or 4-wire mode (see the SPI Data Transfer Protocol section). The HAL can implement any of the supported modes. However, the transceiver must be configured to the same mode as the HAL implementation. The SPI mode configuration of the transceiver is set by the initialization API and the desired SPI mode defined by `taliseSpiSettings_t` of the initialization parameter.

ADIHAL_wait_us

The `ADIHAL_wait_us` HAL function performs a thread blocking/sleeping delay of the specified time in milliseconds. This function must wait for this specified amount of time, at least.

ADIHAL_wait_us Synopsis

The synopsis for `ADIHAL_wait_us` is as follows:

```
uint32_t ADIHAL_wait_us(void *devHalInfo, uint32_t time_us);
```

Parameters include the following:

- `void*devHalInfo` is a void pointer to the targeted device state container. The device state provides details of the targeted device.
- `uint32_t time_us` is the desired amount of time in (in ms) for which the function blocks or thread sleeps.

Return value: an error of type `adiHalErr_t, ADIHAL_OK` indicates successful operation. Any other value may represent an error code to be returned to the application.

Dependencies: the `ADIHAL_wait_us` HAL function is dependent on application and platform specific modules

ADIHAL_wait_us Remarks

The `ADIHAL_wait_us` HAL function is used for delay/sleep between checking status events.

ADIHAL_writeToLog

The `ADIHAL_writeToLog` HAL function writes a message to a log file for debugging purposes. This function uses the `logLevel` parameter to specify what type of message, warning, or error to write to the log file. The API functions call this function to log errors detected during the execution of the transceiver API. Logging and file details are platform specific and the `devHalInfo` parameter is expected to provide this information.

ADIHAL_writeToLog Synopsis

The synopsis for `ADIHAL_writeToLog` is as follows:

```
adiHalErr_t ADIHAL_writeToLog(void *devHalInfo, adiLogLevel_t logLevel,
uint32_t errorCode, const char *comment);
```

Parameters include the following:

- `void*devHalInfo` is a void pointer to the targeted device state container. The device state provides details of the targeted device.
- `adiLogLevel_t logLevel` is an enumerated value to specify the type of log message being logged. The value is one of the following `adiLogLevel_t` enumerator types:
  - ADIHAL_LOG_MSG
  - ADIHAL_LOG_WARN
  - ADIHAL_LOG_ERR
  - ADIHAL_LOG_SPI; refer to adiLogLevel_t section for details
- `uint32_t errorCode` is a 32-bit integer value representing the API error code reference for the error being logged.
- `const char *comment` is a character array that holds the desired comment to represent the error.

Return value: an error of type `adiHalErr_t, ADIHAL_OK` indicates successful operation. Any other value may represent an error code to be returned to the application.

Dependencies: the `ADIHAL_writeToLog` HAL function is dependent on application and platform specific modules

ADIHAL_writeToLog Remarks

This function is used to write a message to a log file for debugging purposes.

ADIHAL_setLogLevel

The `ADIHAL_setLogLevel` HAL function sets the HAL logging options for the device. This function sets the desired logging options for the logging feature. The device API calls this function based on the logging options set by the API user in the API layer. The HAL implementation maintains this logging option and defines its use with the `ADIHAL_writeToLog` function. Logging and file details are platform specific and the `devHalInfo` parameter is expected to provide this information. The `devHalInfo` parameter can be used to maintain the logging options for a particular device.

ADIHAL_setLogLevel Synopsis

The synopsis for `ADIHAL_setLogLevel` is as follows:

```
adiHalErr_t ADIHAL_setLogLevel(void *devHalInfo, uint16_t halLogLevel)
```

Parameters inlcude the following:

- `void*devHalInfo` is a void pointer to the targeted device state container. The device state provides details of the targeted device.
- `uint16_t logLevel` is a bit mask to indicate the desired logging options. The value is one of the following `adiLogLevel_t` enumerator types:
  - ADIHAL_LOG_NONE
  - ADIHAL_LOG_MSG
  - ADIHAL_LOG_WARN
  - ADIHAL_LOG_ERR
  - ADIHAL_LOG_ALL
- `uint32_t errorCode` is a 32-bit integer value representing the API error code reference for the error being logged.
- `const char *comment` is a character array that holds the desired comment to represent the error.

Return value: an error of type `adiHalErr_t, ADIHAL_OK` indicates successful operation. Any other value may represent an error code to be returned to the application.

Dependencies: the `ADIHAL_setLogLevel` HAL function is dependent on application and platform specific modules

ADIHAL_setLogLevel Remarks

See the adiLogLevel_t and ADIHAL_writeToLog sections.

SOFTWARE INTEGRATION

The API package was developed on a Xilinx Zynq ZC706 reference platform and utilizes a Cortex®-A9 processor that runs a Linux variant. Using the provided API in a custom hardware and software environment is readily accomplished because the API was developed abiding by ANSI C constructs and maintaining Linux system call transparency. The ANSI C standard was followed to ensure agnostic processor and operating system integration with the API code.

IMPLEMENTING HARDWARE ABSTRACTION INTERFACE

Users develop code to target custom hardware platforms and therefore use different drivers for the peripherals, for example, SPI and GPIO, in comparison to the drivers chosen for the Analog Devices evaluation platform. The HAL interface is a library of functions that the API uses when the API must access the target platform hardware. The HAL is defined by **adi_hal.h**. The implementation of this interface is platform dependent and is implemented by the developer in the **adi_hal.c** file. The prototypes of the required functions defined in the **adi_hal.h** file must not be modified because doing so breaks the API.

Table 1 lists the functions required by the HAL interface for integration. For full details on the definition and required operation of these functions, see the Hardware Abstraction Layer section and the HAL Interface Definition section.

Table 1. HAL Interface Functions for User Integration

Function Name	Function Description
ADIHAL_openHw	Open and initialize all platform drivers/resources and peripherals required to control the device (SPI, timer, logging)
ADIHAL_closeHw	Close any resources opened by ADIHAL_openHw
ADIHAL_resetHw	Toggle the hardware reset signal for the device
ADIHAL_setTimeout	Set the maximum time in which the API expects the HAL functions to complete and return
ADIHAL_spiWriteByte	Write a single byte of data to a 15-bit register address on the targeted SPI device
ADIHAL_spiReadByte	Read a single byte of data from a 15-bit register address on the targeted SPI device
ADIHAL_spiWriteBytes	Write an array of data bytes to an array of register addresses on the targeted SPI device
ADIHAL_spiReadBytes	Read an array of data bytes from an array of register addresses on the targeted SPI device
ADIHAL_spiWriteField	Perform a read/modify/write to a bit field in a particular SPI register
ADIHAL_spiReadField	Read a particular bit field of data from a SPI register
ADIHAL_wait_us	Perform a wait/thread/sleep operation in units of µs
ADIHAL_setLogLevel	Mask to set the severity of information to write to the log (error/warning/message)
ADIHAL_writeToLog	Log a debug message (message/warning/error) from the API to the platform log

DEVELOPING THE APPLICATION

The **/src/app/example/headless.c** file provides a user example demonstrating top level configuration and control. The example application was written to support the control of one device. The API was written to support the control of multiple devices. Because the API is written in C, a pointer to a `taliseDevice_t` data structure is used to describe or point to a particular device. To support multiple devices, the application layer code must instantiate multiple `taliseDevice_t` structures to describe each physical device. Many initialization settings also have data structures that are defined by the API.

Include Files

The API has multiple **.h** header files. For core API functionality, Table 2 shows the mandatory **.h** header files that must be included in the application layer program. Optional add on API functions can be included if the application requires those features as shown in Table 3. Note that the places `typedef` definitions in files with **_types** suffixes, for example, **talise_types.h**. These **_types.h** files are included within their corresponding **.h** files and do no need to be manually included in the application layer code.

Table 2. Mandatory .h Header Files for the Application Layer

Mandatory Include Files	Description
talise.h	Core run-time functions
talise_error.h	Core error handling functions
talise_arm.h	Arm related functions
talise_cals.h	Calibration related functions
talise_gpio.h	GPIO related functions
talise_jesd204.h	JESD204B interface related functions
talise_radioctrl.h	Functions for controlling the radio
talise_rx.h	Receiver related functions
talise_tx.h	Transmitter related functions

Table 3. Optional .h Header Files for the Application Layer

Optional (Add On) Include Files	Description
talise_agc.h	Add on receiver automatic gain control (AGC) functionality

The **talise_reg_addr_macros.h** and **talise_arm_macros.h** files are not needed by the application layer and are only used directly by the API. The **talise_user.h** file does not need to be included in the application layer unless the application layer is required to access the exported variables contained in the file (receiver gain tables).

Note that the **talise_user.h** and **talise_user.c** files contain the default gain table as well as defines for API timeouts and SPI read intervals, which can be set as needed by the baseband integrated circuit (BBIC). The **talise_user.h** files are the only API files that the developer has permission to change.

API Data Structures

There are two top level data structures used by the API to allow multiple device support: the `taliseDevice_t` structure and the `taliseInit_t` structure. The `taliseDevice_t` structure identifies each instance of a physical device. The `taliseDevice_t` structure has two members: `devHalInfo` and `devStateInfo`.

The data structure is as follows:

```
typedef struct
{
    void *devHalInfo;           /*!< ADI_HAL Hardware layer settings pointer specific to this
Talise instance */
    taliseInfo_t devStateInfo;  /*!< Talise runtime state container */

} taliseDevice_t;
```

The `devHalInfo` member is defined as a void pointer and allows the user to define and pass any platform hardware settings to the platform HAL layer functions. For example, the `devHalInfo` member can contain information, such as which SPI chip select must be used for the physical device. The API does not use the `devHalInfo` member, and therefore does not define what information it should contain. Note that the API functions are shared across all instances of physical devices. The `devHalInfo` structure defined by the developer identifies which physical device is targeted (SPI chip select) when a particular API function is called. It can be necessary for the developer to store other hardware information unique to a particular device in this structure, for example, timer instances and log file information, to allow for multithreading. It is expected that only one thread use the API to a particular device.

The `devStateInfo` member of the `taliseDevice_t` structure is a run-time state container for the API. The application layer must allocate memory for this structure, but only the API writes to the structure. The application layer allocates the `devStateInfo` member with all zeroes. The API uses the `devStateInfo` member to keep up with the current state of the API (if has it been initialized or if the Arm loaded), as well as a debug store for any run-time data, such as error codes and error sources. It is not intended for the application layer to directly access the `devStateInfo` member, as the API functions are provided to access the last error code and source information.

The `taliseInit_t` structure is the second important top level structure. The core initialization settings to configure a device are stored in this structure and passed to the API initialization functions during the initialization phase. This structure contains the receiver/transmitter/observation receiver profile settings, system clock settings, JESD204B settings, and specific SPI controller settings. The application layer passes a pointer to an instance of the `taliseInit_t` structure for a particular device to the `TALISE_initialize()` API function to handle the majority of the device initialization. After initialization is complete, the `taliseInit_t` structure can be disposed of or deallocated.

The data structure is as follows:

```
typedef struct
{
    taliseSpiSettings_t   spiSettings;     /*!< SPI settings data structure */
    taliseRxSettings_t    rx;              /*!< Rx settings data structure */
    taliseTxSettings_t    tx;              /*!< Tx settings data structure */
    taliseObsRxSettings_t obsRx;           /*!< ObsRx settings data structure */
    taliseDigClocks_t     clocks;          /*!< Holds settings for CLKPLL and reference clock */
    taliseJesdSettings_t  jesd204Settings; /*!< Holds the JESD204B data link settings */


} taliseInit_t;
```

Receiver, Transmitter, and Observation Receiver API Profiles

The API is designed to be configured using a use case profile of settings for the receiver, transmitter, and observation receiver. The `taliseInit_t` structure shown in the API Data Structures section holds the profile information, which is used during initialization to configure the device. The same API is used to support the receiver only device as is used to support the transmitter and observation receiver devices and TDD devices in the family of transceivers. The API can be used for a receiver only configuration, as well as for a transmitter and observation receiver configuration. It is not possible to use a transmitter profile only, because the observation receiver configuration is necessary for the transmitter initialization and tracking calibrations to run properly.

For a receiver only device, the receiver member in the `taliseInit_t` structure must be initialized with a valid receiver profile. The transmitter and observation receiver members of the `taliseInit_t` structure are allocated, but these members can be initialized to all zeros if unused. Additionally, for a transceiver and observation receiver device, the receiver member can be initialized to zeros, and the transceiver and observation receiver must be initialized with a valid use case profile. The `TALISE_initialize` API function determines which profiles are valid based on a non-zero I/Q data rate and if the channels are enabled.

For example, `init->tx.txChannels`, `init->obsRx.obsRxChannelsEnable`, and `init->rx.rxChannels` members are used by the API to identify which profiles to verify and use during initialization. The profiles are verified by ensuring that the I/Q data rates for each enabled profile can be accomplished by the common digital clocks available to the datapath. If the profiles are not possible due to inconsistency in the necessary shared clocks, the API returns an error during initialization.

Because the profiles share common digital clock settings, the `taliseInit_t` structure has a single clock member to describe the clock PLL and digital clock information. The JESD204B framer and deframer settings are shared as well, and these settings are pulled out to the `taliseJesdSettings_t` member. The receiver and observation receiver member structures contain a member that allows each profile to select which framer to use.

The latest supported profiles can be obtained from the latest TTES. The profile information is stored in a file called **ProfileLUT.txt** that ships with the software. The software is capable of generating the C code for the initialization structure necessary for the settings chosen in the evaluation software. The autogenerated initialization structure provides the developer a with low risk path to creating a correct initialization structure that works on the Analog Devices evaluation platform with the API.

Initialization Data Structures

The API functions use a specific set of data structures. The application layer code is responsible for initializing these data structures. All API functions use a pointer to a `taliseDevice_t` structure to describe the device of interest, which allows the application layer to control multiple devices in a system using the same API library. It is imperative that structure initialization is complete before attempting system operation. The file **/src/app/example/talise_config.c** illustrates structure initialization. Explanations for each data structure can be found in the **talise.chm** document. Table 4 contains a list of the initialization structures used by the API.

Table 4. Initialization Data Structures Used in the API

Data Structure	Location	Description
taliseInit_t	**/src/devices /talise/talise_types.h**	Top level initialization structure.
taliseSpiSettings_t	**/src/devices/talise/talise_types.h**	This contains the SPI controller configuration, which is typically a subset of the platform hardware SPI settings.
taliseFir_t	**/src/devices/talise/talise_types.h**	Data structure to specify finite infinite response (FIR) filter settings.
taliseRxSettings_t	**/src/devices/talise/talise_types.h**	Data structure to specify receiver datapath settings.
taliseRxProfile_t	**/src/devices/talise/talise_types.h**	Data structure to specify settings for the current receiver specific use case profile.

Data Structure	Location	Description
taliseRxGainControl_t	**/src/devices/talise/talise_types.h**	Data structure to hold receiver gain control settings for initialization.
taliseTxSettings_t	**/src/devices/talise/talise_types.h**	Data structure to specify transceiver datapath settings.
taliseTxProfile_t	**/src/devices/talise/talise_types.h**	Data structure to specify settings for the current transceiver specific use case profile.
taliseObsRxSettings_t	**/src/devices/talise/talise_types.h**	Data structure to specify observation receiver datapath settings.
taliseORxGainControl_t	**/src/devices/talise/talise_types.h**	Data structure to specify observation receiver gain control settings for initialization.
taliseJesdSettings_t	**/src/devices/talise/talise_jesd204_types.h**	Data structure to specify JESD204B framer and deframer configuration information.
taliseJesd204bDeframerConfig_t	**/src/devices/talise/talise_jesd204_types.h**	Data structure to specify the settings for the deserializer and deframer configuration.
taliseJesd204bFramerConfig_t	**/src/devices/talise/talise_jesd204_types.h**	Data structure to specify JESD204B framer configuration settings.
taliseDigClocks_t	**/src/devices/talise/talise_types.h**	Data structure to specify digital clock settings.
taliseAgcCfg_t	**/src/devices/talise/talise_agc_types.h**	Data structure to specify AGC settings for AGC initialization.

Note that the `taliseAgcCfg_t` structure is not part of the `taliseInit_t` structure. The AGC configuration requires a large memory footprint for its initialization structure. Because the AGC may not be used for every use case, it is not required to allocate memory for the structure unless the AGC feature is used. If the AGC feature is desired for the main receiver datapath, allocate the `taliseAgcCfg_t` structure and pass a pointer to the structure to the `TALISE_setupRxAgc` API function.

API Error Handling and Debug

Each API function returns a `uint32_t` value that represents a recovery action. Instead of returning an error code, the API functions tell the application layer what action to take, due to a possible error in the API function call. The list of API recovery actions is short in comparison to the number of error codes used by the API, which allows the application layer to significantly reduce error handling logic. The possible recovery action return values are shown in Table 5.

Table 5. Possible Recovery Action Return Values

Recovery Action Name	Description
TALACT_NO_ACTION	API function completed successfully; no error handling action is required.
TALACT_WARN_RESET_LOG	Warning, log file was unable to be written to in the ADIHAL layer; function completed.
TALACT_WARN_RERUN_TRCK_CAL	Warning, tracking calibration reported an error that can cause RF performance degradation, however the system is still operational.
TALACT_WARN_RESET_GPIO	Warning, API function completed successfully, but had to circumvent use of BBIC GPIO pin.
TALACT_ERR_CHECK_TIMER	Error, the ADIHAL layer reported an error with the timer platform layer code.
TALACT_ERR_RESET_ARM	Error, the API detected an error that requires the Arm processor to be reset and the initialization calbrations to be rerun.
TALACT_ERR_RERUN_INIT_CALS	Error, the API detected an error that requires the initialization calibrations to be rerun. RF performance may be unacceptable.
TALACT_ERR_RESET_SPI	Error, the ADIHAL layer reported a platform hardware SPI failure. The SPI driver must be reset. The calling API function did not complete.
TALACT_ERR_RESET_GPIO	Error, the ADIHAL layer reported a platform hardware BBIC GPIO failure that prevented an API function from completing.
TALACT_ERR_CHECK_PARAM	A parameter range check or NULL pointer check in the API prevented the API function from completing successfully. Verify parameters are correct and retry function.
TALACT_ERR_RESET_FULL	Error, the API detected an error condition that requires a full hard reset of the device. Reinitialize the device completely.
TALACT_ERR_RESET_JESD204FRAMERA	Error, the API detected an error condition that requires reset/initialization of Framer A.
TALACT_ERR_RESET_JESD204FRAMERB	Error, the API detected an error condition that requires reset/initialization of Framer B.
TALACT_ERR_RESET_JESD204DEFRAMERA	Error, the API detected an error condition that requires reset/initialization of Deframer A.
TALACT_ERR_RESET_JESD204DEFRAMERB	Error, the API detected an error condition that requires reset/initialization of Deframer B.
TALACT_ERR_REDUCE_TXSAMPLE_PWR	Error, the API detected transceiver sample power has exceeded the maximum threshold set by the BBIC in the power amplifier (PA) protection feature.
TALACT_ERR_BBIC_LOG_ERROR	Error, the API detected an error condition that may or may not be a true error. The BBIC should log this reported error and determine if this is an actual error or not.

API Recovery Action, TALACT_NO_ACTION

The TALACT_NO_ACTION recovery action is returned when an API function completes successfully. There is no recovery action to be performed.

API Recovery Action, TALACT_WARN_RESET_LOG

The TALACT_WARN_RESET_LOG recovery action is returned if the ADIHAL layer returns a logging error, which can happen if the log file cannot be opened or written to. The API layer does not return this as an error because the error does not directly affect radio performance. Additionally, this recovery action does not prevent the API function from completing. It is recommended that the application layer attempt to close the log file and reopen to resolve the log file access issue.

API Recovery Action, TALACT_WARN_RERUN_TRCK_CAL

The TALACT_WARN_RERUN_TRCK_CAL recovery action is returned if the API detects a failure for a tracking calibration. A tracking calibration error usually is not catastrophic and likely only results in degraded RF performance. The application layer must attempt to recover by resetting the tracking calibration.

The application layer must call TALISE_enableTrackingCals() with a mask parameter that disables the tracking calibrations, and then call TALISE_enableTrackingCals() again enabling the desired tracking calibrations.

API Recovery Action, TALACT_WARN_RESET_GPIO

The TALACT_WARN_RESET_GPIO recovery action is returned in the event that the BBIC GPIO failed to operate correctly, but the API was successful in circumventing the error by using the SPI port or other control mechanism. Because the API was able to successfully complete the API function, the issue is not critical, but the application layer must attempt to debug and fix the issue reported by the ADIHAL with respect to the BBIC GPIO control. The API function TALISE_getErrCode() can be used to return the ADIHAL error code. Verify that the TALISE_getErrCode() function returns an error source of the ADIHAL. If an ADIHAL error code is returned, the application layer can use it to further debug the root cause of the error.

API Recovery Action, TALACT_ERR_CHECK_TIMER

The TALACT_ERR_CHECK_TIMER recovery action is returned if the ADIHAL returns an error reporting that the timer is not working as expected. The API uses the timer ADIHAL functions to perform thread blocking waits to ensure that the API does not poll the SPI bus with 100% utilization. If the timer is reporting an error from the ADIHAL, it is possible that the API function works correctly, but there can be an impact on the system, due to the incorrect usage of system resources.

The suggested application layer action is as follows:

1. Attempt to reset the timer resources.
2. Continue the use of API monitoring for future check timer recover action reports.
3. If continued reports of TALACT_ERR_CHECK_TIMER occur, a system diagnostic can be required for the particular hardware.

API Recovery Action, TALACT_ERR_RESET_ARM

The TALACT_ERR_RESET_ARM recovery action is returned if the API detects an issue with the Arm processor that requires a complete reset and reload of the Arm firmware. This type of action can be required if the communication interface to the Arm processor fails or if the Arm watchdog timer reports an error. These events are not expected in production code, but are failsafe mechanisms in the event of a catastrophic error.

The suggested application layer action is as follows:

1. Issue TALISE_setRxTxEnable() to disable the transmitter to keep the hardware in a safe state. If this action fails, a full reset is required.
2. Set the power amplifier and other RF front-end components in a powered down or initialization state.
3. Call TALISE_getErrCode() to determine the specific ADIHAL error code and verify that ADIHAL is the error source. Log the error code and source.
4. Dump the Arm memory if necessary for debug.
5. Dump the SPI registers if necessary for debug.
6. Reload the stream processor and Arm binary firmware files.
7. Continue with normal initialization sequence to run the initialization calibrations and to enable the tracking calibrations.

API Recovery Action, TALACT_ERR_RERUN_INIT_CALS

The TALACT_ERR_RERUN_INIT_CALS recovery action is returned if the API detects an error with the initialization calibrations. The error severity is high enough that rerunning all of the initialization calibrations is required. A full device reset is not required. It is also not required to reload the Arm firmware.

The suggested application layer action is as follows:

1. Set the power amplifier and other RF front-end components in powered down or initialization state.
2. Call TALISE_getErrCode() to determine the specific ADIHAL error code and verify that ADIHAL is the error source. Log the error code and source.
3. Read the Arm calibration status to log debug information on the calibration failure.
4. Call TALISE_getInitCalStatus().
5. Call TALISE_runInitCals() to rerun the initialization calibrations.
6. Call TALISE_waitInitCals() and TALISE_getInitCalStatus() to confirm that there is no error in the initialization calibrations.

API Recovery Action, TALACT_ERR_RESET_SPI

The TALACT_ERR_RESET_SPI recovery action is returned if the ADIHAL layer reports a HAL error when attempting a SPI read or write transaction. If the ADIHAL function returns a timeout error due to the SPI hardware being busy or used by another thread, the API attempts to retry the SPI operation once. If the SPI transaction fails again, the API reports this recovery action. This action is also returned if an ADIHAL error is returned due to inability to access the driver.

The suggested application layer action is as follows:

1. Call TALISE_getErrCode() to determine the specific ADIHAL error code and verify that ADIHAL is the error source.
2. Log the error code and source.
3. If the ADIHAL error is a timeout, the API function can be retried.
4. If the ADIHAL error is not a timeout, the application tries resetting the SPI driver and retrying the function call.
5. If recovery action persists, verify SPI communication with other SPI devices and assess the need for a BBIC system reset.

API Recovery Action, TALACT_ERR_RESET_GPIO

The TALACT_ERR_RESET_GPIO recovery action is returned if the ADIHAL layer reports a HAL error when attempting to control the BBIC GPIO pins. If the API function cannot circumvent the error, this action is returned. If the API can circumvent the error, only a warning is returned (TALACT_WARN_RESET_GPIO). Currently, the only BBIC GPIO pin ($\overline{\text{RESET}}$) used in the ADIHAL is to reset the device.

If this action is returned, the application layer attempts to reset the BBIC GPIO pins that are used within the ADIHAL layer of code. If the application layer can resolve the GPIO hardware driver issue, normal operation of the API can resume by retrying the failed API function.

API Recovery Action, TALACT_ERR_CHECK_PARAM

The TALACT_ERR_CHECK_PARAM recovery action is returned if an API parameter range check or a NULL parameter check failed. In the event that this recovery action is returned, the API function did not complete. It is expected that this recovery action is only found during the debug phase of developement. During application software development, this recovery action informs the developer to double check the value passed into the API function parameters. When the parameters are corrected to be in the valid range, or NULL pointers are corrected, recalling the function allows the API function to complete.

For debug, the developer can call the TALISE_getErrCode function to retrieve the last API error code and source. This information can then be passed into the TALISE_getErrorMessage function, which returns a string that describes the error in more detail.

If the application software passes the development test but this recovery action is returned in the field, a bug in the application layer is highly possible, causing an out of range or NULL pointer error.

API Recovery Action, TALACT_ERR_RESET_FULL

The TALACT_ERR_RESET_FULL recovery action is returned if an API function cannot complete due to a detected error. If the API cannot correct or circumvent the error, and the severity of the error requires a complete reset of the device, this action is returned.

The suggested application layer action is as follows:

1. Put system hardware in safe state by setting the power amplifier and other RF front-end components in powered down or initialization state, then hard reset the device (TALISE_resetDevice()).
2. Read the API error code information for debug by calling TALISE_getErrCode(). Dump the Arm memory, if necessary, and then dump the SPI registers, if necessary.
3. Reinitialize the device using a normal full initialization sequence.

API Recovery Action, TALACT_ERR_RESET_JESD204FRAMERA

The TALACT_ERR_RESET_JESD204FRAMERA recovery action is returned if an API function cannot complete due to a detected error with Framer A. This error requires reset or initialization of Framer A only, not the entire device, which allows the BBIC to reset Framer A without affecting traffic through Framer B. Possible framer errors that can corrected by a single framer reset or initialization include an invalid data first in, first out (FIFO) pointer offset or a local multiframe counter (LMFC) pointer alignment error. These errors are reported as a framer IRQ and a general-purpose interrupt, GP_INTERRUPT.

The suggested application layer action is as follows:

1. Call the gpIntHandler() function that reports and clears the interrupt.
2. Reset Framer A.
3. Initialize Framer A.

API Recovery Action, TALACT_ERR_RESET_JESD204FRAMERB

The TALACT_ERR_RESET_JESD204FRAMERB recovery action is returned if an API function cannot complete due to a detected error with Framer B. This error only requires reset or initialization of Framer B, not the entire device. This reset feature allows the BBIC to reset Framer B without affecting traffic through Framer A. Possible framer errors that can becorrected by a single framer reset or initialization include an invalid data FIFO pointer offset or an LMFC pointer alignment error. These errors are reported as a framer IRQ and a general-purpose interrupt, GP_INTERRUPT.

The suggested application layer action is as follows:

1. Call gpIntHandler() function that reports and clears the interrupt.
2. Reset Framer B.
3. Initialize Framer B.

API Recovery Action, TALACT_ERR_RESET_JESD204DEFRAMERA

The TALACT_ERR_RESET_JESD204DEFRAMERA recovery action is returned if an API function cannot complete due to a detected error with Deframer A. This error only requires reset or initialization of Deframer A, not the entire device. This reset feature allows the BBIC to reset Deframer A without affecting traffic through Deframer B. Possible deframer errors that can be corrected by a single deframer reset or initialization include counter threshold overflows, and status errors, such as the following:

- Initial lane sync error (ILS)
- Interlane deskew error (ILD)
- Frame sync error (FS)
- Good checksum error (checksum)
- Code group sync error (CSG)

These errors are reported as a deframer IRQ and a general-purpose interrupt, GP_INTERRUPT.

The suggested application layer action is as follows:

1. Call the gpIntHandler() function that reports and clears the interrupt.
2. Reset Deframer A.
3. Initialize Deframer A.

API Recovery Action, TALACT_ERR_RESET_JESD204DEFRAMERB

The `TALACT_ERR_RESET_JESD204DEFRAMERB` recovery action is returned if an API function cannot complete due to a detected error with Deframer B. This error only requires reset or initialization of Deframer B, not the entire device. This reset feature allows the BBIC to reset Deframer B without affecting traffic through Deframer A. Possible deframer errors that can be corrected by a single deframer reset or initialization include counter threshold overflows and status errors. These errors are reported as a deframer IRQ and a general-purpose interrupt, GP_INTERRUPT.

The suggested application layer action is as follows:

1. Call the `gpIntHandler()` function that reports and clears the interrupt.
2. Reset Deframer B.
3. Initialize Deframer B.

API Recovery Action, TALACT_ERR_REDUCE_TXSAMPLE_PWR

The `TALACT_ERR_REDUCE_TXSAMPLE_PWR` recovery action is returned if an API function has determined that the transceiver sample power has exceeded the user specified maximum power threshold that is set in the power amplifier protection feature. The BBIC immediately reduces the transceiver power to protect hardware components in the datapath from damage, such as the power amplifier. These errors are reported as a power amplifier protection IRQ and a general-purpose interrupt, GP_INTERRUPT.

The suggested application layer action is as follows:

1. Call the `gpIntHandler()` function that ramps down the transceiver power, power-down the upconverter, and place the device in radio off. It is recommended that the application place the general-purpose interrupt handler of the BBIC on a high priority event thread to minimize the time required to service this interrupt.
2. Determine and correct the root cause of the over maximum power condition.
3. Reinitialize the device using a normal full initialization sequence.

API Recovery Action, TALACT_ERR_BBIC_LOG_ERROR

The `TALACT_ERR_BBIC_LOG_ERROR` API recovery action is returned if an API function has detected a condition that only the BBIC can determine if it is a true error or not. An example of this condition is a deframer error counter threshold overflow. If a deframer counter overflows once an hour or once a month, only the BBIC is able to determine if the counter overflow constituted an actual error condition.

The suggested application layer action is as follows:

1. Record the error.
2. Perform any BBIC determine recovery actions.

Modifying Receiver/Observation Receiver Gain Tables

The **/src/devices/talise/talise_user.h** and **/src/devices/talise/talise_user.c** files are provided to allow developers to include custom gain tables or other custom data necessary for use with the API. Developers can modify the **talise_user.c** and **talise_user.h** files.

Analog Devices provides default gain table settings with 0.5 dB gain steps for the receiver and the observation receiver in the **talise_user.c** file. The gain tables consist of a 2D array construct where the subarray order for each gain table type is described in a code comment at the beginning of the declaration. Each row of the 2D array specifies the gain breakdown between the RF analog gain, transimpedance amplifier (TIA), ADC, external gain, and digital gain for a particular gain index. The first row of each gain table represents the maximum gain index and is normalized by the API to Gain Index 255. If necessary, customers can modify the default gain tables as needed.

Note that the **talise_user.h** file exports the receiver and observation receiver gain tables as global variables. When using the API with multiple devices, each instance shares the default receiver and observation receiver gain tables in this file. If a custom gain table is required per device, the developer can create a gain table with a custom variable name. During the `TALISE_initialize()` function, the default gain table is loaded from the **talise_user.c** file. After initialization, call `TALISE_programRxGainTable()` by passing a pointer to the custom gain table as a parameter to load the table.

Restrictions

Developers are not permitted to modify any code located in the **/src/devices/*** folder other than changing the **adi_hal.c** code bodies for hardware driver insertion and gain table changes in **talise_user.c**. Analog Devices maintains the code in the **/src/devices/talise** and **/src/devices/ad9528** files. Analog Devices provides new releases to fix any code bugs in these folders.

No direct SPI read/write operation is permitted when configuring an Analog Devices clock chip device. Developers must only use the high level API functions defined in the **/src/devices/talise/talise.h**, **/src/devices/ad9528/ad9528.h**, or other **public .h** files. Users must not directly use any SPI read/write function located in the **adi_hal.h** file in the application layer code for configuration or control. Analog Devices does not support any customer code containing SPI writes that are reverse engineered from the original API.

Multithread and Multidevice Application Considerations

For applications with multiple transceivers, the API requires a reference to the targeted device and its hard and soft particulars, for example, SPI chip select, reset, and configuration status. The `taliseDevice_t` structure is used to identify each instance of a physical device. See the API Data Structures section for more details

For multithreaded applications, it is required that only one thread control and configure instance of a physical device. Concurrent thread configuration of the same instance of a physical device is not supported by the API.

Delay, Wait, and Sleep Operations

A small number of APIs require some time to allow the hardware to complete internal configurations, for example, `TALISE_setRfPllFrequency()`. These APIs request the system to perform a wait or sleep operation by calling the HAL interface function `ADIHAL_wait_us`. If the HAL interface implemention of the target platform chooses to implement a thread/sleep operation, it is not permitted for the application to call another API targeting the same device. The application is required to let the wait/sleep operation and the API to complete before continuing with the configuration of the device.

The wait/sleep periods used by the API are defined in the **talise_user.h** file. The timeout period values are the recommended period required to complete the operation. Modifying these values is not recommended and can impact performance. During this timeout period, the status of the device is polled. The frequency of the polling the status during this timeout period can be modified by the user by adjusting the value of the polling interval.

SPI

The SPI bus provides the mechanism for digital control of the device by a BBP. Each SPI register is 8 bits wide, and each register contains control bits, status monitors, or other settings that control all functions of the device. This section is mainly an information only section that is meant to give the user an understanding of the hardware interface used by the BBP for controlling a device. All control functions are implemented using the API detailed within this reference manual.

SPI CONFIGURATION USING API FUNCTION

The SPI operation is configured by the `TALISE_setSpiSettings (taliseDevice_t *device, taliseSpiSettings_t *spi)` helper function. This function is called by `TALISE_initialize(taliseDevice_t *device, taliseInit_t *init)`. Users can configure SPI settings for the device with different SPI controller configurations by configuring member values of the `taliseSpiSettings_t` data structure.

The `taliseSpiSettings_t` data structure contains the following:

```
typedef struct
{
       uint8_t MSBFirst
uint8_t enSpiStreaming
uint8_t autoIncAddrUp
uint8_t fourWireMode
taliseCmosPadDrvStr_t cmosPadDrvStrength
} taliseSpiSettings_t;
```

Table 6. taliseSpiSettings_t Data Structure Parameters

Structure Member	Value	Function	Default Value
MSBFirst	0x00	Least significant bit first.	0x01
	0x01	Most significant bit first.	
enSpiStreaming	0x00	Disable software feature. See Multibyte Data Transfer section. Not implemented in Analog Devices platform layer.	0x00
	0x01	Enable software feature to improve SPI throughput. See Multibyte Data Transfer section. Not implemented in Analog Devices I platform layer.	
autoIncAddrUp	0x00	Autoincrement. Functionality intended to be used with SPI streaming. Sets address to autoincrement > next address = address − 1. Not implemented in Analog Devices platform layer.	0x01
	0x01	Autodecrement. Functionality intended to be used with SPI streaming. Sets address to autodecrement > next address = address + 1. Not implemented in Analog Devices platform layer.	
fourWireMode	0x00	SPI hardware implementation. Use 3-wire SPI (SDIO pin is bidirectional). Figure 8 shows example of SPI 3-wire mode of operation. Analog Devices field-programmable gate array (FPGA) platform always uses 4-wire mode.	0x01
	0x01	SPI hardware implementation. Use 4-wire SPI. Figure 6 and Figure 7 show examples of SPI 4-wire mode of operation. Default mode for Analog Devices FPGA platform.	

Structure Member	Value	Function	Default Value
cmosPadDrvStrength	TAL_CMOSPAD_DRV_1X	CMOS output pads can drive 2.5 pF load.	TAL_CMOSPAD_DRV_1X
	TAL_CMOSPAD_DRV_2X	CMOS output pads can drive 5 pF load.	
	TAL_CMOSPAD_DRV_3X	CMOS output pads can drive 7.5 pF load.	
	TAL_CMOSPAD_DRV_4X	CMOS output pads can drive 10 pF load.	
	TAL_CMOSPAD_DRV_5X	CMOS output pads can drive 12.5 pF load.	
	TAL_CMOSPAD_DRV_6X	CMOS output pads can drive 15 pF load.	
	TAL_CMOSPAD_DRV_8X	CMOS output pads can drive 20 pF load.	
	TAL_CMOSPAD_DRV_10X	CMOS output pads can drive 25 pF load.	

Any value that is not listed in the Table 6 is invalid.

The SPI error handling is managed by ADIHAL, which can return the `TALACT_ERR_RESET_SPI` error code. For more details, refer to the API Error Handling and Debug section of this document.

SPI BUS SIGNALS

The SPI bus consists of the $\overline{CS}$, SCLK, SDIO, and SDO signals.

$\overline{CS}$ Signal

$\overline{CS}$ is the active low chip select that functions as the bus enable signal driven from the BBP to a device. $\overline{CS}$ is driven low before the first SCLK rising edge and is normally driven high again after the last SCLK falling edge. The device ignores the clock and data signals while $\overline{CS}$ is high. $\overline{CS}$ also frames communication to and from the device and returns the SPI interface of the device to the ready state when it is driven high.

Forcing $\overline{CS}$ high in the middle of a transaction aborts part or all of the transaction. If the transaction is aborted before the instruction is complete or in the middle of the first data word, the transaction is aborted and the state machine returned to the ready state. Any complete data byte transfers prior to $\overline{CS}$ deasserting is valid, but all subsequent transfers in a continuous SPI transaction are aborted.

SCLK Signal

SCLK is the serial interface reference clock driven by the BBP to the device. It is only active when $\overline{CS}$ is low. The minimum SCLK frequency is 1 kHz. The maximum SCLK frequency is 50 MHz.

SDIO and SDO Signals

When configured as a 4-wire bus, the SPI utilizes two data signals: SDIO and SDO. SDIO is the data input line driven from the BBP to the device, and SDO is the data output from the device to the BBP in this configuration. When configured as a 3-wire bus, SDIO is used as a bidirectional data signal that receives and transmits serial data. In this mode, the SDO port is disabled.

The data signals are launched on the falling edge of SCLK and sampled on the rising edge of SCLK by the BBP and the device. SDIO carries the control field from the BBP to the device during all transactions, and the BBP carries the write data fields during a write transaction. In a 3-wire SPI configuration, SDIO carries the returning read data fields from the device to the BBP during a read transaction. In a 4-wire SPI configuration, SDO carries the returning data fields to the BBP.

The SDO and SDIO pins transition to a high impedance state when the $\overline{CS}$ input is high. The device does not provide any weak pull-ups or pull-downs on these pins. When SDO is inactive, it is floated in a high impedance state. If a valid logic state on SDO is required at all times, add an external weak pull-up/pull-down (10 kΩ value) on the printed circuit board (PCB).

SPI DATA TRANSFER PROTOCOL

The SPI is a flexible, synchronous, serial communication bus allowing seamless interfacing to many industry-standard microcontrollers and microprocessors. The serial input/output is compatible with most synchronous transfer formats, including both the Motorola® SPI and Intel® SSR protocols. The control field width for the device is limited to 16 bits, and multibyte input/output operation is allowed. The device cannot be used to control other devices on the bus; it only operates as a slave.

There are two phases to a communication cycle. Phase 1 is the control cycle, which is the writing of a control word into the device. The control word provides the serial port controller of the device with information regarding the data field transfer cycle, which is Phase 2 of the communication cycle. The Phase 1 control field defines whether the upcoming data transfer is read or write. This control field also defines the register address that is accessed.

Phase 1 Instruction Format

The 16-bit control field contains the following information:

- The R/$\overline{W}$ bit (Bit 15) of the instruction word, determines whether a read or write data transfer occurs after the instruction byte write. Logic high indicates a read operation; logic zero indicates a write operation.
- The [D14:D0] bits (Bits A[14:0]) specify the starting byte address for the data transfer during Phase 2 of the input/output operation.

All byte addresses, both starting and internally generated addresses, are assumed to be valid. That is, if an invalid address (undefined register) is accessed, the input/output operation continues as if the address space is valid. For write operations, the written bits are discarded, and read operations result in logic zeros at the output.

Single-Byte Data Transfer

When `enSpiStreaming` = 0, a single-byte data transfer is chosen. In this mode, $\overline{CS}$ goes active low, the SCLK signal activates, and the address is transferred from the BBP to device.

In LSB mode, the LSB of the address is the first bit transmitted from the BBP, followed by the next 14 bits in order from next LSB to MSB. The next bit signifies if the operation is read (set) or write (clear). If the operation is a write, the BBP transmits the next 8 bits from LSB to MSB. If the operation is a read, the device transmits the next 8 bits from LSB to MSB. When the final bit is transferred, the data lines return to their idle state and the $\overline{CS}$ line must be driven high to end the communication session.

In MSB mode, the first bit transmitted is the R/$\overline{W}$ bit that determines if the operation is a read (set) or write (clear). The MSB of the address is the next bit transmitted from the BBP, followed by the remaining 14 bits in order from next MSB to LSB. If the operation is a write, the BBP transmits the next 8 bits from MSB to LSB. If the operation is a read, the device transmits the next 8 bits from MSB to LSB. When the final bit is transferred, the data lines return to their idle state and the $\overline{CS}$ line must be driven high to end the communication session.

Multibyte Data Transfer

Because most registers in the API are not consecutive, using multibyte data transfer mode provides little benefit. The user determines if multibyte data transfer mode enhances control of the device in the end application in comparison to single command format. When `enSpiStreaming` = 1, a multibyte data transfer is allowed. In this mode, data transfers across the bus as long as the $\overline{CS}$ pin is low. The `autoIncAddrUp` controls how the address changes for subsequent writes or reads. When `autoIncAddrUp` = 1, the address increments from the starting address for each subsequent data transfer until $\overline{CS}$ is driven high. If the last register address is reached, the next address accessedis 0x000. When `autoIncAddrUp` = 0, the address decrements from the starting address for each subsequent data transfer. If Address 0x000 is reached, the next address that is accessed is the last register location defined in the register map. Address 0x000 is used to setup SPI interface, as well as functionality to soft reset the device. Uncontrolled data written to Address 0x000 can cause SPI missconfiguration or can reset the device. It is strongly recommended that any data transfer using the multibyte data transfer feature be controlled so that 0x000 is only written once at startup.

For multibyte data transfers in LSB mode, the LSB of the address is the first bit transmitted from the BBP, followed by the next 14 bits in order from next LSB to MSB. The next bit signifies if the operation is read (set) or write (clear). If the operation is a write, the BBP transmits the next 8 bits from LSB to MSB. After the MSB is received, the address increments or decrements based on the `autoIncAddrUp` parameter. The BBP then continues to transfer data in 8-bit words from LSB to MSB, until the operation is terminated by $\overline{CS}$ being driven high. If the operation is a read, the device transmits the next 8 bits from LSB to MSB. The device then changes the address and continues to transfer data in 8-bit words from LSB to MSB, until the operation is terminated by $\overline{CS}$ being driven high.

For multibyte data transfers in MSB mode, the same process is followed, except the first bit transferred indicates if the operation is read (set) or write (clear). The starting address is then transmitted by the BBP from MSB to LSB, followed by the data transfer from MSB to LSB. Address increments or decrements are still controlled by the `autoIncAddrUp` parameter.

Example: LSB First Multibyte Transfer, Autoincrementing Address

To complete a 4-byte write starting at Register 0x02A and ending with Register 0x02D in LSB first format, take the following steps when programming the master:

1. Ensure that `fourWireMode` = 1. The device is configured to work with 4-wire interface.
2. Ensure that `MSBFirst` = 0. SPI works in LSB first mode.
3. Ensure that `autoIncAddrUp` = 1. The address pointer automatically increments.
4. Ensure that `enSpiStreaming` = 1. A multibyte data transfer is allowed.
5. Force the $\overline{CS}$ line low and keep it low until the last byte is transferred.
6. Send the instruction word 0101 0100 0000 000_0 (the last 0 indicates a write operation) to select 0x02A as the starting address.

7. Use the next 32 clock cycles to send the data to be written to the registers from LSB to MSB for each 8-bit word.
8. Ensure that the $\overline{CS}$ line is driven high after the last bit has been sent to 0x02D to end the data transfer.

Example: MSB First Multibyte Transfer, Autodecrementing Address

To complete a 4-byte write starting at Register 0x02A and ending with Register 0x027 in LSB first format, take the following steps when programming the master:

1. Ensure that `MSBFirst` = 1; SPI works in MSB first mode.
2. Ensure that `autoIncAddrUp` = 0; the address pointer automatically decrements.
3. Ensure that `enSpiStreaming` = 1; a multibyte data transfer is allowed.
4. Force the $\overline{CS}$ line low and keep it low until the last byte is transferred.
5. Send the instruction word 0_000 0000 0010 1010 (the first 0 indicates a write operation) to select 0x02A as the starting address.
6. Use the next 32 clock cycles to send the data to be written to the registers from MSB to LSB for each 8-bit word.
7. Ensure the $\overline{CS}$ line is driven high after the last bit has been sent to 0x027 to end the data transfer.

TIMING DIAGRAMS

The diagrams in Figure 6 and Figure 7 illustrate the SPI bus waveforms for a single register write operation and a single register read operation, respectively. In the first figure, the value 0x55 is written to Register 0x00A. In the second value, Register 0x00A is read and the value returned by the device is 0x55. If the same operations are performed with a 3-wire bus, the SDO line in Figure 6 is eliminated, and the SDIO and SDO lines in Figure 7 are combined on the SDIO line. Note that both operations use MSB first mode and all data is latched on the rising edge of the SCLK signal.

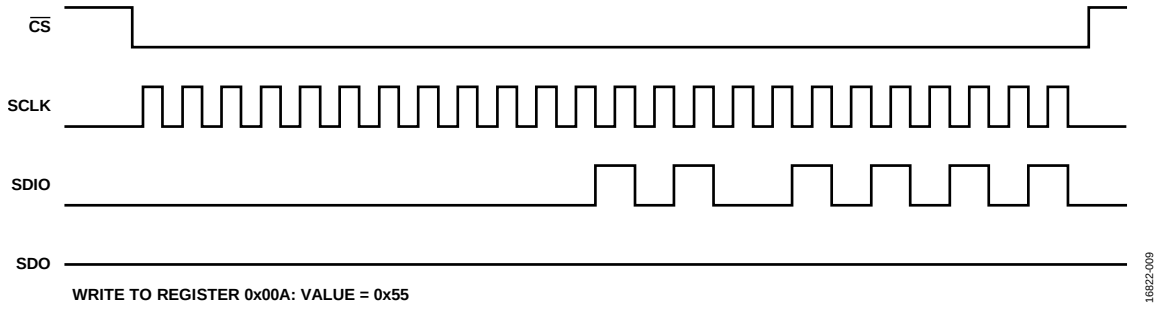

Figure 6. Nominal Timing Diagram, SPI Write

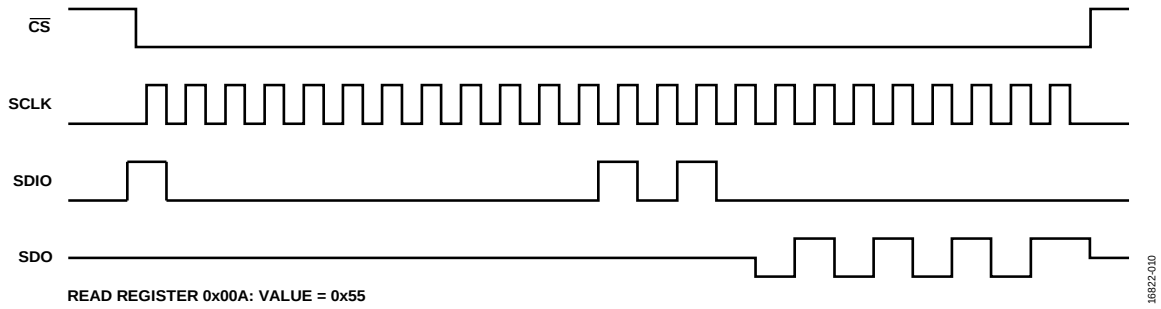

Figure 7. Nominal Timing Diagram, SPI Read

Table 7 lists the timing specifications for the SPI bus. The relationship between these parameters is shown in Figure 8. This diagram shows a 3-wire SPI bus timing diagram with the device returning a value of 0xD4 from Register 0x00A and timing parameters marked. Note that this is a single-read operation, so the bus ready parameter after the data is driven from the device (t_{HZS}) is not shown in the diagram.

Table 7. SPI Bus Timing Constraint Values

Parameter	Min	Typical	Max	Description
t_{CP}	20 ns		1ms	SCLK cycle time (clock period)
t_{MP}	10 ns			SCLK pulse width
t_{SC}	3 ns			$\overline{CS}$ setup time to first SCLK rising edge
t_{HC}	0 ns			Last SCLK falling edge to $\overline{CS}$ hold
t_S	3 ns			SDIO data input setup time to SCLK
t_H	0 ns			SDIO data input hold time to SCLK
t_{CO}	3 ns		8 ns	SCLK falling edge to output data delay (3-wire or 4-wire mode)
t_{HZM}	t_H		t_{CO}	Bus turnaround time after BBP drives the last address bit
t_{HZS}	3 ns		t_{CO}	Bus turnaround time after the device drives the last data bit

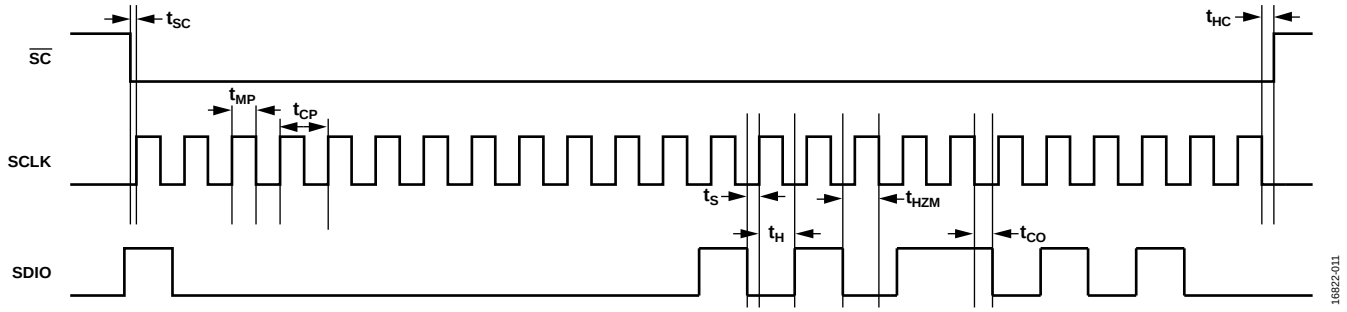

Figure 8. 3-Wire SPI Timing with Parameter Labels, SPI Read

JESD204B INTERFACE

The device employs the JESD204B Subclass 1 standard to transfer ADC and digital-to-analog (DAC) samples between the device and a BBP. The interface supports high speed serial lane rates of up to 12,288 Mbps. An external clock distribution solution provides a device clock and the SYSREF signal to the device and the BBP. The SYSREF signal ensures deterministic latency between the device and the BBP.

Note that the initialization sequence of the part is critical to guarantee deterministic latency. Specifically, the Arm initialization calibrations must be run before the JESD204B links are established, as described in the Device Initialization Sequence section of this document. It is also imperative to check the FIFO depth after the link has been established.

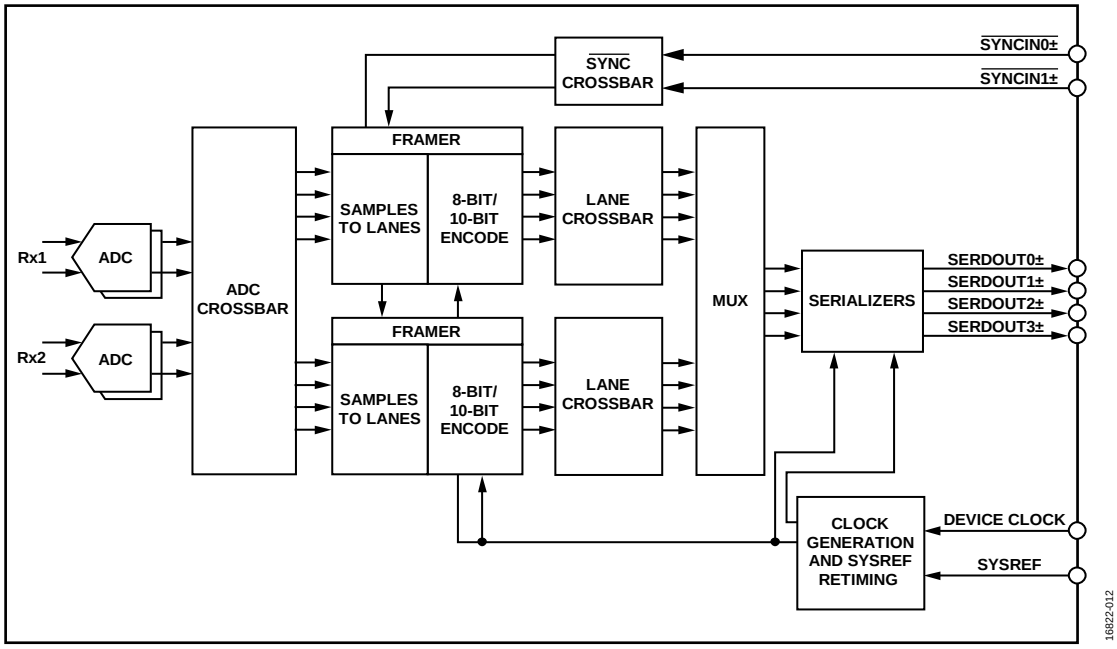

Figure 9. High Level JESD204B Interface Block Diagram (Receiver Only)

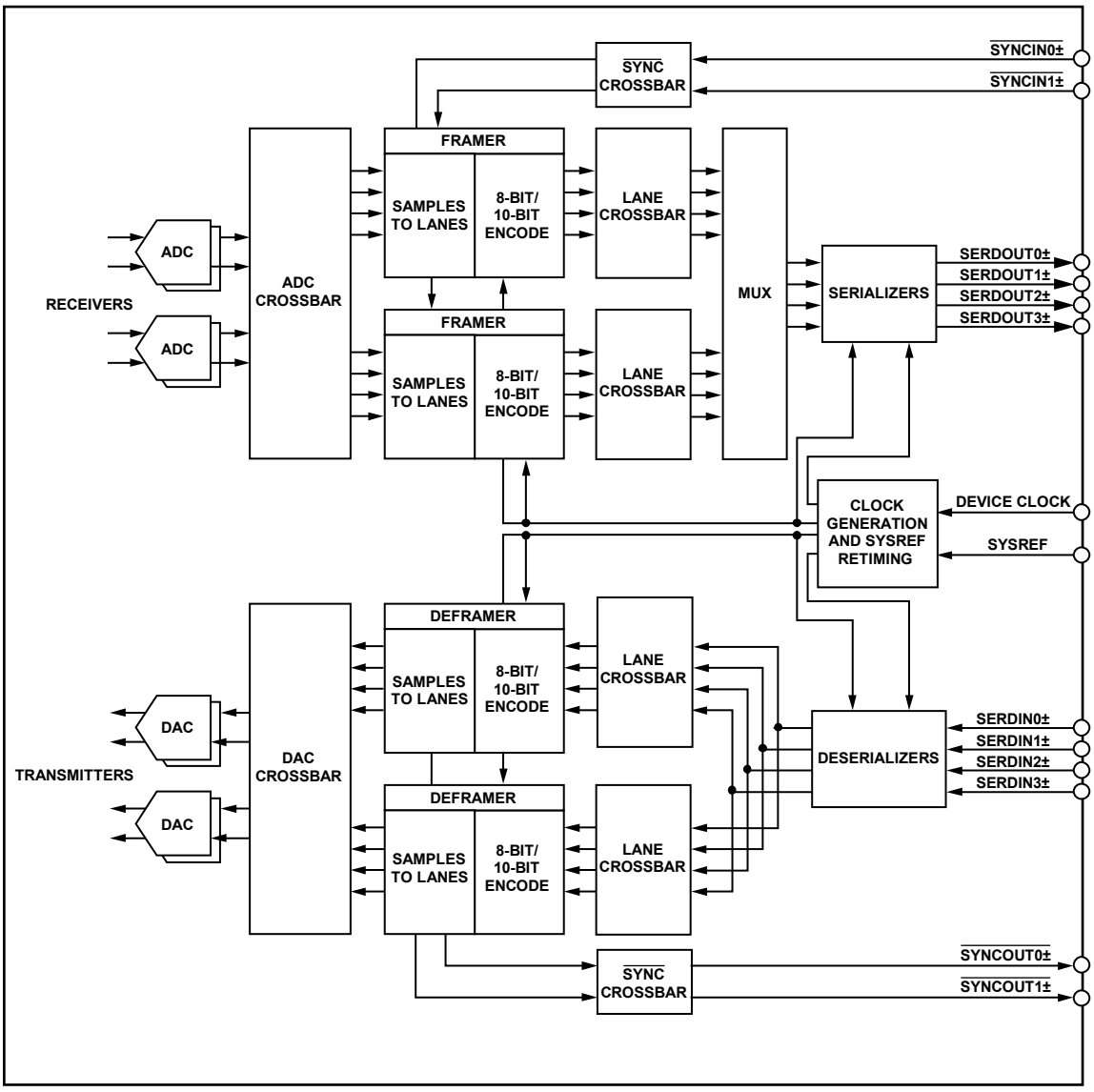

Figure 10. High Level JESD204B Interface Block Diagram (Transmitter Only)

API Software Integration

Configuration of the JESD204B circuitry is handled by the `TALISE_initialize()` API function. Set any JESD204B link options in the `taliseInit_t` data structure before calling `TALISE_initialize()`. The `taliseJesdSettings_t` member of the `taliseInit_t` data structure contains structures for each framer and deframer, serializers, and deserializers. Note that when configuring the `taliseJesdSettings_t` structure for a receiver only device, set the structure members for the two deframers to 0 and set the `taliseInit_t.tx.txChannels` member to 0.

JESD204B API Data Structures

taliseJesdSettings_t

The `taliseJesdsettings_t` data structure contains the information required to properly configure each framer, each deframer, the serializers, and deserializers. Details of each member can be found in the API documentation, which is provided with the software.

The data structure is as follows:

```
typedef struct
{
    taliseJesd204bFramerConfig_t framerA;
    taliseJesd204bFramerConfig_t framerB;
    taliseJesd204bDeframerConfig_t deframerA ;
    taliseJesd204bDeframerConfig_t deframerB ;
    uint8_t serAmplitude;
    unit8_t serPreEmphasis;
    uint8_t serInvertLanePolarity;
    uint8_t desInvertLanePolarity;
    uint8_t desEqSetting;
} taliseJesdSettings_t;
```

Table 8. JESD204B Settings Structure Member Description

Structure Member	Valid Values	Description
framerA	data structure	Framer A configuration data structure.
framerB	data structure	Framer B configuration data structure.
deframerA	data structure	Deframer A configuration data structure.
deframerB	data structure	Deframer B configuration data structure.
serAmplitude	0 to 15	Serializer amplitude setting. Default = 15.
serPreEmphasis	0 to 4	Serializer pre-emphasis setting. Default = 1.
serInvertLanePolarity	0x0 to 0xF	Serializer lane polarity inversion select, one bit per lane. If Bit 0 = 1, Lane 0 is inverted. If Bit 1 = 1, Lane 1 is inverted. If Bit 2 = 1, Lane 2 is inverted. If Bit 3 = 1, Lane 3 is inverted.
desInvertLanePolarity	0x0 to 0xF	Deserializer lane polarity inversion select, one bit per lane. If Bit 0 = 1, Lane 0 is inverted. If Bit 1 = 1, Lane 1 is inverted. If Bit 2 = 1, Lane 2 is inverted. If Bit 3 = 1, Lane 3 is inverted.
desEqSetting	0 to 4	Deserializer equalizer setting. Applied to all deserializer lanes.

RECEIVER (ADC) DATAPATH

The transport layer and link layer for JESD204B are performed in the device framers. The device has two JESD204B framers that multiplex into four serial lanes. Samples from the main receivers can be connected to either framer by the ADC crossbar. Each framer has its own $\overline{\text{SYNC}}$ signal, which allows one link to be brought down for reconfiguration without interrupting the other link.

The two framers are capable of operating at different sample rates. The higher sample rate must be a power of two multiple of the lower sample rate (2×, 4×, or 8×). There are two options to make this work: oversample at the framer input or bit repeat at the framer output.

Oversample mode repeats sample values at the framer input of the link with the slower sample rate, which allows all serializers to run at the same bit rate. In oversample mode, the BBP decimates the data after the transport layer to remove the extra samples.

Bit repeat mode repeats each bit at the framer output in the lane or lanes that carry the slower data before the data enters the serializer. Because this occurs after the 8B10B encoding, it appears as if the lane is running at a slower data rate than the other lanes, which essentially expands the eye of the signal. In bit repeat mode, the BBP must be able to configure the lane rates on the individual lanes independently. The lanes with the slower link must receive data at a slower lane rate than the lanes with the faster link.

Both framers must share the four serializers. Each framer must be configured for 0, 1, 2, or 4 lanes such that the two framers combine for no more than 4 lanes. If one framer uses all four lanes, then the other framer cannot be used.

Each framer is capable of generating a pseudorandom bit sequence (PRBS) on the enabled lanes. When the PRBS is enabled, errors can be injected. Enabling the PRBS generator can disable the normal JESD204B framing and cause the $\overline{\text{SYNC}}$ to deassert (if the ADC is the PRBS test source, $\overline{\text{SYNC}}$ does not deassert).

The serializers can be configured to adjust the amplitude and preemphasis of the physical signal to help combat bit errors due to various PCB trace lengths.

Supported Framer Link Parameters

The device supports a subset of possible JESD204B link configurations. The number of ADCs and the number of JESD204B lanes implemented in the silicon limit these configurations (see Table 9 and Table 10).

Table 9. Maximum JESD204B Framer Parameters

JESD204B Parameter	Device Value	Description
M	4	Number of converters (M can be 1, 2, or 4 per Table 10). JESD204B framer parameters dependent on number of lanes, number of ADCs, and number of samples.
L	4	Number of lanes (L can be 1, 2, or 4 per Table 10). JESD204B framer parameters dependent on number of lanes, number of ADCs, and number of samples.
S	4	Number of samples per converter per frame (S can be 1, 2, or 4 per Table 10). JESD204B framer parameters dependent on number of lanes, number of ADCs, and number of samples.
N	16	Converter resolution (N can be 12, 16, or 24).
N′	24	Total number of bits per sample (N′ can be 12, 16, or 24).
CF	0	Number of control words/frame clock cycles/converter devices.
CS	0	Number of control bits/conversion samples.
HD	0, 1	High density mode (only M1L2S1, M1L4S2, and M2L4S1 uses HD = 1).
K	Variable, suggested: 32	Number of frames in 1 multiframe, ($20 \le F \times K \le 256$), $F \times K$ must be a multiple of 4.

Table 10. JESD204B Framer Parameters Dependent on Number of Lanes, ADCs, and Samples

Number of ADCs (M)	Number of Lanes (L)	Number of Samples (S)	Number of Bits per Sample (N′)	Number of Octets in 1 Frame (F), $F = N'/8 \times M \times S/L$
1	1	1	16	2
1	1	2	16	4
1	1	4	16	8
1	2	1	16	1
1	2	2	16	2
1	2	4	16	4
1	4	2	16	1
1	4	4	16	2
2	1	1	16	4
2	1	2	16	8
2	2	1	16	2
2	2	2	16	4
2	2	4	16	8
2	4	1	16	1
2	4	2	16	2
2	4	4	16	4
4	1	1	16	8
4	2	1	16	4
4	2	2	16	8
4	4	1	16	2

Number of ADCs (M)	Number of Lanes (L)	Number of Samples (S)	Number of Bits per Sample (N')	Number of Octets in 1 Frame (F), F = N'/8 × M × S/L
4	4	2	16	4
4	4	4	16	8
2	1	2	12	6
4	1	1	12	6
4	2	1	12	3
2	2	2	24	6
4	2	1	24	6

For a particular converter sample rate, not all combinations listed in Table 10 are valid. For the JESD204B configuration mode to be valid, the lane rate for that mode must be within the range 3,686.4 Mbps to 12,288 Mbps. The lane rate is the serial bit rate for one lane of the JESD204B link.

Calculate the lane rate using Equation 1.

$$Lane\ Rate = I/Q\ Sample\ Rate \times M \times N' \times (10 \div 8) \div L \quad (1)$$

Serializer Configuration

The amplitude of the serializer is represented by a 4-bit number that is not linearly weighted. Not all settings are unique, and not all settings meet the JESD204B transmitter mask. The JESD204B transmitter mask requires a differential amplitude greater than 360 mV and less than 770 mV. To meet the JESD204B transmitter mask, it is recommended to set the serializer amplitude to a decimal value between 12 to 15. The default amplitude is 15.

The values shown in Table 11 are calculated values based on the design. Measured values are slightly lower than the calculated values. It is always recommended to verify the eye diagram in the system after building a PCB to verify any layout related performance differences.

Table 11. Serializer Amplitude Settings at 9.8304 Gbps

Serializer Amplitude (Decimal)	Average Differential Amplitude (mV p-p)
0	250
1	263
2	276
3	288
4	301
5	314
6	326
7	339
8	352
9	365
10	378
11	391
12	404
13	418
14	431
15	445

The serializer pre-emphasis allows boosting the amplitude any time the serial bit changes state. If no bit transition occurs, the amplitude is de-emphasized. Pre-emphasis helps open the eye diagram for longer PCB traces or when the parasitic loading of connectors has a noticeable effect. In most cases, to find the best setting, a simulation or measuring the eye diagram with a high speed scope at the receiver is recommended. The serializer pre-emphasis is represented by a 3-bit number. The range in differential amplitude can be seen in Table 11 and its effects are shown in Table 12.

Table 12. Pre-Emphasis Amplitude Settings at 9.8 Gbps and Amplitude of 15

Emphasis (Decimal)	Average Differential Amplitude (mV p-p)
0	445
1	457
2	464
3	439
4	365

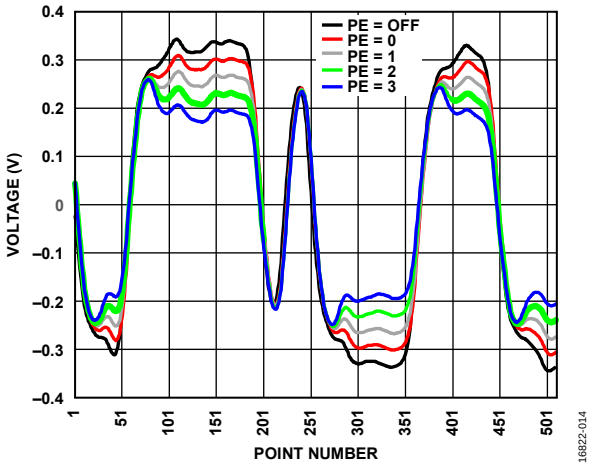

Figure 11. Serializer Preemphasis (PE) Measured on 3 Gbps Serial Data, Serializer

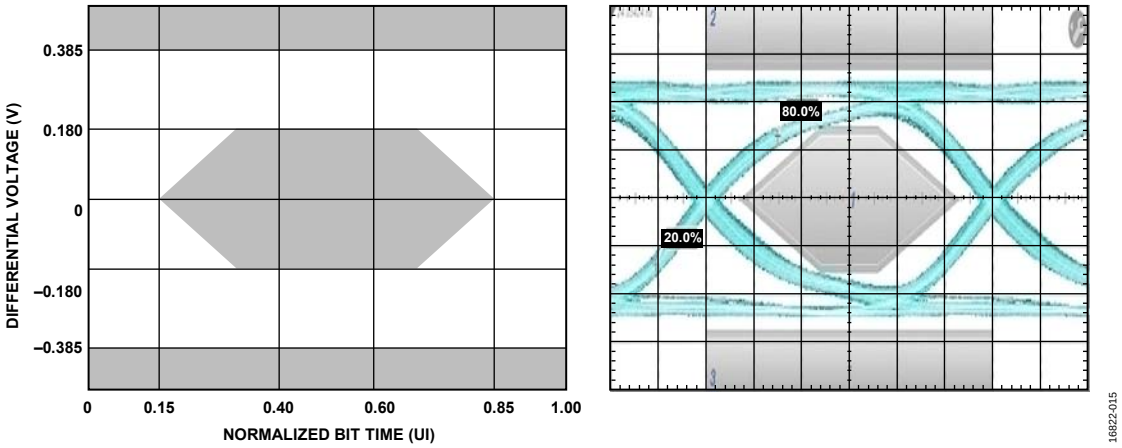

Figure 12. Example 9.8304 Gbps Eye Diagram at Serializer Output

Framer

Each framer receives 16-bit ADC samples and maps them to high speed serial lanes. The mapping changes depending on the JESD204B configuration chosen, specifically the number of lanes, the number of converters, and the number of samples per converter. Figure 13 provides one valid framer configuration for the device.

The responsibilities of the framer are as follows:

- JESD204B link initialization is the link state progresses from code ground synchronization (CGS) to initial lane assignment sequence (ILAS), then to user data.
- Character replacement allows frame and multiframe synchronization during user data.
- Map the ADC samples to the JESD204B lanes.
- Perform 8B10B encoding.

The ADC sample input into the framer passes through a sample crossbar. The sample crossbar allows any ADC output to map to any framed sample location in either framer during the framing process. For example, this can be used to swap I and Q samples or to send Receiver 1 data across one link and Receiver 2 data across the other link. The framer lane data outputs also pass through a lane crossbar, which allows mapping any framer output lane (internal to the silicon) to any physical JESD204B lane at the package pin. The framer packs the ADC samples into lane data following the JESD204B specification. Figure 13 shows the data packing for M = 2, L = 1, S = 1 as an example. For other configurations, refer to the Supported Framer Configurations section.

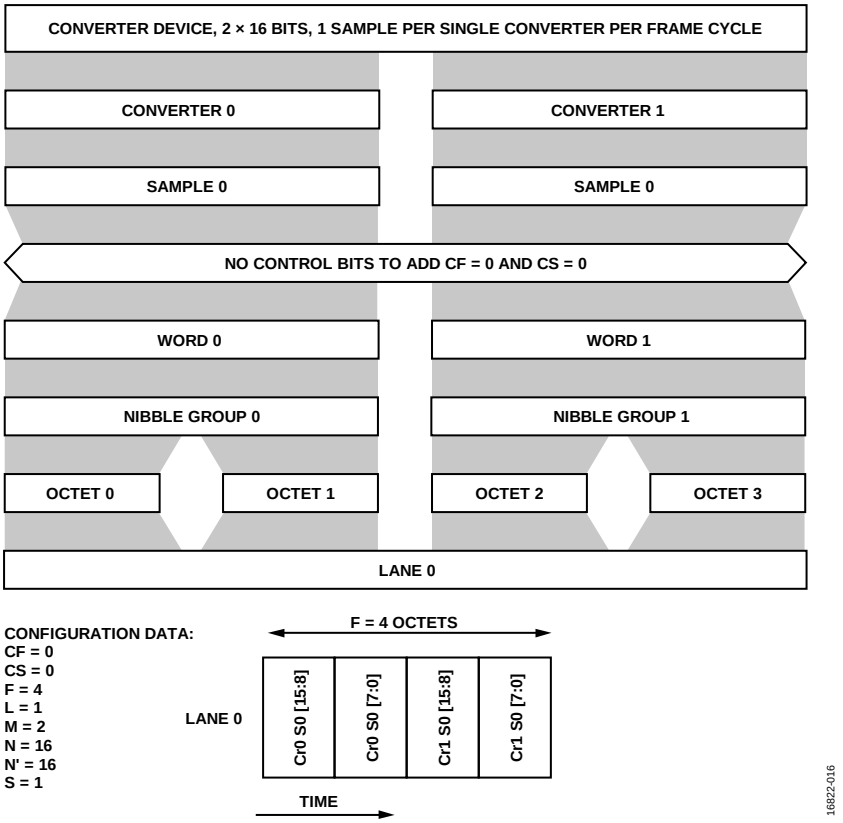

Figure 13. Framer Data Packing for M = 2, L = 1, S = 1

Other Useful Framer IP Features

PRBS Generator

The framer has a built in PRBS test pattern generator to aid in debugging the JESD204B serial link. The pattern generator is capable of generating PRBS7, PRBS9, PRBS15, PRBS23, or PRBS31 patterns. If errors caused by signal integrity exist, it can be difficult to get the JESD204B framer/deframer to work properly. The PRBS generator built into the framer allows the device to output serial data even when the link causes bit errors. With this mode enabled, the serializer amplitude and emphasis can be adjusted to find the best setting to minimize bit errors on the serial link. For this mode to be fully utilized, the baseband processor must have a PRBS checker to check the PRBS sequence for errors.

The typical usage sequence is as follows:

1. Initialize the device as outlined in the Link Establishment section.
2. Run the `TALISE_enableFramerTestData()` with the required framer, set the test data source to the desired PRBS order, and set the injection point to serializer input.
3. Enable the PRBS checker on the BBP and reset its error count.
4. Wait a specific amount of time to allow a good number of samples to be transmitted, and then check the PRBS error count of the BBP.

API Software Integration

The configuration of the serializer and both framers are all handled by the `TALISE_initialize()` API function. Set any JESD204B link options in the `taliseInit_t` data structure before calling `TALISE_initialize()`. After initialization, there are some other API functions to aid in debug and monitoring the status of the JESD204B link.

JESD204B Framer API Data Structures

taliseJesd204bFramerConfig_t

The `taliseJesd204bFramerConfig_t` data structure contains the information required to properly configure each framer. Details of each member can be found in API Documentation within the TTES software. The TTES has the option to output example data structures with values chosen from the configuration tab of the software.

The data structure is as follows:

```
typedef struct
{
    uint8_t bankId;
    uint8_t deviceId;
    uint8_t lane0Id;
    uint8_t M;
    uint8_t K;
    uint8_t F;
    uint8_t Np;
    uint8_t scramble;
    uint8_t externalSysref;
    uint8_t serializerLanesEnabled;
    uint8_t serializerLaneCrossbar;
    uint8_t lmfcOffset;
    uint8_t newSysrefOnRelink;
    uint8_t syncbinSelect;
    uint8_t overSample;
} taliseJesd204bFramerConfig_t;
```

Table 13. JESD204B Framer Configuration Structure Member Description

Structure Member	Valid Values	Description
bankId	0 to 15	JESD204B configuration bank ID, extension to device ID.
deviceId	0 to 255	JESD204B configuration device ID, link identification number.
lane0Id	0 to 31	JESD204B configuration lane I, if more than one lane is used, each subsequent lane increments from this number.
M	0, 2, 4	Number of ADC converters, 2 converters per receive chain.
K	1 to 32	Number of frames in a multiframe, default is 32. F × K must be a multiple of 4.
F	1, 2, 3, 4, 6, 8	Number of octets per frame.
NP	12, 16, 24	Number of bits per sample.
scramble	0 to 255	Scrambling enabled. If scramble = 0, scrambling is disabled. If scramble > 0, scrambling is enabled.
externalSysref	0 to 255	External SYSREF enabled. If externalSysref = 0, use internal SYSREF (not currently valid). If externalSysref > 0, use external SYSREF.
serializerLanesEnabled	0x0 to 0xF	Serializer lane enabled, one bit per lane. If Bit 0 = 0, Lane 0 is disabled. If Bit 0 = 1, Lane 0 is enabled. If Bit 1 = 0, Lane 1 is disabled. If Bit 1 = 1, Lane 1 is enabled. If Bit 2 = 0, Lane 2 is disabled. If Bit 2 = 1, Lane 2 is enabled. If Bit 3 = 0, Lane 3 is disabled. If Bit 3 = 1, Lane 3 is enabled.

Structure Member	Valid Values	Description
serializerLaneCrossbar	0x0 to 0xFF	Serializer lane crossbar, two bits per lane. Bits[1:0] identify the framer lane that connects to serializer Lane 0. Bits[3:2] identify the framer lane that connects to serializer Lane 1. Bits[5:4] identify the framer lane that connects to serializer Lane 2. Bits[7:6] identify the framer lane that connects to serializer Lane 3.
lmfcOffset	0 to 31	LMFC offset value for deterministic latency setting, set such that 0 ≤ lmfcOffset ≤ (K − 1)
newSysrefOnRelink	0 to 255	New SYSREF on relink, flag to indicate that a SYSREF is required to reestablish the link. If newSysrefOnRelink = 0, no SYSREF is required. If newSysrefOnRelink > 0, SYSREF is required.
syncinbSelect	0 to 1	$\overline{\text{SYNC}}$ selection, selects which $\overline{\text{SYNCINx±}}$ input is connected to the framer. If syncinBselect = 0, $\overline{\text{SYNCIN0±}}$ is connected to the framer. If syncinBselect = 1, $\overline{\text{SYNCIN1±}}$ is connected to the framer.
overSample	0 to 1	Oversample mode, selects which method is chosen when oversample or bit repeat is needed. If overSample = 0, bit repeat mode is selected. If overSample = 1, oversample is selected.

JESD204B Framer Enumerated Types

taliseFramerDataSource_t

The `taliseFramerDataSource_t` is an enumerated data type to select the framer test data source. The allowable values are listed in Table 14.

Table 14. Framer Data Source Enumeration Description

Enumeration Value	Description
FTD_ADC_DATA	Framer test data ADC data source.
FTD_CHECKERBOARD	Framer test data checkerboard data source. The output sequence of subsequent N-bits sample is generated by inverting the previous N-bits of the same data pattern at that clock cycle, counting from LSB to MSB.
FTD_TOGGLE0_1	Framer test data toggle 0 to 1 data source.
FTD_PRBS31	Framer test data PRBS31 data source.
FTD_PRBS23	Framer test data PRBS23 data source.
FTD_PRBS15	Framer test data PRBS15 data source.
FTD_PRBS9	Framer test data PRBS9 data source.
FTD_PRBS7	Framer test data PRBS7 data source.
FTD_RAMP	Framer test data ramp data source.

taliseFramerInjectPoint_t

The `taliseFramerInjectPoint_t` is an enumerated data type to select the framer test data injection point. The allowable values are listed in Table 15.

Table 15. Framer Injection Point Enumeration Description

Enumeration Value	Description
FTD_FRAMERINPUT	Framer test data injection point at framer input
FTD_SERIALIZER	Framer test data injection point at serializer input
FTD_POST_LANEMAP	Framer test data injection point after lane mapping

taliseFramerSel_t

The `taliseFramerSel_t` is an enumerated data type to select the desired framer. The allowable values are listed in Table 16.

Table 16. Framer Selection Enumeration Description

Enumeration Value	Description
TAL_FRAMER_A	Framer A selection.
TAL_FRAMER_B	Framer B selection.
TAL_FRAMER_A_AND_B	Used for cases where Rx1 uses one framer and Rx2 uses the second framer.

JESD204B Framer API Functions

TALISE_enableSysrefToFramer()

This function enables or disables the external SYSREF JESD204B signal connection to the framers of the transmitter. The function is as follows:

```
taliseErr_t TALISE_enableSysrefToFramer(taliseDevice_t *device, taliseFramerSel_t framerSel,
uint8_t enable);
```

For the framer to retime its LMFC, a SYSREF rising edge is required. The external SYSREF signal at the pin can be gated off internally so the framer does not see a potential invalid SYSREF pulse before it is configured correctly.

By default, the device has the SYSREF signal ungated. However, the multichip sync state machine does not allow the external SYSREF signal to reach the framer until the other stages of multichip sync have completed. As long as the external SYSREF signal is correctly configured before performing multichip sync, this function may not be needed by the BBIC because the multichip sync state machine gates the SYSREF signal to the framer.

The precondition for this function is that it must be called after the device has been initialized and the JESD204B framer is enabled.

This dependency of this function is `device->devHalInfo`.

This function has the following parameters:

- `device` is a pointer to the device settings structure.
- `framerSel` is the select framer to enable and disable SYSREF input for (valid `TAL_FRAMER_A`, `TAL_FRAMER_B` or `TAL_FRAMER_A_AND_B`).
- `enable` = 1 enables SYSREF to framer, and `enable` = 0 disables SYSREF to framer.

The return values for this function are as follows:

- `TALACT_WARN_RESET_LOG` is the recovery action for log reset.
- `TALACT_ERR_CHECK_PARAM` is the recovery action for bad parameter check.
- `TALACT_ERR_RESET_SPI` is the recovery action for SPI reset required.
- `TALACT_NO_ACTION` is the function completed successfully, no action required.

TALISE_readFramerStatus ()

This function reads back the status of the selected framer to determine the state of the JESD204B link. The `framerStatus` return value returns an 8-bit status word. The function is as follows:

```
taliseErr_t TALISE_readFramerStatus(taliseDevice_t *device, taliseFramerSel_t framerSel, uint8_t
*framerStatus);
```

Table 17. Framer Status Values

framerStatus Bit	Description
5	JESD204B negative edge count for $\overline{\text{SYNCIN0}}\pm$ = not greater than zero, and 1 = greater than zero.
4	Reserved (0).
3	Selects which $\overline{\text{SYNCINx}}\pm$ pin is used by the requested framer (0 = $\overline{\text{SYNCIN0}}\pm$ and 1 = $\overline{\text{SYNCIN1}}\pm$).
2	Current $\overline{\text{SYNCINx}}\pm$ level (1 = high, 0 = low).
1	SYSREF phase error, a new SYSREF had different timing than the first that set the LMFC timing.
0	Framer has received the SYSREF and has retimed its LMFC.

Precondition: the receiver JESD204B link(s) must be configured and running to use this function

Dependencies: `device->devHalInfo`.

Parameters include the following:

- `device` is a pointer to the device settings structure.
- `framerSel` is read back the framer status of the selected framer (Framer A or Framer B).
- `framerStatus` is the receiver framer status byte read.

Return values include the following:

- TALACT_WARN_RESET_LOG is the recovery action for log reset.
- TALACT_ERR_CHECK_PARAM is the recovery action for bad parameter check.
- TALACT_ERR_RESET_SPI is the recovery action for SPI reset required.
- TALACT_NO_ACTION is the function completed successfully, no action required.

TALISE_enableFramerTestData()

This function selects the PRBS type and enables or disables the receiver framer PRBS generation. This is a debug function to be used for debug of the receiver JESD204B lanes. Receiver data transmission on the JESD204B link(s) is not possible when the framer test data is activated. The function is as follows:

```
taliseErr_t TALISE_enableFramerTestData(taliseDevice_t *device, taliseframerSel_t framerSelect,
taliseFramerDataSource_t testDataSource, taliseFramerInjectPoint_t injectPoint);
```

Precondition: this function can be called any time after device initialization.

Dependencies: device->devHalInfo.

Parameters include the following:

- device is a pointer to the device settings structure.
- framerSel selects the framer of interest.
- testDataSource selects the desired test data pattern (normal datapath = FTD_ADC_DATA).
- injectPoint is the point in the datapath to inject the test data. Inject PRBS data into serializer for physical layer testing.

Return values include the following:

- TALACT_WARN_RESET_LOG is the recovery action for log reset.
- TALACT_ERR_CHECK_PARAM is the recovery action for bad parameter check.
- TALACT_ERR_RESET_SPI is the recovery action for SPI reset required.
- TALACT_NO_ACTION is the function completed successfully, no action required.

TALISE_injectFramerTestDataError

This function injects an error into the framer test data by inverting the data. This is the debug function to be used for debug of the receiver JESD204B lanes. Receiver data transmission on the JESD204B link(s) is not possible when the framer test data is activated. The function is as follows:

```
taliseErr_t TALISE_injectFramerTestDataError(TaliseDevice_t *device, taliseFramerSel_t
framerSelect);
```

Precondition: this function is called after the framer test data is enabled.

Dependencies: device->devHalInfo.

Parameters include the following:

- device is a pointer to the device settings structure.
- framerSel selects the framer of interest.

Return values include the following:

- TALACT_WARN_RESET_LOG is the recovery action for log reset.
- TALACT_ERR_CHECK_PARAM is the recovery action for bad parameter check.
- TALACT_ERR_RESET_SPI is the recovery action for SPI reset required.
- TALACT_NO_ACTION indicates that the function completed successfully, no action required.

TALISE_enableFramerLink

This function enables and disables the JESD204B framer. This function is normally not necessary. In the event that the link must be reset, this function allows a framer to be disabled and reenabled. The function is as follows:

```
TaliseErr_t TALISE_enableFramerLink(TaliseDevice_t *device, taliseFramerSel_t framerSelect,
uint8_t enable);
```

Precondition: this function can be called any time after device initialization.

Dependencies: `device->devHalInfo`.

Parameters include the following:

- `device` is a pointer to the device settings structure.
- `framerSel` selects the framer of interest.
- `enable` = 0, disables the selected framer, and `enable` = 1, enables the selected framer link.

Return values include the following:

- `TALACT_WARN_RESET_LOG` is the recovery action for log reset.
- `TALACT_ERR_CHECK_PARAM` is the recovery action for bad parameter check.
- `TALACT_ERR_RESET_SPI` is the recovery action for SPI reset required.
- `TALACT_NO_ACTION` is the function completed successfully, no action required.

TALISE_setupAdcSampleXbar()

This function sets the ADC sample crossbar to map the I/Q data from Receiver 1/Receiver 2/Observation Receiver 1/Observation Receiver 2 to the chosen JESD204B converter of the framer. The function is as follows:

```
uint32_t TALISE_setupAdcSampleXbar(taliseDevice_t *device, taliseFramerSel_t framerSel,
taliseAdcSampleXbar_t adcXbar);
```

Precondition: this function is called during JESD204B initialization.

Dependencies: `device->devHalInfo`.

Parameters include the following:

- `device` is a pointer to the device settings structure.
- `framerSel` selects Framer A or Framer B to set the ADC crossbar on its input where only `TAL_FRAMER_A` or `TAL_FRAMER_B` are valid choices.
- `adcXbar` is the ADC crossbar setting for the framer of choice.

Return values include the following:

- `TALACT_WARN_RESET_LOG` is the recovery action for log reset.
- `TALACT_ERR_CHECK_PARAM` is the recovery action for bad parameter check.
- `TALACT_ERR_RESET_SPI` is the recovery action for SPI reset required.
- `TALACT_NO_ACTION` indicates that the function completed successfully, no action required.

TRANSMITTERS (DAC) DATAPATH

The device has two JESD204B deframers that share four physical lanes. The two deframers feed a sample crossbar that connects to four DAC converters. All converters must run at the same sample rate and all lanes must run at the same data rate. The deframer is capable of receiving a PRBS sequence and accumulating error counts. The deserializers have adjustable equalization circuits (fixed setting, not adaptive) to counteract the insertion loss due to various PCB trace lengths and material.

Supported Deframer Link Parameters

The device supports a subset of possible JESD204B link configurations. The modes are limited by the number of DACs and the number of JESD204B lanes implemented in the silicon (see Table 18 and Table 19).

Table 18. Maximum JESD204B Deframer Parameters

JESD204B Parameter	Device Value	Description
M	4	Number of converters (M can be 1, 2, or 4 per Table 19).
L	4	Number of lanes (L can be 1, 2, or 4 per Table 19).
S	1	Samples transmitted per single converter per frame cycle.
N	16	Converter resolution (N can be 12 or 16).
N′	16	Total number of bits per sample (N′ can be 12 or 16).
CF	0	Number of control words/frame clock cycle/converter device.
CS	0	Number of control bits/conversion sample.
HD	0 or 1	High density mode. 0 is used for all cases except where specifically identified. Only M1L2 and M2L4 uses HD = 1 mode.
K	Variable, suggested: 32	Number of frames in 1 multiframe, (20 ≤ F × K ≤ 256), F × K must be a multiple of 4, K ≤ 32

Table 19. JESD204B Deframer Parameters Dependent on Number of Lanes and Number of DACs

Number of DACs (M)	Number of Lanes (L)	Number of Bits per Sample (N′)	Number of Octets in 1 Frame (F) (F = N′/8 × M/L)
1	1	16	2
1	2	16	1
2	1	16	4
2	2	16	2
2	4	16	1
4	1	16	8
4	2	16	4
4	4	16	2
4	2	12	3

For a particular converter sample rate, not all combinations listed in Table 19 are valid. For the JESD204B configuration mode to be valid, the lane rate for that mode must be within the range 2457.6 Mbps to 12,288 Mbps. The lane rate is the serial bit rate for one lane of the JESD204B link. Calculate the lane rate using Equation 1.

The deserializer link is allowed to run at a different lane rate than the serializer link, under the condition that both lane rates are possible with respect to the clock divider settings. Both the deserializer and serializer link rates are derived from the same clock PLL, but there are separate dividers to generate the deserializer clock and data recovery (CDR) clock and the serializer clock.

Deserializer Configuration

The deserializer includes an equalizer that can be set to a fixed setting to help in compensate for signal integrity distortions for each physical channel due to PCB trace length and impedance. Table 20 summarizes the amount of insertion loss each equalizer (EQ) setting can overcome. EQ settings can range from 0 (maximum boost) to 2 (default). Note that the measured length is the value at which the eye diagram is nearly failing the receive mask for each EQ setting at 10 Gbps.

Table 20. Measured Deserializer EQ Correction (Nomimal 1.3 V, 25°C)

EQ Settings	3 GHz Loss (dB)	6 GHz Loss (dB)	FR408HR Board Material Length (inches)	FR4 Board Material Length (inches)
0	17	31	30	25
1	15.5	26	20	20
2	12	21	15	15

Low power mode (equalizer setting = 2) is recommended if the insertion loss of the JESD204B PCB channels is less than 12 dB (at 6 Gbps) and 21 dB (at 10 Gbps). If the insertion loss is greater than this, one of the other settings can be appropriate. At 10 Gbps operation, typically around 4 mW, more power per lane is consumed for each step in the equalizer settings. Note that either setting can be used in conjunction with transmitter preemphasis to ensure functionality and/or to optimize for power. The equalizer setting can be changed in the API using the `desEqSetting` parameter in the function `InitJesd204bSerDes()`.

Deframer

The deframers receive 8B10B encoded data from the deserializer and decodes the data into 16-bit DAC samples. The deserializer to DAC sample mapping changes depending on the JESD204B link configuration setting.

The responsibilities of the deframer are as follows:

- Monitor the JESD204B link for running disparity errors (control $\overline{\text{SYNCOUTx±}}$ pin to reset link or report errors).
- Control the JESD204B interrupt signal (can output on GP_INTERRUPT pin) to signal BBP when certain JESD204B error conditions arise.
- Remove character replacement.
- Perform 8B10B decoding.
- Map JESD204B lane data to DAC samples.

A lane crossbar provides the ability to reorder the lanes into each deframer input. A sample crossbar provides the ability to reorder the DAC samples at the output of the deframers. The lane and sample crossbars enable flexibility on which physical lanes are used and which data is on each link. Figure 44 to Figure 52 show the allowable deframer configurations.

Other Useful Deframer IP Features

PRBS Checker

The deframer has a built in PRBS checker. The PRBS checker can self synchronize and check for PRBS errors on a PRBS7, PRBS15, or PRBS31 sequence. Because this mode can work in the midst of potential bit errors on each lane, the physical link can be debugged even when the deframer is unable to work properly. This mode can check the robustness of the physical link during initial testing and/or factory test. For this mode to be fully utilized, the BPP must have a PRBS generator capable of creating PRBS7, PRBS15, or PRBS31 data.

A typical usage sequence is as follows:

1. Initialize the device as outlined in the Link Establishment section.
2. Enable the PRBS generator on the BBP with the same PRBS sequence that is used on the device.
3. Call the API `TALISE_enableDeframerPrbsChecker()` function, transferring the actual device being evaluated, the PRBS sequence to check, and location where the check is to be done.
4. After some amount of time, call the API function `TALISE_readDeframerPrbsCounters()` to check the PRBS errors. This function transfers the actual device being evaluated and the counter selection lane to be read, and the error count is returned in the third parameter that is transferred.

To prove an error count of 0 is valid, the BBP can have a PRBS error inject feature. Alternatively, the BBP amplitude and emphasis settings can be set to a setting where errors occur. To reset the error count, call the API function that clears the counters, `TALISE_clearDeframerPrbsCounters()`.

API Software Integration

The configuration of the deserializer and the Transmitter 1/Transmitter 2 deframer is handled by the `TALISE_initialize()` API function. Set any JESD204B link options in the `TaliseInit_t` data structure before calling `TALISE_initialize()`. After initialization, there are some other API functions to aid in debug and monitoring the status of the JESD204B link.

JESD204B Deframer API Data Structures

TaliseJesd204bDeframerConfig_t

The `taliseJesd204bDeframerConfig_t` data structure contains the information required to properly configure each deframer. Details of each member can be found in the API documentation. The TTES has the option to output example data structures with values chosen from the **Configuration** tab of the software.

The data structure is as follows:

```
typedef struct
{
    uint8_t bankId;
    uint8_t deviceId;
    uint8_t lane0Id;
    uint8_t M;
    uint8_t K;
    uint8_t Np;
    uint8_t scramble;
    uint8_t externalSysref;
    uint8_t deserializerLanesEnabled;
    uint8_t deserializerLaneCrossbar;
    uint8_t lmfcOffset;
    uint8_t newSysrefOnRelink;
    uint8_t syncbOutSelect;
} TaliseJesd204bDeframerConfig_t;
```

Table 21. JESD204B Deframer Config Structure Member Description

Structure Member	Valid Values	Description
bankId	0 to 15	JESD204B configuration bank ID, extension to device ID.
deviceId	0 to 255	JESD204B configuration device ID, link identification number.
lane0Id	0 to 31	JESD204B configuration lane ID, if more than one lane is used, each subsequent lane increments from this number.
M	0, 2, 4	Number of ADC converters, 2 converters per receive chain.
K	1 to 32	Number of frames in a multiframe, default is 32. F × K must be a multiple of 4.
Np	12, 16	Number of bits per sample.
scramble	0 to 255	Scrambling enabled. If scramble = 0, scrambling is disabled. If scramble > 0, scrambling is enabled.
externalSysref	0 to 255	External SYSREF enabled. If externalSysref = 0, use internal SYSREF (not currently valid). If externalSysref > 0, use external SYSREF.
deserializerLanesEnabled	0x0 to 0xF	Deserializer lane enabled, one bit per lane. If Bit 0 = 0, Lane 0 is disabled, and if Bit 0 = 1, Lane 0 is enabled. If Bit 1 = 0, Lane 1 is disabled, and if Bit 1 = 1, Lane 1 is enabled. If Bit 2 = 0, Lane 2 is disabled, and if Bit 2 = 1, Lane 2 is enabled. If Bit 3 = 0, Lane 3 is disabled, and if Bit 3 = 1, Lane 3 is enabled.

Structure Member	Valid Values	Description
deserializerLaneCrossbar	0x0 to 0xFF	Deserializer lane crossbar, two bits per lane. Bits[1:0] identify the deserializer lane that connects to Deframer Lane 0. Bits[3:2] identify the deserializer lane that connects to Deframer Lane 1. Bits[5:4] identify the deserializer lane that connects to Deframer Lane 2. Bits[7:6] identify the deserializer lane that connects to Deframer Lane 3
lmfcOffset	0 to 31	LMFC offset value for deterministic latency setting, set such that 0 ≤ lmfcOffset ≤ (K − 1)
newSysrefOnRelink	0 to 255	New SYSREF on relink, flag to indicate that a SYSREF is required to reestablish the link. If newSysrefOnRelink = 0, no SYSREF is required. If newSysrefOnRelink > 0, SYSREF is required.
syncbOutSelect	0 to 1	$\overline{\text{SYNC}}$ selection, selects which $\overline{\text{SYNCOUTx±}}$ output is driven by the deframer. If syncbOutSelect = 0, the deframer drives $\overline{\text{SYNCOUT0±}}$. If syncbOutSelect = 1, the deframer drives $\overline{\text{SYNCOUT1±}}$.

JESD204B Deframer Enumerated Types

taliseDeframerSel_t

The `taliseDeframerSel_t` is an enumerated data type to select the desired deframer. The allowable enumerator values are listed in Table 22.

Table 22. Deframer Selection Enumerator Description

Enumeration Value	Description
TAL_DEFRAMER_A	Deframer A selection
TAL_DEFRAMER_B	Deframer B selection
TAL_DEFRAMER_A_AND_B	Used for cases where Tx1 uses one framer and Tx2 uses the second framer

taliseDeframerPrbsOrder_t

The `taliseDeframerPrbsOrder_t` is an enumerated data type to select the desired deframer PRBS pattern. The allowable enumerator values are listed in Table 23.

Table 23. Deframer PRBS Polynomial Order Enumerator Description

Enumerator Value	Description
TAL_PRBS_DISABLE	Deframer PRBS pattern disable
TAL_PRBS7	Deframer PRBS7 pattern select
TAL_PRBS15	Deframer PRBS15 pattern select
TAL_PRBS31	Deframer PRBS31 pattern select

taliseDefPrbsCheckLoc_t

The `taliseDefPrbsCheckLoc_t` is an enumerated data type to select the desired location within the deframer to check the PRBS pattern. The allowable enumerator values are listed in Table 24.

Table 24. Deframer PRBS Check Location Enumerator Description

Enumerator Value	Description
TAL_PRBSCHECK_LANEDATA	Deframer PRBS data check at lane data location
TAL_PRBSCHECK_SAMPLEDATA	Deframer PRBS data check at sample data location

JESD204B Deframer API Functions

TALISE_enableSysrefToDeframer()

This function enables or disables the external SYSREF to the deframers of the transmitter. The function is as follows:

```
TaliseErr_t TALISE_enableSysrefToDeframer(TaliseDevice_t *device, taliseDeframerSel_t
deframerSel, uint8_t enable);
```

For the deframer to retime its LMFC, a SYSREF rising edge is required. The external SYSREF signal at the SYSREF pin can be gated off internally so that the deframer does not see a potential invalid SYSREF pulse before the deframer is configured correctly.

By default, the device has the SYSREF signal ungated. However, the multichip sync state machine does not allow the external SYSREF signal to reach the deframer until the other stages of multichip sync have completed. As long as the external SYSREF is correctly configured before performing multichip sync, this function may not be needed by the BBIC because the multichip sync state machine gates the SYSREF signal to the deframer.

Precondition: this function is called after the device has been initialized and the JESD204B deframer is enabled.

Dependencies: `device->devHalInfo`.

Parameters include the following:

- `device` is a pointer to the device settings structure.
- `deframerSel` selects the deframer of interest.
- `enable` = 1 enables SYSREF to framer, and `enable` = 0 disables SYSREF to framer.

Return values include the following:

- `TALACT_WARN_RESET_LOG` is the recovery action for log reset.
- `TALACT_ERR_CHECK_PARAM` is the recovery action for bad parameter check.
- `TALACT_ERR_RESET_SPI` is the recovery action for SPI reset required.
- `TALACT_NO_ACTION` indicates that the function completed successfully, no action required.

TALISE_enableDeframerLink()

This function enables and disables the JESD204B deframer. The function is as follows:

```
taliseErr_t TALISE_enableDeframerLink(taliseDevice_t *device, taliseDeframerSel_t deframerSel,
uint8_t enable)
```

This function is normally not necessary. If the link must be reset, this function allows a deframer to be disabled and reenabled. During disable, the lane FIFOs for the selected deframer are also disabled. When the deframer link is enabled, the lane FIFOs for the selected deframer are reenabled (reset). The BBIC sends valid serializer data before enabling the link so that the CDR is locked.

Precondition: this function can be called any time after device initialization.

Dependencies: `device->devHalInfo`.

Parameters include the following:

- `device` is a pointer to the device settings structure.
- `deframerSel` selects the deframer of interest
- `enable` = 0 disables the selected deframer, and `enable` = 1 enables the selected deframer link.

Return values include the following:

- `TALACT_WARN_RESET_LOG` is the recovery action for log reset.
- `TALACT_ERR_CHECK_PARAM` is the recovery action for bad parameter check.
- `TALACT_ERR_RESET_SPI` is the recovery action for SPI reset required.
- `TALACT_NO_ACTION` indicates that the function completed successfully, no action required.

TALISE_readDeframerStatus()

After bringing up the device deframer JESD204B link, the BBP can check the status of the deframer. The function is as follows:

```
taliseErr_t TALISE_readDeframerStatus(TaliseDevice_t *device, taliseDeframerSel_t deframerSel,
uint16_t *deframerStatus);
```

Table 25. Deframer Status

deframerStatus Bit	Bit Name	Description
7	Valid checksum	This bit is equal to 1 if the checksum calculated by the device matches the checksum sent in the ILAS data.
6	EOF event	This bit captures the internal status of the framer end of frame event. Value = 1 if framing error during ILAS.
5	EOMF event	This bit captures the internal status of the framer end of multi-frame event. Value = 1 if framing error during ILAS.
4	FS lost	This bit captures the internal status of the framer frame symbol event. Value = 1 if framing error during ILAS or user data (invalid replacement characters).
3	LMFC out	Not useful to read across SPI.
2	User data valid	This bit is equal to 1 when in user data (deframer link is up and sending valid DAC data).
1	SYSREF received	Deframer has received the external SYSREF signal.
0	Sync error	A link synchronization error occurred.

Precondition: the transceiver JESD204B link(s) must be configured and running to use this function.

Dependencies: `device->devHalInfo`.

Parameters include the following:

- `device` is a pointer to the device settings structure.
- `deframerSel` selects the deframer of interest.
- `deframerStatus` is the 8-bit deframer status word return value.

Return values include the following:

- `TALACT_WARN_RESET_LOG` is the recovery action for log reset.
- `TALACT_ERR_CHECK_PARAM` is the recovery action for bad parameter check.
- `TALACT_ERR_RESET_SPI` is the recovery action for SPI reset required.
- `TALACT_NO_ACTION` indicates that the function completed successfully, no action required.

TALISE_enableDeframerPrbsChecker()

This function configures and enables or disables the lane or sample PRBS checker of the transceiver. Use this debug function to debug of the transceiver JESD204B lanes. The transmitter link(s) must be configured and operational to use this function. If the `checkerLocation` is `TAL_PRBSCHECK_LANEDATA`, the PRBS is checked at the output of the deserializer. If the `checkerLocation` is `TAL_PRBSCHECK_SAMPLEDATA`, the PRBS data is expected to be framed JESD204B data and the PRBS is checked after the JESD204B data is deframed. For the sample data, there is only a PRBS checker on Deframer Output 0. The lane PRBS has a checker on each deserializer lane. The function is as follows:

```
TaliseErr_t TALISE_enableDeframerPrbsChecker(TaliseDevice_t *device, taliseDeframerPrbsOrder_t
polyOrder, taliseDefPrbsCheckLoc_t checkerLocation);
```

Precondition: this function can be called any at time after device initialization.

Dependencies: `device->devHalInfo`.

Parameters include the following:

- `device` is a pointer to the device settings structure.
- `polyOrder` selects the PRBS type based on enumerator values (`TAL_PRBS_DISABLE`, `TAL_PRBS7`, `TAL_PRBS15`, and `TAL_PRBS31`).
- `checkerLocation` is the check at deserializer (Deframer Input 0 to Deframer Input 3) or sample (Deframer Output 0).

Return values include the following:

- TALACT_WARN_RESET_LOG is the recovery action for log reset.
- TALACT_ERR_CHECK_PARAM is the recovery action for bad parameter check.
- TALACT_ERR_RESET_SPI is the recovery action for SPI reset required.
- TALACT_NO_ACTION indicates that the function completed successfully, no action required.

TALISE_clearDeframerPrbsCounters()

This function allows the BBP to clear the deframer PRBS counters and resets the PRBS error counters for all lanes. It is recommended to clear the error counters after enabling the deframer PRBS checker. The function is as follows:

```
taliseErr_t TALISE_clearDeframerPrbsCounters(TaliseDevice_t *device);
```

Precondition: the transceiver JESD204B link(s) must be configured and running to use this function.

Dependencies: device->devHalInfo.

Parameter: device is a pointer for the device settings structure.

Return values include the following:

- TALACT_WARN_RESET_LOG is the recovery action for log reset.
- TALACT_ERR_CHECK_PARAM is the recovery action for bad parameter check.
- TALACT_ERR_RESET_SPI is the recovery action for SPI reset required.
- TALACT_NO_ACTION indicates that the function completed successfully, no action required.

TALISE_readDeframerPrbsCounters()

After enabling the deframer PRBS checker and clearing the PRBS error counters, use this function to read back the PRBS error counters. The lane parameter allows the BBP to select which lane error counter to read. Only one lane error counter can be read at a time. To read error counters for all four lanes, the BBP must call this function four times. The function is as follows:

```
taliseErr_t TALISE_readDeframerPrbsCounters(TaliseDevice_t *device, uint8_t lane, uint8_t
*prbsErrorCount, uint8_t *prbsInvertedStatus);
```

In the case that the PRBS checker is set to check at the deframer output sample, there is only a checker on the Deframer Sample 0 output. In this case, the lane function parameter is ignored and the Sample 0 PRBS counter is returned.

Precondition: the transmitter JESD204B link(s) must be configured and running to use this function.

Dependencies: device->devHalInfo.

Parameters include the following:

- device is a pointer to the device settings structure.
- lane specifies which lane to read the counter back for (valid 0, 1, 2, or 3). First silicon only reads back counter on Lane 0.
- prbsErrorCount is a return value of the 8-bit PRBS error count.
- prbsInvertedStatus is a return value of the bit mask indicating PRBS sequence is valid but inverted.
  - Bit 3 = Lane 3 PRBS sequence is valid but inverted.
  - Bit 2 = Lane 2 PRBS sequence is valid but inverted.
  - Bit 1 = Lane 1 PRBS sequence is valid but inverted.
  - Bit 0 = Lane 0 PRBS sequence is valid but inverted.

Return values include the following:

- TALACT_WARN_RESET_LOG is the recovery action for log reset.
- TALACT_ERR_CHECK_PARAM is the recovery action for bad parameter check.
- TALACT_ERR_RESET_SPI is the recovery action for SPI reset required.
- TALACT_NO_ACTION indicates that the function completed successfully, no action required.

TALISE_getDfrmIlasMismatch()

This function compares the received Lane 0 ILAS configuration to the deframer configuration and returns a 32-bit mask that indicates values that are mismatched. The actual Lane 0 ILAS configuration and deframer configuration values can be obtained by passing a pointer to the `taliseJesd204bLane0Config_t` structure type to the `dfrmCfg` and `dfrmIlas` function parameters individually or together. Passing a NULL pointer to either of these parameters results in no values returned for that parameter (see Table 26). The function is as follows:

```
Uint32_t TALISE_getDfrmIlasMismatch(TaliseDevice_t *device, taliseDeframerSel_t deframerSelect,
uint32_t *mismatch, taliseJesd204bLane0Config_t *dfrmCfg, taliseJesd204bLane0Config_t
*dfrmIlas);
```

Table 26. Deframer ILAS Mismatch

Mismatch Mask Bit	Description
17	Lane 3 checksum, 0 = match, 1 = mismatch
16	Lane 2 checksum, 0 = match, 1 = mismatch
15	Lane 1 checksum, 0 = match, 1 = mismatch
14	Lane 0 checksum, 0 = match, 1 = mismatch
13	HD, 0 = match, 1 = mismatch
12	CF, 0 = match, 1 = mismatch
11	S, 0 = match, 1 = mismatch
10	NP, 0 = match, 1 = mismatch
9	CS, 0 = match, 1 = mismatch
8	N, 0 = match, 1 = mismatch
7	M, 0 = match, 1 = mismatch
6	K, 0 = match, 1 = mismatch
5	F, 0 = match, 1 = mismatch
4	SCR, 0 = match, 1 = mismatch
3	L, 0 = match, 1 = mismatch
2	LID0, 0 = match, 1 = mismatch
1	BID, 0 = match, 1 = mismatch
0	DID, 0 = match, 1 = mismatch

Precondition: the transceiver JESD204B link(s) must be configured and running to use this function.

Dependencies: `device->devHalInfo`.

Parameters include the following:

- `device` is a pointer to the device settings structure.
- `deframerSel` is an enumerator indicating which deframer to address.
- `mismatch` is a pointer to a single `uint32_t` variable for reporting the ILAS match status, which is always returned.
- `dfrmCfg` is a pointer to a data structure that returns the deframer configuration settings. If this is returned as NULL, data is not returned to the pointer.
- `dfrmIlas` is a pointer to a data structure that returns the received Lane 0 ILAS settings. If this is returned as NULL, no data is returned to this pointer.

Return values include the following:

- `TALACT_WARN_RESET_LOG` is the recovery action for log reset.
- `TALACT_ERR_CHECK_PARAM` is the recovery action for bad parameter check.
- `TALACT_ERR_RESET_SPI` is the recovery action for SPI reset required.
- `TALACT_NO_ACTION` indicates that the function completed successfully, no action required.

TALISE_setupDacSampleXbar()

This function sets the DAC sample crossbar. In the event that less than four DACs are enabled for deframing, the least significant deframer outputs are used. The function is as follows:

```
uint32_t TALISE_setupDacSampleXbar(taliseDevice_t *device, taliseTxChannels_t channelSel,
taliseDacSampleXbar_t dacXbar);
```

Precondition: this function is called during JESD204B initialization.

Dependencies: `device->devHalInfo`.

Parameters include the following:

- `device` is a pointer to the device settings structure.
- `channelSel` is a `taliseTxChannels_t` enumerated data type for DAC crossbar channel selection, where only Transmitter 1 or Transmitter 2 are valid choices.
- `dacXbar` is an enumerated data type used to map any deframer output to a specific DAC channel I/Q converter input for the transceiver.

Return values:

- `TALACT_WARN_RESET_LOG` is the recovery action for log reset.
- `TALACT_ERR_CHECK_PARAM` is the recovery action for bad parameter check.
- `TALACT_ERR_RESET_SPI` is the recovery action for SPI reset required.
- `TALACT_NO_ACTION` indicates that the function completed successfully, no action required.

MULTICHIP SYNCHRONIZATION

Mulitchip synchronization is necessary when working with multiple transceivers or with one transceiver that requires deterministic latency between the transmit and observation or main receive JESD204B datapaths. Perform a mulitchip sync after the device is initialized. Additionally, this synchronization is used to achieve RF PLL phase synchronization. Refer to the RF PLL Phase Synchronization section for details.

When multichip sync is enabled, the function is performed in four stages; each one is initiated with a rising SYSREF edge. The first two SYSREF rising edges synchronize the device clock dividers. This portion of the synchronization requires some amount of time for the clock PLL outputs to settle. The third SYSREF rising edge synchronizes the high speed digital clock dividers. The fourth SYSREF rising edge synchronizes the numerically controlled oscillators (NCOs), the JESD204B LMFC, and the RF PLL phase synchronization.

For proper synchronization, the rising edge of SYSREF must be received at each device during the same device clock cycle. It is recommended to disable the SYSREF, enable multichip sync, and then reenable the SYSREF signal. The SYSREF signal can either be issued in individual pulses or be free running. If a periodic pulse is used, the phase must not change between rising edges. Single pulse mode can be implemented by gating a free running SYSREF after the falling edge to prevent runt pulses.

Multichip Synchronization API Functions

TALISE_enableMultichipSync()

This function sets up the chip for multichip sync and cleans up after the synchronization. The function is as follows:

```
TaliseErr_t TALISE_enableMultichipSync(TaliseDevice_t *device, uint8_t enableMCS, uint8_t
*mcsStatus);
```

When the `enableMcs` parameter = 1, this function resets the multichip sync state machine in the device. Calling the function again resets the state machine and expects the multichip sync sequence to start over.

When the `enableMcs` parameter = 0, the multichip sync state machine is not changed, allowing the user to read back the multichip sync status in the `mcsStatus` parameter.

Table 27. Multichip Sync Status

mcsStatus Value	Bit Name	Description
3	Device clock divider sync	This bit is equal to 1 if multichip sync device clock divider synchronization occurrs.
2	CLKPLL SDM sync	1 if multichip sync clock PLL Σ-Δ modulator (fractional mode) synchronization occurrs.
1	Digital clocks sync	1 if multichip sync digital clock divider synchronization occurrs.
0	JESD SYSREF sync	1 if multichip sync JESD204B SYSREF synchronization occurrs.

Precondition: this function is called after the device has been initialized and PLL lock status has been verified.

Dependencies: `device->devHalInfo`.

Parameters include the following:

- `device` is a pointer to the device settings structure.
- `enableMCs` = 1 enables/resets the multichip sync state machine.
- `mcsStatus` is an optional parameter, if pointer is not NULL, the function returns the `mcsStatus` word described in Table 27.

Return values include the following:

- `TALACT_WARN_RESET_LOG` is the recovery action for log reset.
- `TALACT_ERR_CHECK_PARAM` is the recovery action for bad parameter check.
- `TALACT_ERR_RESET_SPI` is the recovery action for SPI reset required.
- `TALACT_NO_ACTION` indicates that the function completed successfully, no action required.

LINK ESTABLISHMENT

After applying power to the device, the serializers, framers, deserializers, deframers, and the rest of the JESD204B circuits are powered down. Several steps are required to successfully power up the JESD204B link.

Suggested JESD204B API Initialization Sequence

The steps required to initialize the JESD204B links (both ADC and DAC datapaths) are as follows:

1. Initialize the `taliseInit_t` API data structure and substructures with the desired settings. The TTES can output **.c** or **.h** files with the data structures initialized to the values from the **Configuration** tab of the software.
2. Call the `TALISE_initialize()` API command to configure the device. This sets up the device to use the receiver, transceiver, or observation receiver profiles chosen, program the clock PLL and digital clocks, and set up the serializer, framer, deserializer, and deframer, for the profiles that are valid.
3. Initiate multichip synchronization by resetting the multichip sync state machine, followed by the first three SYSREF pulses. After each SYSREF pulse, monitor the multichip sync state machine status to verify that the synchronization stage is complete.

   Refer to the following code:

```
//Disable SYSREF from clock device if free running

uint8_t mcsStatus = 0;
uint32_t talAction = TALACT_NO_ACTION; //default to no action required

//Enable MCS in Talise.
talAction = TALISE_enableMultichipSync(pTaliseDevice, 1, &mcsStatus);
if (talAction != TALISE_NO_ACTION)
{
   //function threw action code
}
//Request first two SYSREF pulses from clock device
Ad9528.requestSysref(1);
Ad9528.requestSysref(1);
do
{
   //Poll MCS Status to determine if the Device Clock dividers have synchronized
   talAction = TALISE_enableMultichipSync(pTaliseDevice, 0, &mcsStatus);
if (talAction != TALISE_NO_ACTION)
{
          //function threw action code
}
```

```
} while (!(mcsStatus & 0x8))

//Request third SYSREF pulse from clock device
Ad9528.requestSysref(1);
do
{
   //Poll MCS Status to determine if the Digital Clock dividers have synchronized
talAction = TALISE_enableMultichipSync(pTaliseDevice, 0, &mcsStatus);
if (talAction != TALISE_NO_ACTION)
{
          //function threw error code
}
} while (!(mcsStatus & 0x2))
```

4. Complete the normal sequence to load the Arm processor and to run the initialization calibrations. The JESD204B link initialization is usually done at the end of initialization.
5. If the BBP requires the DAC transmit datapath, instruct the BBP to run the required initialization for the BBP. Enable the JESD204B serializer in the BBP to the state in which it outputs CGS K characters.
6. Perform a reset to the deframer to clear any disparity bit errors previously detected. Additionally, if the serializer/deserializer (SERDES) PLL inside the BPP resets, it can cause the device lane FIFOs to overflow/underflow, which requires a deframer reset. The code for this reset is as follows:

```
TALISE_enableDeframerLink(pTaliseDevice, pTaliseDeframerSel, 0);
TALISE_enableDeframerLink(pTaliseDevice, pTaliseDeframerSel, 1);
```

7. Enable the JESD204B blocks to accept a SYSREF signal for an internal LMFC timing reset. Only the calls to the desired framers/deframers are necessary. Send the fourth SYSREF pulse to the BBP and the devices to reset the JESD204B LMFC timing locally in each device to guarantee deterministic latency. The device does not reset its LMFC timing on any future SYSREF pulses (unless the `newSysrefOnRelink` option is enabled in the framer/deframer data structures). The code to enable the JESD204B blocks is as follows:

```
TALISE_enableSysrefToFramer(pTaliseDevice, pTaliseFramerSel, 1);
TALISE_enableSysrefToDeframer(pTaliseDevice, pTaliseDeframerSel, 1);

//Request fourth SYSREF pulse from clock device
Ad9528.requestSysref(1);
do
{
   //Poll MCS Status to determine if the JESD LMFC has synchronized
   talAction = TALISE_enableMultichipSync(pTaliseDevice, 0, &mcsStatus);
if (talAction != TALISE_NO_ACTION)
{
          //function threw action code
}
} while (!(mcsStatus & 0x1))
```

When establishing a JESD204B link, it is desirable that the data arriving to the deframer does not arrive very close to an LMFC boundary. If this does happen, the deterministic latency can vary from system to system if the data on one system arrives just before an LMFC event, and arrives on another system just after an LMFC event. If this happens, there is an LMFC period difference in the latency between the systems. Furthermore, the architecture in the device does not support a very small delay through the FIFO, and data corruption occurs if the delay is too small. Therefore, it is important that the FIFO depth be checked after the link is established, and the link is adjusted to achieve a FIFO depth that is close to the medium depth. The FIFO depth can be checked in Register 0x15CE for Deframer 0 and in Register 0x161E for Deframer 1. Write to the appropriate register with a value of 0x80 to latch the current FIFO depth, and then read back. The readback value is reported as a signed, twos complement number located in Bits[D5:D0] of the register with valid values from +K2 to −(K/2 − 1). The value reported is the difference of the number of frames between the read and write pointers. If the value is found to be close to 0, for example, −2, −1, 0, +1, or +2, adjust the depth by varying the LMFC offset parameter on either end of the JESD204B link. Note that across multiple system starts, the depth in the FIFO can vary by one or two frames. This variation is expected because sampling phase uncertainties are absorbed by the FIFO to give deterministic latency. Consider this fact when optimizing the JESD204B link and performing multiple system starts to find the worst case depth values for a given LMFC offset.

Device Framers

At this point (after the framer has received a SYSREF pulse), the JESD204B serializer sends the CGS K characters when waiting for the BBP to deassert the configured $\overline{\text{SYNCIN}}$x± signal (goes high). The device then transmits the ILAS sequence at the following LMFC boundary, after the ILAS the framer sends the received ADC data. Refer to the TX Device portion of Figure 36 in the JEDEC Standard No. 204B. For details on ILAS refer to Figure 35 of the JEDEC Standard No. 204B.

Device Deframer

When the deframer has received a SYSREF pulse and is receiving the CGS, the deframer drives $\overline{\text{SYNCOUT}}$x± high at the beginning of the next LMFC period. When $\overline{\text{SYNCOUT}}$x± is driven high, the deframer looks for the BBP to switch from transmitting the CGS to transmitting the ILAS. Details of the link initialization can be seen in Figure 14. When the ILAS transmission is complete, the deframer begins decoding DAC samples, as long as no errors are detected during the ILAS transmission. Refer to the RX Device portion of Figure 36 in the JEDEC Standard No. 204B.

Ensure that the scrambling setting matches in the BBP and in the device. It is possible for the JESD204B link to successfully link and for the data to appear corrupt because only the data is scrambled, not the ILAS. This data scrambling can result in a transmit spectrum that looks like noise.

COMPATIBILITY WITH XILINX JESD204B FPGA IP

Analog Devices uses the Xilinx JESD204B bundled with the XC7Z045 FFG900 for demonstration with the provided Analog Devices evaluation platform.

Some versions of the Xilinx JESD204B include a watchdog timer that resets the high speed serial PLLs if the configured $\overline{\text{SYNCOUT}}$x± signal is held low for more than 10 ms. This feature causes the lane FIFOs in the deserializers to overflow/underflow, because the lane FIFOs derive the write clock from the recovered CDR clock. When the FPGA resets its SERDES PLLs, the CDR clock in the device unlocks and causes the lane FIFO to underflow/overflow. Typically, this is not a problem, because $\overline{\text{SYNCOUT}}$x± is not held low for longer than 10 ms in normal use. In debug mode, however, the user can choose to hold $\overline{\text{SYNCOUT}}$x± low to test the link. It is recommended to disable the 10 ms watchdog reset in the Xilinx IP wrapper to prevent unnecessary issues caused by randomly resetting PLLs in the system.

LINK SHARING IN TDD MODE

In TDD mode, a single link can be shared between receiver and observation receiver modes of operation.

Figure 14 and Figure 15 show path of data flow in observation receiver and receiver modes, respectively, in link sharing mode. For Figure 14 and Figure 15, the same framer is shared between observation receiver and receiver modes. The observation receiver mode data flow is indicated in red in Figure 14, and the receiver mode data flow is indicated in green in Figure 15. To use the same framer, the observation I/Q sample rate must be the same as the receiver sample rate: 2× the receiver sample rate, or 4× the receiver sample rate.

In Figure 14 and Figure 15, the receiver is set in the 200 MHz/245.76 MSPS profile and the observation receiver is assumed to be set in the 450 MHz/491.52 MSPS profile.

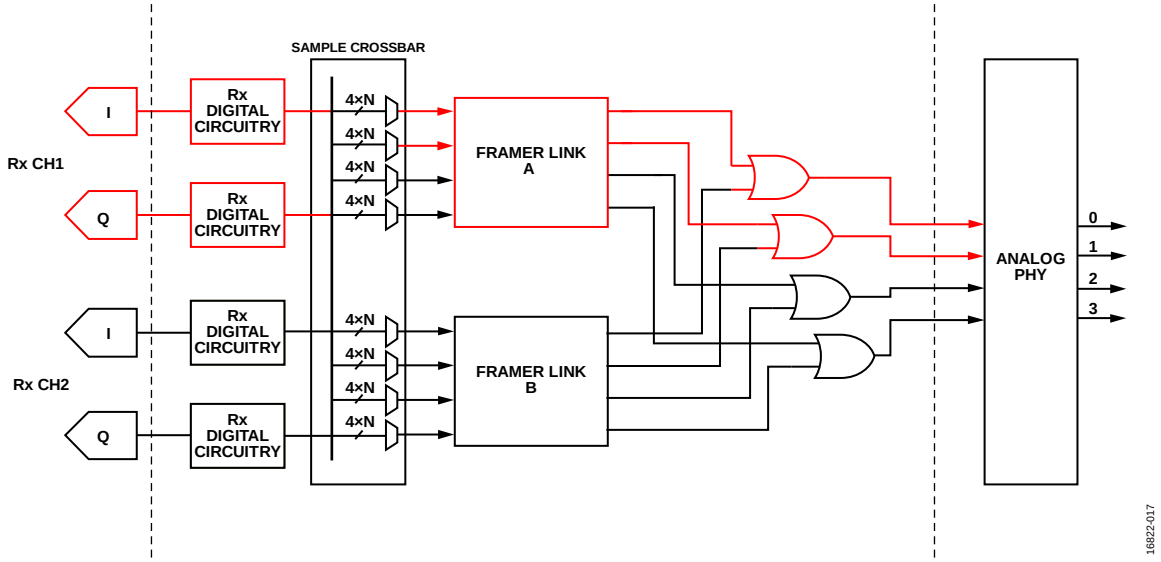

Figure 14. Single Channel Observation Receiver Mode Operation Using Framer A (Link Sharing Mode)

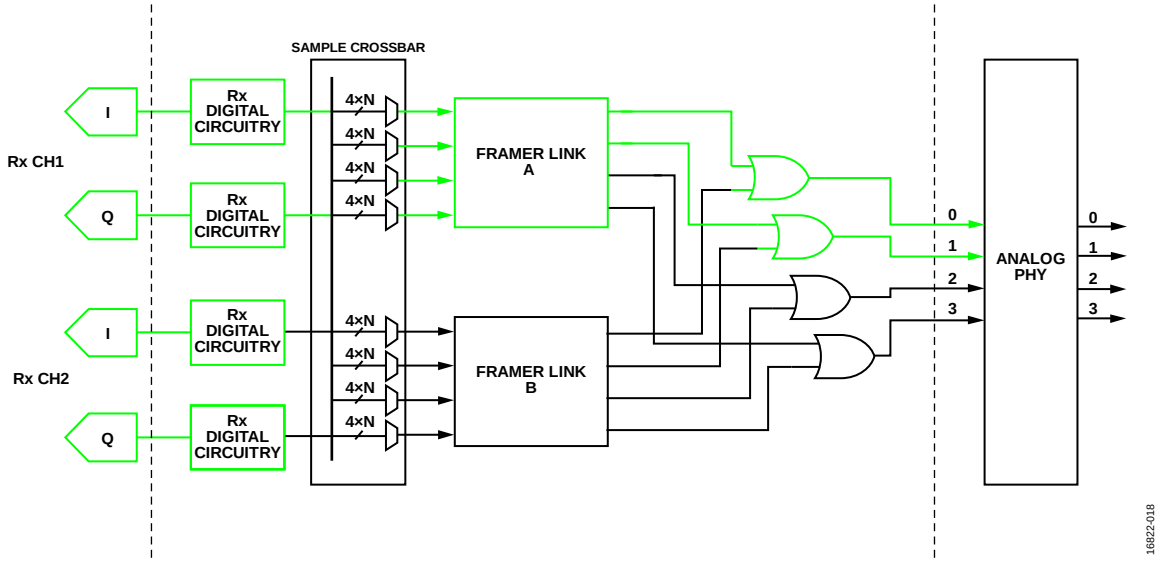

Figure 15. Dual-Channel Receiver Mode Operation Using Framer A (Link Sharing Mode)

Figure 16 shows link parameters and mapping of data onto lanes for the TDD in link sharing mode. The BBIC can now change the M and S parameters on the device the fly when switching between receiver and observation receiver modes.

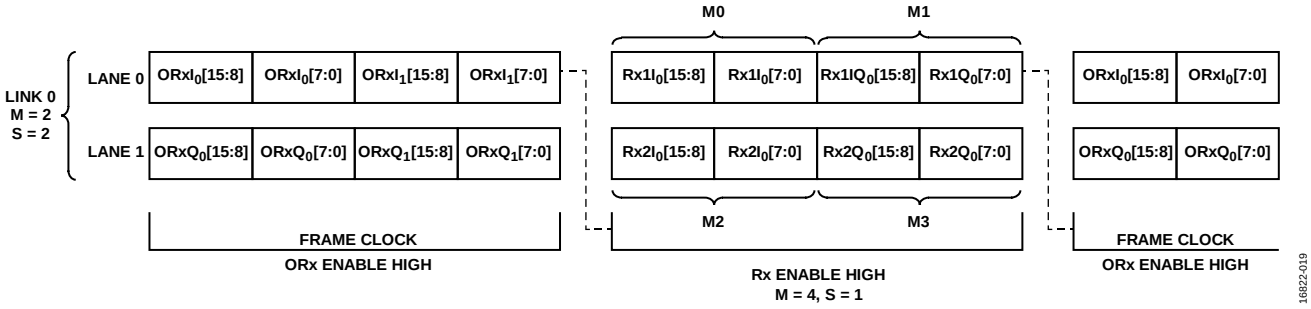

Figure 16. Mapping of Observation Receiver/Receiver Data onto Link in Link Sharing Mode

Enable Link Sharing Mode

Link sharing mode can be enabled using the GUI and the API. The process is as follows:

1. Using the GUI, on the JESD204B setup page, select the same framer and lane for the receiver and observation receiver.
2. Using the API, select the same framer and lanes that were selected during the receiver and observation receiver profile setup.

After setting up the profiles, the stream file is set up such that the JESD204B M and S parameters are toggled with receiver and observation receiver enable signals, such that F is kept the same (the link does not drop). For this use case, take the following steps:

1. During receiver enable, M = 4, and S = 1.
2. During observation receiver enable, M = 2, and S = 2.

JESD204B CONFIGURATION DIAGRAMS

Supported Framer Configurations

See Table 9 for terms used in Figure 17 to Figure 43.

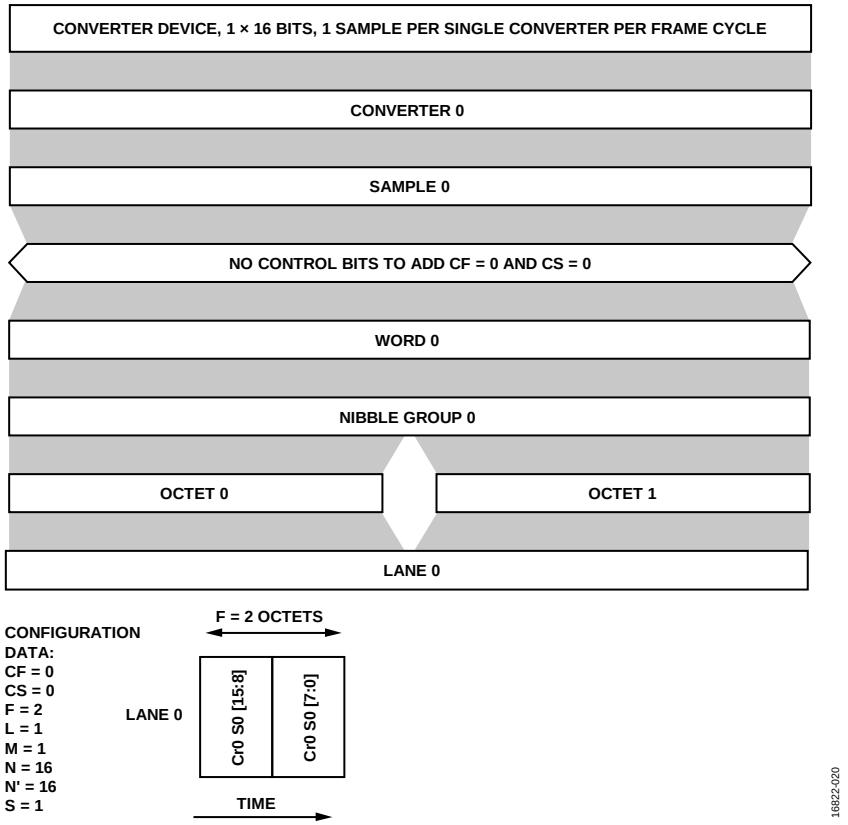

Figure 17. JESD204B Framer Configuration (M = 1, L = 1, S = 1)

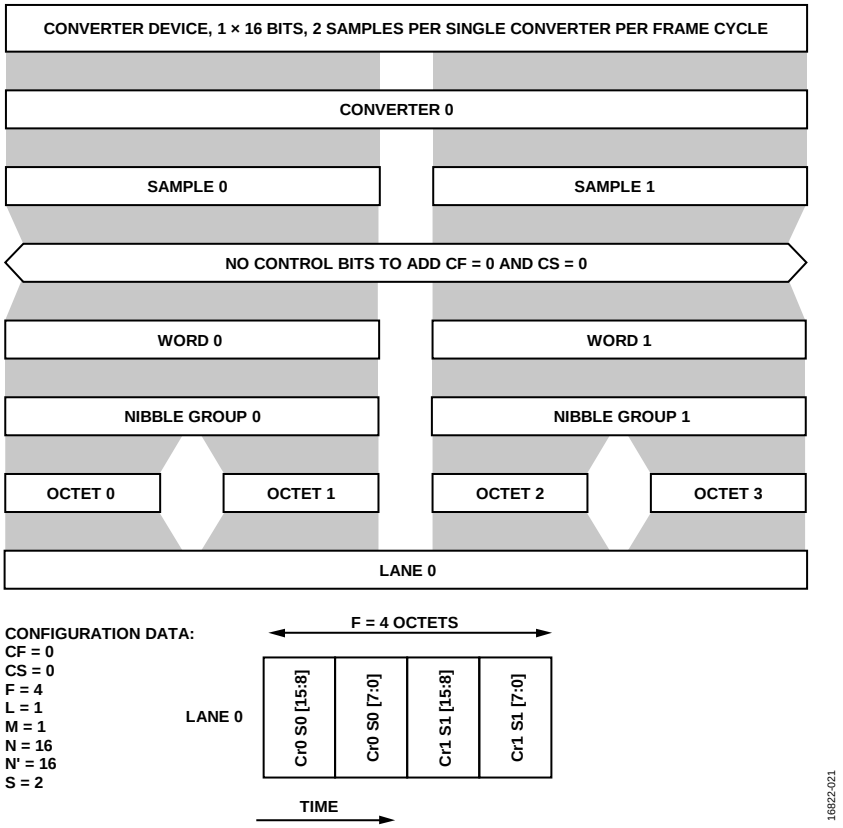

Figure 18. JESD204B Framer Configuration (M = 1, L = 1, S = 2)

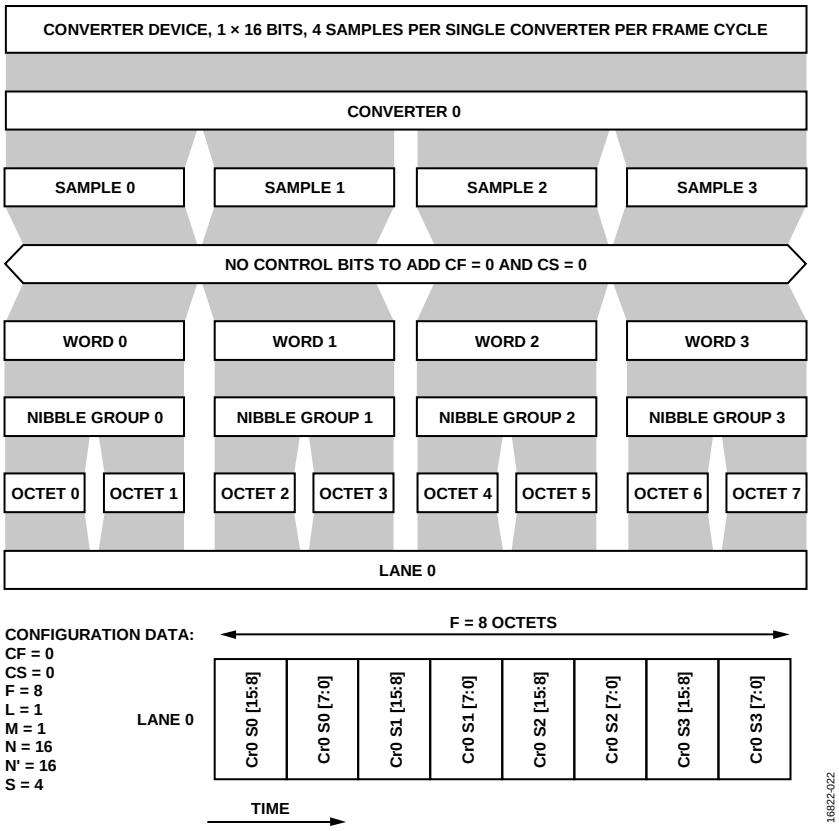

Figure 19. JESD204B Framer Configuration (M = 1, L = 1, S = 4)

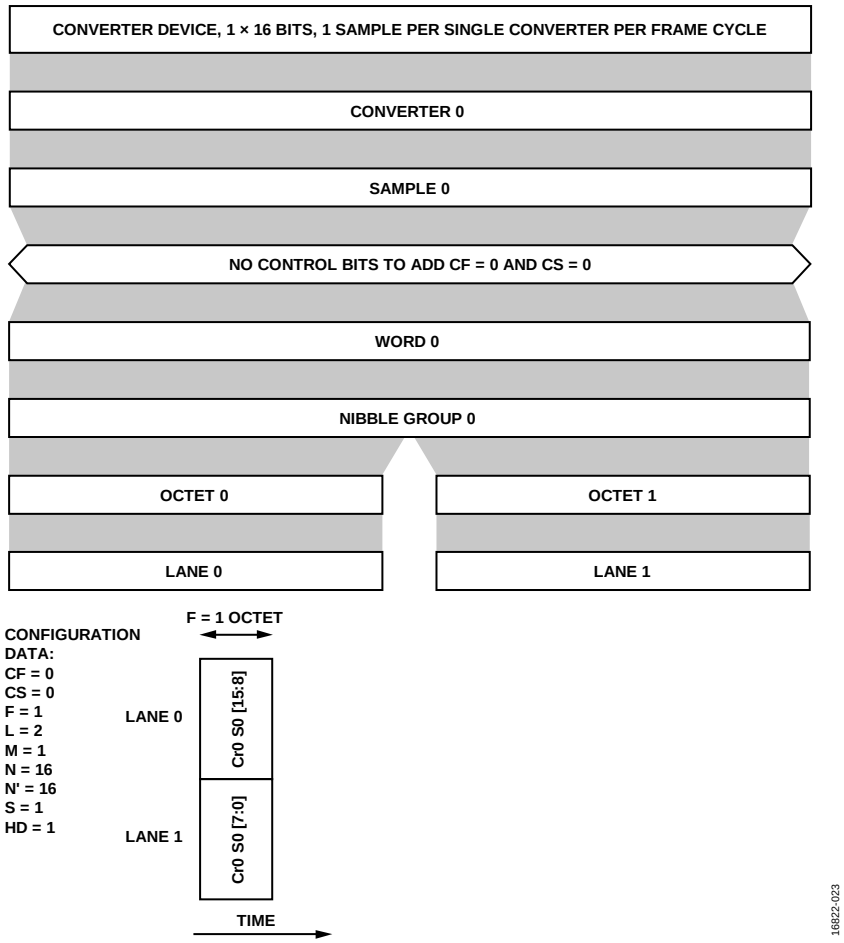

Figure 20. JESD204B Framer Configuration (M = 1, L = 2, S = 1)

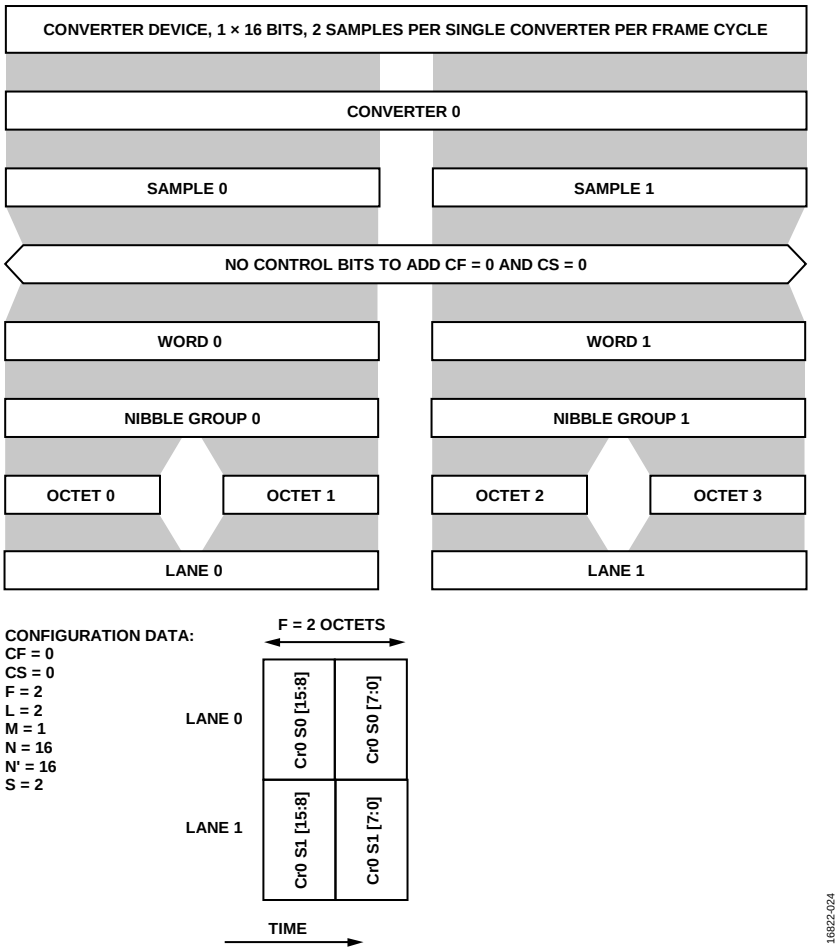

Figure 21. JESD204B Framer Configuration (M = 1, L = 2, S = 2)

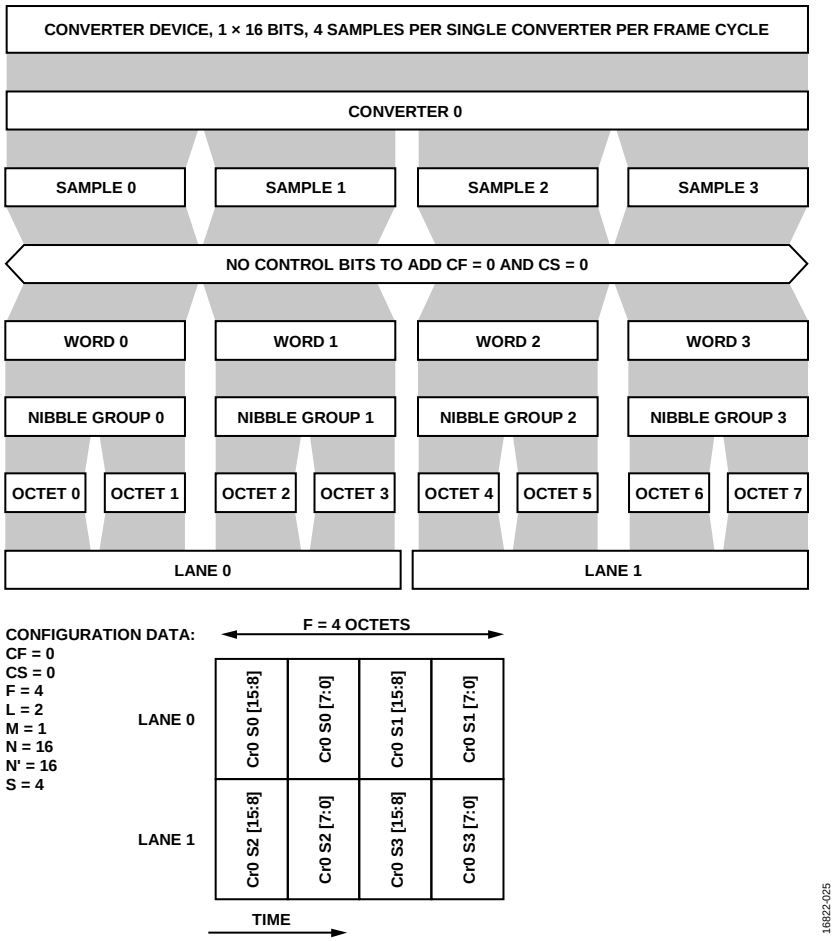

Figure 22. JESD204B Framer Configuration (M = 1, L = 2, S = 4)

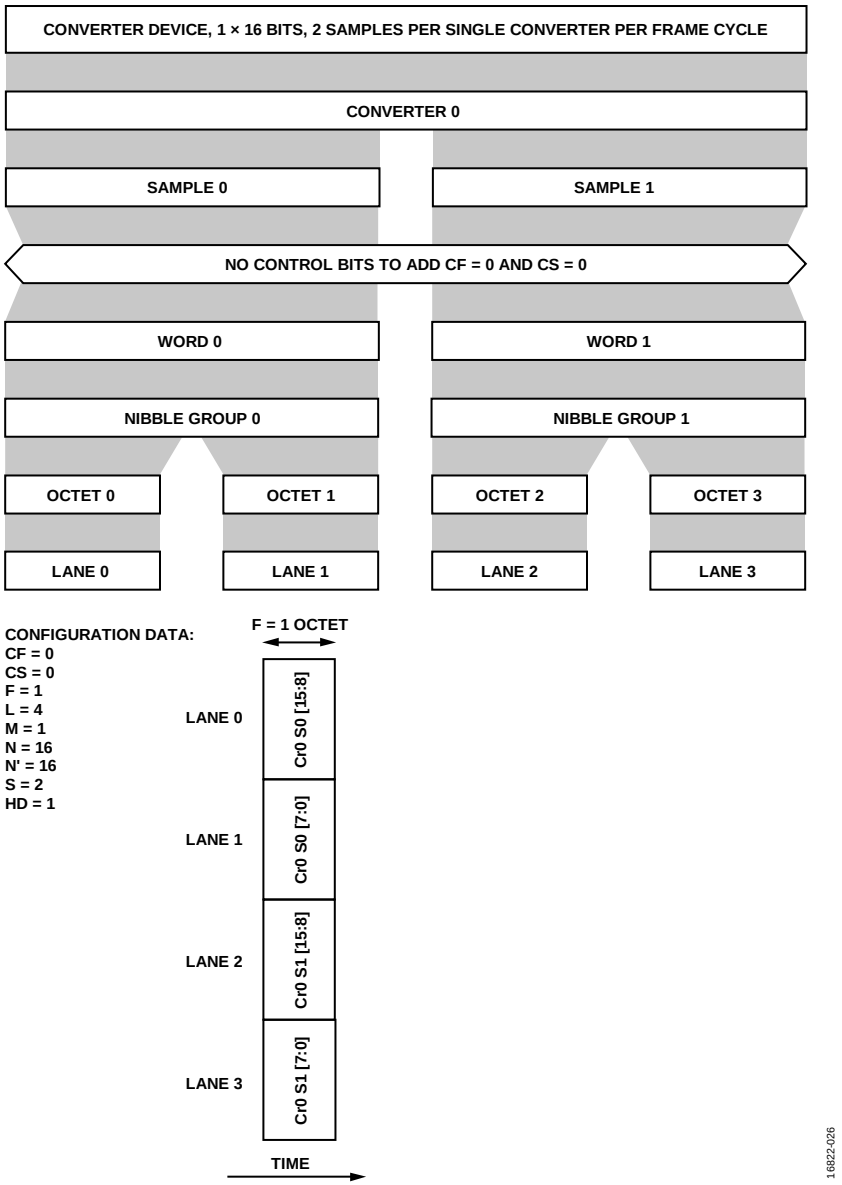

Figure 23. JESD204B Framer Configuration (M = 1, L = 4, S = 2)

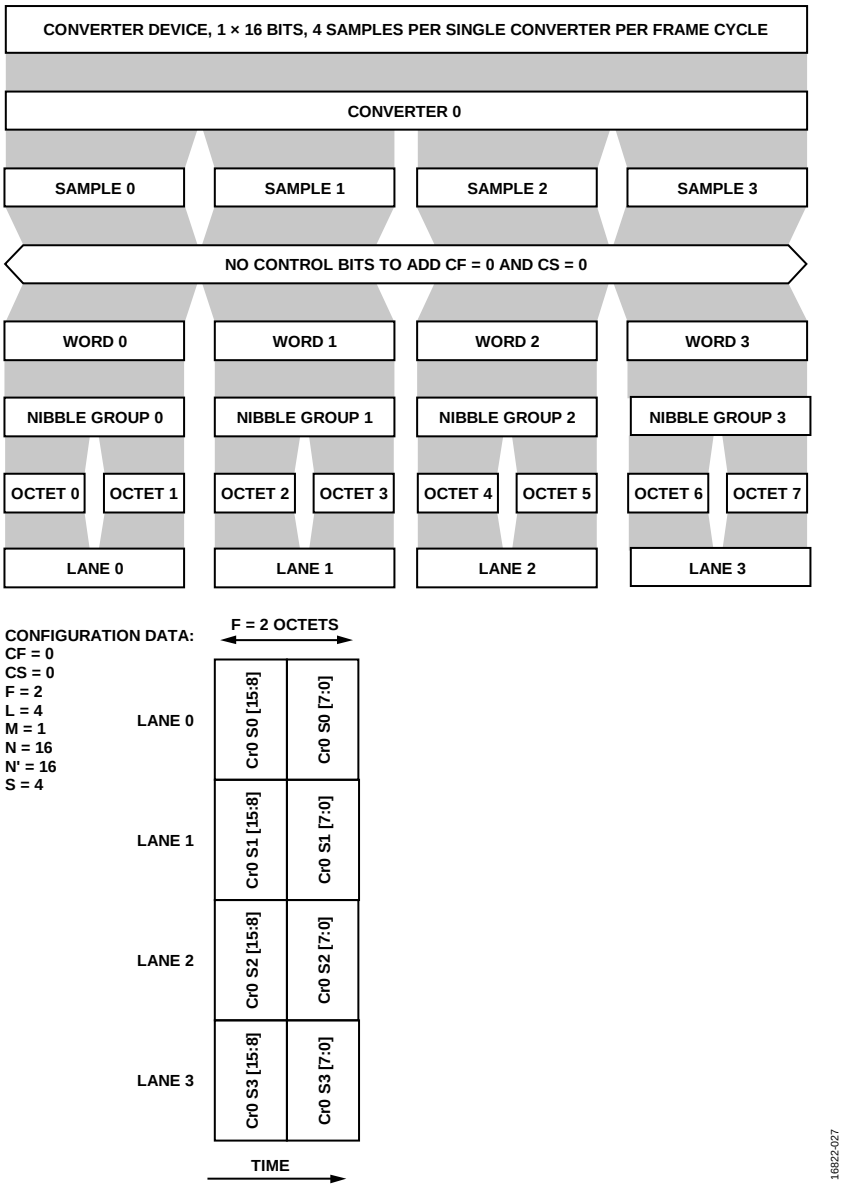

Figure 24. JESD204B Framer Configuration (M = 1, L = 4, S = 4)

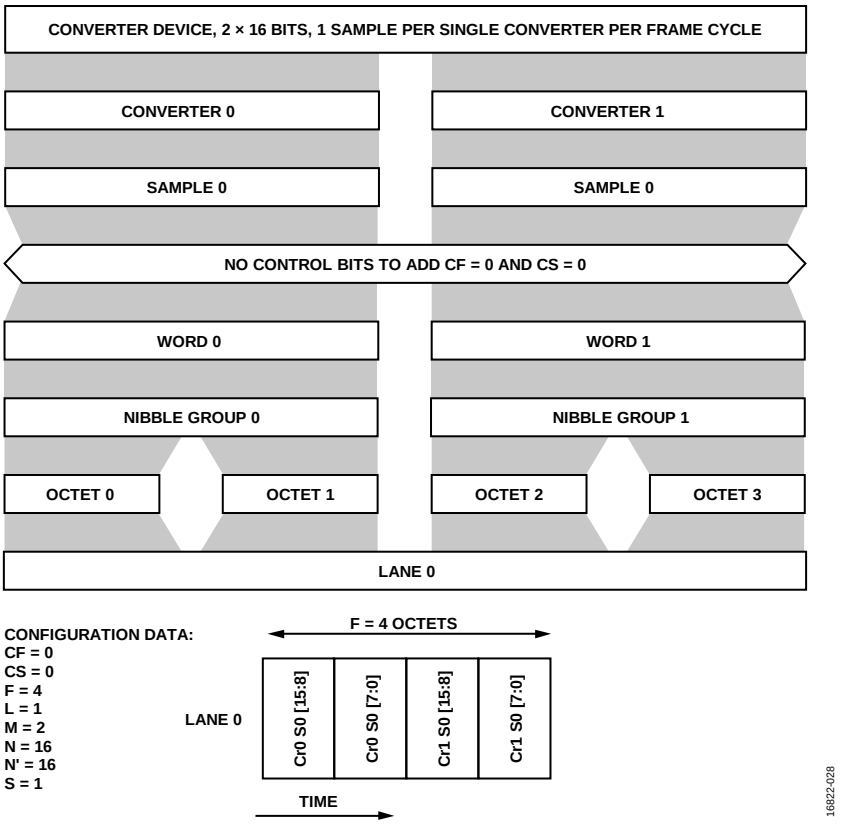

Figure 25. JESD204B Framer Configuration (M = 2, L = 1, S = 1)

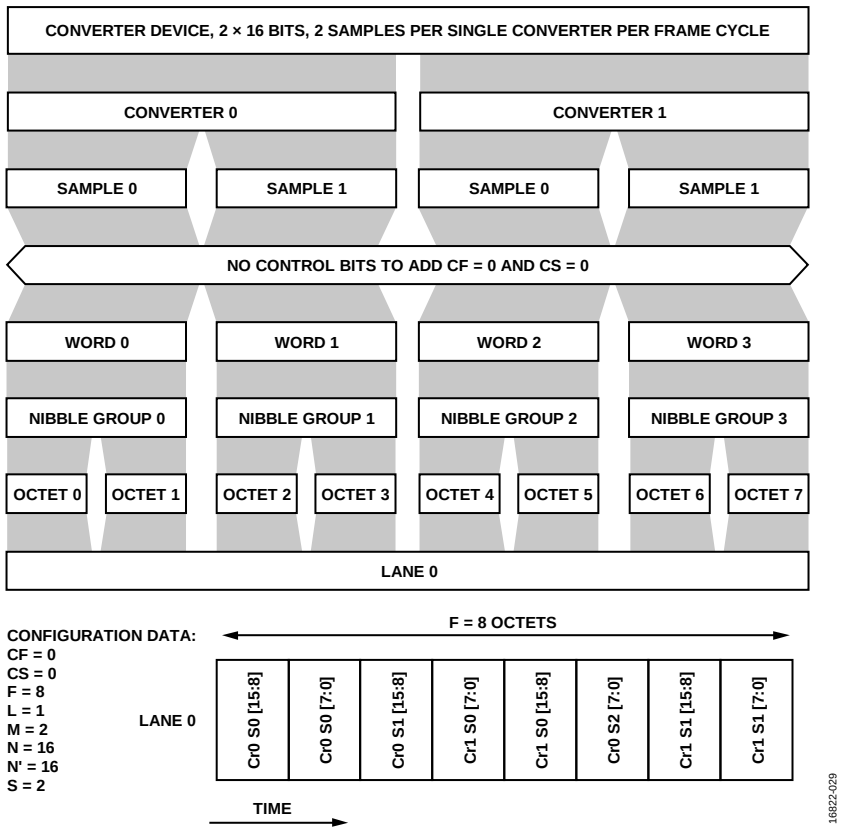

Figure 26. JESD204B Framer Configuration (M = 2, L = 1, S = 2)

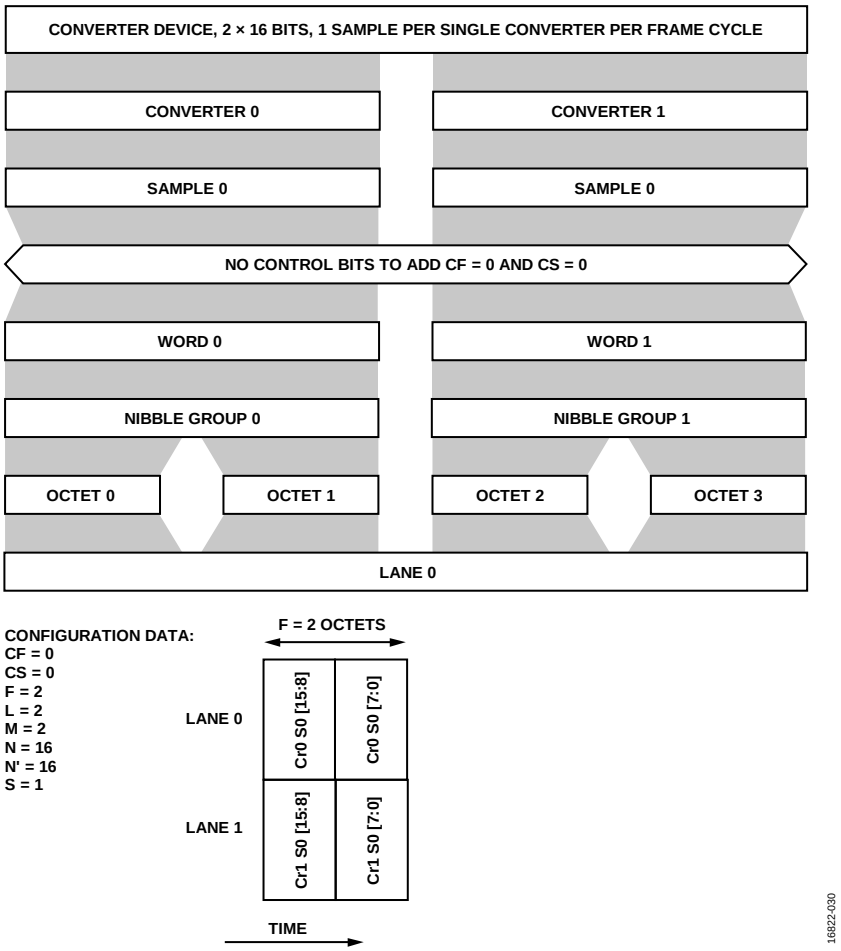

Figure 27. JESD204B Framer Configuration (M = 2, L = 2, S = 1)

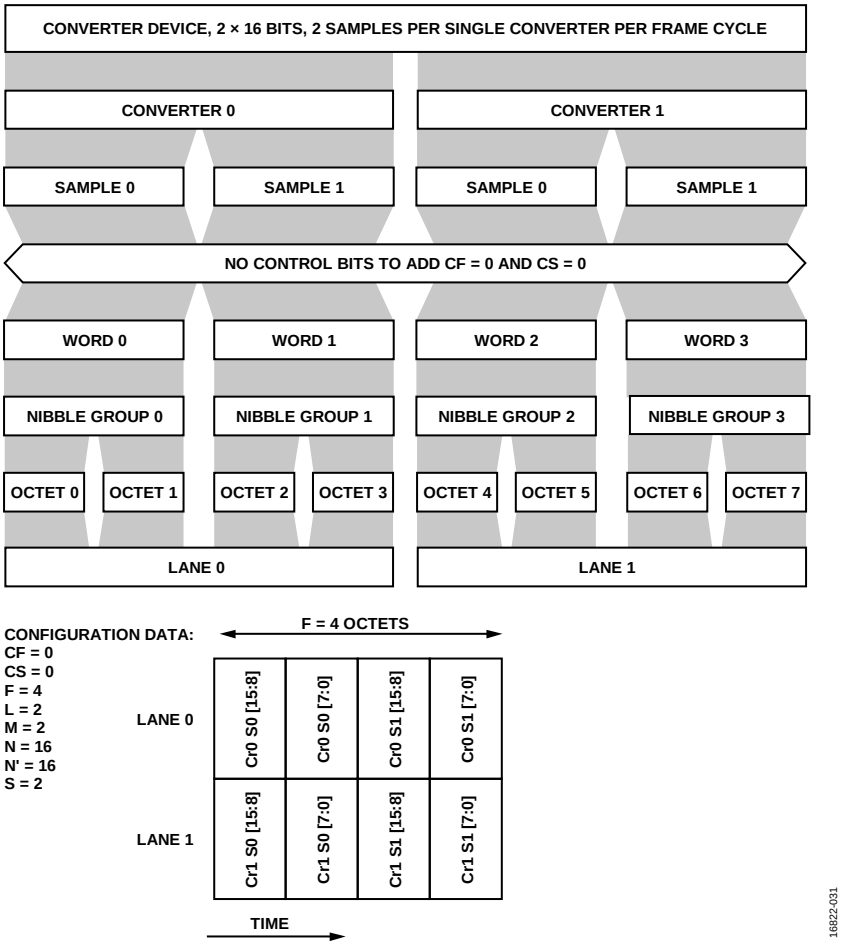

Figure 28. JESD204B Framer Configuration (M = 2, L = 2, S = 2)

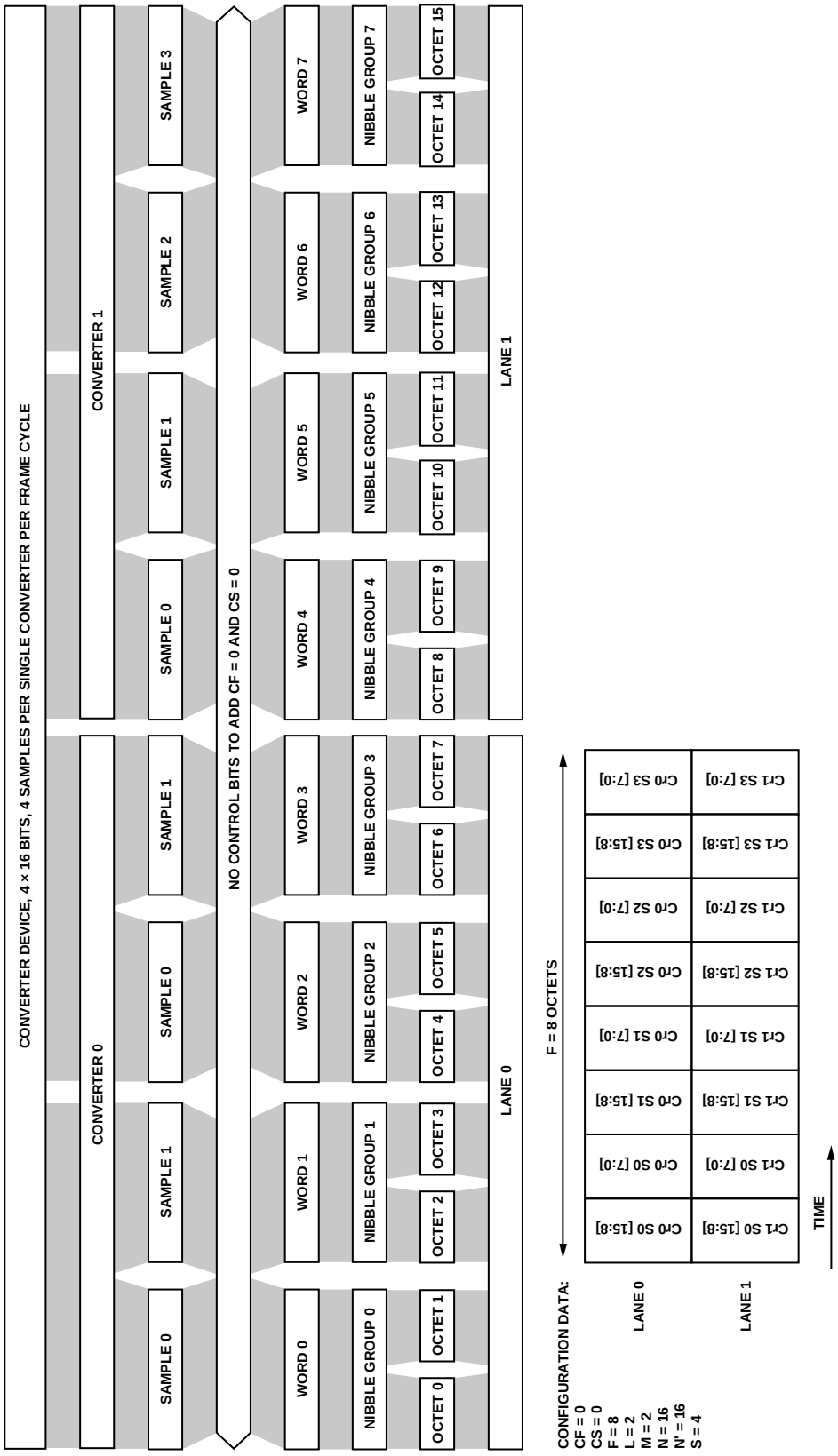

Figure 29. JESD204B Framer Configuration (M = 2, L = 2, S = 4)

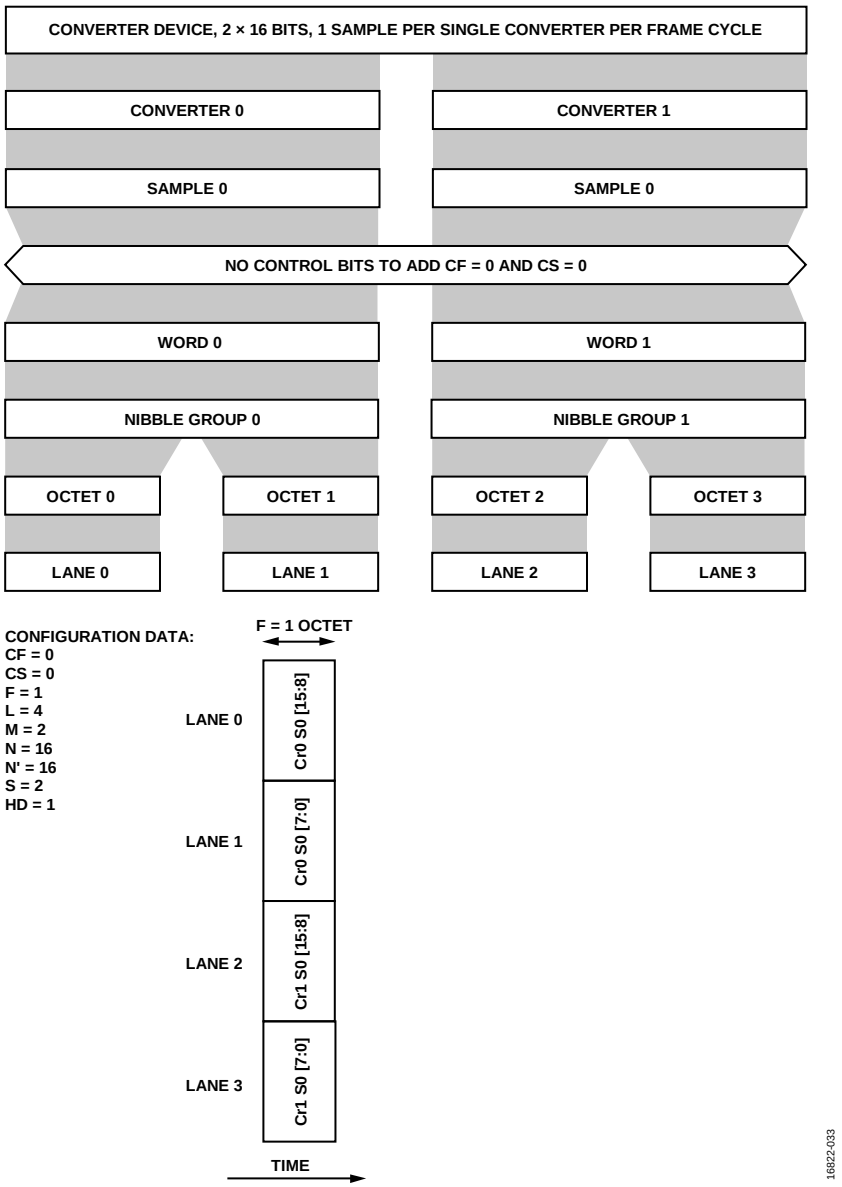

Figure 30. JESD204B Framer Configuration (M = 2, L = 4, S = 2)

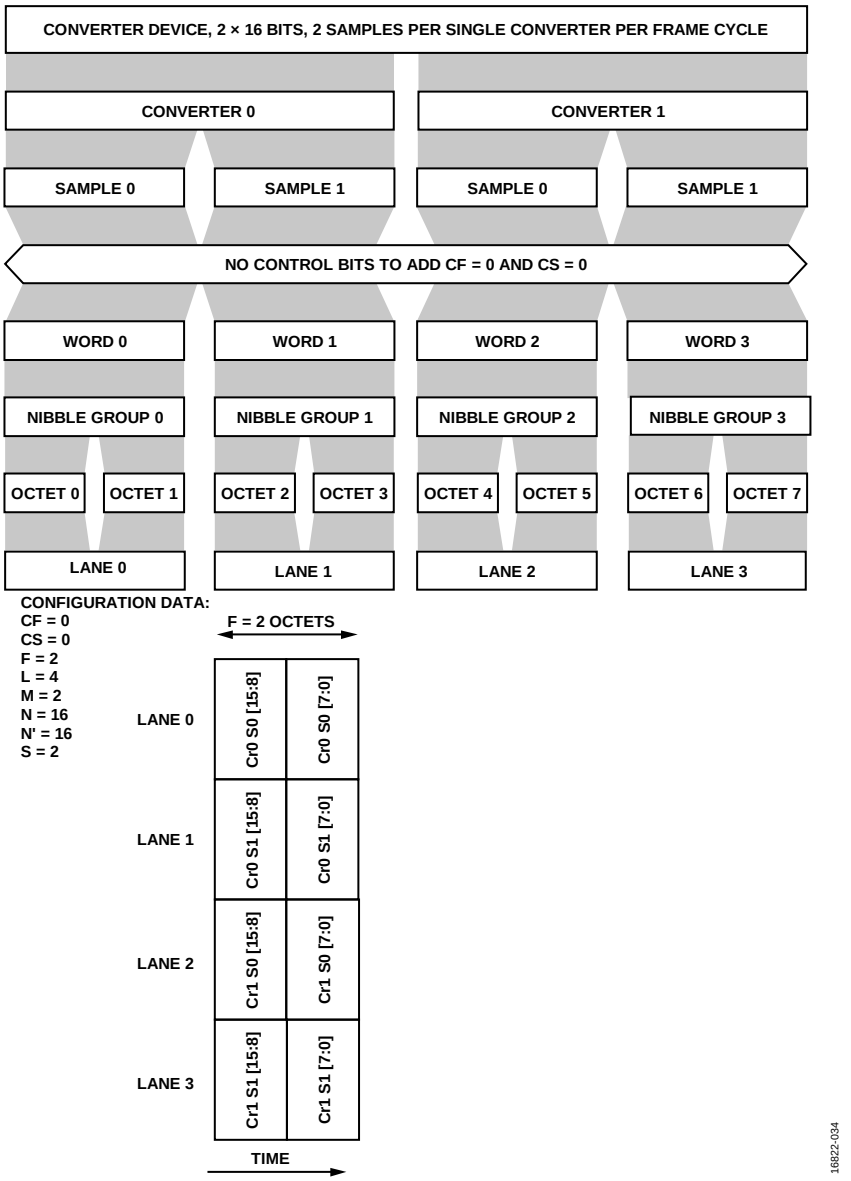

Figure 31. JESD204B Framer Configuration (M = 2, L = 4, S = 2)

Figure 32. JESD204B Framer Configuration (M = 2, L = 4, S = 4)

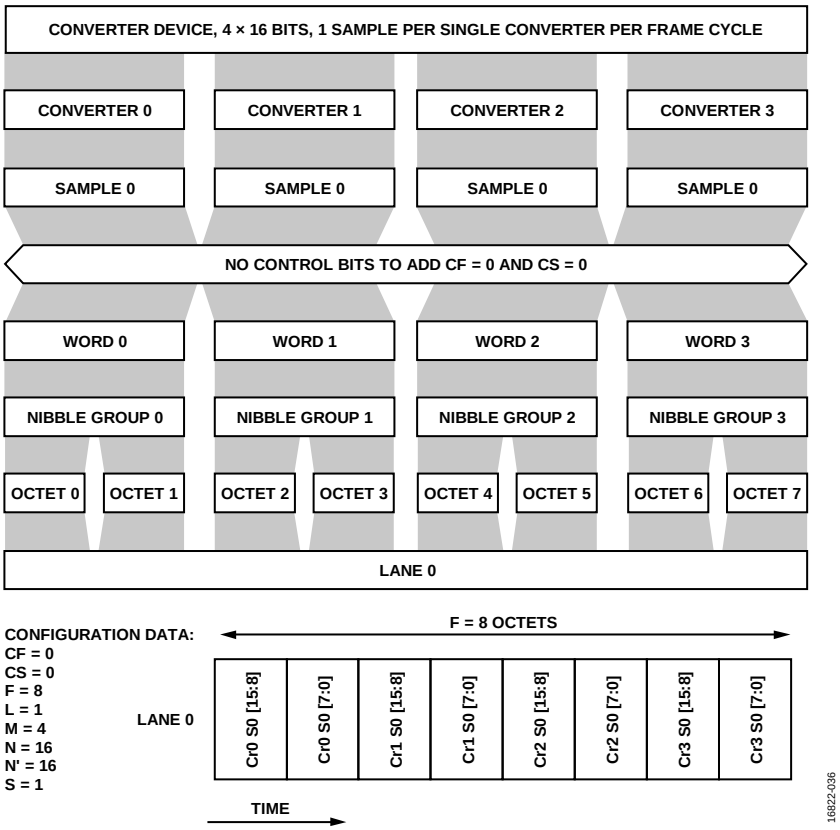

Figure 33. JESD204B Framer Configuration (M = 4, L = 1, S = 1)

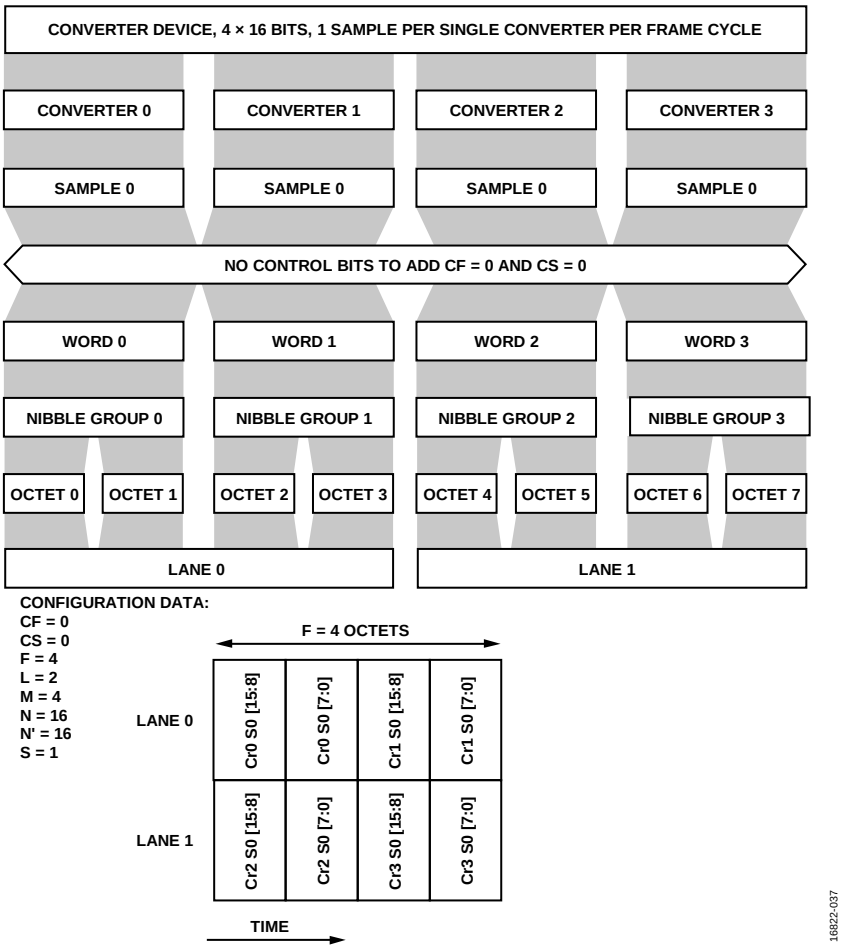

Figure 34. JESD204B Framer Configuration (M = 4, L = 2, S = 1)

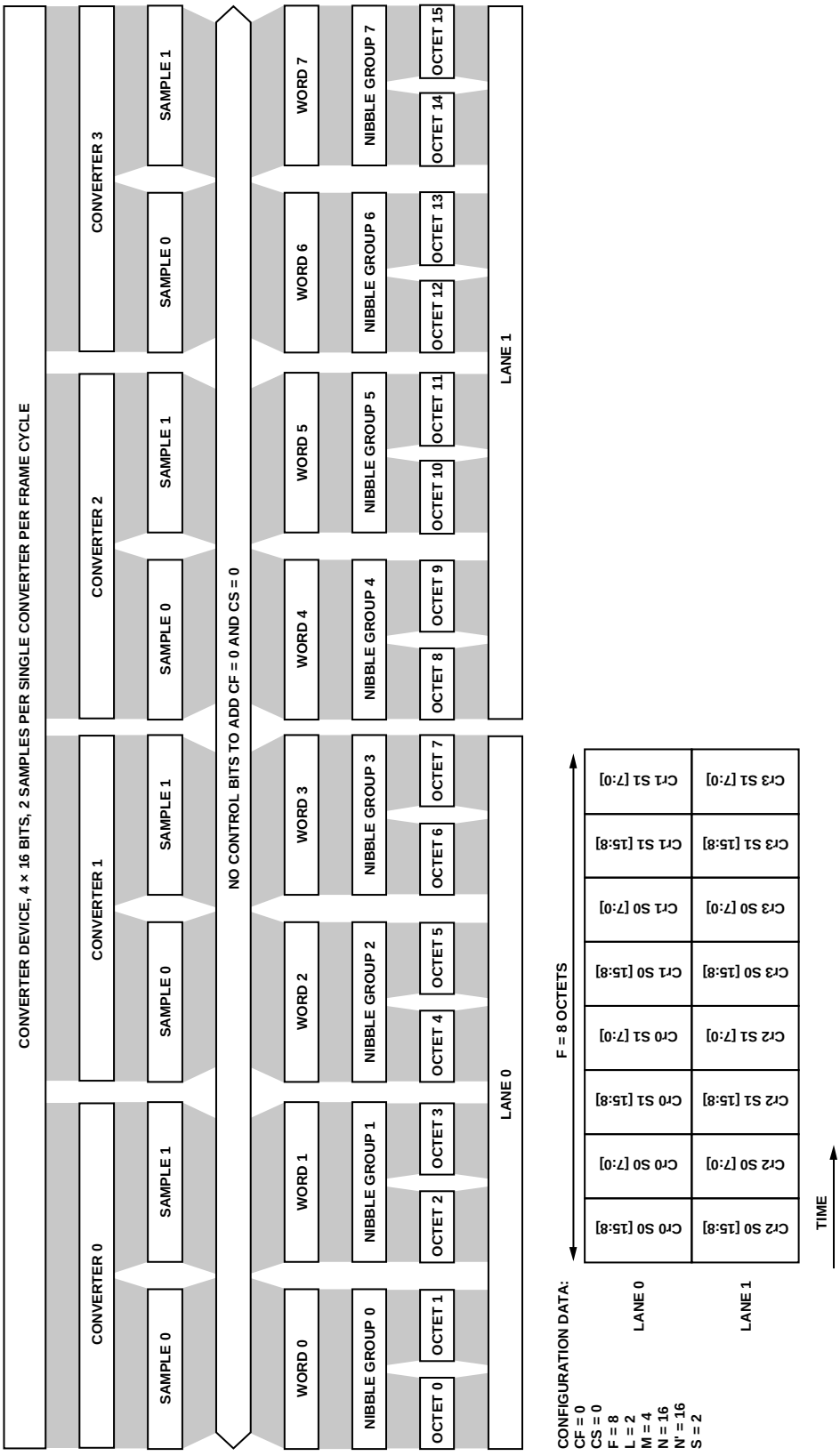

Figure 35. JESD204B Framer Configuration (M = 4, L = 2, S = 2)

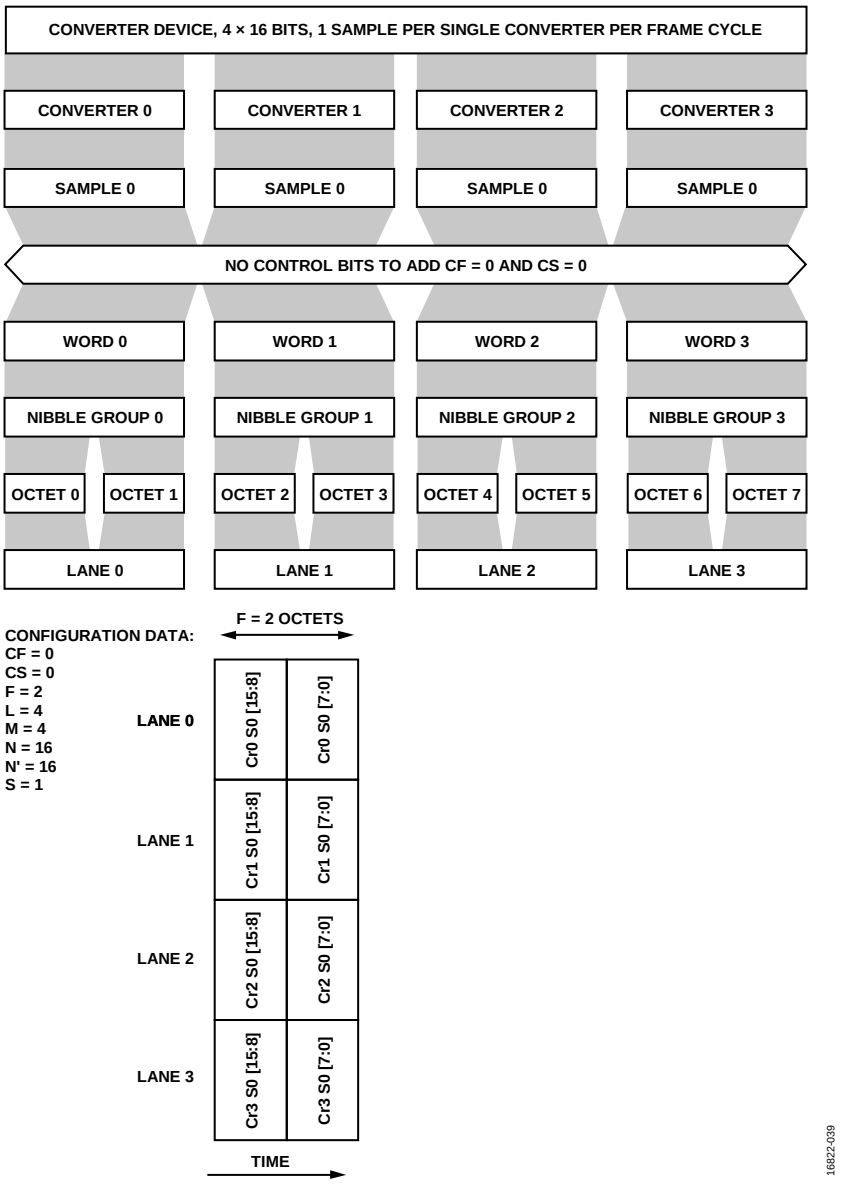

Figure 36. JESD204B Framer Configuration (M = 4, L = 4, S = 1)

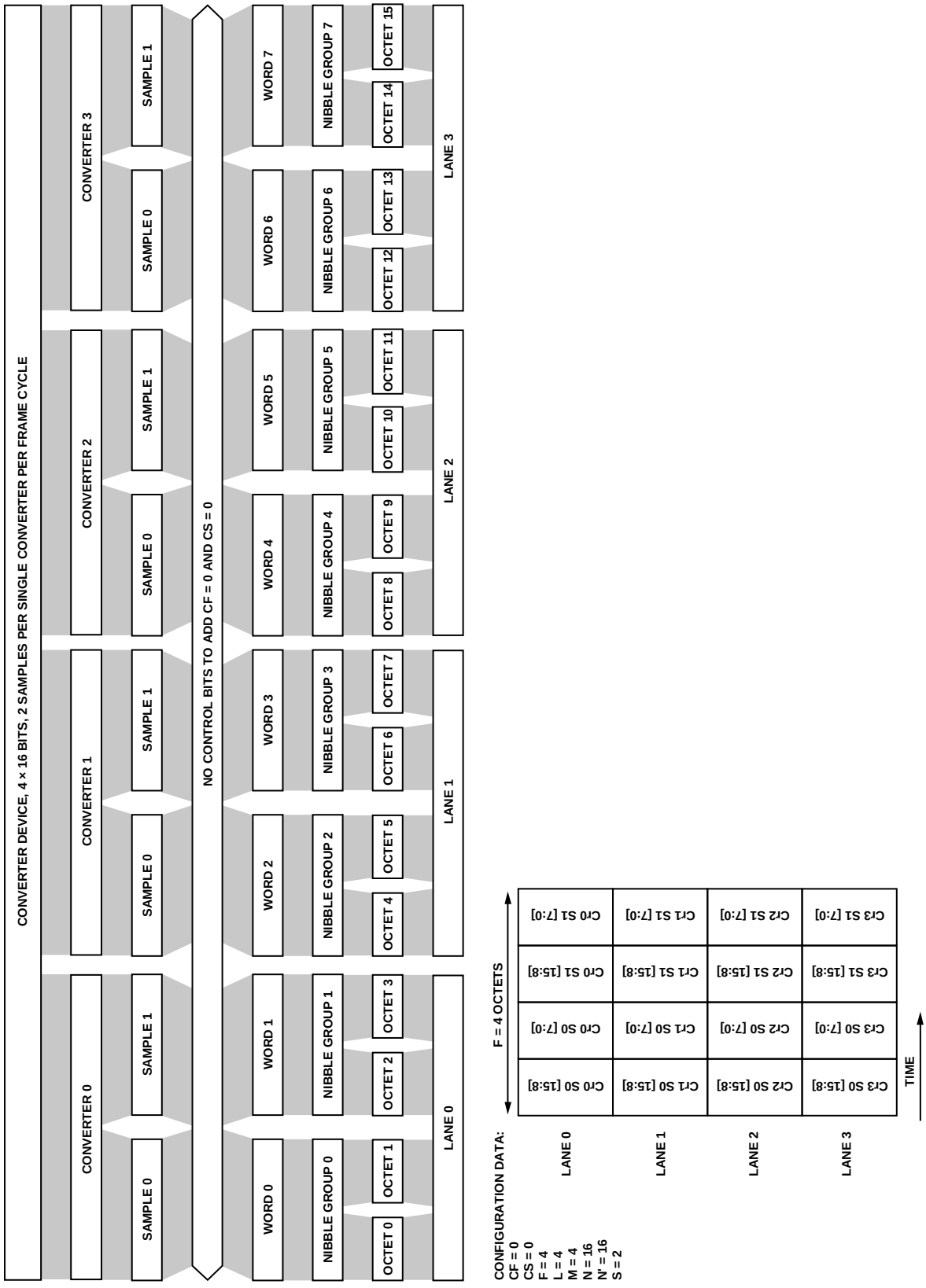

Figure 37. JESD204B Framer Configuration (M = 4, L = 4, S = 2)

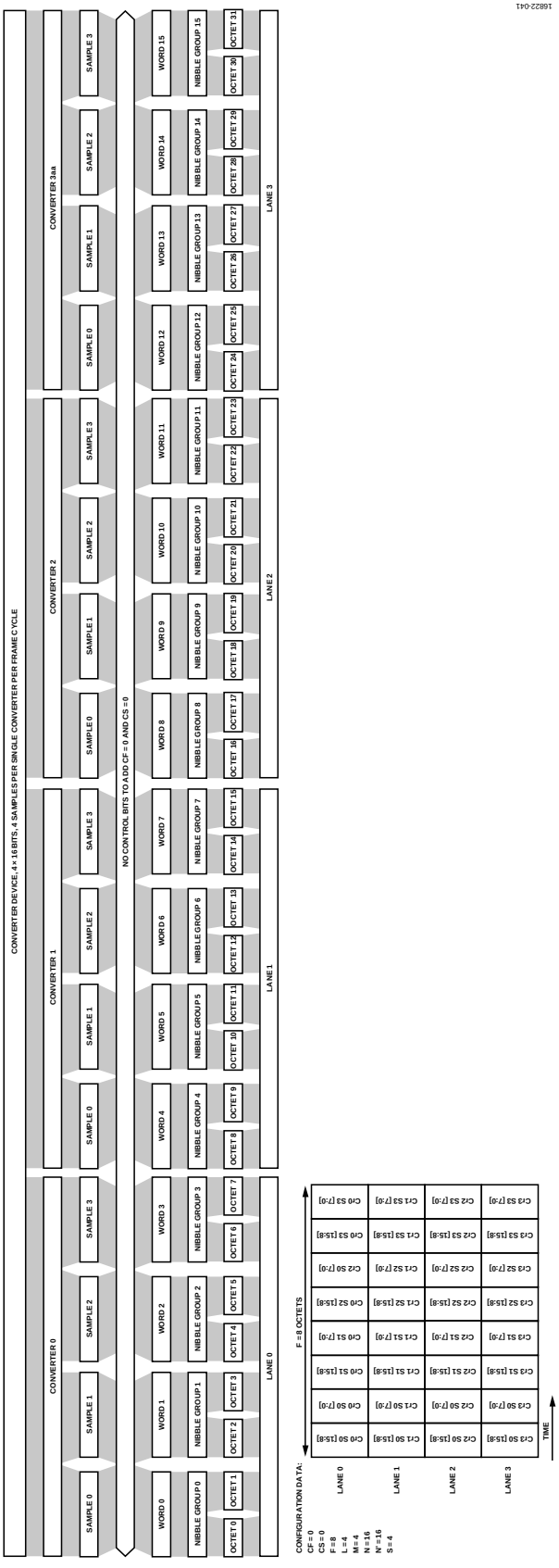

Figure 38. JESD204B Framer Configuration (M = 4, L = 4, S = 4)

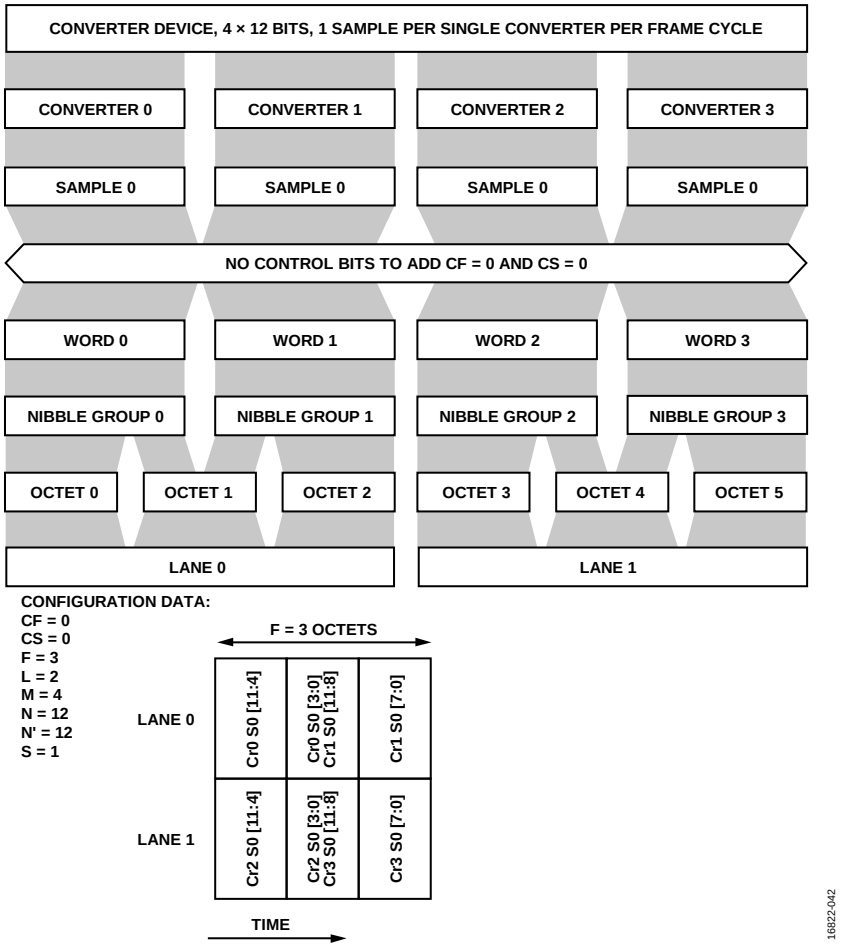

Figure 39. JESD204B Framer Configuration (M = 4, L = 2, S = 1, N′ = 12)

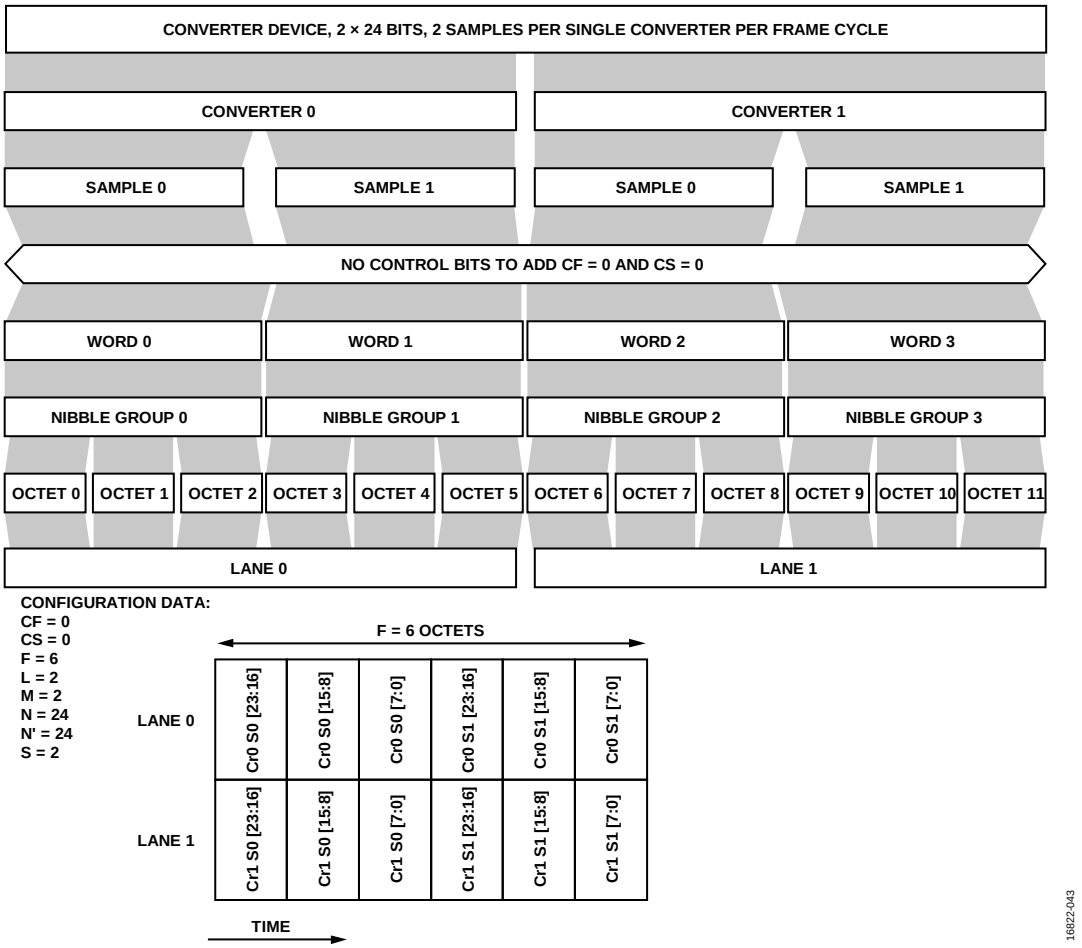

Figure 40. JESD204B Framer Configuration (M = 2, L = 2, S = 2, N′ = 24)

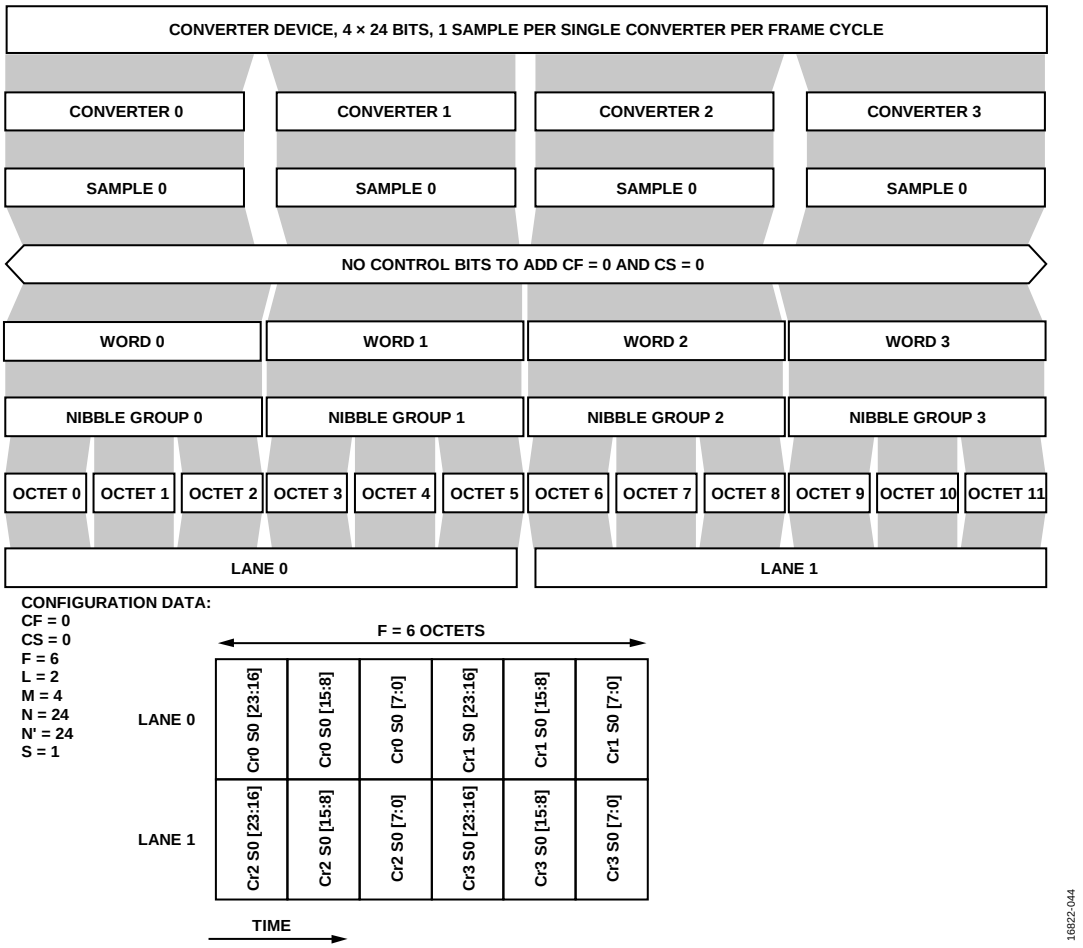

Figure 41. JESD204B Framer Configuration (M = 4, L = 2, S = 1, N' = 24)

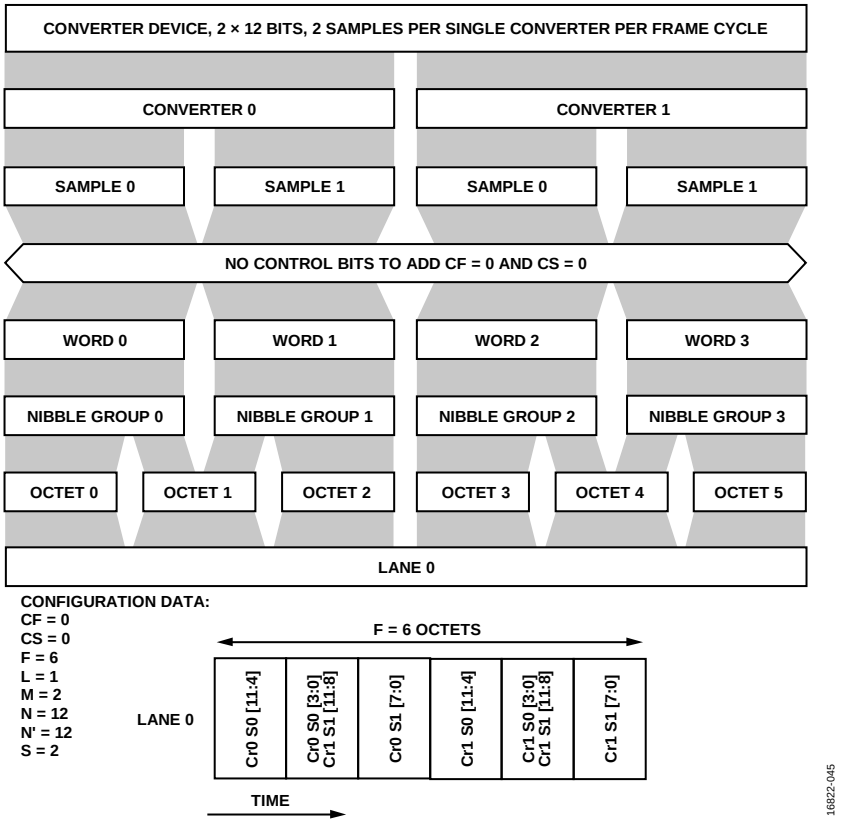

Figure 42. JESD204B Framer Configuration (M = 2, L = 1, S = 2, N′ = 12)

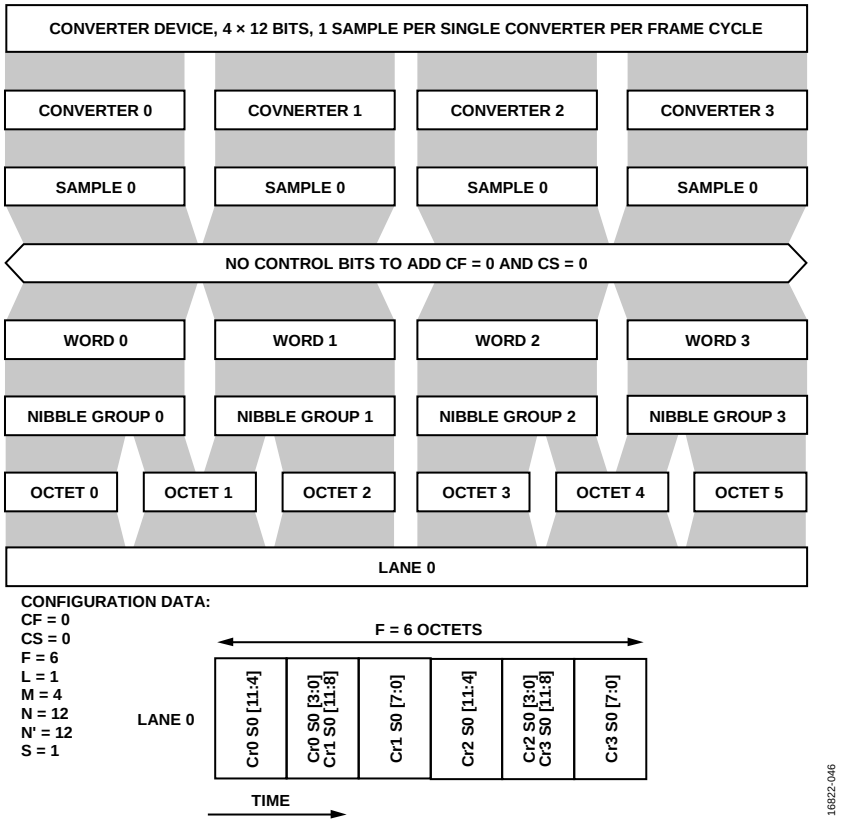

Figure 43. JESD204B Framer Configuration (M = 4, L = 1, S = 1, N′ = 12)

Supported Deframer Configurations

See Table 9 for terms used in Figure 44 to Figure 52.

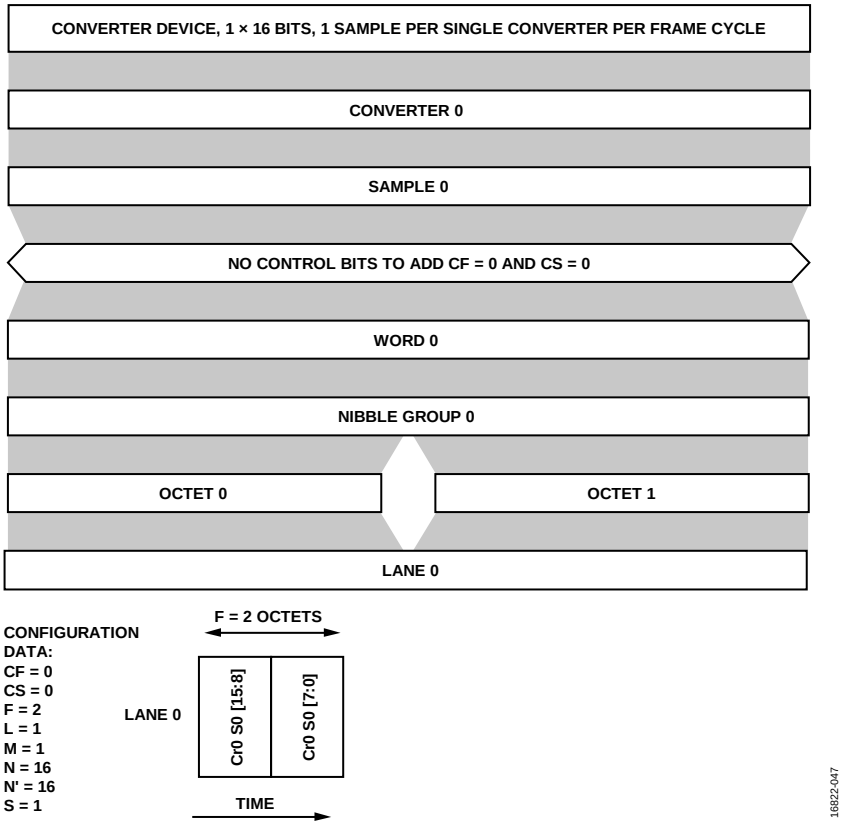

Figure 44. JESD204B Deframer Configuration (M = 1, L = 1, S = 1)

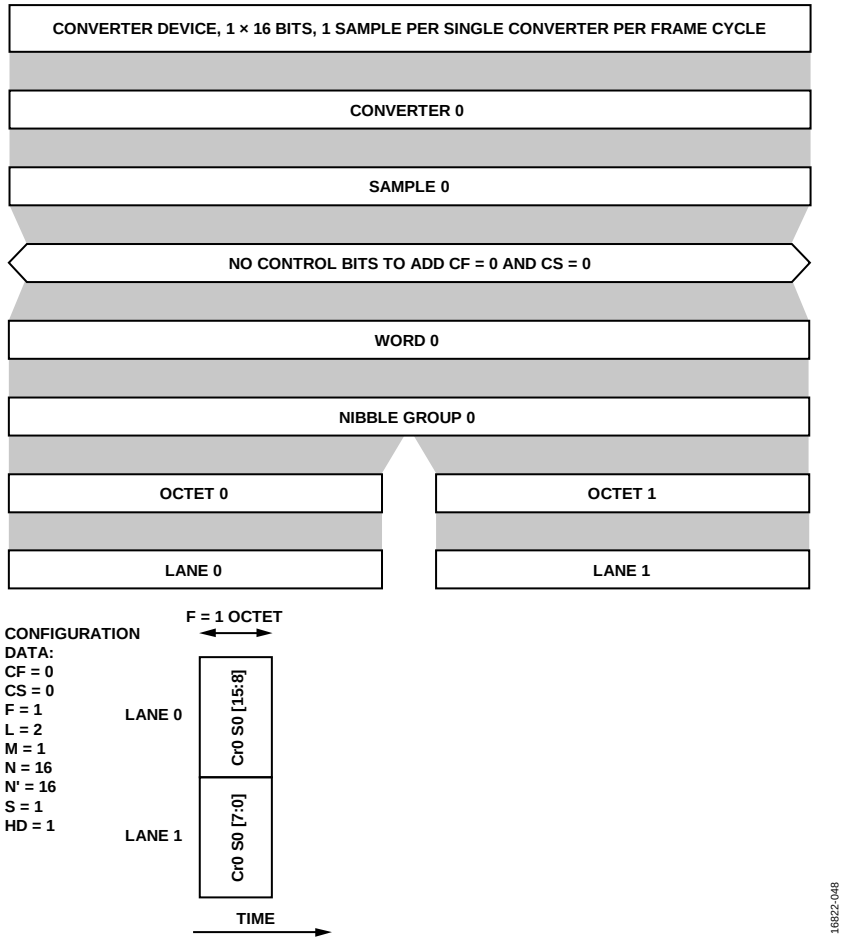

Figure 45. JESD204B Deframer Configuration (M = 1, L = 2, S = 1)

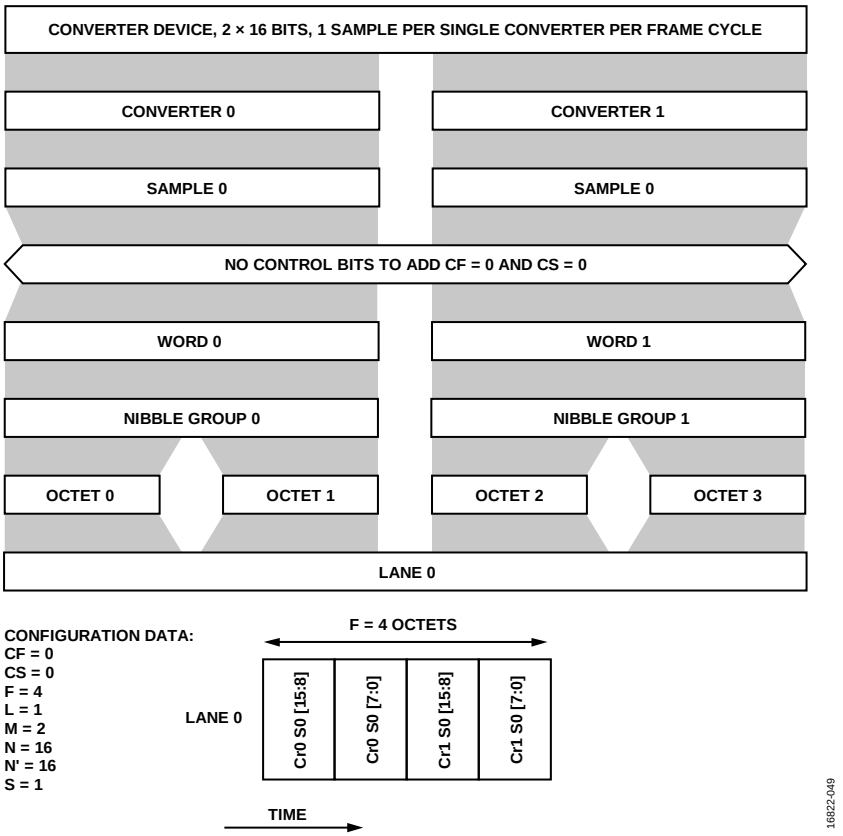

Figure 46. JESD204B Deframer Configuration (M = 2, L = 1)

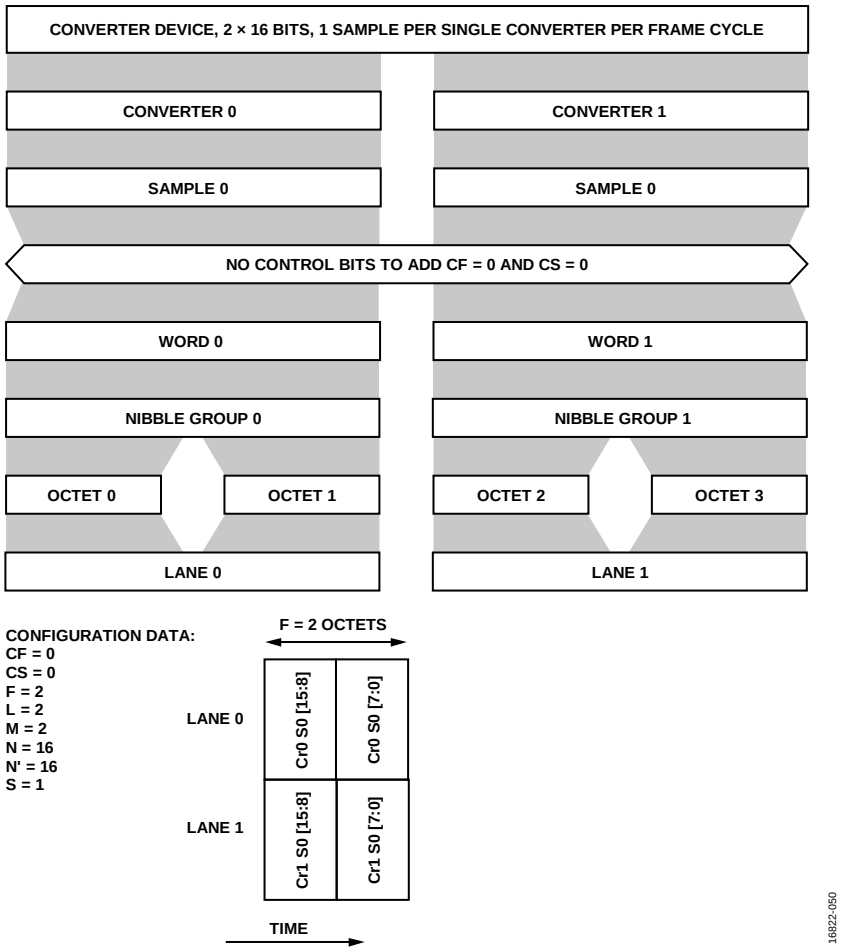

Figure 47. JESD204B Deframer Configuration (M = 2, L = 2)

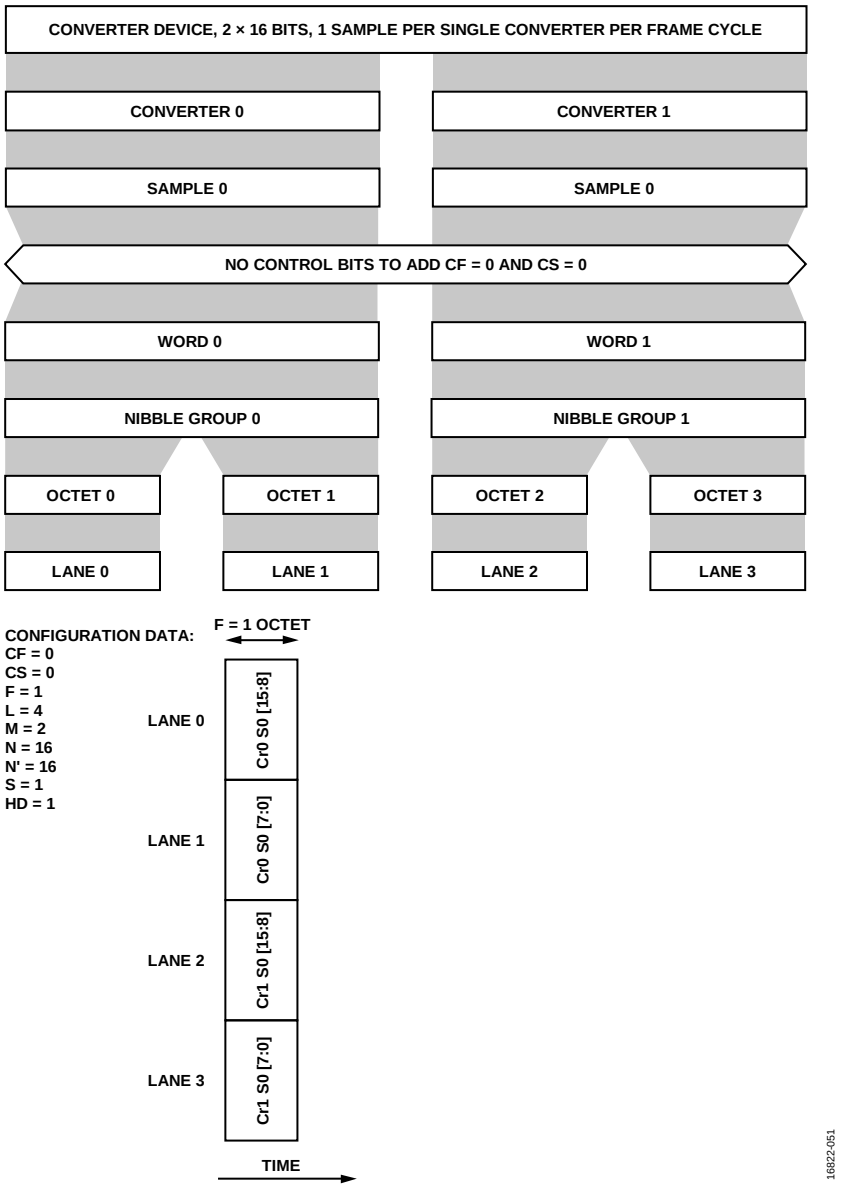

Figure 48. JESD204B Deframer Configuration (M = 2, L = 4)

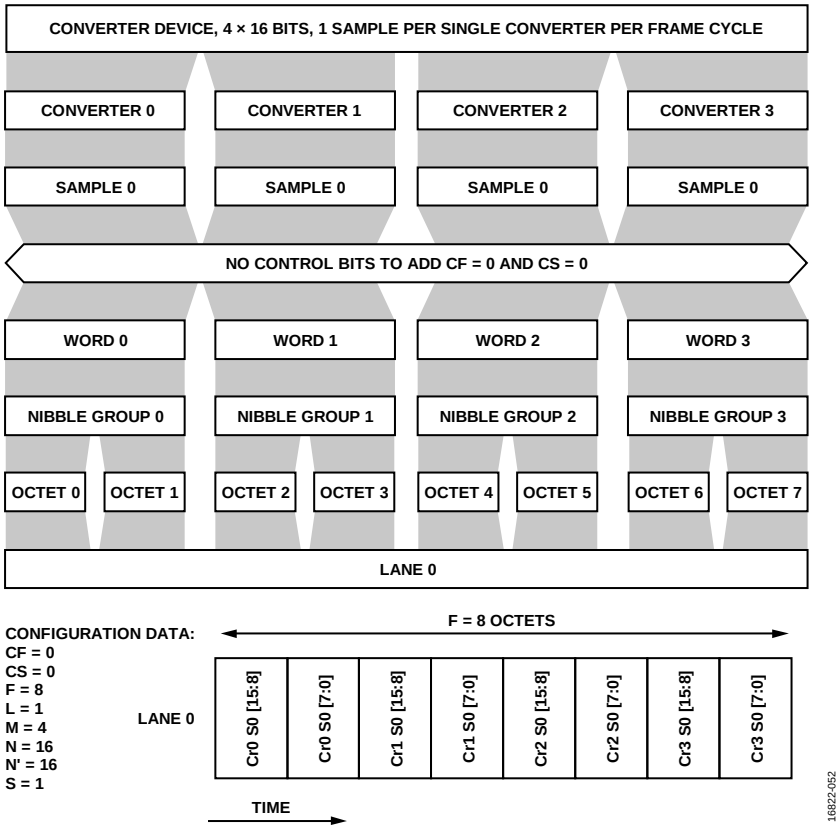

Figure 49. JESD204B Deframer Configuration (M = 4, L = 1)

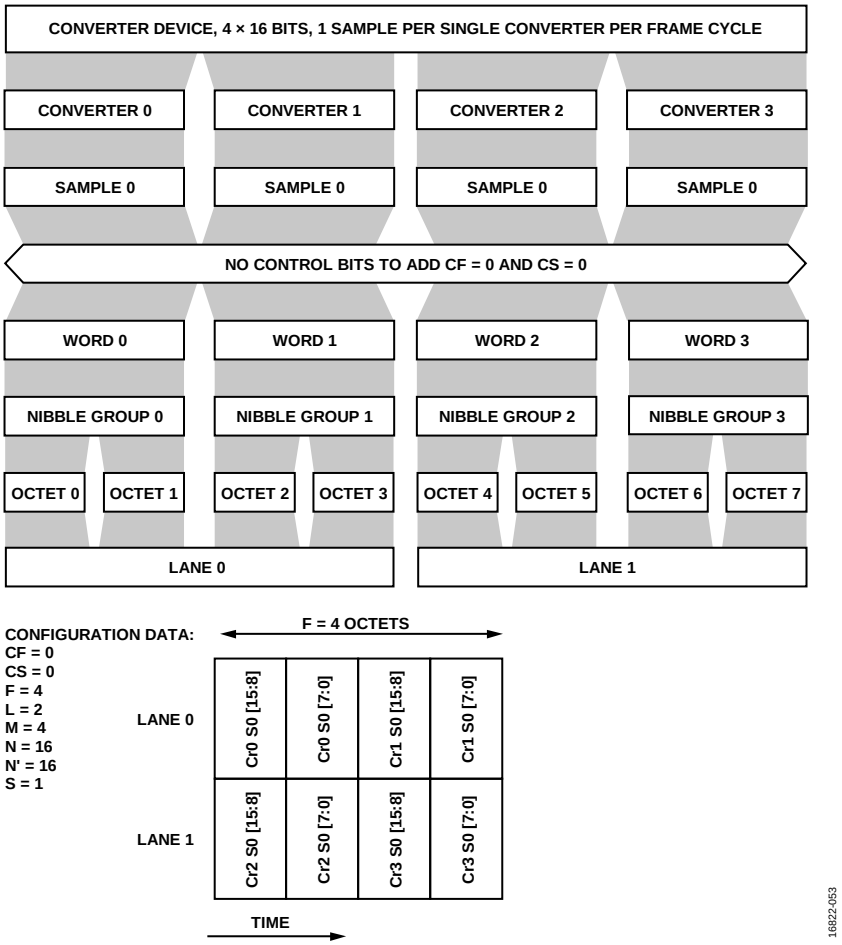

Figure 50. JESD204B Defframer Configuration (M = 4, L = 2)

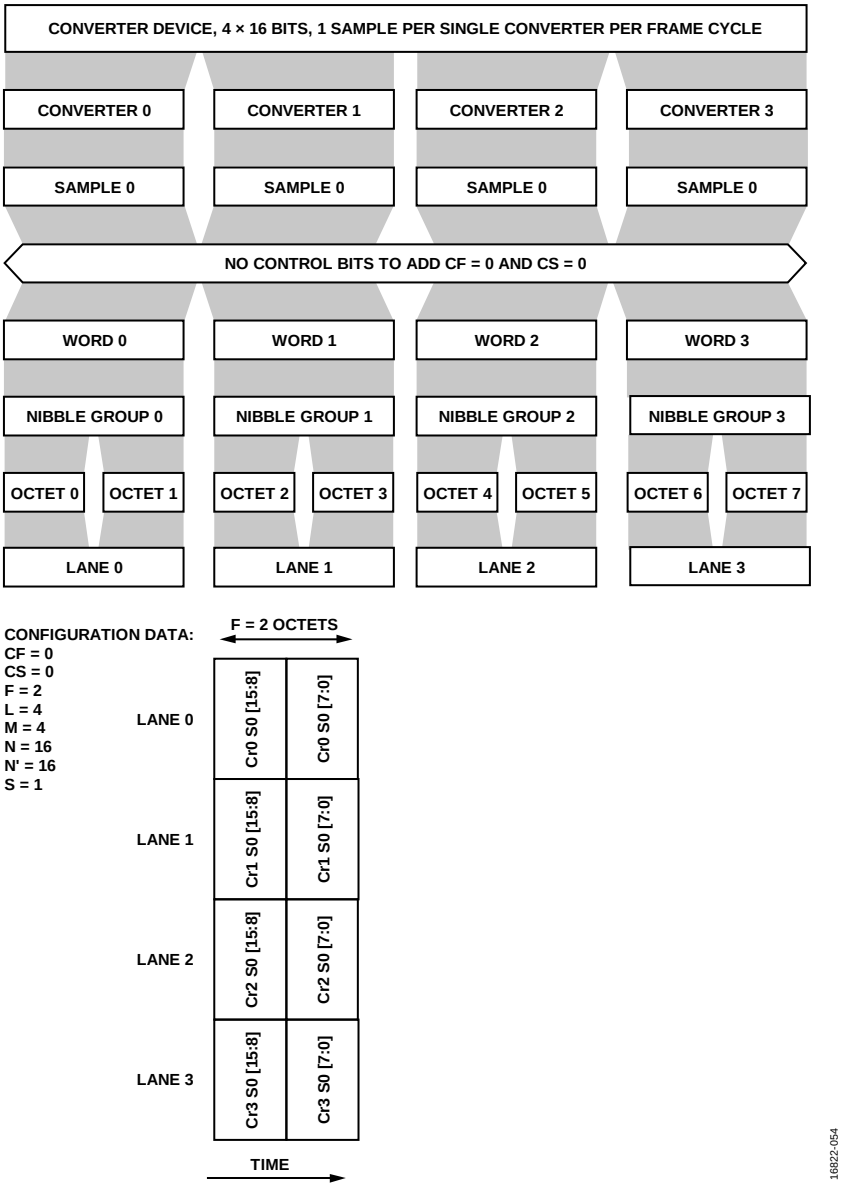

Figure 51. JESD204B Deframer Configuration (M = 4, L = 4)

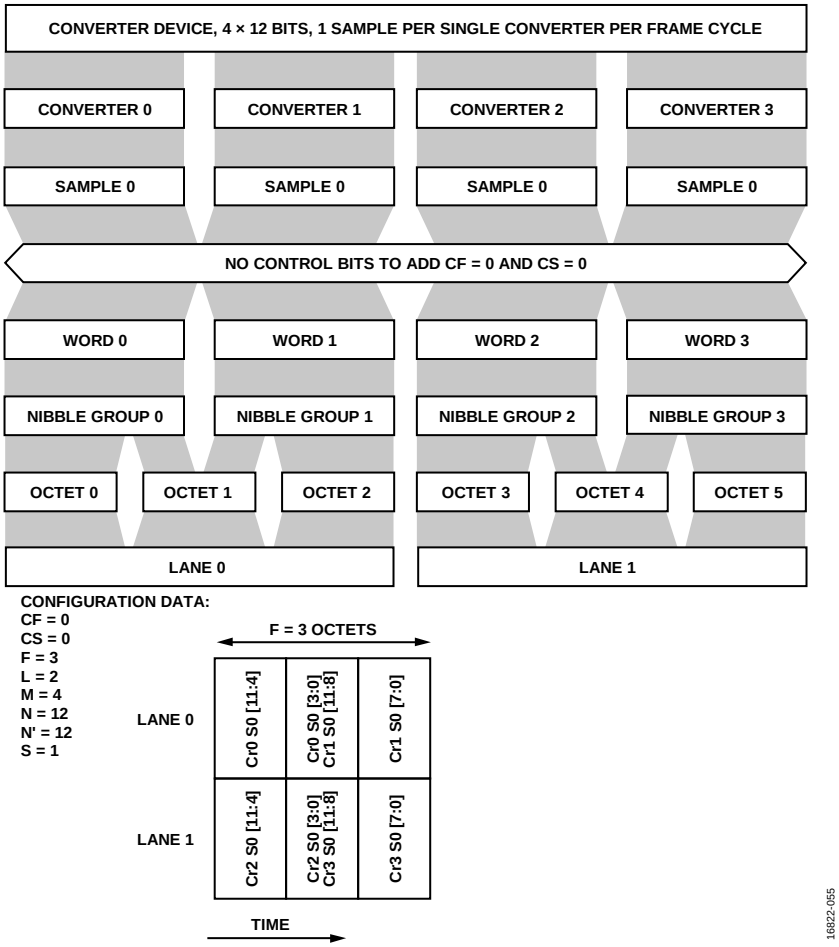

Figure 52. JESD204B Deframer Configuration (M = 4, L = 2, N' = 12)

SYSTEM INITIALIZATION

This section provides information about the initialization process for the device utilizing the API developed by Analog Devices. This section does not explain the API library functions. Detailed information regarding the API functions can be found in the device API doxygen document (**talise.chm**) located at **/src/doc**. This section does not describe API integration and the hardware abstraction interface.

DEVICE INITIALIZATION SEQUENCE

The initialization sequence is comprised of API calls intermixed with user defined function calls that are specific to the hardware platform. The API functions perform all of the necessary tasks for transceiver configuration, calibration, and control. The user is required to insert their code into the initialization sequence, specific to the hardware platform requirements. These platform requirements include but are not limited to: user clock device, user FPGA/application specific integrated circuit (ASIC)/BBIC JESD204B interface, datapath control, and various system checks that are governed by the application. The source code contained in the **/src/example/headless.c** file provides a basic initialization sequence with code comments to help guide the user with the insertion of their application specific code.

Device Initialization Sequence Order

The initialization sequence is as follows:

1. Instantiate all data structures and load their members required by the user application.
2. Initialize and setup of all clocks (platform clock source and JESD204B SYSREF signals are set up).
3. Initialize hardware platform (hardware dependent devices such as FPGA/ASIC/BBIC interfaces are initialized).
4. Initialize hardware for API (call `TALISE_openHw`).
5. Reset the device (call `TALISE_resetDevice` for reset of transceiver device in preparation for initialization).
6. Initialize the device (call `TALISE_initialize` function for configuration of the device).
7. Check clock PLL status for lock (call `TALISE_getPllLockStatus` and perform check with user defined code).
8. Multichip synchronization (necessary for JESD204B deterministic latency requirements).
9. Check clock PLL status for lock (call `TALISE_getPllLockStatus` and perform check with user defined code).
10. Initialize the Arm processor (call `TALISE_initArm`).
11. Initialize the stream processor (call `TALISE_initStreamProc`).
12. Load the stream processor binary file (call `TALISE_loadStreamProcessor`).
13. Load the Arm binary file (call `TALISE_loadArmFromBinary` with user defined, binary array pointer).
14. Set the RF PLL frequencies (call `TALISE_setRfPllFrequency`).
15. Perform RF PLL lock check (call `TALISE_checkPllLockStatus`).
16. Run the initialization calibrations (call `TALISE_runInitCals` and `TALISE_waitInitCals` with user defined code).
17. Enable the SYSREF signal detection for the device (call `TALISE_enableSysrefToFramer` functions).
18. Send the SYSREF signal to bring up the JESD204B interface.
19. Check the device JESD204B (deframer and/or framer) status (call `TALISE_readDeframerStatus` and `TALISE_readRxFramerStatus`).
20. Verify sync and link status for hardware platform.
21. Enable the tracking calibrations (call `TALISE_enableTrackingCals`).
22. Turn the radio on (call `TALISE_radioOn`).
23. Power up desired transmitters and receivers (call `TALISE_ setRxTxEnable`).

DEVICE INITIALIZATION EXAMPLE CODE

For an example code, refer to the **headless.c** file located in the **/src/app/example/headless.c** folder.

SYSTEM SHUTDOWN

This section provides information about the shutdown process for the device that is utilizing the API developed by Analog Devices. This section does not explain the API library functions. Detailed information regarding the API functions can be found in the device API doxygen document (**talise.chm**) located at **/src/doc**.

DEVICE SHUTDOWN SEQUENCE

The API library provides two main APIs that can perform a system shutdown procedure. The `TALISE_shutdown` function resets the device into a safe state for powering down the device. The `TALISE_closeHw` function performs a hardware shutdown for the device. This function calls `ADIHAL_closeHw`, which shuts down all of the external hardware blocks required to operate the device. This hardware shutdown procedure is defined by the user's implementation of ADIHAL.

Device Shutdown Sequence Order

The device shutdown sequence is as follows:

1. Turn the radio off (call `TALISE_radioOff`).
2. Put the device into safe state for shutdown (call `TALISE_shutdown`).
3. Shut down the external hardware for the device (call `TALISE_closeHw`).

STREAM PROCESSOR AND SYSTEM CONTROL

STREAM PROCESSOR

The device supports quick configuration from idle states to operation, or, given the shared nature of the observation receiver and receiver paths, supports a quick transition between receive and observation modes. The stream processor is a processor within the device that is tasked with performing a series of configuration tasks upon an external request. When a request is made, the stream processor performs a series of defined actions defined in the image that is loaded into the stream during device initialization.

Therefore, the stream processor executes a series of tasks, also known as streams, for the following operations:

- Transmitter 1 enable/Transmitter 1 disable, Transmitter 2 enable/Transmitter 2 disable
- Receiver 1 enable/Receiver 1 disable, Receiver 2 enable/Receiver 2 disable
- Observation Receiver 1 enable/Observation Receiver 1 disable, Observation Receiver 2 enable/Observation Receiver 2 disable

Enabling and disabling paths is done typically using pins. However, these functions can also be controlled via the SPI bus (see the System Control section for details). The stream is not limited to path enabling events and can react to other events, such as a GPIO input signal.

The device and the stream are flexible in configuration. In the same way that the initialization structures change with the profile, the stream processor image must change with the configuration. For example, the stream that enables Observation Receiver 1 is different if a 450 MHz or 200 MHz profile is chosen. For this reason, it is necessary to save a stream image for each configuration of the device. When the user saves configuration files (**.c**) using the GUI, a stream image is also saved automatically. Use this stream file when using the configuration files.

The following are examples of why the stream files differ:

- The framer choices for the observation receiver and the receiver.
- If link sharing is used or not between the observation receiver and receiver.
- If the observation receiver stitching is used or not.
- The DAC mode choice in TDD modes (for example, whether or not the DAC is powered off when the transmitter is disabled).
- If floating point formatting is used on the receiver and observation receiver paths.

SYSTEM CONTROL

The signal paths within the device can be controlled either through the API or through the pin controls. If the device is controlled through the API, this mode of control is reliant on the SPI communication bus. Therefore, for critical time alignment of powering on/off chains, pin control is recommended. Each path is independently controlled with the enable signals defined in Table 28.

Table 28. Signal Chain Enable Signals

Enable Signal	Applicable Devices
RX1_ENABLE	ADRV9008-1, ADRV9009
RX2_ENABLE	ADRV9008-1, ADRV9009
TX1_ENABLE	ADRV9008-2, ADRV9009
TX2_ENABLE	ADRV9008-2, ADRV9009
ORX1_ENABLE	ADRV9008-2, ADRV9009
ORX2_ENABLE	ADRV9008-2, ADRV9009

API Control

To control the signal paths through the API, the use the following command:

```
TALISE_ setRxTxEnable(taliseDevice_t *device, taliseRxORxChannels_t rxOrxChannel,
taliseTxChannels_t txChannel)
```

See Table 29 and Table 30.

Table 29. taliseTxChannels_t Enumeration Definitions

taliseTxChannels_t Elements	Enabled Channel
TAL_TXOFF	Not applicable
TAL_TX1	Tx1
TAL_TX2	Tx2
TAL_TX1TX2	Tx1 and Tx2

Table 30. taliseRxORxChannels_t Enumeration Definitions

taliseRxORxChannels_t Elements	Enabled Channel
TAL_RXOFF_EN	Not applicable
TAL_RX1_EN	Rx1
TAL_RX2_EN	Rx2
TAL_RX1RX2_EN	Rx1 and Rx2
TAL_ORX1_EN	ORx1
TAL_ORX2_EN	ORx2
TAL_ORX1ORX2_EN	ORx1 and ORx2

The ADRV9008-1 does not have transmitter paths. Therefore, `TAL_TxOff` must be selected for the transmitter channel. Do not call the observation receiver options for the `taliseRxORxChannels_t` selection.

The ADRV9008-2 does not have any receiver paths. Therefore, Receiver 1, Receiver 2, and Receiver 1/Receiver 2 must not be chosen.

When `TALISE_setRxTxEnable` is called, the requested channels are enabled. The channels remain active until further instruction with this command. Note that if the observation receiver is enabled continuously and is not returned to `TAL_RXOFF_EN` for any time, the transmitter tracking calibrations are not able to function (as explained in the System Considerations for Arm Calibrations section).

Pin Control

The device chains can also be controlled using a series of enable pins. When these pins are toggled high, the relevant signal chain is enabled (see Table 31).

Table 31. Enable Signal Pin Numbers

Device	Enable Signal Name	Pin Number
ADRV9008-1	RX1_ENABLE	M5
	RX2_ENABLE	M7
ADRV9008-2	TX1_ENABLE	M6
	TX2_ENABLE	M8
	ORX1_ENABLE	M5
	ORX2_ENABLE	M7
ADRV9009	RX1_ENABLE	M5
	RX2_ENABLE	M7
	TX1_ENABLE	M6
	TX2_ENABLE	M8
	ORX1_ENABLE	Requires a GPIO pin (see Table 32)
	ORX2_ENABLE	Requires a GPIO pin (see Table 32)

As noted in Table 31, the ADRV9009 requires its ORXx_ENABLE signals to be supplied through GPIO_x pins. The GPIO_x pin options are shown in Table 32.

Table 32. Permissible GPIO Pins for Observation Receiver Control in the ADRV9009

Option	GPIO Pin Assignment
Option 1	GPIO_0, enables ORx1
	GPIO_1, enables ORx2
Option 2	GPIO_4, enables ORx1
	GPIO_5, enables ORx2
Option 3	GPIO_8, enables ORx1
	GPIO_9, enables ORx2

The following API command advises which enable signals are controlled with the pins, and also in the case of the ADRV9009, which GPIO_x pins are used for the ORXx_ENABLE signals:

```
TALISE_ setRadioCtlPinMode(taliseDevice_t *device, uint8_t pinOptionsMask, taliseRadioCtrlCfg2_t orxEnGpioPinSel)
```

In this command, `pinOptionsMask` is comprised of the `taliseRadioCtlCfg1_t` enumerated types described in Table 33 to configure the use of the enable pins. `orxEnGpioPinSel` is a `taliseRadioCtlCgf2_t` enumerated type that is described in Table 34, that indicates which GPIO_x pins are assigned for the ORXx_ENABLE pins in the ADRV9009.

Table 33. taliseRadioCtlCfg1_t Enumeration Definition

taliseRadioCtlCfg1_t Elements	Description
TAL_TXRX_PIN_MODE	Configures the device for Rx and Tx path control via the RXx_ENABLE and TXx_ENABLE pins.
TAL_ORX_PIN_MODE	Configures the device for ORx path control via the ORXx_ENABLE pins.
TAL_ORX_USES_RX_PINS	Utilize for the ADRV9008-2. This element configures the device to use the M5 and M7 pins, as shown in Table 31. Do not use this element for the ADRV9009 because these pins are utilized for Rx enabling.
TAL_ORX_SEL	Selects ORx1 when set to 0, and ORx2 when set to 1.
TAL_ORX_SINGLE_CHANNEL	This element is used when there is a single ORXx_ENABLE pin and the ORx paths are switched between ORx1 and ORx2 through API control.
TAL_ORX_ENAB_SEL_PIN	Selects the enable pin to be utilized for the TAL_ORX_SINGLE_CHANNEL use case.

Table 34. taliseRadioCtlCfg2_t Enumeration Definition

taliseRadioCtlCfg2_t Elements	Description
TAL_ORX1ORX2_PAIR_01_SEL	Selects Option 1 defined in Table 32 for the ORx enable pins in the ADRV9009.
TAL_ORX1ORX2_PAIR_45_SEL	Selects Option 2 defined in Table 32 for the ORx enable pins on the ADRV9009.
TAL_ORX1ORX2_PAIR_89_SEL	Selects Option 3 defined in Table 32 for the ORx enable pins on the ADRV9009.
TAL_ORX1ORX2_PAIR_NONE_SEL	No pins selected. Use this command for the ADRV9008-1 and the ADRV9008-2.

The `pinOptionsMask` is created by observation receiving the appropriate elements as detailed in Table 33. Assign a single element (listed in Table 34) to the `orxEnGpioPinSel`. For example, for the ADRV9009 that is utilizing the pin control mode of the receiver, transmitter, and observation receiver paths, and choosing GPIO_0 and GPIO_1 for the observation receiver enable pins, the masks is as follows:

```
unit8_t pinOptionsMask = TAL_TXRX_PIN_MODE | TAL_ORX_PIN_MODE;
uint8_t orxEnGpioPinSel = TAL_ORX1ORX2_PAIR_01_SEL;
```

USE CASES

This section details example use cases for the various devices and explains how the device typically operates to ensure that calibrations run.

ADRV9008-2, Two Transmitter, One Observation Receiver Use Case

This use case considers two types of feedback paths to the observation receiver input: one for digital predistortion (DPD) data and one utilized for voltage standing wave ratio (VSWR) reflections. Note that all switches in Figure 53 are shown in high position (Logic 1). In this case, the Observation Receiver 1 path was chosen as the observation path. However, it is equally valid for the Observation Receiver 2 path to be chosen.

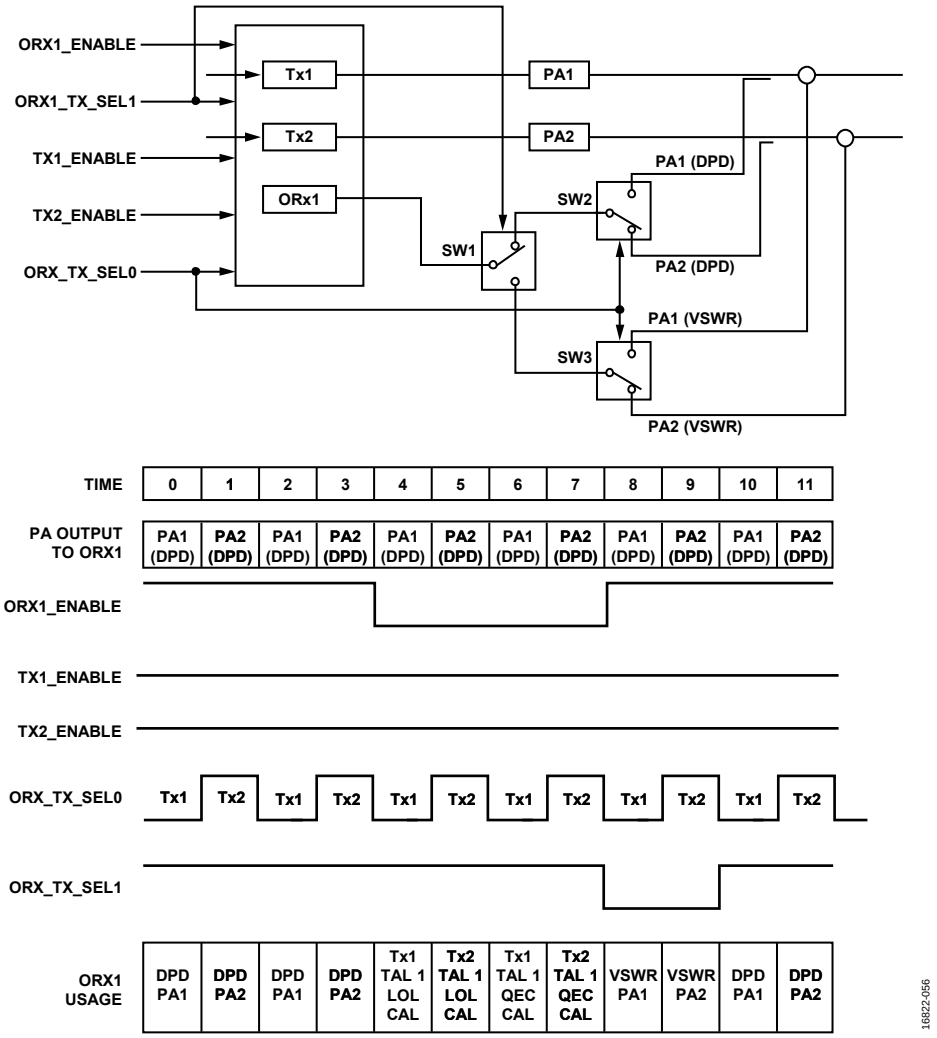

Figure 53. ADRV9008-2, Two Transmitter, One Observation Receiver Use Case

Figure 53 considers a frequency division duplex (FDD) use case, so TX1_ENABLE and TX2_ENABLE are high the entire time. The ORX1_ENABLE signal advises when the Observation Receiver 1 path used by the user, or when the path is available for the device to perform transmitter tracking calibrations. Tracking calibrations can only run when ORX1_ENABLE is low.

As described in Table 32, the transmitter local oscillator (LO) leakage calibration is dependent on the status of the GPIO pins that are configured to the ORX_TX_SEL0 and ORX_TX_SEL1 functions. The device can only run Transmitter 1 LO leakage tracking when there is a connection through the DPD path from Transmitter 1 to Observation Receiver1. Likewise, the device can only run Transmitter 2 LO leakage tracking when there is connection through the DPD path from Transmitter 2 to Observation Receiver 1. Therefore, as shown in Figure 53, it is equally valid to show the LO leakage calibrations where the QEC calibrations are shown in Figure 53. The scheduler determines, at any time, which calibration must be run (which ones are pending), and based on the enable and GPIO signals, which calibration can run.

The transmitter QEC calibration is not dependent on which transmitter is being externally looped back to the Observation Receiver 1 input. This calibration is only dependent on the utilized observation receiver path (in this case, Observation Receiver 1) being available for calibration, because the calibration uses an internal feedback path. The transmitter QEC calibration can run where the transmitter LO leakage tracking calibration is shown in Figure 53, and unlike the transmitter LO leakage tracking calibrations, the transmitter QEC tracking calibrations can also swap positions because the transmitter QEC tracking calibrations are not dependent on external feedback paths.

ORX1_TX_SEL1 advises if there is a valid feedback path between the Transmitter 1 or Transmitter 2 of this device and the observation receiver input being utilized. As shown in Figure 53, the ORX_TX_SEL1 is used to select between the DPD and VSWR paths. When the external LO leakage tracking calibration is running, it is important that the exact feedback path does not alternate between iterations of the calibration because the calibration algorithm learns the channel. Therefore, if in one instance of a Transmitter 1 LO leakage tracking calibration, Transmitter 1 is fed back to Observation Receiver 1 through the DPD path, but in another instance, it is fed back through the VSWR path, this channel alternation affects the performance of the algorithm. This means that only the DPD feedback must be utilized, and that no condition is allowed where the transmitter LO leakage tracking calibrations can run when VSWR is being fed back to the observation receiver input. These conditions are guaranteed in Figure 53 because the ORX_TX_SEL1 is used to switch between the DPD and VSWR paths.

ORX1_TX_SEL0 advises the Arm processor whether Transmitter 1 or Transmitter 2 is being fed back to the observation receiver input. In Figure 53, it also controls the switch that selects between the Transmitter 1 and Transmitter 2 paths.

The observation receiver QEC calibration (not shown in Figure 53) runs when the observation receiver path is enabled. This calibration does not run when the observation receiver path is disabled, for example, when the observation receiver path is available for transmitter calibrations.

ADRV9009, Two Receiver, Two Transmitter, One Observation Receiver Use Case

The ADRV9009 use case is very similar to the ADRV9008-2, two transmitter, one observation receiver use case. See the ADRV9008-2, Two Transmitter, One Observation Receiver Use Case section for details on transmitter calibration details and the status of the GPIO pins/enable signals required for the transmitter calibrations to operate properly.

As noted in the System Considerations for Tracking Calibrations section, the tracking calibrations must be assigned a minimum of 500 µs of continuous time on the observation path at any one time. The 500 µs observation time is the principle constraint on the TDD timing, with the receiver and transmitters enabled for a minimum of 500 µs when the calibration is scheduled to track. It is permissible to have special frames that are smaller than 500 µs as long as it is understood that the calibrations do not update based on observations of less than 500 µs.

In the ADRV9009, the receiver and observation receiver inputs share baseband paths. In the widest bandwidth setting, a stitching algorithm is utilized to form a single quadrature channel for the observation receiver path from four ADCs. These ADCs are also used to form quadrature channels for Receiver 1 and Receiver 2 during receiver periods. The ADCs must not be enabled together at any time. The two setups are mutually exclusive.

Receiver QEC tracking (not shown in Figure 54) is run continuously during the receiver data periods. This tracking is paused when the receiver channels are disabled and resumed when the receiver channels are reenabled. The observation receiver QEC calibration runs when the observation receiver path is enabled.

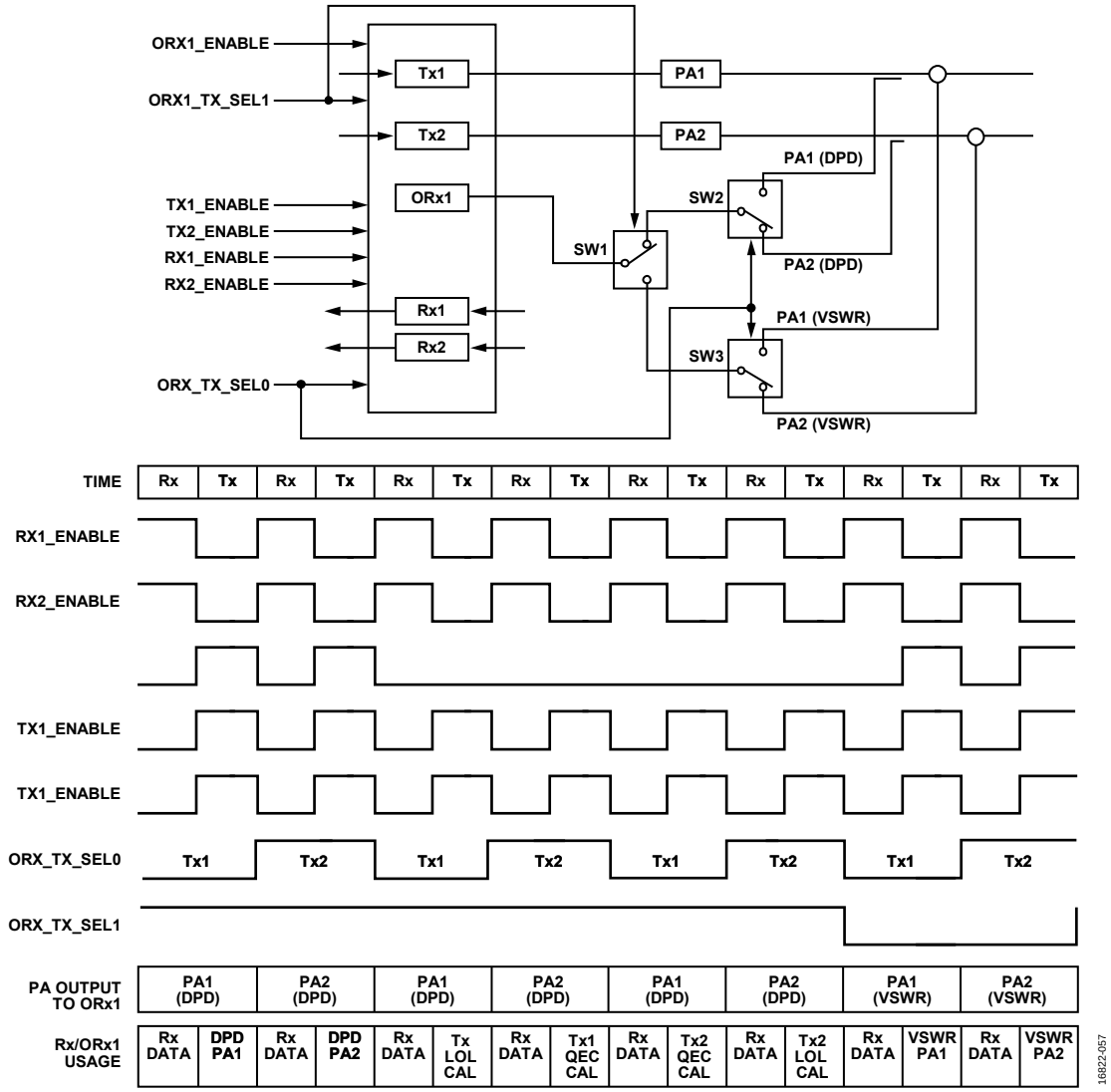

Figure 54. ADRV9009, Two Receiver, Two Transmitter, One Observation Receiver Use Case

GSM USE CASES

The device requires specific frequency planning to support multicarrier global systems for mobile communications (GSM) scenarios. These frequency plans ensure optimal sensitivity and harmonic rejection performance on the receiver. In these frequency plans, second-order harmonic distortion (HD2) falls in band and is corrected by an HD2 correction algorithm, and higher order harmonics fall out of band. For the transmitter, the frequency plans place LO leakage, the upper sideband, and the baseband third-order harmonic distortion (HD3) out of band.

These frequency plans require the use of a specific profile configuration. For the receiver, use the 200 MHz, I/Q rate of 245.76 MHz, decimate by 4 (DEC4) profile, or its low IF variant, the 100 MHz, low IF receiver profile with an I/Q rate of 122.88 MHz, DEC4. The low IF variant identically configures the device to the 200 MHz/245.76 MHz receiver profile, with the exception that the low IF variant uses a digital IF conversion stage to frequency shift and decimate the received signal for JESD204B link transmission at 122.88 MSPS. For the transmitter, use the widest bandwidth 200 MHz/450 MHz transmitter profile with an I/Q rate of 491.52 MHz.

GSM 1800 DIGITAL CELLULAR SYSTEM (DCS) BAND

The frequency plan for the GSM 1800 DCS band is shown in Table 35.

Table 35. Frequency Plan for the 1800 DCS Band

Function Name	LO Frequency (MHz)	IF Offset (MHz)	Low Edge (MHz)	High Edge (MHz)
Receiver	1695	52.5	15	90
Transmitter	1900	−57.5	−95	−20

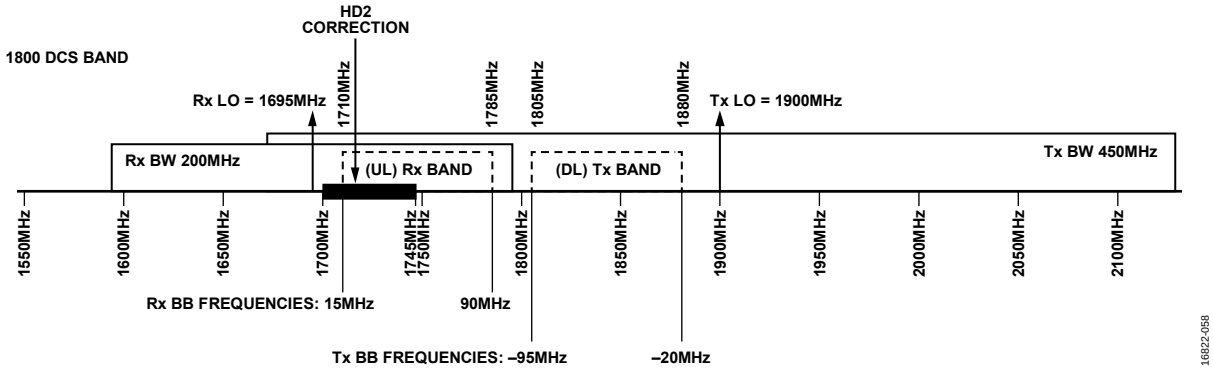

Figure 55. Frequency Plan for the 1800 DCS Band

GSM 1900 PERSONAL COMMUNICATIONS SERVICE (PCS) BAND

The frequency plan for the GSM 1900 PCS band is shown in Table 36.

Table 36. Frequency Plan for the 1900 PCS Band

Function Name	LO Frequency (MHz)	IF Offset (MHz)	Low Edge (MHz)	High Edge (MHz)
Receiver	1820	60	30	90
Transmitter	2010	−50	−80	−20

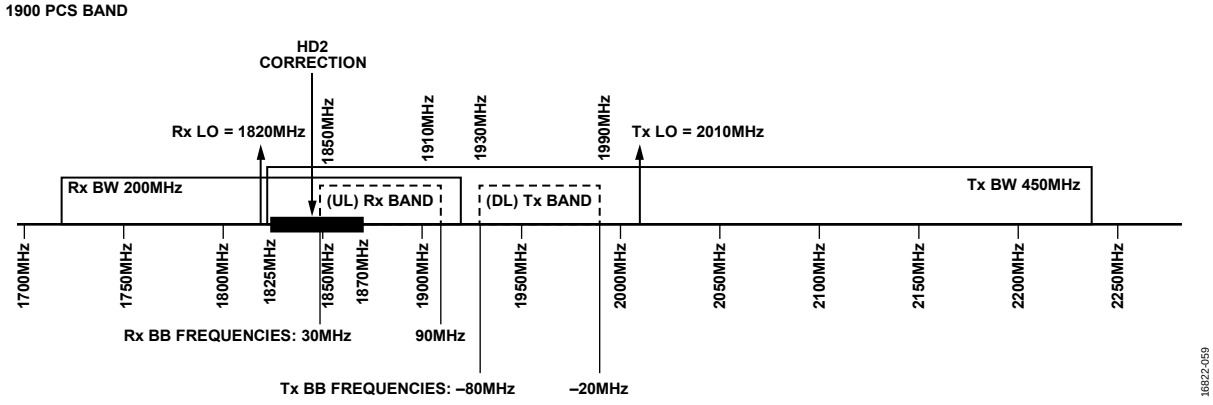

Figure 56. Frequency Plan for the 1900 PCS Band

GSM 850 BAND

The frequency plan for the GSM 850 band is shown in Table 37.

Table 37. Frequency Plan for the GSM 850 Band

Function Name	LO Frequency (MHz)	IF Offset (MHz)	Low Edge (MHz)	High Edge (MHz)
Receiver	796	40.5	28	53
Transmitter	914	−32.5	−45	−20

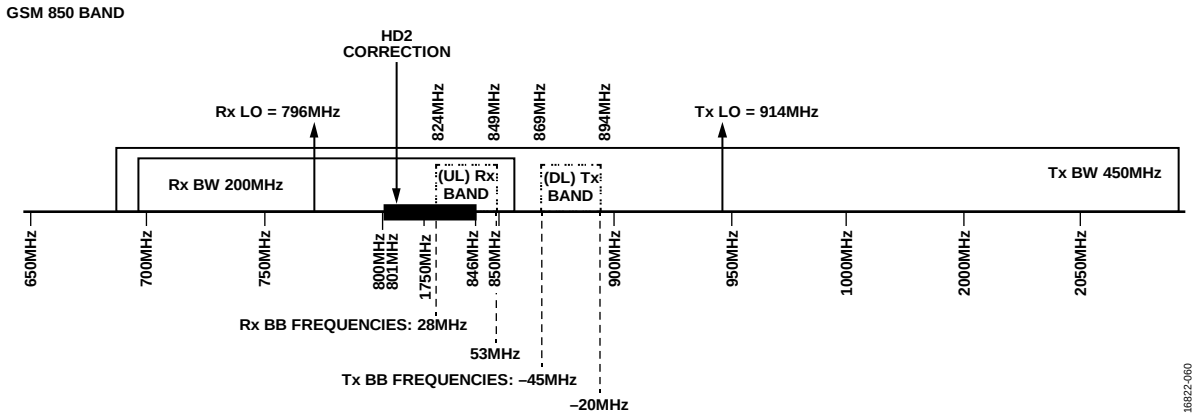

Figure 57. Frequency Plan for the GSM 850 Band

GSM 900 BAND

The frequency plan for the GSM 900 band is shown in Table 38.

Table 38. Frequency Plan for the GSM 900 Band

Function Name	LO Frequency (MHz)	IF Offset (MHz)	Low Edge (MHz)	High Edge (MHz)
Receiver	847	50.5	33	68
Transmitter	980	−37.5	−55	−20

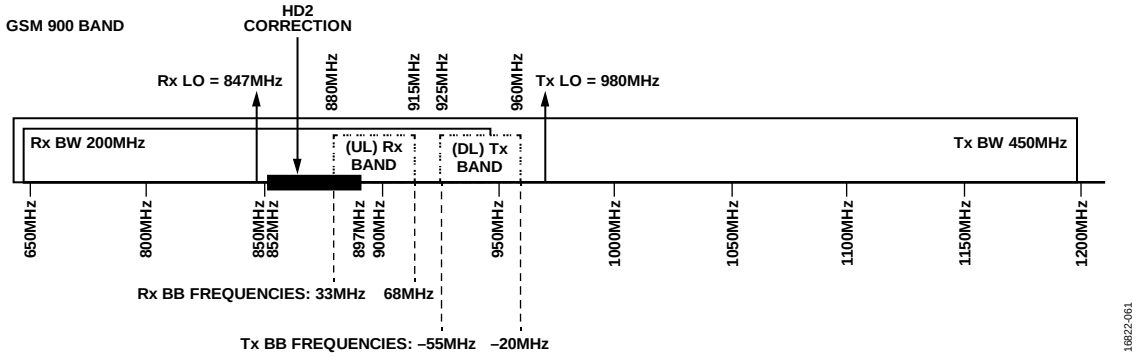

Figure 58. Frequency Plan for the GSM 900 Band

SYNTHESIZER CONFIGURATION

The device contains three RF PLL synthesizers: the RF LO synthesizer, auxiliary synthesizer, and the clock synthesizer. Figure 59 illustrates these synthesizers and their interconnectivity within the device. Each PLL synthesizer employs a fractional–N architecture with a completely integrated voltage controlled oscillator (VCO) and loop filter. No external components are required to cover the entire frequency range of the device. This configuration allows the use of any convenient reference frequency for operation on any channel with any sample rate. The fundamental frequency of each of the PLLs is from 6 GHz to 12 GHz. The LO frequency is created by dividing down the PLL VCO frequency. The reference frequency for the PLL is scaled from the reference clock applied to the chip REF_CLK_IN± pins.

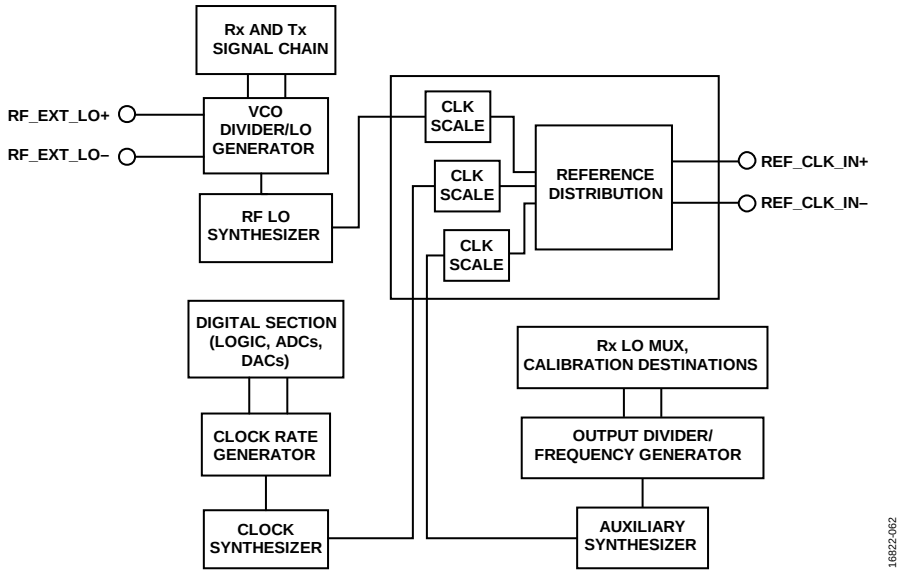

Figure 59. Synthesizer Interconnection Block Diagram

The clock synthesizer is used to generate all the clocking signals necessary to run the device. The reference frequency for the PLL is scaled from the reference clock that is applied to the chip REF_CLK_IN± pins. Note that although the PLL is of fractional–N architecture, the signal sampling relationships to the JESD204B interface rates typically require that the clock synthesizer operates in integer mode. Profiles included in the TTES configure the clock synthesizer appropriately. Reconfiguration of the clock synthesizer is typically not necessary after initialization. The most direct approach to clock synthesizer configuration is to follow the recommended programming sequence and utilize the provided API functions to set the clock synthesizer to the required mode of operation.

An auxiliary synthesizer is integrated into the device to generate the signals necessary to calibrate the device. The reference frequency for the auxiliary synthesizer is scaled from the device clock that is applied to the chip REF_CLK_IN± pins. The output signal is connected to a switching network and injected into the various circuits to calibrate filter bandwidth corners, or injected into the receiver signal chain as an offset LO. A number of calibrations are executed during the initialization sequence at startup. No signals are present at the receiver/observation receiver input during the tone calibration time, and calibrations are fully autonomous. During the calibrations, the auxiliary synthesizer is controlled solely by the internal Arm processor, and the synthesizer does not require any user interactions.

CONNECTIONS FOR EXTERNAL CLOCK (REF_CLK_IN± PINS)

The external clock is used as the reference clock for the RF PLL and the clock PLL on the device and must be a clean clock source. Connect the external clock inputs to the REF_CLK_IN+ pin and the REF_CLK_IN− pin via ac coupling capacitors. Terminate the differential signal prior to the capacitors with 100 Ω as shown in Figure 60. The inputs of the device are biased to a 618 mV voltage level. The inputs are high impedance, with less than 1 pF and 20 kΩ each. The frequency range of the REF_CLK signal must be between 10 MHz and 1000 MHz. Ensure that the external clock peak-to-peak amplitude does not exceed 2 V. Note that for best spurious performance, the REF_CLK signal input level must not exceed 1 V). For best synthesizer performance, a high slew rate signal is best, with fast rise and fall times. A clipped sine wave type signal is recommended.

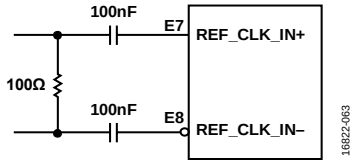

Figure 60. Reference Clock Input Connections

REF_CLK_IN± SIGNAL PHASE NOISE REQUIREMENTS

To prevent performance degradation, the REF_CLK_IN± reference must be a clean signal. Optimal performance from the synthesizer results if the applied reference is ideal. However, this is unrealistic because ideal sources do not exist, and if the sources did, these sources would be exorbitantly expensive. Table 39 lists the required phase noise of the REF_CLK_IN± signal for a 1 dB system phase noise degradation compared to an ideal REF_CLK_IN± signal. For different REF_CLK_IN± signal frequencies, the information shown in Table 39 can be scaled appropriately. A clock source with phase noise performance as specified in Table 39 (or better) allows the device to achieve the specifications listed in the ADRV9008-1, ADRV9008-2, and ADRV9009 data sheets. Using a higher phase noise source can degrade performance delivered by the transceiver.

Table 39. REF_CLK_IN± Signal Phase Noise Requirements for 1 dB System Phase Noise Degradation Compared to an Ideal Reference Clock (Narrow PLL Loop Bandwidth)

	Narrow PLL Loop Bandwidth (Approximately 50 kHz), Default, Typically Used <3 GHz		
Frequency Offset from Carrier	**122.88 MHz (dBc/Hz)**	**153.6 MHz (dBc/Hz)**	**245.76 MHz (dBc/Hz)**
100 Hz	−113.02	−111.08	−107.00
1000 Hz	−125.02	−123.08	−119.00
10 kHz	−133.02	−131.08	−127.00
100 kHz	−137.02	−135.08	−131.00
1 MHz	−133.02	−131.08	−127.00
10 MHz	−104.02	−102.08	−98.00

Table 40. REF_CLK_IN± Signal Phase Noise Requirements for 1 dB System Phase Noise Degradation Compared to an Ideal Reference Clock (Wide PLL Loop Bandwidth)

	Wide PLL Loop Bandwidth (Approximately 300 kHz), User Configured, Typically Used >3 GHz		
Frequency Offset from Carrier	**122.88 MHz (dBc/Hz)**	**153.6 MHz (dBc/Hz)**	**245.76MHz (dBc/Hz)**
100 Hz	−114.02	−112.08	−108.00
1000 Hz	−127.02	−125.08	−121.00
10 kHz	−138.02	−136.08	−132.00
100 kHz	−146.02	−144.08	−140.00
1 MHz	−147.02	−145.08	−141.00
10 MHz	−118.02	−116.08	−112.00

SYNTHESIZER SOFTWARE CONFIGURATION

The configuration of the device is dependent on application requirements. When using an external LO, use the TTES software to generate initial values for the API structure members.

Figure 59 outlines a high level synthesizer block diagram. There is an option to provide an external LO for the receiver and transmitter signal chains. Figure 61 shows where the user can select an external LO from the **Ext. LO** dropdown list in the **Configuration** tab of the TTES. An external LO can also be selected using API commands. Before the user begins to initialize the device, the `taliseDigClocks_t` structure data field `rfPllUseExternalLo` can be set to 0 or 1 to select an internal or external LO source. The external source must be 2× the desired LO.

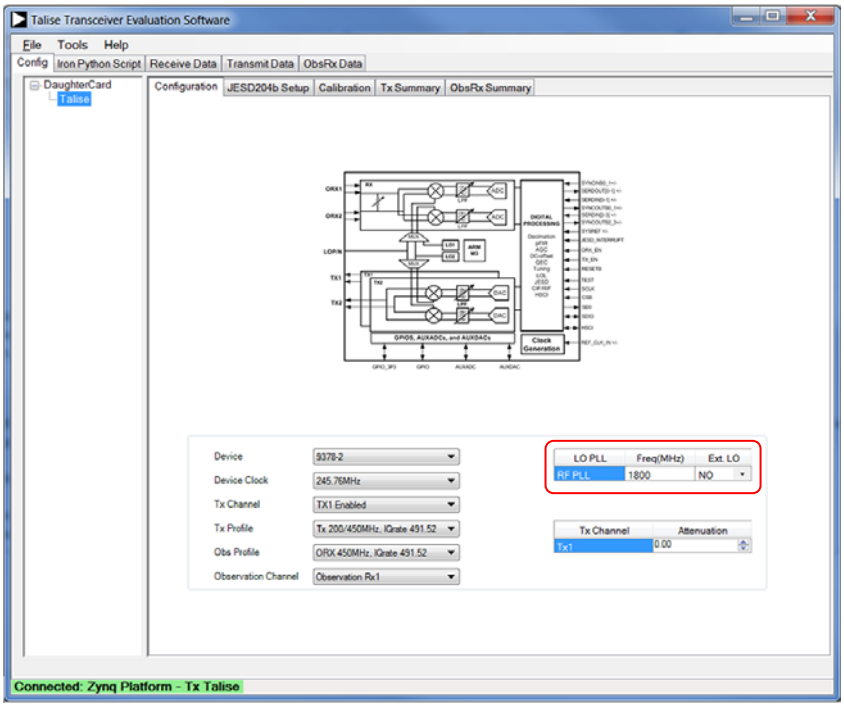

Figure 61. Internal and External LO Configuration in TTES

Part of the device initialization procedure includes the set up of an internal clock generation. All internal clocks are generated based on the selected profile (reference frequency applied to the device, JESD204B lane rates, and bandwidth mode). Therefore, there is no need to reconfigure the clock synthesizer when the device initialization sequence is complete.

The API function that initializes the clock rate generator block (see Figure 59) is as follows:

```
uint32_t TALISE_initialize(taliseDevice_t *device, taliseInit_t *init)
```

The initialization sequence calculates and updates the clock synthesizer and loop filter settings based on a VCO frequency lookup table (LUT). The VCO frequency break points for the synthesizer LUT can be found in an array, `vcoFreqArrayHz`, which can be found in the **talise.c** file.

Synthesizer API Data Structures

This section describes the synthesizer data structures.

taliseDigClocks_t

The synthesizer configuration is stored in the `taliseDigClocks_t` data structure. The data structure contains the types that are described in Table 41.

Table 41. taliseDigClocks_t Data Structure Description

Type	Data Field	Permissible Values	Description
uint32_t	deviceClock_kHz	122.88 MHz 184.32 MHz 245.76 MHz	Clock PLL and device reference clock frequency in kHz.
uint32_t	clkPllVcoFreq_kHz	6,000,000 to 12,000,000	Clock PLL VCO frequency in kHz.
taliseHsDiv_t	clkPllHsDiv	TAL_HSDIV_2 TAL_HSDIV_2P5 TAL_HSDIV_3 TAL_HSDIV_4 TAL_HSDIV_5	Clock PLL high speed clock divider, permissible values defined by the taliseHsDiv_t enum.
uint8_t	rfPllUseExternalLo	0, 1	1 = external LO for RF PLL. 0 = internal LO generator for RF PLL.
taliseRfPllMcs_t	rfPllPhaseSyncMode	TAL_RFPLLMCS_NOSYNC TAL_RFPLLMCS_INIT_AND_SYNC TAL_RFPLLMCS_INIT_AND_1TRACK TAL_RFPLLMCS_INIT_AND_CONTTRACK	Sets RF PLL phase sync mode. Adds extra time to lock RF PLL when PLL frequency change. Permissible values defined by taliseRfPllMcs_t enum.

Table 42. PLL Enumerators

Parameter Name	Enumerator Value	Description
pllName	TAL_RF_PLL	Selects RF PLL for receiver and transmitter.
	TAL_CLK_PLL	Selects clock PLL for receiver and transmitter.
	TAL_AUX_PLL	Selects auxiliary PLL for receiver and transmitter.

Synthesizer API Functions

The public functions described in this section are provided to the user to configure and observe the device synthesizer settings.

TALISE_getRfPllFrequency()

Use this function to get the current operating frequency of the PLL. A `taliseRfPllName_t` enumerated type is passed for the desired PLL frequency to read. The function is as follows:

```
TALISE_getRfPllFrequency(taliseDevice_t* device, taliseRfPllName_t pllName, uint64_t*
rfPllLoFrequency_Hz)
```

Precondition: this function can be used after the device has been initialized and the PLLs are configured. For the auxiliary PLL or RF PLL, the Arm firmware must also be loaded and running to read back the PLL frequencies.

Parameters include the following:

- `device` is a pointer to the device data structure containing settings.
- `pllName` is the name of the desired PLL to read the frequency.
- `rfPllLoFrequency_Hz` is a 4-byte pointer to return the current LO frequency in Hz for the specified PLL.

TALISE_setRfPllFrequency()

Use this function to set the operating frequency of a PLL. A `taliseRfPllName_t` enumerated type is passed for the desired PLL to read. This function is as follows:

```
TALISE_setRfPllFrequency(taliseDevice_t* device, taliseRfPllName_t pllName, uint64_t
rfPllLoFrequency_Hz)
```

Precondition: this function can be called after the Arm processor has been initialized and the device must be in radio off state.

Parameters include the following:

- `device` is a pointer to the device data structure containing settings.
- `pllName` is the name of the desired PLL to read the frequency.
- `rfPllLoFrequency_Hz` is the desired RF LO frequency in Hz.

TALISE_setRfPllLoopFilter()

Use this function to set the configuration of the RF PLL loop filter. The function is as follows:

```
TALISE_setRfPllLoopFilter(taliseDevice_t* device, uint16_t loopBandwidth_kHz, uint8_t stability)
```

Precondition: this function can be called after the Arm has been initialized. The device must also be in the radio off state. This function must be followed with a `TALISE_setRfPllFrequency()` command for the `TAL_RF_PLL` enumerator value to set up the RF PLL with the new loop filter configuration.

Parameters include the following:

- `device` is a pointer to the device data structure containing settings.
- `loopBandwidth_kHz` is a desired loop bandwidth in kHz. Valid range is between 50 kHz and 750 kHz.
- `stability` is a factor that impacts noise and stability of the loop filter. Valid range is between 3 and 15. Lower values decrease stability and increase rejection of noise.

TALISE_getRfPllLoopFilter()

Use this function to get the current loop bandwidth and stability factor for the RF PLL. The function is as follows:

```
TALISE_getRfPllLoopFilter(taliseDevice_t* device, uint16_t* loopBandwidth_kHz, uint8_t*
stability)
```

Precondition: this function can be used after the device has been initialized and the RF PLL has been configured. The Arm firmware must also be loaded and running.

Parameters include the following:

- `device` is a pointer to the device data structure containing settings.
- `loopBandwidth_kHz` is a 2-byte pointer to value of loop bandwidth in kHz. Valid range is between 50 kHz and 750 kHz.
- `stability` is a 1-byte pointer to stability setting of loop filter. Impacts noise and stability of the loop filter. Valid range is between 3 and 15. Lower values decrease stability and increase rejection of noise.

RF PLL FREQUENCY CHANGE PROCEDURE

This section describes the procedure to use when an RF PLL change is required under specific conditions. If the user wishes to change the transmitter, receiver, and observation receiver frequencies, if the frequency step change is less than 100 MHz, and if the frequency step does not cross the VCO frequency break points (defined in `vcoFreqArrayHz`, located in the **talise.c** file) by two boundaries, use the following procedure:

1. Move the device to the radio off state by executing the following command:

```
if   ((talError = TALISE_radioOff(&talDevice)) != TALACT_NO_ACTION)
    {
   /*** < Info: errorString will contain log error string in order to debug failure > ***/
   talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
   errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
    }
```

2. Program the new LO frequency. For example, set the LO to 2550 MHz by executing the following commands:

```
if   ((talError = TALISE_setRfPllFrequency(&talDevice, TAL_RF_PLL , 2550000000)) !=
TALACT_NO_ACTION)
    {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
    }
   /*** < Action: wait 1ms for PLLs to lock > ***/
if  ((talError = TALISE_getPllsLockStatus(&talDevice, &pllLockStatus))  != TALACT_NO_ACTION)
  {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
  }
if ((*pllLockStatus &  0x0F) == 0x07)
  {
   /*** < Info: RF, CLK, and AUX PLL locked >     ***/
  }
else
  {
                /*** < Info: RF, CLK, or AUX PLL not locked > ***/
   /*** < Action: Ensure lock before proceeding  - User code here> ***/
  }
```

3. Reset the external channel by executing the following command:

```
if   ((talError = TALISE_resetExtTxLolChannel (&talDevice, TAL_TX1TX2)) != TALACT_NO_ACTION)
      {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
      }
```

4. Move the device back to the radio on state by executing the following command:

```
if   ((talError = TALISE_radioOn(&talDevice)) != TALACT_NO_ACTION)
      {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
      }
```

If an RF PLL change is required with the following conditions: if the user wishes to change the transmitter, receiver, and observation frequencies, if the frequency step change is more than 100 MHz, or if the frequency step crosses the VCO frequency break points (defined in `vcoFreqArrayHz`, located in the **talise.c** file), use the following procedure:

1. Move the device to the radio off state by executing the following command:

```
if   ((talError = TALISE_radioOff(&talDevice)) != TALACT_NO_ACTION)
    {
   /*** < Info: errorString will contain log error string in order to debug failure > ***/
   talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
   errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
    }
```

2. Program the new LO frequency. For example, set the LO to 2550 MHz by executing the following commands:

```
if   ((talError = TALISE_setRfPllFrequency(&talDevice, TAL_RF_PLL , 2550000000)) !=
TALACT_NO_ACTION)
    {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
    }
   /*** < Action: wait 1ms for PLLs to lock > ***/
if  ((talError = TALISE_getPllsLockStatus(&talDevice, &pllLockStatus))  != TALACT_NO_ACTION)
  {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
  }
if ((*pllLockStatus &  0x0F) == 0x07)
  {
   /*** < Info: RF, CLK, and AUX PLL locked >     ***/
  }
else
  {
                /*** < Info: RF, CLK, or AUX PLL not locked > ***/
   /*** < Action: Ensure lock before proceeding  - User code here> ***/
  }
```

3. Rerun the initialization calibrations by calling `TALISE_runInitCals` and `TALISE_waitInitCals` with user defined code.
4. Move the device back to the radio on state by executing the following command:

```
if   ((talError = TALISE_radioOn(&talDevice)) != TALACT_NO_ACTION)
     {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
     }
```

RF PLL LOOP FILTER RECOMMENDATIONS

For optimal phase noise and error vector magnitude (EVM) performance, the RF PLL loop filter bandwidth settings listed in Table 43 are recommended.

Table 43. Recommended RF PLL Bandwidth vs. Operating Frequency

RF PLL Frequency (MHz)	Loop Filter Bandwidth (kHz)
75 to 3000	50 (default)
3000 to 6000	300

Note that the device firmware defaults to a 50 kHz loop filter bandwidth and must be changed using the provided API according to design requirements.

RF PLL LOOP FILTER CHANGE PROCEDURE

This section describes the procedure that must be used when a change to the RF PLL loop filter is required.

1. Move the device to the radio off state by executing the following command:

```
if   ((talError = TALISE_radioOff(&talDevice)) != TALACT_NO_ACTION)
    {
   /*** < Info: errorString will contain log error string in order to debug failure > ***/
   talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
   errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
    }
```

2. Change RF PLL loop filter settings. For example, set the loop bandwidth to 150 kHz with stability factor of 5 by executing the following commands:

```
if   ((talError = TALISE_setRfPllLoopFilter(&talDevice, 150, 5)) != TALACT_NO_ACTION)
    {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
    }
```

3. Follow RF PLL loop filter change with a `setRfPllFrequency` command at the current operating frequency:

```
if   ((talError = TALISE_setRfPllFrequency(&talDevice, TAL_RF_PLL , 2550000000)) !=
TALACT_NO_ACTION)
    {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
    }
   /*** < Action: wait 1ms for PLLs to lock > ***/
if  ((talError = TALISE_getPllsLockStatus(&talDevice, &pllLockStatus))  != TALACT_NO_ACTION)
  {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
  }
if ((*pllLockStatus &  0x0F) == 0x07)
  {
   /*** < Info: RF, CLK, and AUX PLL locked >     ***/
  }
else
```

```
{
                /*** < Info: RF, CLK, or AUX PLL not locked > ***/
  /*** < Action: Ensure lock before proceeding  - User code here> ***/
}
```

4. Move the device back to the radio on state by executing the following command:

```
if   ((talError = TALISE_radioOn(&talDevice)) != TALACT_NO_ACTION)
      {
/*** < Info: errorString will contain log error string in order to debug failure > ***/
talError = TALISE_getErrorCode(&talDevice, &talErrSrc, &talErrCode)
errorString = TALISE_getErrorMessage(*talErrSrc, *talErrCode)
      }
```

RF PLL RESOLUTION

The `TALISE_getRfPllFrequency()` and `TALISE_setRfPllFrequency()` commands have frequency parameters with a 1 Hz resolution. The real frequency that the RF PLL is tuned to can vary by a small amount, depending on the frequency of operation. The actual frequency steps available that the RF PLL can be tuned to are limited by the fractional word, modulus, and reference clock frequency. A modulus of 8,386,560 is used to be an exact frequency on at least a 5 kHz raster using the reference clocks that are shown in Table 45. Table 45 outlines the RF PLL frequency step variations vs. the RF operating band. Note that the upper limit is noninclusive; if the operating frequency is at the limit, use the next step size where the limit is lower.

The following examples shown in the Example 1 section through the Example 4 section show how to use Table 45 to determine the correct LO frequency setting.

Table 44. Divide by 2 Boundaries vs. Desired RF Operating Frequency

Limit	Divide by 32 (MHz)	Divide by 16 (MHz)	Divide by 8 (MHz)	Divide by 4 (MHz)	Divide by 2 (MHz)
Lower Limit	187.5	375	750	1500	3000
Upper Limit	375	750	1500	3000	6000

Table 45. LO Step Sizes vs. Desired RF Operating Frequency

Desired RF Operating Frequency Range, Lower Limit to Upper Limit (Hz)	Reference Clock Frequency (MHz), REF_CLK_IN±	PLL Sample Rate	Exact Decimal Frequency Raster (Hz)	LO Step Size (Hz)
75 to 93.75	307.2 or 153.6	76.8	625	0.07154304
	245.76 or 122.88	61.44	500	0.057234432
	184.32 or 92.16	46.08	125	0.042925824
93.75 to 187.5	307.2 or 153.6	76.8	625	0.143086081
	245.76 or 122.88	61.44	500	0.114468864
	184.32 or 92.16	46.08	125	0.085851648
187.5 to 375	307.2 or 153.6	76.8	625	0.286172161
	245.76 or 122.88	61.44	500	0.228937729
	184.32 or 92.16	46.08	125	0.171703297
375 to 750	307.2 or 153.6	76.8	625	0.572344322
	245.76 or 122.88	61.44	500	0.457875458
	184.32 or 92.16	46.08	125	0.343406593
750 to 1500	307.2 or 153.6	76.8	625	1.144688645
	245.76 or 122.88	61.44	500	0.915750916
	184.32 or 92.16	46.08	125	0.686813187
1500 to 3000	307.2 or 153.6	76.8	625	2.289377289
	245.76 or 122.88	61.44	500	1.831501832
	184.32 or 92.16	46.08	125	1.373626374
3000 to 6000	307.2 or 153.6	76.8	625	4.578754579
	245.76 or 122.88	61.44	500	3.663003663
	184.32 or 92.16	46.08	125	2.747252747

Example 1

In this example, the REF_CLK_IN± input reference frequency is 184.32 MHz, and the user wishes to tune the LO to a frequency equal to 3,600,000,002 Hz. For this example, the LO step size for this range is 2.747252747 Hz. The count number for the code required to obtain this frequency is as follows:

Count = 3,600,000,002 Hz ÷ 2.747252747 Hz = 1,310,400,000.849 Hz

Round this quotient to 1,310,400,001 Hz.

Actual LO Frequency = (1,310,400,001 Hz) × (2.747252747 Hz) = 3,600,000,000.416 Hz

Example 2

In this example, the REF_CLK_IN± input is 245.76 MHz, and the user wishes tune the LO to a frequency equal to 3,200,000,005 Hz. For this example, the LO step size for this range is 3.663003663 Hz. The count number for the code required to obtain this frequency is as follows:

Count = 3,200,000,005 Hz ÷ 3.663003663 Hz = 873,600,001.366 Hz

Round this quotient to 873,600,001 Hz.

Actual LO Frequency = (873,600,001 Hz) × (3.6630003663 Hz) = 3,200,000,003.6598 Hz

Example 3

In this example, the REF_CLK_IN± input is 245.76 MHz, and the user wishes to tune the LO to a frequency equal to 400,000,001 Hz. For this example, the LO step size for this range is 0.457875458 Hz. The count number for the code required to obtain this frequency is as follows:

Count = 400,000,001 Hz ÷ 0.457875458 Hz = 873,600,001,946 Hz

Round this quotient to 873,600,002 Hz.

Actual LO Frequency = (873,600,002 Hz) × (0.457875458 Hz) = 400,000,001.0246 Hz

Example 4

In this example, for any of the reference clocks listed in Table 45, the user wishes to tune the LO to a frequency of 1921.6 MHz. Because the desired LO frequency divided by the step size results in an integer, the actual LO frequency is exactly 1921.6 MHz.

RF PLL LOCK STATUS

The lock status of the clock PLL, RF PLL, and auxillary PLL is provided through an API command. Additionally, the PLL lock status can be set to asset via the general-purpose interrupt pin (GP_INTERRUPT).

RF PLL Lock API Functions

TALISE_getPllsLockStatus()

This function returns the status of the PLLs via the `pllLockStatus` pointer. The three LSBs of the `uint8_t` value at the `pllLockStatus` represent the lock status of the clock PLL, RF PLL, and auxiliary PLL. Bit 0 is the clock PLL lock status, Bit 1 is the RF PLL lock status, and Bit 2 is the auxiliary PLL lock status. A bit value of 1 indicates that the corresponding PLL is locked. A bit value of 0 indicates that the corresponding PLL is unlocked. The function is as follows:

```
TALISE_getPllsLockStatus(taliseDevice_t* device, uint8_t* pllLockStatus)
```

Precondition: this function can be called any time after the PLLs have been configured and are operational.

Parameters include the following:

- `device` is a pointer to the device data structure.
- `pllLockstatus` is the PLL lock status byte pointer to return the bitwise representation of the PLL lock status.

RF PLL Lock Status, General-Purpose Interrupt

Another outlet for the lock status of the PLLs is the GP_INTERRUPT pin. The interrupt mask enumeration and the GP_INTERRUPT API functions related to the GP_INTERRUPT can be found in the General-Purpose Interrupt Operation section.

CONNECTIONS FOR EXTERNAL LO

The RF_EXT_LO_I/O± pins can work in two modes. As an output, these pins provide access to the signal generated by the internal RF LO. As an input, these pins allow the user to provide an external LO signal into the device.

RF_EXT_LO_I/O± as an Input

Unlike the internal synthesizers that always operate from 6 GHz to 12 GHz regardless of the RF tune frequency, when an external LO is used, the frequency applied must be 2× the desired RF tune frequency.

AC couple the differential signal applied to the RF_EXT_LO_I/O± inputs as shown in Figure 62. The RF_EXT_LO_I/O± input represents small signal input impedance (Z_{IN}) near 50 Ω differential in parallel with 1.6 pF. The ac coupling capacitor must have a much smaller impedance at the external VCO frequency than Z_{IN}. Take the loss in an on-board balun into account when calculating the input power to the RF_EXT_LO_I/O± input.

Take care when selecting an on-board balun for this application. The combination of amplitude and phase balance performance of the balun can affect quardrature error performance. Additionally, duty cycle and differential second-order harmonic distortion impacts the ability of the device to correct a quadrature error. The recommended minimum requirement is a combination of no more than 5° of differential phase error, 1 dB differential amplitude error, 2% duty-cycle error, and less than −50 dBc even-order harmonics (primarily second-order). Refer to the ADRV9008-1, ADRV9008-2, and ADRV9009 data sheets for specifications.

Set the external LO source modulus to the same value as the internal LO source modulus. The modulus must be equal to 8,386,560. Additionally, any external LO source must be phase-locked to the device clock.

Note that in the GSM receiver application, a low noise external LO source is required to meet the system phase noise requirement at 800 kHz.

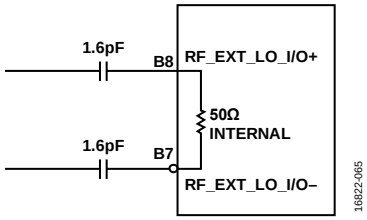

Figure 62. RF_EXT_LO_I/O± as an Input, External Components

A higher frequency external LO requires a higher input power. Generally, a higher input power (P_{IN}) results in improved phase noise. Use the minimum input power that results in phase noise that meets requirements (with some margin).

Enabling the External LO as Input Using the API

This section describes how to enable the external LO as an input.

Prior to initialization, set `rfPllUseExternalLo` in the `taliseInit_t` data structure.

```
taliseDigClocks =
    {
        deviceClock_kHz     = 245760,                /* CLKPLL and device reference clock
frequency in kHz */
        clkPllVcoFreq_kHz   = 9830400,                /* CLKPLL VCO frequency in kHz */
        clkPllHsDiv         = TAL_HSDIV_2,           /* CLKPLL high speed clock divider */
        rfPllUseExternalLo = 1,                      /*  1= Use external LO for RF PLL, 0 = use
internal LO generation for RF PLL */
        rfPllPhaseSyncMode = TAL_RFPLLMCS_NOSYNC     /* RFPLL MCS (Phase sync) mode */
    },
```

RF_EXT_LO_I/O± as an Output

The external LO output is only provided for testing purposes and is not approved for daisy-chain operation. This section provides configuration details. Internal synthesizers always operate from 6 GHz to 12 GHz. The user can observe the internal LO frequency on the RF_EXT_LO_I/O± pins. Users can gain access to the internal synthesizer VCO frequency (f_{VCO}) output via the RF_EXT_LO_I/O± pins. The output frequency on the RF_EXT_LO_I/O± pins ranges from f_{VCO} divided by 2 to f_{VCO} divided by 64.

For example, when the LO operates at 4 GHz, the internal VCO operates at 8 GHz (f_{VCO}). When outputting an internal VCO signal divided by 2, the user can observe 4 GHz ($f_{VCO}/2$) at the RF_EXT_LO_I/O± pins.

The hardware configuration for the RF_EXT_LO_I/O± pins operating as outputs is shown in Figure 63. To operate the RF_EXT_LO_I/O± pins as outputs, differential lines of 50 Ω, an ac-coupled capacitor with an impedance at the desired RF_EXT_LO_I/O± output frequency of less than 25 Ω (for example, 300 pF at 1 GHz (0.5 Ω)), and a differential load of 50 Ω is required.

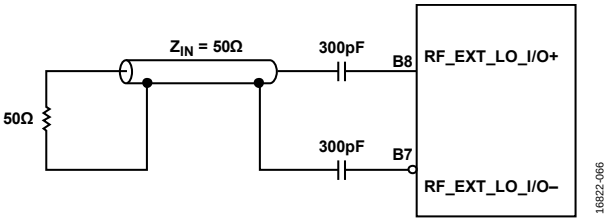

Figure 63. RF_EXT_LO_I/O± as an Output, External Components

RF PLL PHASE SYNCHRONIZATION

The RF PLL phase synchronization function allows the internally generated LO to be phase synchronized and aligned to the applied reference clock. In multidevice systems, this function allows all devices to align the RF PLL to the same point. Therefore, the phase between each device is aligned at startup so that phasing between devices is fixed and can be repeated. At startup, the standard JESD204B multichip synchronization mechanism, implemented with the device clock (REF_CLK_IN±) and system reference signals (SYSREF), resets the data converter clocks and all other clocks at the baseband rate. The REF_CLK_IN± and SYSREF signals are also used to initialize the on-chip counter that is used later during PLL programming to synchronize the LO phase. No additional signals are required to take advantage of the LO phase synchronization mechanism. A digital representation of the desired LO phase can be computed at each PLL reference clock edge from the on-chip counter and a PLL fractional word programming. This digital representation is remembered in the digital phase accumulator (DPA).

The LO phase synchronization hardware operates by directly sampling the LO signal (in quadrature) using the PLL reference clock signal (REF_CLK_IN±). Averaging is required to increase the accuracy of the LO phase measurement. Therefore, at every sample, the observed LO phase is derotated by the digitally desired phase by performing a vector multiplication of the complex conjugate of the digital phase. The result of these operations is a vector representing the phase difference between the LO and the digitally desired phase, and these vectors can be averaged over many REF_CLK_IN± cycles to obtain an accurate measurement of the phase adjustment required.

After the phase difference is measured, the adjustment can be applied into the first stage Σ-Δ modulator of the PLL by adding the adjustment to the first stage modulator input. The total adjustment amount is added over many reference clock cycles to stay within the PLL loop bandwidth and to not cause the PLL to come unlocked. To counteract temperature effects after calibration, a PLL phase tracking mode can be activated.

Figure 64 shows a block diagram of the phase synchronization system.

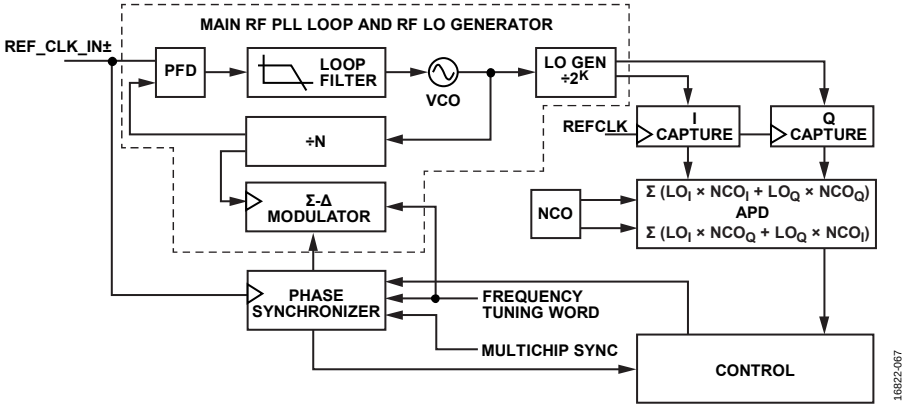

Figure 64. LO Phase Synchronization Functional Diagram

System Level Considerations

The overall phase synchronization is determined by a number of factors, including the board level clock routing (t_{CLK}), the on-chip reference path routing ($t_{REFPATH}$), the PLL and LO divider path (t_{PLL}), and the RF and antenna paths (t_{RF}). In a beamforming/multiple input, multiple output (MIMO) system, a system level antenna calibration is performed to equalize the sum of these paths between all channels (see Figure 65). Additionally, the following supply pins are highly sensitive and can affect the phase synchronization performance, especially during chip mode changes, for example, during TDD operation. Care must be taken to isolate the VDDA1P3_RF_SYNTH, VDDA1P3_CLOCK_SYNTH, and VDDA1P3_RF_LO pins from the VDDA_1P3_ANALOG supply. This isolation is critical to achieving best phase synchronization performance.

The goals of this transceiver mechanism are to reduce the complexity of the antenna calibration by initializing to a more consistent startup condition with a deterministic PLL phase and LO divider state, to reduce the temperature dependence of the system phase synchronization to allow the antenna calibration to run less frequently during operation, and to allow transceivers to be stopped and started in an operational system and hot synchronize with the other transceiver elements.

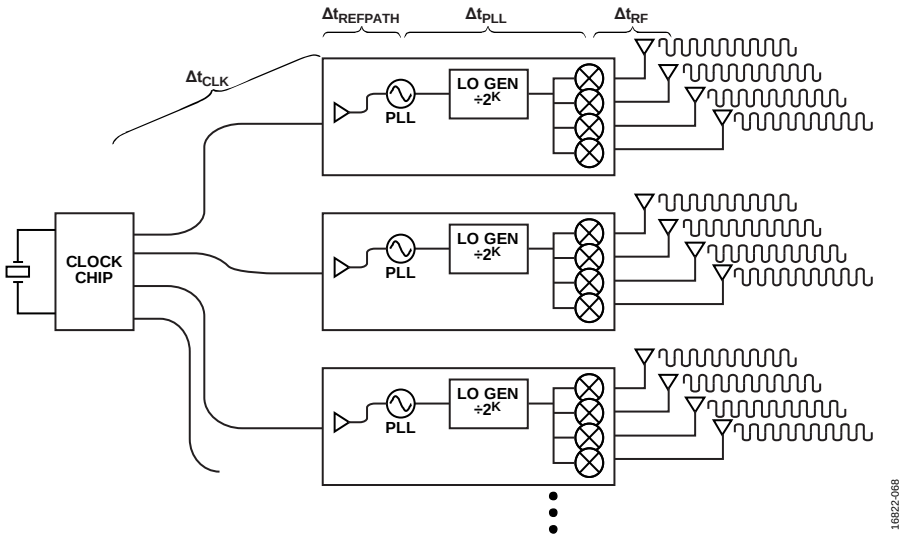

Figure 65. High Level Contributors to System Phase per Antenna

The LO phase synchronization method addresses the initial PLL phase and LO divider state, and reduces the temperature dependence of the PLL phase and LO divider state to a negligible amount in comparison to other sources of phase drift in the system.

Enabling the LO Phase Sync Function Using the API

To enable the LO phase synchronization function, take the following steps:

1. Set the phase sync bit in the `taliseInit_t` data structure.
2. Perform a multichip synchronization to set JESD204B deterministic latency using SYSREF signal pulses as normal. The LO phase synchronization uses existing signaling and the SYSREF signal to accomplish LO phase synchronization.

```
.clocks =
    {
        .deviceClock_kHz    = 245760,                   /* CLKPLL and device reference clock
frequency in kHz */
        .clkPllVcoFreq_kHz  = 9830400,             /* CLKPLL VCO frequency in kHz */
        .clkPllHsDiv        = TAL_HSDIV_2,              /* CLKPLL high speed clock divider
*/
        .rfPllUseExternalLo = 0,                        /*  1= Use external LO for RF PLL, 0
= use internal LO generation for RF PLL */
        .rfPllPhaseSyncMode = TAL_RFPLLMCS_INIT_AND_SYNC       /* RFPLL MCS (Phase sync)
mode . See enum values below.
    },
```

Possible enumerator values are:

```
/**
* \brief Enumerated list of RFPLL phase synchronization modes
*
* RFPLL Phase sync requires extra time to sync each time the RFPLL frequency
* is changed. If RFPLL phase sync is not required, it may be desired to
* disable the feature to allow the RFPLL to lock faster.
*
* Depending on the desired accuracy of the RFPLL phase sync, several options
* are provided.
*/
typedef enum
{
    TAL_RFPLLMCS_NOSYNC = 0,                          /*!< Disable RFPLL phase synchronization */
    TAL_RFPLLMCS_INIT_AND_SYNC = 1,                   /*!< Enable RFPLL phase sync init only */
    TAL_RFPLLMCS_INIT_AND_1TRACK = 2,           /*!< Enable RFPLL phase sync init and track
once  */
    TAL_RFPLLMCS_INIT_AND_CONTTRACK = 3        /*!< Enable RFPLL phase sync init and track
continuously */
} taliseRfPllMcs_t;
```

RF PLL Phase Synchronization Demo Setup

A vector network analyzer is used to measure the phase difference between the evaluation board output and a reference source, which is phase-locked to the device clock for the RF PLL on the device. Figure 66 shows the test setup. It is important to use the same reference for all the equipment in the setup. All the equipment shown in Figure 66 is locked to the same 10 MHz reference. Figure 66 applies to the ADRV9009 and the ADRV9008-2. For the ADRV9008-1, use RF_EXT_LO_I/O± as the input to the vector network analyzer.

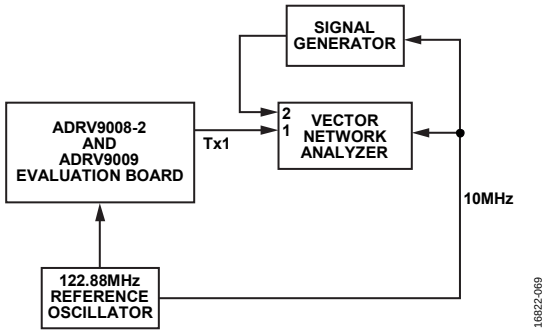

Figure 66. RF PLL Phase Synchronization Test Setup

The GUI provides two options to set the RF PLL phase synchronization: **Disable** or **Init & Track Continuously**, as shown in Figure 67.

Figure 67 shows the both phase synchronization modes in the dropdown list in the **RFPLL Phase Sync** pane. Set this control to **Disable** mode to baseline the design before the function is enabled.

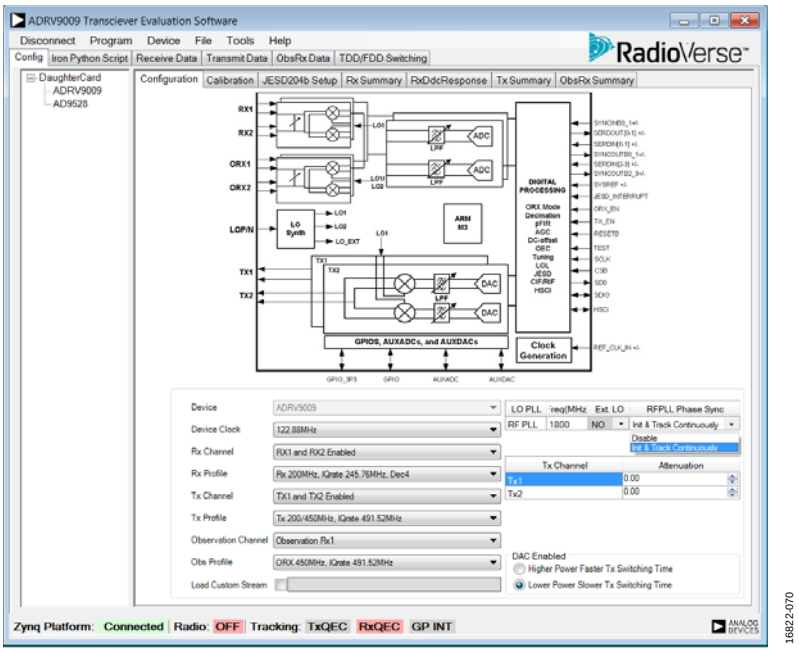

*Figure 67. GUI Setup Screen, **RFPLL Phase Sync** Set to **Disable***

Figure 68 shows five power cycles of the same device with the phase synchronization function beginning in the disabled state. At each power-up, the phase of the transmitter output compared to the signal generator reference is a random value on each of the five power cycles. Figure 68 also shows initialization and tracking results, which brings the intial random phase to a repeatable value.

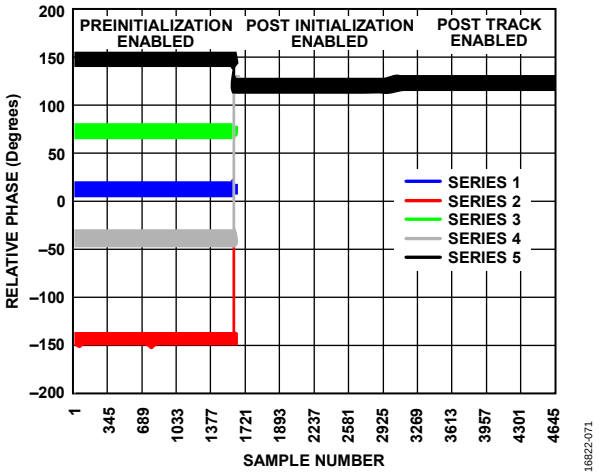

Figure 68. Random RF PLL Phases Compared to Laboratory Signal Generator Reference—RF PLL Phase Synchronization Transitions from Disabled Through Inititialization and into Tracking Mode, Five Independent Power-Up Sequences

Next, enable the phase sync function. In the GUI, select the **Init & Track Continuously** from the dropdown list. When the device is programmed, this function is enabled and uses the existing SYSREF signals for JESD204B bus alignment to trigger the RF PLL phase sync at each power-up cycle.

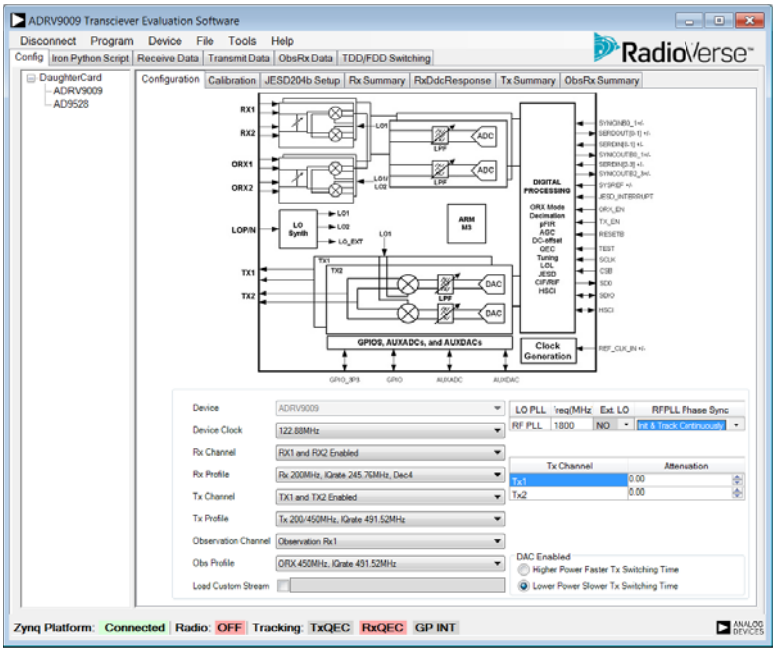

*Figure 69. GUI Setup Window, **RFPLL Phase Sync** Set to **Init & Track Continuously***

Figure 70 shows the phase synchronization transitioning from initialization to tracking on the same device over five power-up cycles.

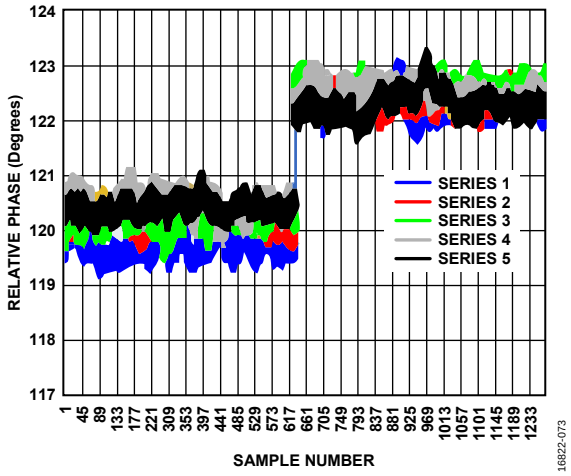

Figure 70. RF PLL Phase Synchronization—Initialization to Tracking Results

RF PLL Phase Synchronization Demo Setup with 2 Two Evaluation Platforms

A vector network analyzer is used to measure the phase difference between the evaluation board Transmitter 1 output and another evaluation board Transmitter 1 output. The vector network analyzer is phase-locked to the reference clock for the RF PLL on the device. Figure 71 shows the test setup. It is important to use the same reference for all the equipment in the setup. In Figure 71, all equipment is locked to the same 10 MHz reference, and the RF tune frequency is 1800 MHz.

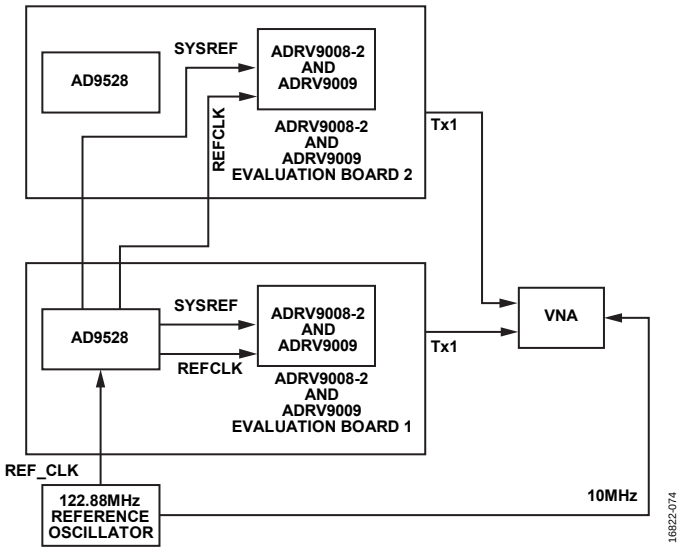

Figure 71. RF PLL Phase Sync Test Setup

In this test, five power cycles show the random initial phase difference between the presynchronized evaluation boards. Figure 72 shows the transition from preinitialization through initialization and tracking. It is observed that the phase difference reduces to almost zero after initialization and tracking is enabled.

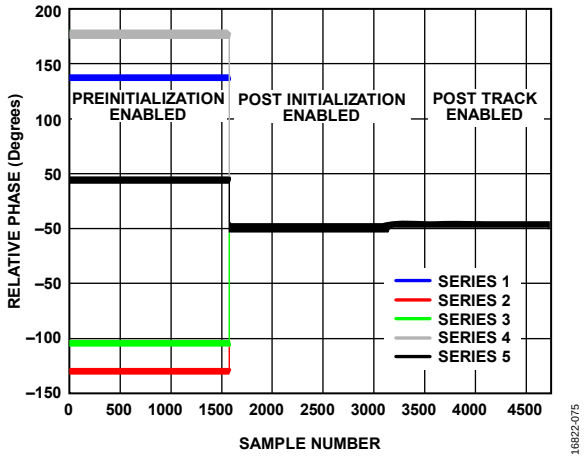

Figure 72. Transmitter Output Phase Comparison through RF PLL Phase Synchronization Cycle (RF Tune Frequency = 1800 MHz)

Figure 73 shows the transistion from initialization to tracking. The tracking maintains approximately 2.5 to 3.0 degrees of phase delay between the Transmitter 1 output of each chip.

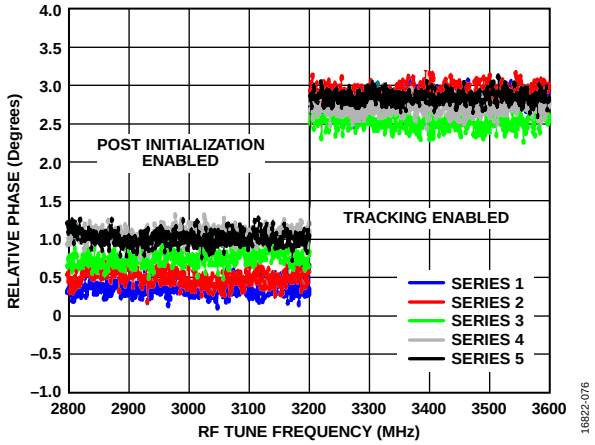

Figure 73. RF PLL Phase Synchronization Intialization to Tracking Transition (RF Tune Frequency = 1800 MHz)

Over temperature, continuous tracking mode is capable of maintaining phase of the LO (see Table 46).

Table 46. LO Sync Phase Difference Over Temperature

LO Frequency (MHz)	Typical Time Delay Change (sec/°C)	Typical Phase Change (Degrees of Phase Delay/°C)
450	1.61×10^{-12}	0.27
900	1.58×10^{-12}	0.54
1800	1.56×10^{-12}	1.09
3600	1.56×10^{-12}	2.08

Figure 74 shows the LO phase difference in comparison to a signal generator reference vs. die temperature. The data shown in Figure 74 has been normalized to the value obtained at 20°C.

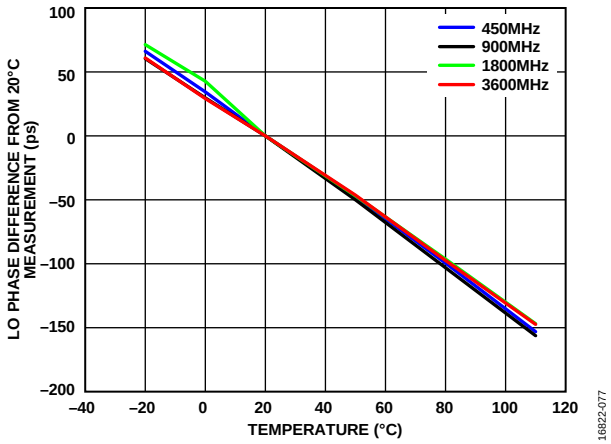

Figure 74. LO Phase Difference Compared to Signal Generator Reference vs. Temperature, from 20°C Measurement

RF PLL FREQUENCY HOPPING

In the device, the RF PLL can change frequency quickly to another predetermined LO frequency set by the user. This section describes how to set up the device to achieve quick switching of the LO frequency (LO hopping). Note that using this feature can cause tradeoffs in RF performance in certain cases. Minimum hopping time can be profile dependent, but is generally limited to approximately 60 µs.

There are two use cases through which a user can control frequency hopping:

- GPIO mode. In this mode, the user can switch the RF PLL frequency via a user assigned GPIO pin. The user must configure the frequency hopping GPIO during the initialization time. The next hopping frequency must be set via an API command (see the RF PLL Frequency Hopping, Software Configuration section).
- Non-GPIO mode (API mode). In this mode, the user can switch the RF PLL frequency via an API command during run time (see the RF PLL Frequency Hopping, Software Configuration section).

To control frequency hopping in GPIO mode, assign a GPIO pin to the `fhmGpioPin` variable in the `taliseFhmConfig_t` structure to function as the trigger to change the frequency. To operate in non-GPIO mode, set the `fhmGpioPin` variable to `TAL_GPIO_INVALID`. Set the bounds of the frequency hopping in the same structure such that the LO cannot hop to a frequency outside the band of operation. The API command `TALISE_setFhmConfig()` configures the device settings defined in the `taliseFhmConfig_t`. The API command `TALISE_setFhmConfig()` can only be called when the device is in the radio off state.

After setting up the `taliseFhmConfig_t` structure, continue by setting up the `taliseFhmMode_t` structure to enable/disable the RF PLL frequency hopping feature. The user can choose to enable multichip synchronization when changing LO frequency, operate in GPIO mode or non-GPIO mode (API Mode), as well as choose the first frequency to hop to when fast hopping mode is enabled. The user can also select if the previous loop filter bandwidth is restored or if the fast hopping mode loop bandwidth of 600 kHz is to be maintained when exiting fast hopping mode. The API command `TALISE_setFhmMode()` can be called when the device is in the radio on state.

After the data structure setup and the associated API commands are run, the RF PLL frequency hopping mode is in enabled. If operating the device in GPIO mode, use the `Talise_setFhmHop()`function to transfer the frequency that the device hops to next, and then a low to high transition of the selected GPIO pin causes the device to change the LO frequency to the one specified by the user. If operating in non-GPIO mode, the device changes the LO frequency to the frequency specified in the `Talise_setFhmHop()` function when the API command runs. The `Talise_setFhmHop()` function can only be called when the device is in the radio on state.

RF PLL Frequency Hopping, Software Configuration

This section describes the software configuration for the RF PLL frequency hopping function.

API Data Structures for RF PLL Frequency Hopping

Table 47. taliseFhmConfig_t Data Structure Descriptions

Member Name	Data Type	Description	Value Range
fhmGpioPin	taliseGpioPinSel_t	GPIO input pin to the device to trigger frequency hopping	GPIO_0 to GPIO_15. To unassign a GPIO pin as a frequency hopping trigger, set this pin to TAL_GPIO_INVALID
fhmMinFreq_Hz	uint64_t	Minimum scan frequency for frequency hopping	Dependent on the RF design of customer board
fhmMaxFreq_Hz	uint64_t	Maximum scan frequency for frequency hopping	Dependent on the RF design of customer board

Table 48. taliseFhmMode_t Data Structure Description

Member Name	Data Type	Description	Value Range
fhmEnable	uint8_t	Signal to enable/disable frequency hopping.	0: disable frequency hopping mode (FHM). 1: enable FHM
enableMcsSync	uint8_t	Signal to enable multichip synchronization for frequency hopping.	0: disable multichip synchronization. 1: enable multichip synchronization Ignored if fhmEnable is set to 0. For frequency hopping, multichip synchronization calibration parameters are modified to speed up calibration but are restored on exiting FHM mode.

Member Name	Data Type	Description	Value Range
fhmTriggerMode	taliseFhmTriggerMode_t	This member configures the mode of the frequency hopping trigger to either GPIO mode (for timing critical operations) or non-GPIO mode (Arm command through SPI interface to switch frequency).	TAL_FHM_GPIO_MODE, TAL_FHM_NON_GPIO_MODE Ignored if fhmEnable is set to 0.
fhmExitMode	taliseFhmModeExit_t	This member sets the frequency hopping mode of exit to either full exit cycle or quick exit cycle. Note that this member is only valid if fhmEnable member in this structure is set to 0 (disable FHM). If fhmEnable is set to 1 (enable FHM), fhmExitMode is ignored.	TAL_FHM_QUICK_EXIT, TAL_FHM_FULL_EXIT. In full exit cycle, RF PLL loop bandwidth is restored to narrow-band, the RF and auxiliary PLLs are recalibrated, and tracking calibrations are resumed. In quick exit cycle, RF PLL loop bandwidth is unchanged. This is ignored if fhmEnable is set to 1. The PLL loop bandwidth s increased to 600 kHz for FHM.
fhmInitFrequency_Hz	uint64_t	First hop frequency on enabling Frequency Hopping	First frequency to hop to when FHM is enabled. This is ignored if fhmEnable is set to 0.

Table 49. taliseFhmStatus_t Data Structure Description

Member Variable	Data Type	Description
currentFhmCmdErrorStatus	uint16_t	Current FHM error status when entering FHM mode
currentFhmHopErrorStatus	uint16_t	Current FHM active errors when hopping
numFhmHops	uint32_t	Total number of frequency changes that occurred since entering FHM
numFhmNoErrorEvents	uint32_t	Total number of no FHM error events
lastFhmNoErrorFreq_Hz	uint64_t	Last frequency for which no error was encountered
numFhmHopsOutsideScanRange	uint32_t	Total number of hops outside FHM scan range
lastFreqOutsideScanRange_Hz	uint64_t	Last frequency outside FHM scan range
numInvalidFhmHopFrequencies	uint32_t	Number of times that an invalid hop frequency occurred
lastInvalidHopFreq_Hz	uint64_t	Last invalid hop frequency
compPllError	uint32_t	Number of times a PLL LO computation error occurred
compPllErrorFreq_Hz	uint64_t	PLL LO computation error frequency
rfPllLockFailed	uint32_t	Number of times RF PLL lock failed
rfPllLockFailedFreq_Hz	uint64_t	Last frequency for which RF PLL lock failed

API Functions for RF PLL Frequency Hopping

This section provides detailed information about the API functions used to set up, enable, and read back status information for RF PLL frequency hopping.

TALISE_setFhmConfig()

This command sets up the frequency hopping trigger GPIO pin and the frequency hopping range. The function is as follows:

`TALISE_setFhmConfig(taliseDevice_t *device, taliseFhmConfig_t *fhmConfig)`

Precondition: this function can be called to set up the `taliseFhmstructure` configuration structure before the Arm processor is initialized. After initialization, this function can only be called when the device is in the radio off state.

Parameters include the following:

- `device` is a pointer to the data structure that contains settings.
- `taliseFhmConfig` is a configuration structure of the frequency hopping mode.

TALISE_getFhmConfig()

This function retrieves the current trigger GPIO pin and frequency hopping range. The function is as follows:

`TALISE_getFhmConfig(taliseDevice_t *device, taliseFhmConfig_t *fhmConfig)`

Precondition: this function can be called to set up the `taliseFhmstructure` configuration structure before the Arm processor is initialized. After initialization, this function can be called when the device is in the radio off and radio on states.

Parameters include the following:

- `device` is a pointer to the data structure that contains settings.
- `taliseFhmConfig` is a configuration structure of the frequency hopping mode.

TALISE_setFhmMode()

This function enables/disables frequency hopping, enables/disables multichip synchronization, and configures exit mode. This function also configures the initial hopping frequency. The function is as follows:

`TALISE_setFhmMode(taliseDevice_t *device, taliseFhmMode_t *fhmMode)`

If `fhmEnable` is set to 1 (enable frequency hopping), the following sequence executes:

1. FHM trigger mode (GPIO/non-GPIO) is configured.
2. FHM initialization frequency is written to the Arm mailbox.
3. A command is sent to the Arm to enable FHM mode (with `fhmMode.enableMcsSync` flag).

Note that the `fhmMode.fhmExitMode` is ignored if `fhmEnable` is set to 1.

If `fhmMode.fhmEnable` is set to 0, the Arm processor is commanded to disable frequency hopping with the exit mode selected by the user via the `fhmMode.fhmExitMode` command. Parameters other than `fhmExitMode` are ignored if `fhmEnable` is set to 0.

Precondition: this function can only be called after the Arm processor is initialized. After initialization, this function can be called when the device is in the radio off and radio on states.

Parameters include the following:

- `device` is a pointer to the device data structure that contains settings.
- `fhmMode` is a mode structure of frequency hopping mode.

TALISE_getFhmMode()

The function retrieves the current frequency hopping enable state, multichip synchronization configuration, and the exit mode configuration. The function is as follows:

`TALISE_getFhmMode(taliseDevice_t *device, taliseFhmMode_t *fhmMode)`

Precondition: this function can only be called after the Arm processor is initialized. After initialization, this function can be called when the device is in the radio off and radio on states.

Parameters include the following:

- `device` is a pointer to the device data structure that contains settings.
- `fhmMode` is a pointer to the mode structure of frequency hopping mode.

TALISE_setFhmHop()

This function triggers frequency hopping via an Arm command instead of a GPIO pulse.

`TALISE_setFhmHop(taliseDevice_t *device, uint64_t nextRfPllFrequency_Hz)`

Precondition: this function can only be called after the Arm processor is initialized. After initialization, this function can be called when the device is in the radio off and radio on states.

Parameters include the following:

- `device` is a pointer to the device data structure that contains settings.
- `nextRfPllFrequency_Hz` is the next frequency that the user wishes to hop to in Hz.

TALISE_getFhmStatus()

This function returns the current frequency hopping status.

`TALISE_getFhmStatus(taliseDevice_t *device, taliseFhmSts_t *fhmSts`

Precondition: this function can only be called after the Arm processor is initialized. After initialization, this function can be called when the device is in the radio off and radio on states.

Parameters include the following:

- `device` is a pointer to the device data structure that contains settings.
- `fhmSts` is a pointer to the status structure of frequency hopping mode.

RECEIVER GAIN CONTROL

The ADRV9008-1 and ADRV9009 (Receiver 1 and Receiver 2) feature automatic and manual gain control modes that allow flexible gain control in a wide array of applications. Automatic gain control (AGC) allows receivers to autonomously adjust the receiver gain depending on variations of the input signal, for example, the onset of a strong interferer that can overload the receiver datapath. AGC mode controls the gain of the device based on information from a number of signal detectors (peak/power detectors). The AGC can control the gain of the device with fine resolution, if required. The receivers are also capable of operating in manual gain control (MGC) mode, where changes in gain are initiated by the BBP. The gain control blocks of the device are configured by the API data structures and several API functions exist that allow user interaction with the gain control mechanisms.

The AGC is highly flexible and can be configured in a number of ways. For base transceiver station (BTS) receivers, the received signal is a multicarrier signal in most cases. Only perform a gain change under large overrange or underrange conditions. Gain changes do not occur often for typical 3G/4G operation, and so, peak detect mode operation is sufficient. If an asynchronous blocker does appear, a fast attack mode exists that can reduce the gain at a fast rate.

Alternatively, for support for GSM blockers and radar pulses that have fast rise and rapid fall times, a fast attack, fast recovery, peak detect only mode exists. This mode can recover quickly in addition to fast attack mode.

RECEIVER DATAPATH

Figure 75 shows the receiver datapath and gain control blocks of the device. The receiver has front-end attenuators prior to the mixer stage that are used to attenuate the signal in the RF domain to ensure the signal does not overload the receiver chain. In the digital domain, there are two options: digital attenuation or digital gain. This digital gain block is also utilized for gain compensation.

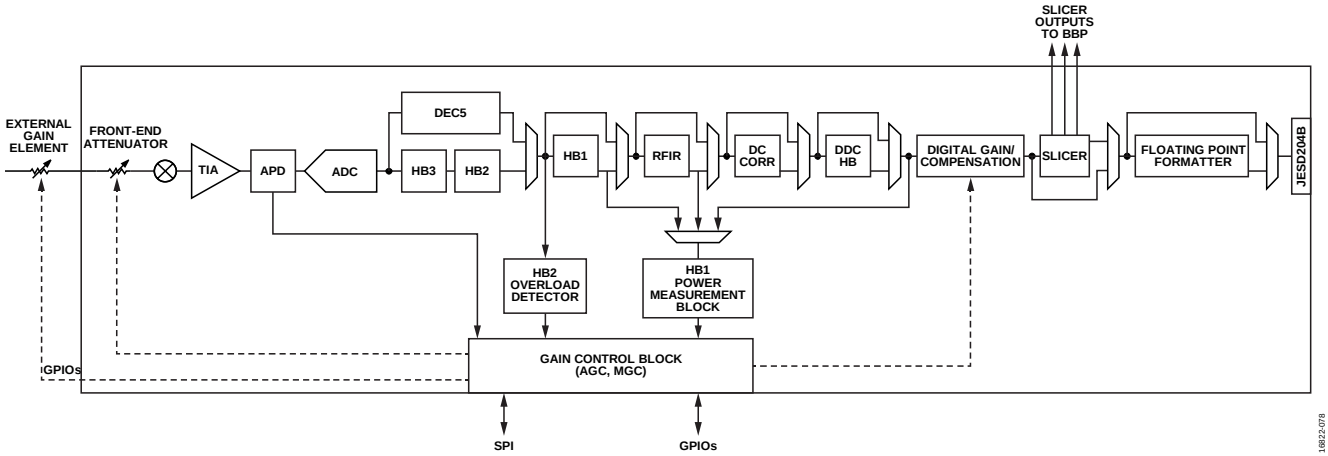

Figure 75. Receiver Datapath and Gain Control Blocks

The receiver chain has a number of observation elements that can monitor the incoming signal. These observation elements can be used in MGC or AGC mode. An analog peak detector (APD) exists prior to the ADC. Because the APD is in the analog baseband, this peak detector sees signals first and also has visibility of the blocker signals that can overload the ADC but can be filtered as the blocker signals progress through the digital chain. The second peak detector is the Half-Band 2 (HB2) overload detector, defined as such because this detector monitors the data at the output of the HB2 filter in the receiver chain.

A power measurement detection block is also provided in the receiver chain, which takes the rms power of the received signal over a configurable period of time. The power measurement detection block can observe power at one of a number of configurable locations.

The device can also control an external gain element through the use of the receiver gain table and the GPIOs.

The gain control block is shown in Figure 75 with multiple inputs that provide information. The gain control block controls the gain of the signal chain using a gain table.

This gain table is user programmable, and each row of the table provides a unique combination of a front-end attenuator, external gain element (if used), and digital gain settings. The gain control block updates the variable gain elements based on the row of this table selected either by the user in MGC mode, or automatically by the device when in AGC mode. The user can control the gain control block using the SPI bus (configuration of AGC or MGC) and the GPIOs.

Table 50. Sample Rows from the Default Receiver Gain Table

Gain Table Index	Front-End Attenuator Bits[7:0]	External Gain Control Bits[3:0]	Transimpedance Amplifier (TIA) /ADC Gain	Signed Digital Gain/Attenuation Bits[10:0]	Phase Offset
255	0	0	0	14	0
254	14	0	0	14	0
253	27	0	0	15	0

The gain table index is the reference for each unique combination of the gain settings in the programmable gain table. It is possible to have different gain tables for Receiver 1 and Receiver 2, although typically the same gain table is used. The possible range of the gain table is 255 to 128. However, typically, only a subset of this gain range is used. The gain table must be assigned in order of decreasing gain, starting with the highest gain in the maximum gain index (for example, 255) to the lowest gain in the minimum gain index. The loading of the gain table and the specification of the maximum/minimum gain indices is described in the Gain Control API Programming section.

The front-end attenuator has an 8-bit control word. The amount of attenuation applied depends on the value set in the front-end attenuator column of the selected gain table index.

The following equation demonstrates a relationship between the internal attenuator and the front-end attenuation value (N) programmed in the gain table:

$$Attenuation\ (\text{dB}) = 20\log_{10}\left(\frac{256 - N}{256}\right)$$

The external gain control column controls four 3.3 V GPIOs for each receiver. Receiver 1 uses GPIO_3P3_3 to GPIO_3P3_0, where Receiver 2 uses GPIO_3P3_7 to GPIO_3P3_4. These 3.3 V GPIOs must be enabled as outputs and set for external gain functionality. The programmed 4-bit value is directly related to the status of these GPIO pins, for example, if the Receiver 1 table is programmed to 3 decimal (0011b) in the selected gain index, GPIO_3P3_0 and GPIO_3P3_1 are high, and GPIO_3P3_2 and GPIO_3P3_3 are low.

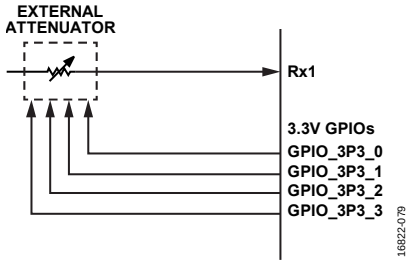

Figure 76. GPIO Control of an External Gain Element to Receiver 1

The signed digital gain/attenuation is used to digitally apply gain or attenuation. The range of the digital gain is 0 dB to 41.9 dB, and the range of the digital attenuation is 0 dB to 18.05 dB. The resolution of the steps is 0.05 dB in both directions. Therefore, a value of +14 provides a 0.7 dB gain, and a value of −14 provides 0.7 dB of attenuation.

The TIA/ADC gain is not used and must be set to zero in all rows.

The gain table loaded during the initialization of the device is stored in the **talise_user.c** file. This table can be modified by the user based on the required use case. The default table provided within **talise_user.c** file provides 0.5 dB gain steps over a 30 dB range. The default table does not support gain compensation, and the table must be updated for such a use case.

```
 * \brief Default Rx gain table settings
 *
 * Order: {FE table, External Ctl, TIA/ADC, signed Digital Gain/Atten[10:0], Phase offset}
 */
taliseRxGainTable_t rxGainTable [61] =
{
    {0, 0, 0, 14, 0},     /* Gain index 255 */
    {14, 0, 0, 14, 0},    /* Gain index 254 */
    {27, 0, 0, 15, 0},    /* Gain index 253 */
    {41, 0, 0, 14, 0},    /* Gain index 252 */
    {53, 0, 0, 13, 0},    /* Gain index 251 */
    {67, 0, 0, 14, 0},    /* Gain index 250 */
```

16822-080

*Figure 77. Sample of the Gain Table Contained in the **talise_user.c** File*

GAIN CONTROL MODES

The gain control mode is selected with the following API function:

```
TALISE_setRxGainControlMode(taliseDevice_t *device, taliseGainMode_t mode)
```

Parameter: `mode` is an enumerator that indicates what gain mode is to be used, as shown in Table 51.

Table 51. taliseGainMode_t Gain Control Modes

Enumerator Name	Gain Mode Description
TAL_MGC	Manual gain mode.
TAL_AGCFAST	Do not use. See TAL_AGCSLOW, which is used for all AGC operation.
TAL_AGCSLOW	Automatic gain control mode. This mode can be configured in fast and slow configurations.
TAL_HYBRID	Do not use. Not implemented in API.

MGC Mode

The gain control block applies the settings from the selected gain index in the gain table. In MGC mode, the BBP is in control of selecting the gain index. There are two options in MGC mode: API command mode or pin control mode. By default, if MGC is chosen, the device is configured for API commands.

In API command mode, use the following API to select a gain index in the gain table:

```
TALISE_setRxManualGain(taliseDevice_t *device, taliseRxChannels_t rxChannel, uint8_t gainIndex)
```

Parameters include the following:

- `rxChannel` is an enumerator that indicates which channel's gain is updated.
- `gainIndex` is the gain table index chosen for that particular receiver.

Use the following function to read back the gain index for a selected channel:

```
TALISE_getRxGain(taliseDevice_t *device, taliseRxChannels_t rxChannel, uint8_t *rxGainIndex)
```

Pin control mode within MGC mode is useful when real-time control of the gain is required. In this mode, 4 GPIO_x pins are used: two for each receiver, one to increase the gain table index, and one to decrease the gain table index. Specify the increment and decrement step size. A pulse is applied to the relevant GPIO_x pin to trigger an increment or decrement in gain, as shown in Figure 78. This pulse must be held high for at least two AGC clock cycles for a gain change to occur (see the AGC Clock and Gain Block Timing section for details).

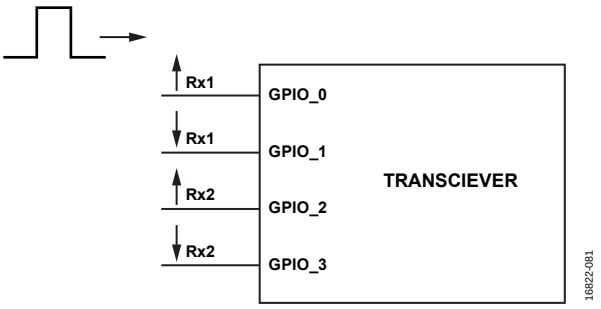

Figure 78. MGC Pin Mode Configured for Use with GPIO_0 to GPIO_3

The following function enables the pin control mode within MGC mode:

```
TALISE_setRxGainCtrlPin(taliseDevice_t *device, taliseRxChannels_t rxChannel,
taliseRxGainCtrlPin_t *rxGainCtrlPin)
```

Parameter: `rxGainCtrlPin` is a structure outlined in Table 52.

Table 52. taliseRxGainCtrlPin_t Data Structure Description

Member Name	Description
uint8_t incStep	Increment in gain index applied when the increment gain is pulsed. Acceptable values for this parameter are 0 to 7. However, a value of 1 is added to what is programmed into this parameter, resulting in step sizes of 1 to 8.
uint8_t decStep	Decrement in gain index applied when the decrement gain is pulsed. Acceptable values for this parameter are 0 to 7. However, a value of 1 is added to what is programmed into this parameter, resulting in step sizes of 1 to 8.
taliseGpioPinSel_t.rxGainIncPin	Acceptable values for Rx1 are GPIO_0 or GPIO_10 and acceptable values for Rx2 are GPIO_03 or GPIO_13.
taliseGpioPinSel_t.rxGainDecPin	Acceptable values for Rx1 are GPIO_01 or GPIO_11 and acceptable values for Rx2 are GPIO_04 or GPIO_14.
unit8_t enable	1 = enable (pin control) and 0 = disable (SPI control).

The overload detectors can be monitored over the GPIO pins. Principally, the GPIOs can be configured to report when an upper or lower threshold is exceeded on the APD or HB2 detectors (see the GPIO Configuration section). For details on what causes an overrange condition, see the Peak Detect Mode section.

AGC Mode

In AGC mode, a built in state machine automatically controls the gain based on the user defined configuration. The AGC can be configured in peak detect mode, where only the overload detectors are used to make gain changes or in peak/power detect mode, where information from the power detector and the overload detectors are used to make gain changes.

The `agcPeakThreshGainControlMode` parameter of the `taliseAgcCfg_t` AGC configuration structure selects the individual modes of the AGC operation (see Table 53).

Table 53. agcPeakThreshGainControlMode settings

agcPeakThreshGainControlMode Setting	Description
0	AGC in peak/power mode
1	AGC is peak detect mode

Peak Detect Mode

In this mode, only the overload detectors are used to inform the AGC to make gain changes. This section explains the basic premise of the operation.

The APD and HB2 detector have high thresholds (`apdHighThresh` and `hb2HighThresh`) and low thresholds (`apdLowThresh` and `hb2UnderRangeHighThresh`). These thresholds and the number of times a threshold must be exceeded for an overrange condition to be flagged are user programmable. The high thresholds are used as limits on the incoming signal level and are principally set based on the maximum input of the ADC. When an overrange condition occurs, the AGC reduces the gain (gain attack).

The low thresholds are used as lower limits on the signal level. If the signal peaks are not exceeding the lower threshold, this is indicative of a low power signal, and the AGC increases the gain (gain recovery), which is defined as an underrange. The AGC stable state (where gain is not adjusted) occurs when neither an underrange nor overrange (overload) condition occurs, for example, the signal peaks are less than the high threshold and greater than the lower threshold.

Each overload/underrange condition has its own attack and recovery gain step (see Table 54).

Table 54. Peak Detector Overrange/Underrange Gain Steps

Overload/Underrange Condition	Gain Step
apdHighThresh Overload	Reduces gain by apdGainStepAttack.
apdLowThresh Underrange	Increases gain by apdGainStepRecovery.
hb2HighThresh Overload	Reduces gain by hb2GainStepAttack.
hb2UnderRangeHighThresh Underrange	Increases gain by hb2GainStepHighRecovery.

An overrange condition occurs when the high thresholds are exceeded for a configurable number of times within a configurable period. An underrange condition occurs when the low thresholds are not exceeded for a configurable number of times within the same configurable period. These counters make the AGC more or less sensitive to peaks in the input signal, ensuring that a single peak exceeding a threshold does not necessarily cause the AGC to react. This sensitivity allows the user to tradeoff the bit error rate with the signal-to-noise ratio (SNR). Table 55 describes the counter parameters for the individual overload/underrange conditions.

Table 55. Peak Detector Overrange/Underrange Counter Settings

Overload/Underrange Condition	Counter Name
apdHighThresh Overload	apdUpperThreshPeakExceededCnt
apdLowThresh Underrange	apdLowerThreshPeakExceededCnt
hb2HighThresh Overload	hb2UpperThreshPeakExceededCnt
hb2UnderRangeHighThresh Underrange	hb2LowerThreshPeakExceededCnt

The AGC uses a gain update counter to time the gain changes, where gain changes are made when the counter expires. The counter value, and therefore the time spacing between possible gain changes, is user programmable through the `agcGainUpdateCounter_us` parameter. The user specifies the period (in µs) that gain changes can be made. Typically, this period is set to frame or subframe boundary periods. The total time between gain updates is the combination of the `agcSlowLoopSettlingDelay` and the `agcGainUpdateCounter_us` parameters.

When the gain update counter expires, all overload counters are reset. Therefore, the gain update period is a decision period, which means that the overload thresholds and gain update counters are set based on the number of overloads that are considered acceptable for the application within the gain update period.

Figure 79 shows an example of the AGC response to a signal vs. the APD threshold levels (the APD is considered in isolation). Initially, the peaks of the signal are within the `apdHighThresh` and `apdLowThresh` thresholds, and no gain changes are made. Then, an interferer suddenly appears, whose peaks exceed the `apdHighThresh` threshold. On the next expiration of the gain update counter (assuming a sufficient number of peaks has occurred to exceed the counter), the AGC decrements the gain index (reduces the gain) by using the `apdGainStepAttack` recovery gain step. This reduction in gain is not sufficient to get the signal peaks within the threshold levels, and so the gain is decremented again, with the peaks now between the two thresholds. The gain is stable in this current gain level until the interfering signal is removed, and the peaks of the wanted signal are now below the `apdLowThresh` threshold, which results in an underrange condition. Therefore, the AGC increases gain with the `apdGainStepRecovery` recovery gain step at the next expiration of the gain update counter, and continues to increase the gain until the peaks of the signal are within the two thresholds.

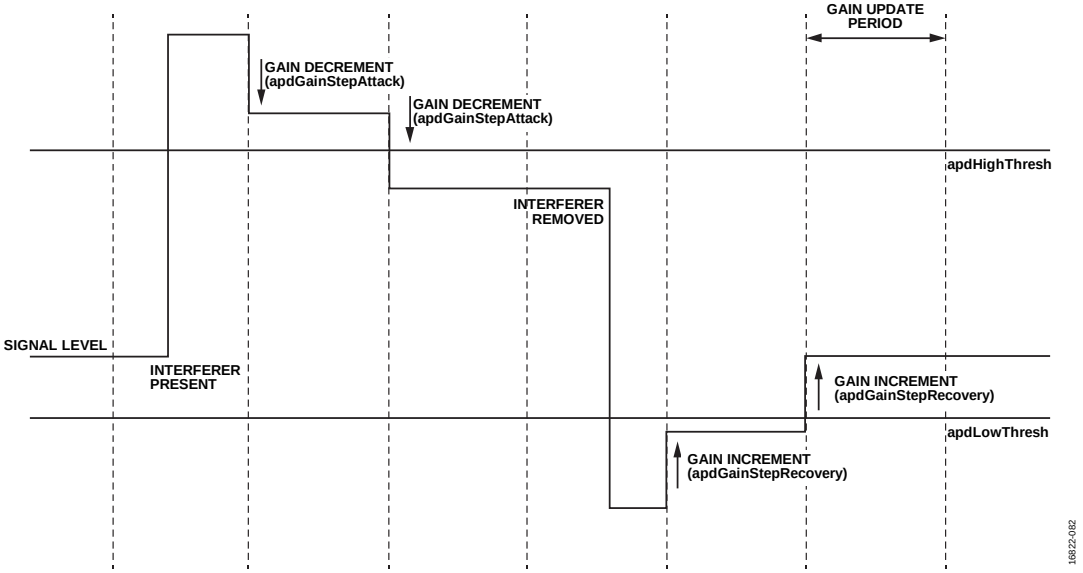

Figure 79. APD Thresholds and Gain Changes Associated with Underrange and Overrange Conditions

Figure 80 shows the same scenario from the viewpoint of the HB2 detector that is considered in isolation.

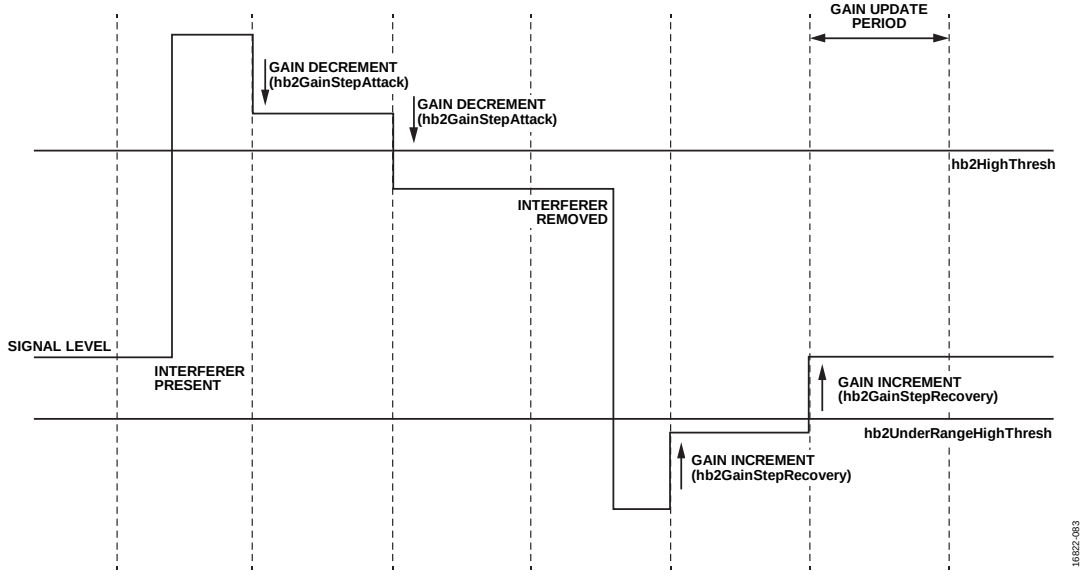

Figure 80. HB2 Thresholds and Gain Changes Associated with Underrange and Overrange Conditions

It is possible to enable a fast attack mode in which the AGC is instructed to reduce gain immediately when an overrange condition occurs, rather than wait until the next expiration of the gain update counter, using the `agcGainChangeIfThreshHigh` threshold. This parameter has independent controls for the APD and HB2 detectors. Values from 0 to 3 are valid (see Table 56).

Table 56. agcGainChangeIfThreshHigh Settings

agcChangeGainIfThreshHigh Setting, Bits[1:0]	Gain Change Following APD Overrange	Gain Change Following HB2 Overrange
00	After expiry of agcGainUpdateCounter_us	After expiry of agcGainUpdateCounter_us
01	After expiry of agcGainUpdateCounter_us	Immediately
10	Immediately	After expiry of agcGainUpdateCounter_us
11	Immediately	Immediately

Figure 81 shows the reaction of the AGC when the `agcChangeGainIfThreshHigh` is set for APD. In this case, when the interferer appears, the gain is updated as soon as the number of peaks exceed the peak counter. The AGC does not wait for the next expiry of the gain update counter. A number of gain changes can be made in quick succession, which provides a faster attack than the default operation. The assumption is that if the ADC is overloaded, it is best to decrease the gain quickly rather than wait for a suitable moment in the received signal to change the gain.

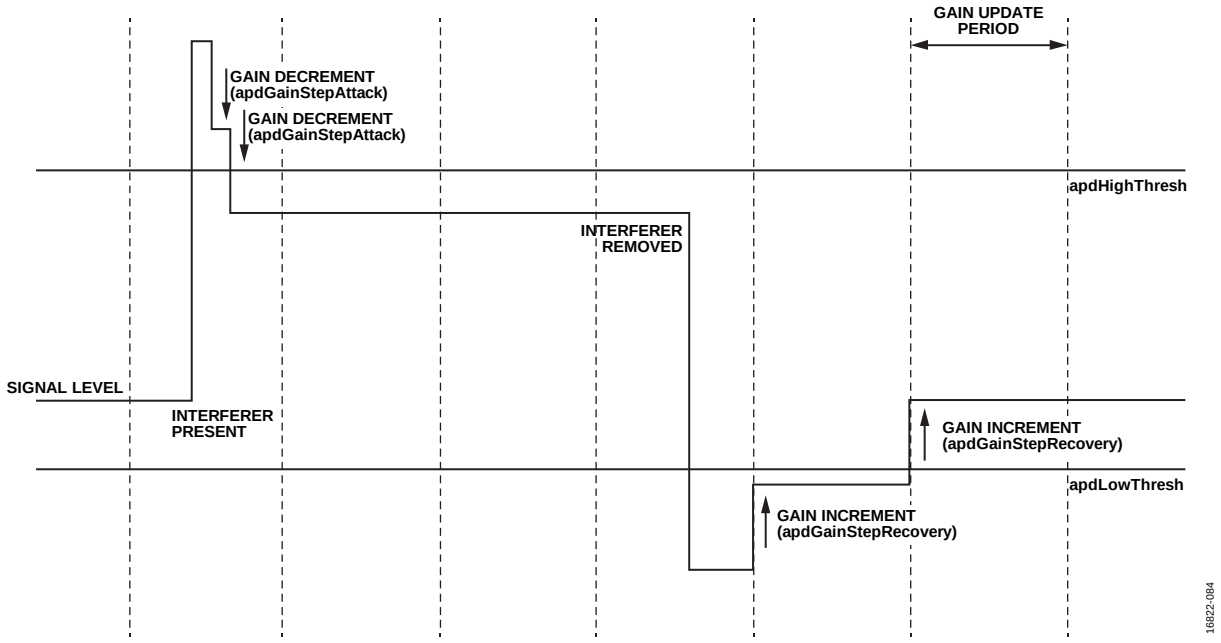

Figure 81. APD Gain Changes with Fast Attack Enabled

Figure 82 shows the same scenario from the viewpoint of the `agcChangeGainIfThreshHigh` threshold set for HB2.

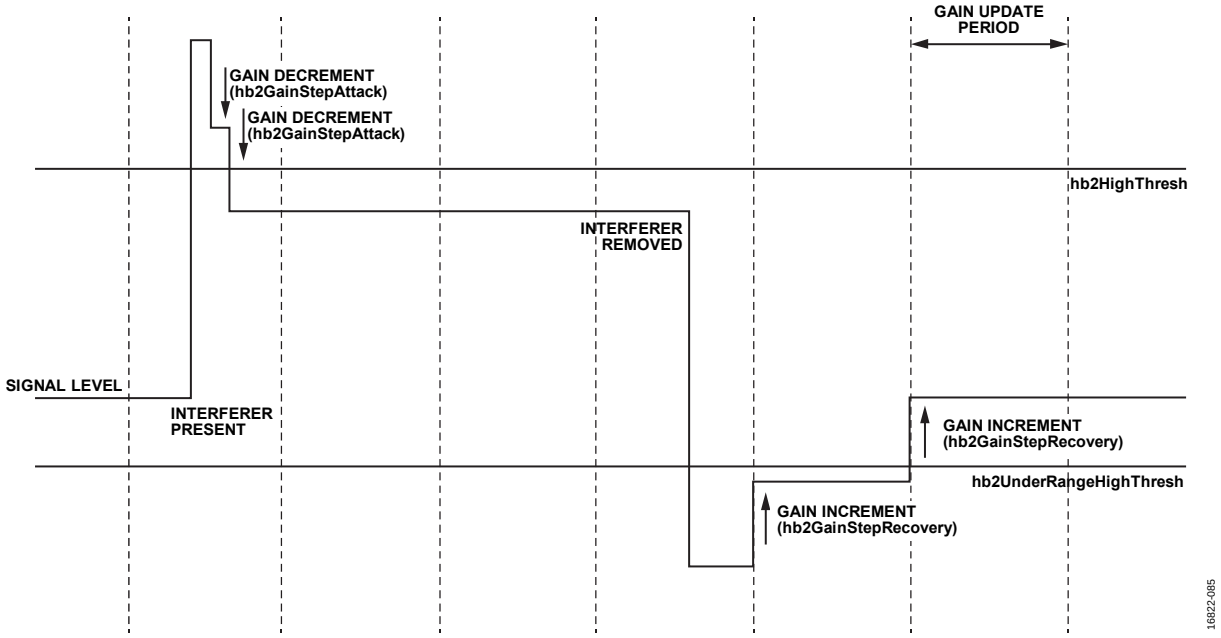

Figure 82. HB2 Gain Changes with Fast Attack Enabled

It is also possible to enable a fast recovery mode in which gain recovery occurs before the expiration of the gain update counter. This functionality is enabled with the `agcEnableFastRecoveryLoop` parameter in the HB2 overrange detector. The operation is as shown in Figure 83. When the signal level falls below the `hb2UnderRangeLowThresh` threshold, the gain is incremented following the expiration of the `agcUnderRangeLowInterval` counter. After sufficient gain increases to bring the signal level above the `hb2UnderRangeLowThresh` threshold, the gain is incremented by `hb2GainStepMidRecovery` after the expiration of the `hb2GainStepMidRecovery` counter. Finally, when the signal level is increased more than the `hb2UnderRangeMidThresh` threshold, the gain is incremented by `hb2GainStepHighRecovery` following the expiration of the `agcUnderRangeHighInterval` counter. The fast recovery loop is designed so that as the desired signal level is approached, the magnitude of the gain adjustments are reduced, and the time interval between gain changes is increased, as shown in Figure 83. This design allows for a fast gain recovery when the signal is far from the desired level, with the adjustments reducing to ensure convergence to the required level.

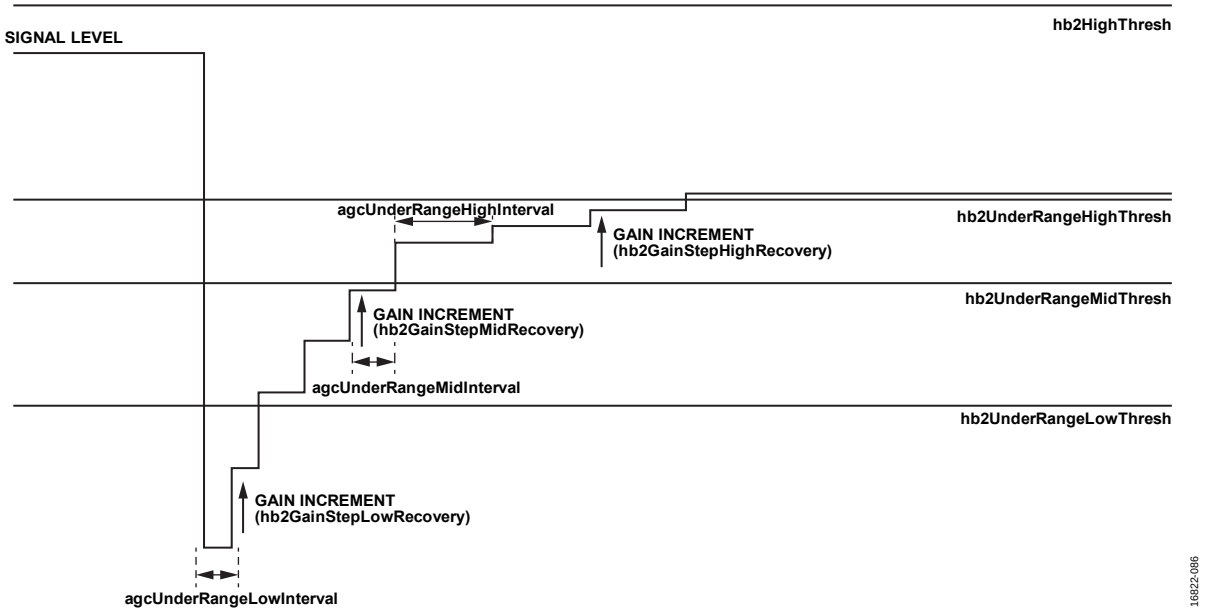

Figure 83. AGC Operation with HB2 Detector in Fast Recovery Mode

Priorities and Overall Operation

It is highly recommended that the apdHighThresh and hb2HighThresh thresholds are set to an equivalent dBFS value. Likewise, it is highly recommended that the apdLowThresh and the hb2UnderRangeHighThresh thresholds are set to equivalent values. This equivalence is approximate because these thresholds have unique threshold settings and are not exactly equal. This section discusses the relevant priorities between the detectors, and how the AGC reacts when multiple threshold detectors have been exceeded.

Table 57 shows the priorities between the detectors when multiple overranges occur.

Table 57. Priorities of Attack Gain Steps

apdHighThresh Overrange	hb2HighThresh Overrange	Gain Change
No	No	No gain change
No	Yes	Gain change by hb2GainStepAttack
Yes	No	Gain change by apdGainStepAttack
Yes	Yes	Gain change by apdGainStepAttack

For recovery, the number of thresholds is dependent on whether fast recovery is enabled or not. Considering the fast recovery scenario, the priority of the thresholds in descending order is the following:

1. hb2UnderRangeLowThresh underrange condition
2. hb2UnderRangeMidThresh underrange condition
3. hb2UnderRangeHighThresh underrange condition
4. apdLowThresh underrange condition

When one underrange condition occurs, the AGC changes the gain by the corresponding gain step size of this condition. However, if multiple conditions occur simultaneously, the AGC acts based on the priorities indicated, for example, if the hb2UnderRangeLowThresh threshold reports an underrange condition, the AGC adjusts the gain by hb2GainStepLowRecovery, with two exceptions.

The apdLowThresh threshold has priority in terms of preventing recovery. If apdLowThresh threshold reports an overrange condition, for example, if the signal is exceeding its threshold, no further recovery is allowed. Configure the apdLowThresh and hb2UnderRangeHighThresh thresholds as close to the same value of dBFS as possible, assuming some small difference between the thresholds, and then, as soon as the apdLowThresh threshold is exceeded, recovery no longer occurs. The reverse is not true. The hb2UnderRangeHighThresh threshold does not prevent the gain recovery towards the apdLowThresh threshold. Given the recommendation that the apdLowThresh and hb2UnderRangeHighThresh thresholds are set equally, a condition where the apdLowThresh is at a lower dBFS level than the hb2UnderRangeLowThresh or hb2UnderRangeMidThresh thresholds does not occur.

Another exception is if the recovery step size for a detector is set to zero. In this case, the AGC makes the gain change of the highest priority detector with a non-zero recovery step. Figure 84 shows a flowchart of the decisions that the AGC makes when recovering the gain in peak detect mode.

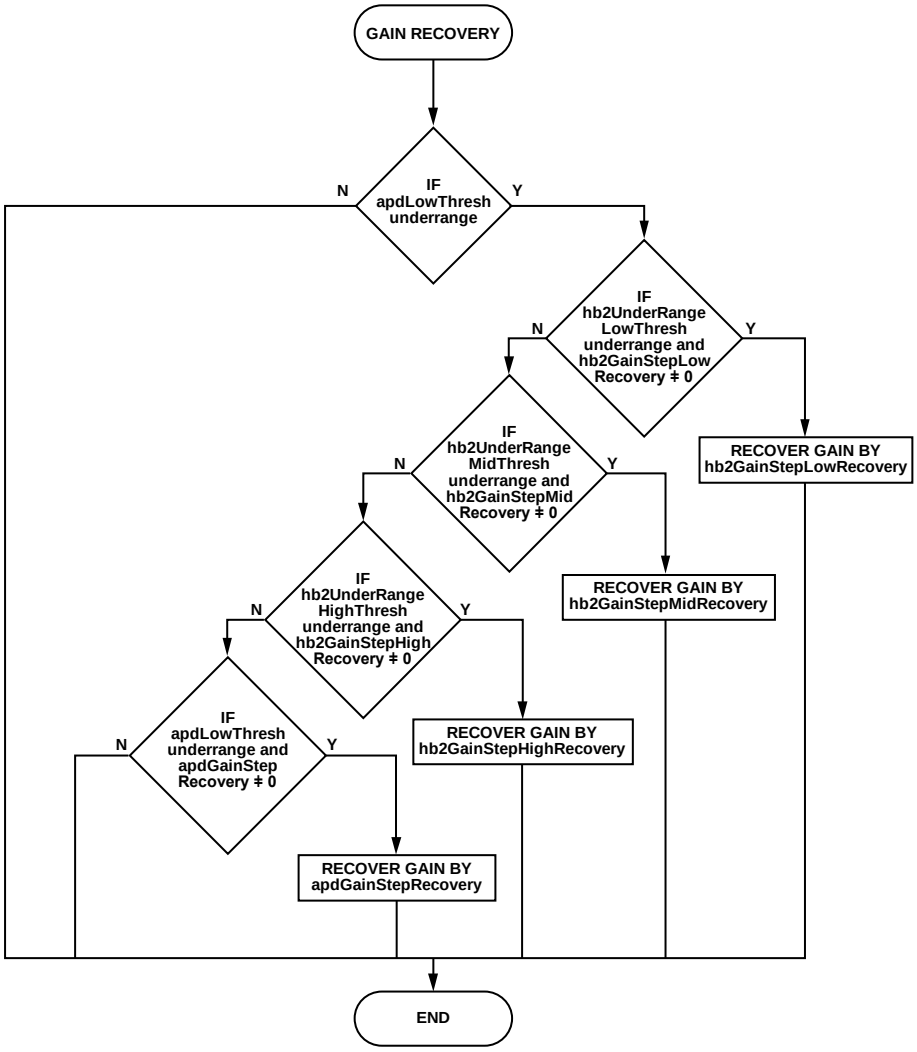

Figure 84. Flowchart for AGC Recovery in Peak Detect AGC Mode

Peak/Power Detect Mode

In this mode, the power detector measurement is also used to control the gain of the receiver chain. In the event of an overrange, the peak detectors and the power detector can instantiate a gain decrement. In the event of an underrange, only the power detector can increment the gain. The power detector changes gain solely at the expiration of the gain update counter. The peak detectors can be set in one of two modes (depending on the setting of `agcGainChangeIfThreshHigh`): the AGC waits for the gain update counter to expire before initiating a gain change, or the AGC immediately updates the gain as soon as the overrange condition occurs (see Figure 83 to Figure 85).

The power measurement block provides the rms power of the receiver data at the measurement location. The power measurement block can be configured to monitor the signal in one of three locations, as shown in Figure 75. In power detect mode, the AGC compares the measured signal level to programmable thresholds that provide a second-order control loop, where the gain can be changed by larger amounts when the signal level is further from the target level and make smaller gain changes when the signal is closer to the target level.

Figure 85 shows the operation of the AGC when using the power measurement detector. The AGC does not modify the gain when the signal level is between `upper0PowerThresh` and `underRangeHighPowerThresh`. This range is the target range for the power measurement.

When the signal level goes below `underRangeLowPowerThresh`, the AGC waits for the next gain update counter expiration and then increments the gain using `underRangeLowPowerGainStepRecovery`. When the signal level is greater than `underRangeLowPowerThresh` but below `underRangeHighPowerThresh`, the AGC increments the gain using

underRangeHighPowerGainStepRecovery. Likewise, when the signal level goes above upper1PowerThresh, the AGC decreases the gain using upper1GainStep, and when the signal level is between upper0PowerThresh and upper1PowerThresh, the AGC decreases the gain using upper0GainStep.

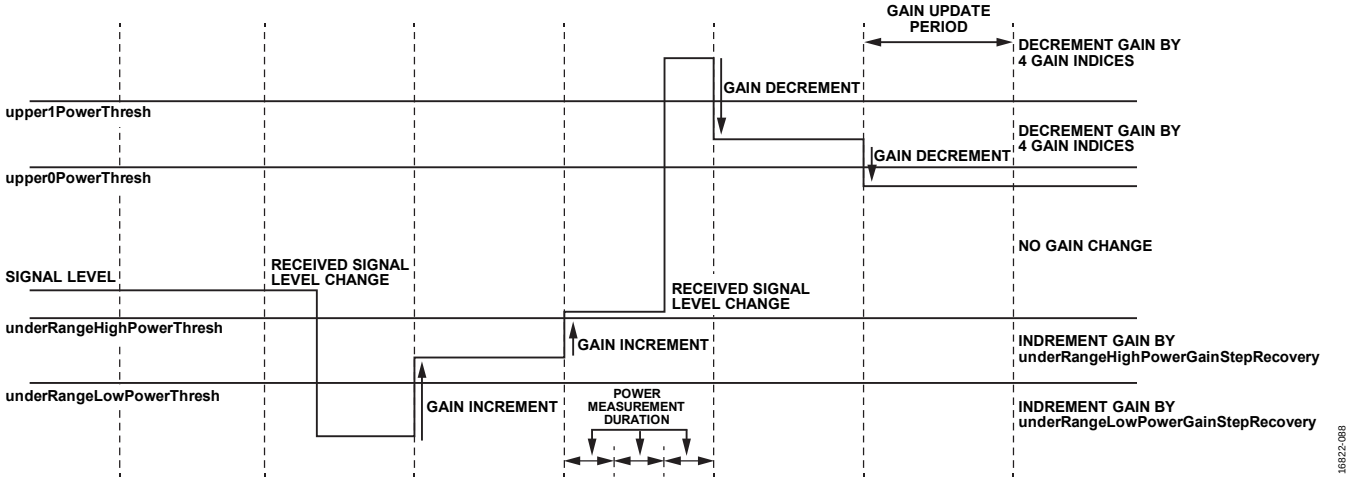

Figure 85. PMD Thresholds and Gain Changes for Underrange and Overrange Conditions

It is possible for the AGC to get contrasting requests from the power and peak detectors, for example, a blocker that is visible to the analog peak detector but is significantly attenuated by the power measurement block. In this case, the APD can request a gain decrement while the power measurement block can request a gain increment. The priority scheme when the AGC is in power detect mode is as follows:

1. APD overrange (upper level)
2. HB2 overrange (upper level)
3. APD lower level overrange
4. HB2 lower level overrange
5. Power measurement

In this example, the gain is decremented because the APD overrange has a higher priority than the power measurement. It is important to note the APD and HB2 lower level overranges. In peak detect mode, the lower level thresholds for these detectors indicate an underrange condition, which causes the AGC to increase the gain. In power detect more, these detectors are not used for gain recovery but can be used to control gain recovery by setting the API parameter, agcLowThreshPreventGain. In this mode, if the signal level is exceeding a lower level threshold, the AGC is prevented from increasing the gain regardless, of the power measurement.

Setting the agcLowThreshPreventGain parameter prevents an oscillation condition that can otherwise occur to a blocker that is visible to an overload detector but is filtered before the power measurement block. In this case, the overload detector can cause the AGC to decrease the gain. The overload detector does this until the blocker is no longer exceeding the defined threshold. At this point, the power measurement block can request an increase in gain and does so until the overload threshold of the detector is exceeded, which decreases the gain. By using these lower level thresholds, the AGC is prevented from increasing gain as the signal level approaches an overload condition, which provides a stable gain level for the receiver chain under such a condition.

AGC CLOCK AND GAIN BLOCK TIMING

The AGC clock is the clock that drives the AGC state machine. A number of the programmable counters are used by the AGC, which are clocked at this rate. The maximum frequency of the AGC clock is 250 MHz. The clock is derived from the Receive Half-Band 1 (RHB1) clock, the input clock to the RHB1 filter. This clock can be scaled by 1, 2, or 4 to ensure that it is less than 250 MHz. For example, in a receiver profile of 200 MHz with an I/Q rate of 245.76 MSPS, the frequency of the RHB1 clock is 491.52 MHz, which results in an AGC clock of 245.76 MHz. The API automatically determines the appropriate scaling to use.

The gain update counter is used to space gain updates when fast attack or fast recovery is not used. The total time between gain updates (gain update period) is a combination of both the agcSlowLoopSettlingDelay and the agcGainUpdateCounter_us periods.

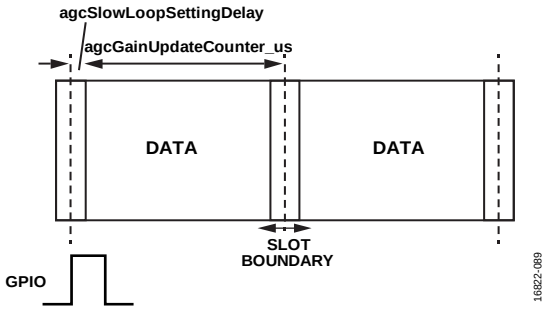

Figure 86. Gain Update Period

Figure 86 outlines the usage of the gain update counter. The dashed lines indicate where a gain decision is made, for example, at these instances, the AGC changes the gain if required (note that fast attack and fast recovery happen asynchronously to the gain update counter). After the expiration of the gain update counter, the AGC enters a settling delay period, defined by `agcSlowLoopSettlingDelay`, where the power measurement and overload detector blocks are disabled for a short time to allow any gain transients to flow through the receiver path. The power measurement and overload detector blocks are enabled at the start of the gain update counter period, and a gain decision is made at the end of the gain update period. At the expiration of the gain update counter, all measurement blocks are reset, and any peak detector counts are reset to zero.

In Figure 86, the gain update counter is shown to expire at slot boundaries, which are suitable points in the received signal where gain changes are acceptable. When the gain update counter first begins, its expiration is at any arbitrary phase to these slot boundaries. To align the expiry to the slot boundaries, set the `agcEnableSyncPulseForGainCounter` parameter. While this bit is set, the AGC monitors a user selected GPIO pin to find a synchronization pulse. This pulse causes the expiration of the counter at this point in time. If the user supplies a GPIO pulse time aligned to these slot boundaries, the expiration of the counter is aligned to slot boundaries. When the receiver is enabled, the AGC can be kept inactive for a number of AGC clock cycles by using `agcRxAttackDelay`, which means that the user can specify one delay for AGC reaction when entering receiver mode and one delay for after a gain change occurs (`agcSlowLoopSettlingDelay`). It is also possible to reset the gain when the receiver is disabled so that the gain is at its maximum by setting `agcResetOnRxon` at the start of each receiver period.

APD

The APD is located in the analog domain, prior to the ADC input. The APD functions by comparing the signal level to programmable thresholds. When a threshold is exceeded for a programmable number of times in a gain update period, the detector flags that the threshold is overloaded.

There are two APD thresholds, as shown in Figure 87. These thresholds are contained in the `agcPeak` API structure, `apdHighThresh` and `apdLowThresh`, respectively.

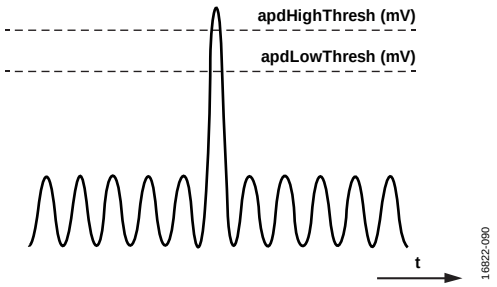

Figure 87. Analog Peak Detector Thresholds

The thresholds are typically considered relative to the full-scale voltage of the ADC, which is 850 mV peak. Determine the mV setting of the APD thresholds by using the following equations:

apdHighThresh (mV) = (*apdHighThresh* + 1) × 16 mV

apdLowThresh (mV) = (*apdLowThresh* + 1) × 16 mV

To determine the setting of the APD thresholds in terms of the closest possible setting in terms of dBFS of the ADC (ADCdBFS), use the following equations:

$$apdHighThresh = round\left(\frac{\left(e^{\left(\frac{\ln(10)\times ADCdBFS}{20}\right)}\times 850\right)-16}{16}\right)$$

$$apdLowThresh = round\left(\frac{\left(e^{\left(\frac{\ln(10)\times ADCdBFS}{20}\right)}\times 850\right)-16}{16}\right)$$

The APD threshold must exceed a programmable number within a gain update counter period before an overrange condition occurs. The upper and lower thresholds have a programmable counter in the `agcPeak` API structure, as indicated in Table 58.

Table 58. APD Programmable Threshold Counters

Threshold Name	Counter Name
Upper Threshold (apdHighThresh)	apdUpperThreshPeakExceededCnt
Lower Threshold (apdLowThresh)	apdLowerThresPeakExceededCnt

As described in the AGC Mode section, the APD is used for gain attack and gain recovery in peak detect mode. In power detect mode, the APD is used for gain attack and to prevent overloading during gain recovery.

In AGC mode, the APD has programmable gain attack and gain recovery step sizes (see Table 59).

Table 59. APD Attack and Recovery Step Sizes

Gain Change	Step Size
Gain Attack	apdGainStepAttack
Gain Recovery	apdGainStepRecovery

The step size refers to the number of indices of the gain table that the gain is changed. The gain table is programmed with the largest gain in the maximum gain index (typically index 255), with decreasing gain for decreasing gain index. If the APD gain attack step size is programmed to 6, the gain index is reduced by 6 when the `apdHighThresh` is exceeded more than `apdUpperThreshPeakExceededCnt` times. For example, if the gain index is 255 before this overrange condition, the gain index is reduced to 249. The amount of gain reduction that this gain index equates to is dependent on the gain table in use. The default table has 0.5 dB steps, which in this example, equate to a 3 dB gain reduction in the event of an APD overrange condition.

The APD is held in reset for a configurable amount of time following a gain change to ensure that the receiver path is settled at the new gain setting before monitoring the paths for overranges. The time that the ADP is held in reset is configured using the `agcPeakWaitTime` parameter.

Note that although it is always recommended that thresholds be optimized for a particular use case, for LO operation lower than 200 MHz, the APD thresholds can require extra attention. In this range, the APD detector threshold levels must be configured approximately 3 dB less than is required for LO frequencies that are greater than 200 MHz to trigger on the same input signal level.

HB2 PEAK DETECTOR

The HB2 peak detector is located in the digital domain at the output of the HB2 filter and is also referred to as the decimated data overload detector because this detector works on decimated data. The HB2 peak detector functions by comparing the signal level to programmable thresholds, similar to the APD detector.

The HB2 detector monitors the received signal level by observing individual I/Q samples over a period of time and comparing these samples to the threshold. If a sufficient number of samples exceeds the threshold in the sample period, the threshold is flagged as exceeded by the detector. The duration of the HB2 measurement is controlled by the `hb2OverloadDurationCnt` setting, and the number of samples that exceeds the threshold in that period is controlled by the `hb2OverloadThreshCnt` setting.

When the required number of samples exceeds the threshold in the duration required, the detector records that the threshold was exceeded. The HB2 detector requires a programmable number of instances for the threshold to be exceeded in a gain update period before the detector flags an overrange condition, similar to the APD detector.

Figure 88 shows a two level approach to detect an overrange condition. Figure 88 shows the gain update counter period, the time that is broken into subsets of time based on the setting of the `hb2OverloadDurationCnt` value. Each of these periods of time is considered separately, and the `hb2OverladThreshCnt` individual samples must exceed the threshold within the `hb2OverloadDurationCnt` to declare an overrange. Two examples are shown in Figure 88. One example shows that the number of samples exceeding the threshold is sufficient for the HB2 peak detector to declare an overrange. The second example shows that the number of samples exceeding the threshold is not sufficient to declare an overrange. The number of overranges are counted, and if the number of overranges of the `hb2HighThresh` exceed the `hb2UpperThreshPeakExceededCnt` in a gain update counter period, an overrange condition is flagged. Similarly, if the number of overloads of the `hb2UnderRangeHighThresh` does not exceed the `hb2LowerThreshPeakExceededCnt`, an underrange condition is flagged.

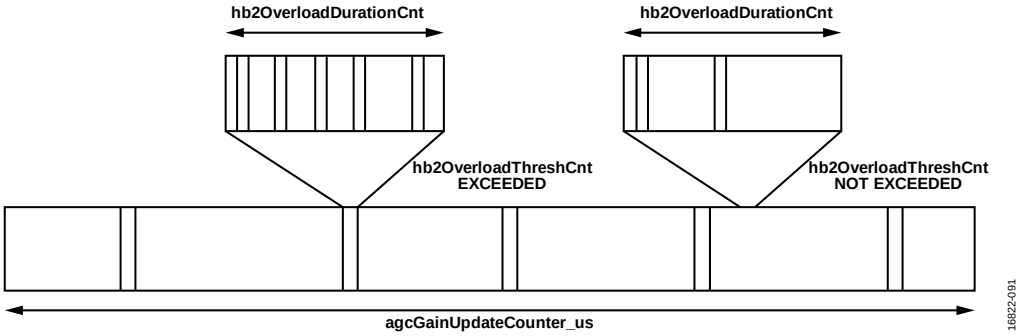

Figure 88. HB2 Detector, Two Level Approach for an Overrange Condition

The HB2 detector has a number of programmable thresholds. Some of these thresholds are only used in the fast recovery mode of the peak detect AGC configuration, as detailed in Table 60.

Table 60. HB2 Overrange Thresholds and Functions

HB2 Threshold Name	Function
hb2HighThresh	Used for gain attack in both peak and power detect AGC modes.
hb2UnderRangeHighThresh	Used for gain recovery in peak detect AGC mode. Used to prevent overloads during gain recovery in power detect AGC mode.
hb2UnderRangeMidThresh	Used only when the fast recovery option of the peak detect AGC mode is being utilized.
hb2UnderRangeLowThresh	Used only when the fast recovery option of the peak detect AGC mode is being utilized.

For more details on these thresholds, Figure 80, Figure 82, and Figure 83.

The HB2 overrange thresholds are related to an ADCdBFS value, which is calculated with the following equations:

$$hb2HighThresh = 256 \times 10^{\left(\frac{hb2HighThresh\ \text{dBFS}}{20}\right)}$$

$$hbd2UnderRangeHighThresh = 256 \times 10^{\left(\frac{hbd2UnderRangeHighThresh\ \text{dBFS}}{20}\right)}$$

$$hbd2UnderRangeMidThresh = 256 \times 10^{\left(\frac{hbd2UnderRangeMidThresh\ \text{dBFS}}{20}\right)}$$

$$hbd2UnderRangeLowThresh = 256 \times 10^{\left(\frac{hbd2UnderRangeLowThresh\ \text{dBFS}}{20}\right)}$$

Each HB2 threshold has an associated counter that does not flag an overrange condition until the threshold is exceeded for a specified amount of times in a gain update period.

Table 61. Gain Steps for an HB2 Overrange and Underrange Condition

HB2 Threshold Name	Counter Value
hb2HighThresh	hb2UpperThreshPeakExceededCnt
hb2UnderRangeHighThresh	hb2LowerThreshPeakExceededCnt
hb2UnderRangeMidThresh	Set to 3
hb2UnderRangeLowThresh	Set to 3

In AGC mode, the HB2 has programmable gain attack and gain recovery step sizes (see Table 62).

Table 62. HB2 Attack and Recovery Step Sizes

Gain Change Name	Step Size
Gain Attack	hb2GainStepAttack
Gain Recovery (hb2UnderRangeHighThresh)	hb2GainStepHighRecovery
Gain Recovery (hb2UnderRangeMidThresh)	hb2GainStepMidRecovery
Gain Recovery (hb2UnderRangeLowThresh)	hb2GainStepLowRecovery

The HB2 peak detector is held in reset for a configurable amount of time following a gain change to ensure that the receiver path is settled at the new gain setting before monitoring the paths for overrange conditions. This duration is configured using the `agcPeakWaitTime` parameter.

POWER DETECTOR

The power detector measures the rms power of the incoming signal. The power detector can monitor the signal level at different locations, namely the HB2 output, the RFIR output, and the output of the dc correction block. To choose a location to monitor, the `powerUseRfirOut` and `powerUseBBDC2` API parameters are utilized.

Table 63. Location of the Decimated Power Measurement

Power Measurement Location	powerUseRfirOut Setting	powerUseBBDC2 Setting
HB2 Output	0	0
RFIR Output	1	0
Baseband DC (BBDC) Output	0	1

The number of samples that are used in the power measurement calculation is configurable using the `powerMeasurementDuration` parameter:

$$\textit{Power Measurement Duration}\ (\textit{Receiver Sample Clocks}) = 8 \times 2^{powerMeasurementDuration}$$

It is important that the power measurement duration does not exceed the gain update counter. The gain update counter resets the power measurement block and a valid power measurement must be available before this event. In the case of multiple power measurements occurring in a gain update period, the AGC uses the last fully completed power measurement, and any partial measurements are discarded.

The power measurement block has a dynamic range of 40 dB by default. This range can be extended to 60 dB by enabling the `powerLogShift` in the power measurement configuration.

GAIN CONTROL API PROGRAMMING

The API programming sequence for the gain control blocks in the device is shown in Figure 89. The configuration of these blocks is one of the last items to be configured before the device is operational. The data structures are defined before initialization of the device begins. When the device initialization has proceeded to the JESD204B configuration, the gain control configuration begins.

The `TALISE_setupRxAgc()` function configures the gain control blocks, for example, the peak detectors, power detector, and the AGC (if used). Run the `TALISE_setupRxAgc()` function in all gain control modes. The peak and power detectors must be configured if they are used. This function requires the following structures to be configured prior to this function being called:

- `The taliseAgcCfg_t` structure, which contains all the gain control settings. Principally, this structure has AGC control structures. However, parameters such as the `agcGainUpdateCounter_us` counter are important in MGC mode because the overload detectors use this counter. This structure also contains the peak and power detector structures.
- The `taliseAgcPeak_t` structure, which contains parameters that are used to configure the APD and HB2 peak detectors.
- The `taliseAgcPower_t` structure, which contains parameters that are used to configure the power measurement block.

The `TALISE_setRxGainControlMode()` function that configures the device in AGC or MGC mode.

The final step in the AGC configuration is to configure any GPIOs as necessary, for example, the monitor outputs that allow real-time monitoring of the peak detector outputs, the GPIO inputs that allow the AGC gain update counter to be synchronized to a slot boundary, or the GPIOs that directly control the gain index. See the GPIO Configuration section for details.

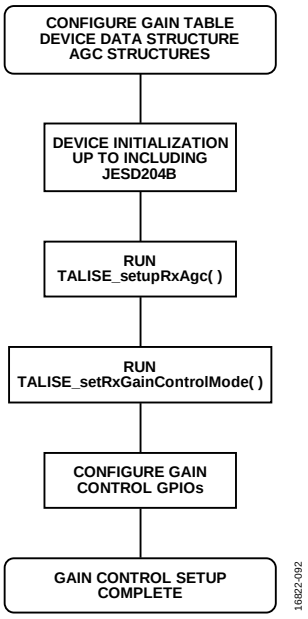

Figure 89. Gain Control Programming Flowchart

GAIN CONTROL DATA STRUCTURES

Figure 90 shows the member structure of the `taliseAgcCfg_t` structure and of its substructures, `taliseAgcPeak_t` and `taliseAgcPower_t`. Each of the parameters are briefly explained in Table 64 to Table 66.

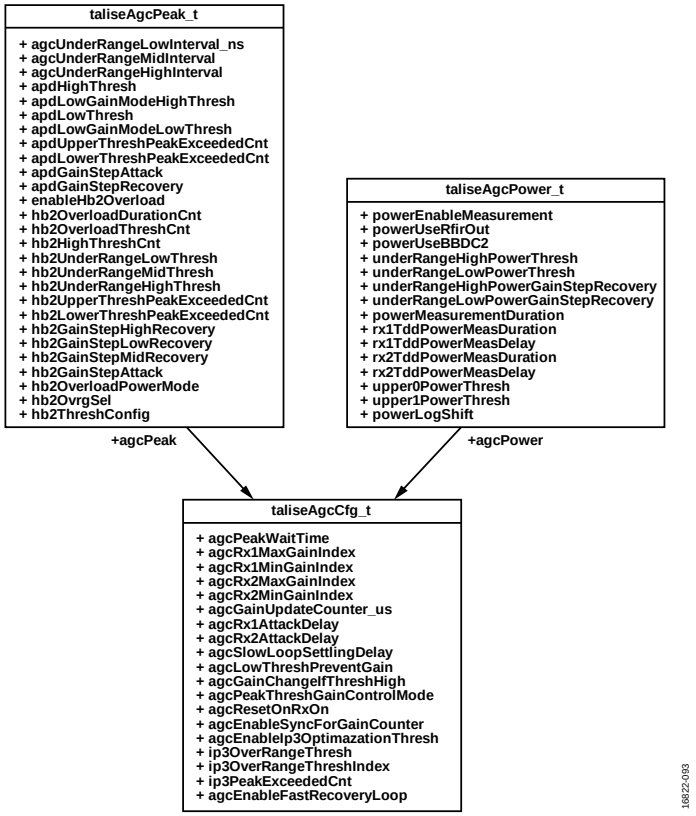

Figure 90. Member Listing of taliseAgcCfg_t, taliseAgcPeak_t, and taliseAgcPower_t Data Structures

Table 64. taliseAgcCfg_t Structure Definition

Parameter Name	Description	Minimum Value	Maximum Value
agcPeakWaitTime	Number of AGC clock cycles to wait before enabling peak/overload detectors after a gain change.	0	31
agcRx1MaxGainIndex	Maximum Rx1 gain index allowed in AGC mode. Must be greater than AGC Rx1 minimum gain index and must be a valid gain index.	128	255
agcRx1MinGainIndex	Minimum Rx1 gain index allowed in AGC mode. Must be less than AGC Rx1 minimum gain index and must be a valid gain index.	128	255
agcRx2MaxGainIndex	Maximum Rx2 gain index allowed in AGC mode. Must be greater than AGC Rx2 minimum gain index and must be a valid gain index.	128	255
agcRx2MinGainIndex	Minimum Rx2 gain index allowed in AGC mode. Must be less than AGC Rx2 maximum gain index and must be a valid gain index.	128	255
agcGainUpdateCounter_us	Used as a decision period, with the peak detectors reset on this period. Gain changes in AGC mode can also be synchronized to this period (the expiration of this counter). The full period is a combination of the agcGainUpdateCounter_us and agcSlowLoopSettlingDelay parameters.	Depends on overload detector settings	4,194,304 AGC clock cycles (in µs)
agcRx1AttackDelay	The duration the AGC is held in reset when the Rx1 path is enabled.	0	63
agcRx2AttackDelay	The duration the AGC is held in reset when the Rx1 path is enabled.	0	63
agcSlowLoopSettlingDelay	Number of I/Q data rate clock cycles to wait after a gain change before peak/power measurements resume.	0	31
agcLowThreshPreventGain	Only relevant in peak/power detect AGC operation. 1: if AGC is in peak power detect mode, gain increments requested by the power detector are prevented if there are sufficient peaks (APD/HB2 low threshold exceeded count) above the apdLowThresh or hb2UnderRangeHighThresh. 0: apdLowThresh and hb2UnderRangeHighThresh are don't cares for gain recovery.	0	1
agcChangeGainIfThreshHigh	Applicable in both peak and peak/power detect modes. 0: gain changes wait for the expiration of the gain update counter if a high threshold count has been exceeded on either the APD or HB2 detector. 1: gain changes occur immediately when initiated by HB2. Gain changes initiated by the APD wait for the gain update to expire. 2: gain changes occur immediately when initiated by APD. Gain changes initiated by HB2 wait for the gain update to expire. 3: gain changes occur immediately when initiated by APD or HB2 detectors.	0	1
agcPeakThreshGainControlMode	1: AGC in peak AGC mode, power based gain changes are disabled. 0: AGC in power AGC mode where both peak detectors and power detectors are utilized.	0	1
agcResetOnRxon	1: AGC state machine is reset when receiver is disabled. 0: AGC state machine maintains its state when receiver is disabled.	0	1
agcEnableSyncPulseForGainCounter	1: allows synchronization of AGC gain update counter to the time slot boundary. GPIO setup required. 0: AGC gain update counter free runs.	0	1
agcEnableIp3OptimizationThresh	This parameter is not utilized. API ignores its setting	Not applicable	Not applicable

Parameter Name	Description	Minimum Value	Maximum Value
ip3OverRangeThresh	This parameter is not utilized. API ignores its setting	Not applicable	Not applicable
ip3OverRangeThreshIndex	This parameter is not utilized. API ignores its setting	Not applicable	Not applicable
ip3PeakExceededCnt	This parameter is not utilized. API ignores its setting	Not applicable	Not applicable
agcEnableFastRecoveryLoop	1: enables the fast recovery AGC functionality using the HB2 overload detector. Only applicable in peak detect mode. 0: AGC fast recovery is not enabled.	0	1
agcPower	Structure containing all the power detector settings. See Table 66.	Not applicable	Not applicable
agcPeak	Structure containing all the peak detector settings. See Table 65.	Not applicable	Not applicable

Table 65. taliseAgcPeak_t Structure Definition

Parameter Name	Description	Minimum Value	Maximum Value
agcUnderRangeLowInterval_ns	Sets the time that the AGC takes to recover when the signal peaks are less than hb2UnderRangeLowThresh. Only applicable when the fast recovery option is enabled in peak detect AGC mode.	Depends on HB2 detector settings	65,535 I/Q samples (in ns)
agcUnderRangeMidInterval	Sets the time constant that the AGC takes to recover when the signal peaks are less than hb2UnderRangeMidThresh. Calculated as: (agcUnderRangeMidInterval + 1) × agcUnderRangeLowInterval_ns Only applicable when the fast recovery option is enabled in peak detect AGC mode.	0	63
agcUnderRangeHighInterval	Sets the time constant that the AGC takes to recover when the signal peaks are less than hb2UnderRangeHighThresh. Calculated as: (agcUnderRangeHighInterval + 1) × agcUnderRangeMidInterval_ns Only applicable when the fast recovery option is enabled in peak detect AGC mode.	0	63
apdHighThresh	Sets the upper threshold of the analog peak detector. When the input signal exceeds this threshold for a programmable number of times (set by its corresponding overload counter) within a gain update period, the overload detector flags. In AGC modes, the gain reduces when this overload occurs. Calculated as: apdHighThresh (mV) = (apdHighThresh + 1) × 16 mV	apdLowThresh	63
apdLowGainModeHighThresh	Not utilized.	Not applicable	Not applicable
apdLowThresh	This sets the lower threshold of the analog peak detector. When the input signal exceeds this threshold a programmable number of times (set by its corresponding overload counter) within a gain update period, the overload detector flags. In peak AGC mode, the gain is increased when this overload is not occurring. In power AGC mode, this threshold can be used to prevent further gain increases if the agcLowThreshPreventGain bit is set. Calculated as: apdLowThresh (mv) = (apdLowThresh + 1) × 16 mV	0	apdHighThresh
apdLowGainModeLowThresh	This parameter is not utilized.		
apdUpperThreshPeakExceededCnt	Sets number of peaks to detect above apdHighThresh to cause an APD high overload event. In AGC modes, a gain decrement is set by apdGainStepAttack.	0	255
apdLowerThreshPeakExceededCnt	Sets number of peaks to detect above apdLowThresh to cause an APD low overload event. In peak detect AGC mode, if an APD low overload event is not occurring, a gain increment is set by apdGainStepRecovery.	0	255
apdGainStepAttack	The number of indices that the gain index pointer is decreased in the event of an APD high overload in AGC modes. The step size in dB depends on the gain step resolution of the gain table (default = 0.5 dB per index step).	0	31

Parameter Name	Description	Minimum Value	Maximum Value
apdGainStepRecovery	The number of indices that the gain index pointer is increased in the event of no APD high overload event occurring in peak detect AGC mode. The step size in dB depends on the gain step resolution of the gain table (default = 0.5 dB per index step).	0	31
enableHb2Overload	1: HB2 overload detector enabled. 0: HB2 overload detector disabled.	0	1
hb2OverloadDurationCnt	The number of clock cycles (at the HB2 output rate) within which hb2OverloadThreshCnt must be exceeded for an overload to occur. An HB2 overload flag is only raised when the number of these overloads exceeds hb2UpperThreshPeakExceededCnt or hb2LowerThreshPeakExceededCnt within a gain update period. The number of clocks is calculated as: $2^{(hbd2OverloadDurationCnt + 1)}$	0	6
hb2OverloadThreshCnt	Sets the number of individual samples exceeding hb2HighThresh or hb2LowThresh necessary within hb2OverloadDurationCnt for an overload to occur. The HB2 overload flag is only raised when the number of these overloads exceeds hb2UpperThreshPeakExceededCnt or hb2LowerThreshPeakExceededCnt within a gain update period.	1	15
hb2HighThresh	Sets the upper threshold of the HB2 detector.	0	255
hb2UnderRangeLowThresh	Sets the lower threshold of the HB2 underrange threshold detectors. Only used when the fast recovery option of the peak detect AGC mode is utilized.	0	255
hb2UnderRangeMidThresh	Sets the middle threshold of the HB2 underrange threshold detectors. Only used when the fast recovery option of the peak detect AGC mode is utilized.	0	255
hb2UnderRangeHighThresh	Peak detect mode, threshold used for gain recovery. Peak detect with fast recovery mode, sets the highest threshold of the HB2 underrange threshold detectors. Power detect mode, threshold used to prevent further gain increases if agcLowThreshPreventGain is set.	0	255
hb2UpperThreshPeakExceededCnt	Sets number of individual overloads above hb2HighThresh (number of times hb2OverloadThreshCnt was exceeded in hb2OverloadDuractionCnt) to cause an HB2 high overload event. In AGC modes, a gain decrement is set by hb2GainStepAttack.	0	255
hb2LowerThreshPeakExceededCnt	Sets number of individual overloads above hb2UnderRangeHighThresh (number of times hb2OverloadThreshCnt was exceeded in hb2OverloadDuractionCnt) to cause an HB2 high overload event. In peak detect AGC mode, a gain increment is set by hb2GainStepHighRecovery.	0	255
hb2GainStepHighRecovery	The number of indices that the gain index pointer is increased in the event of an HB2 underrange high threshold overload event in AGC modes. The step size in dB depends on the gain step resolution of the gain table (default = 0.5 dB per index step).	0	31
hb2GainStepLowRecovery	The number of indices that the gain index pointer is increased in the event of an HB2 underrange low threshold overload event in AGC modes. The step size in dB depends on the gain step resolution of the gain table (default = 0.5 dB per index step). Only used when the fast recovery option of the peak detect AGC mode is utilized.	0	31
hb2GainStepMidRecovery	The number of indices that the gain index pointer is increased in the event of an HB2 underrange mid threshold overload event in AGC modes. The step size in dB depends on the gain step resolution of the gain table (default = 0.5 dB per index step). Used only when the fast recovery option of the peak detect AGC mode is utilized.	0	31
hb2GainStepAttack	The number of indices that the gain index pointer is decreased in the event of an HB2 high overload in AGC mode. The step size in dB depends on the gain step resolution of the gain table (default = 0.5 dB per index step).	0	31
hb2OverloadPowerMode	Sets the measurement mode of the HB2 detector.	0	1
hb2OvrgSel	Set to 0 to enable the HB2 overload detector.	0	0
Hb2ThreshConfig	Set to 3.	3	3

Table 66. taliseAgcPower_t Structure Definition

Parameter Name	Description	Minimum Value	Maximum Value
powerEnableMeasurement	1: power measurement block enabled. 0: power measurement block disabled.	0	1
powerUseRfirOut	Uses RFIR output for power measurement.	0	1
powerUseBBDC2	Uses dc offset block output for power measurement.	0	1
underRangeHighPowerThresh	Threshold in dBFS (negative value assumed), which defines the lower boundary on the stable region (no gain change based on power measurement) of the power detect gain control mode.	0	127
underRangeLowPowerThresh	Offset (negative value assumed) from underRangeHighPowerThresh, which defines the outer boundary of the power based AGC convergence. Typically, recovery is set to larger steps than when the power measurement is greater than this threshold.	0	31
underRangeHighPowerGainStepRecovery	The number of indices that the gain index pointer is decreased in the event that the power measurement is less than underRangeHighPowerThresh but greater than underRangeLowPowerThresh.	0	31
underRangeLowPowerGainStepRecovery	The number of indices that the gain index pointer is decreased in the event that the power measurement is less than underRangeLowPowerThresh.	0	31
powerMeasurementDuration	Number of I/Q samples on which to perform the power measurement. The number of samples corresponding to the 4-bit word is calculated as: $8 \times 2^{(pmdMeasDuration[3:0])}$ This value must be less than the AGC gain update counter.	0	31
rx1TddPowerMeasDuration	Following an Rx enable, the power measurement block can be requested to perform a power measurement for a specific period of a frame. This request is applicable in TDD modes. Sets the duration of this power measurement for Rx1. A value of 0 causes the power measurement to run until the next gain update counter expires.	0	65,535 AGC clock cycles
rx1TddPowerMeasDelay	Following an Rx Enable, the power measurement block can be requested to perform a power measurement for a specific period of a frame. This request is applicable in TDD modes. Sets the delay between the Rx enable and the power measurement, starting on Rx1.	0	65,535 AGC clock cycles
rx2TddPowerMeasDuration	Following an Rx enable, the power measurement block can be requested to perform a power measurement for a specific period of a frame. This request is applicable in TDD modes. Sets the duration of this power measurement for Rx2. A value of 0 causes the power measurement to run until the next gain update counter expires.	0	65,535 AGC clock cycles
rx2TddPowerMeasDelay	Following an Rx enable, the power measurement block can be requested to perform a power measurement for a specific period of a frame. This request is applicable in TDD modes. Sets the delay between the Rx enable and the power measurement starting on Rx2.	0	65,535 AGC clock cycles
upper0PowerThresh	Threshold in dBFS (negative value assumed), which defines the upper boundary on the stable region (no gain change based on power measurement) of the power detect gain control mode.	0	127

Parameter Name	Description	Minimum Value	Maximum Value
upper1PowerThresh	Offset (positive value assumed) from upper0PowerThresh, which defines the outer boundary of the power based AGC convergence. Typically, attack is set to larger steps than when the power measurement is greater than this threshold.	0	15
powerLogShift	Enable increase in dynamic range of the power measurement from 40 dB to approximately 60 dB.	0	1

SAMPLE PYTHON SCRIPTS

It is recommended to use peak detect mode with fast attack and fast recovery for the AGC.

The following is a sample python script provided to enable the AGC in peak detect mode and power measurement mode. The user can use this sample script as a starting point to enable AGC in the device.

For peak detect mode with fast attack and fast recovery options, the sample script provided configures the AGC in peak detect mode with a fast attack option. This script can be executed using the **Iron Python** tab in the GUI.

```
########################
#ADI Demo Python Script
########################

#Import Reference to the DLL
import clr
import array
clr.AddReferenceToFileAndPath("C:\\Program Files (x86)\\Analog Devices\\ADRV900x Transceiver
Evaluation Software\\AdiCmdServerClient.dll")
from AdiCmdServerClient import AdiCommandServerClient

#Create an Instance of the Class
Link = AdiCommandServerClient.Instance

gen_AGC = Link.Talise.RxAgcControl
Peak_AGC = Link.Talise.RxAgcPeak
Power_AGC = Link.Talise.RxAgcPower
AgcSlowMode = Link.Talise.GainMode.Agc
#Connect to the Zynq Platform
if(Link.hw.Connected == 1):
    Connect = 0
else:
    Connect = 1
    Link.hw.Connect("192.168.1.10", 55555)

#Read the Version
print Link.Version()
#######################################
# Program data structure to cmd_server #
#######################################

###############################################
# Program data structure to AGC Control class #
###############################################
```

```
Link.Talise.GetAgcCtrlRegisters(gen_AGC)
gen_AGC.agcPeakWaitTime = 2
gen_AGC.agcRx1MaxGainIndex=255
gen_AGC.agcRx1MinGainIndex=195
gen_AGC.agcRx2MaxGainIndex=255
gen_AGC.agcRx2MinGainIndex=195
gen_AGC.agcGainUpdateCounter_us = 500
gen_AGC.agcRx1AttackDelay = 0
gen_AGC.agcRx2AttackDelay = 0
gen_AGC.agcSlowLoopSettlingDelay = 16
gen_AGC.agcLowThreshPreventGain = 0
gen_AGC.agcChangeGainIfThreshHigh = 1
gen_AGC.agcPeakThreshGainControlMode=1
gen_AGC.agcResetOnRxon=0
gen_AGC.agcEnableSyncPulseForGainCounter=0
gen_AGC.agcEnableFastRecoveryLoop=0
Link.Talise.InitAgcCtrlRegisters(gen_AGC)

########################################
# Program data structure to peakclass #
########################################
Link.Talise.GetAgcPeakRegisters(Peak_AGC)

Peak_AGC.agcUnderRangeLowInterval_ns=4000
Peak_AGC.agcUnderRangeMidInterval=2
Peak_AGC.agcUnderRangeHighInterval=4
Peak_AGC.apdHighThresh=41
Peak_AGC.apdLowThresh=26
Peak_AGC.apdUpperThreshPeakExceededCnt=6
Peak_AGC.apdLowerThreshPeakExceededCnt=3
Peak_AGC.apdGainStepAttack=4
Peak_AGC.apdGainStepRecovery=2
Peak_AGC.enableHb2Overload=1
Peak_AGC.hb2OverloadDurationCnt=1
Peak_AGC.hb2OverloadThreshCnt=1
Peak_AGC.hb2HighThresh=203
Peak_AGC.hb2UnderRangeLowThresh=80
Peak_AGC.hb2UnderRangeMidThresh=100
Peak_AGC.hb2UnderRangeHighThresh=128
Peak_AGC.hb2UpperThreshPeakExceededCnt=6
Peak_AGC.hb2LowerThreshPeakExceededCnt=3
Peak_AGC.hb2GainStepHighRecovery=2
Peak_AGC.hb2GainStepMidRecovery=4
Peak_AGC.hb2GainStepLowRecovery=8
Peak_AGC.hb2GainStepAttack=4
Peak_AGC.hb2OverloadPowerMode=0
Peak_AGC.hb2OvrgSel=0
```

```
Peak_AGC.hb2ThreshConfig=3

Link.Talise.InitAgcPeakRegisters(Peak_AGC)

########################################
# Program data structure to powerclass #
########################################
Power_AGC.powerEnableMeasurement=0
Power_AGC.powerUseRfirOut=1
Power_AGC.powerUseBBDC2=0
Power_AGC.underRangeHighPowerThresh=0x0E
Power_AGC.underRangeLowPowerThresh=0x02
Power_AGC.underRangeHighPowerGainStepRecovery=0x02
Power_AGC.underRangeLowPowerGainStepRecovery=0x04
Power_AGC.powerMeasurementDuration=0x05
Power_AGC.rx1TddPowerMeasDuration=0x05
Power_AGC.rx1TddPowerMeasDelay=0x01
Power_AGC.rx2TddPowerMeasDuration=0x05
Power_AGC.rx2TddPowerMeasDelay=0x01
Power_AGC.upper0PowerThresh=0x0A
Power_AGC.upper1PowerThresh=0x02
Power_AGC.powerLogShift=1

Link.Talise.InitAgcPowerRegisters(Power_AGC)

################################################
# Program data structure to Talise, enable AGC #
################################################
Link.Talise.SetupRxAgc()

Link.Talise.SetRxGainControlMode(AgcSlowMode)

#Disconnect from the Zynq Platform
if(Connect == 1):
       Link.hw.Disconnect()
```

For power measurement detect mode with fast attack option, the sample script provided configures the AGC in power measurement detect mode with a fast attack option. This script can be executed using the **Iron Python** tab in the GUI.

```
########################
#ADI Demo Python Script
########################

#Import Reference to the DLL
import clr
import array
clr.AddReferenceToFileAndPath("C:\\Program Files (x86)\\Analog Devices\\ADRV900x Transceiver
Evaluation Software\\AdiCmdServerClient.dll")
```

```
from AdiCmdServerClient import AdiCommandServerClient

#Create an Instance of the Class
Link = AdiCommandServerClient.Instance

gen_AGC = Link.Talise.RxAgcControl
Peak_AGC = Link.Talise.RxAgcPeak
Power_AGC = Link.Talise.RxAgcPower
AgcSlowMode = Link.Talise.GainMode.Agc
#Connect to the Zynq Platform
if(Link.hw.Connected == 1):
    Connect = 0
else:
    Connect = 1
    Link.hw.Connect("192.168.1.10", 55555)

#Read the Version
print Link.Version()
########################################
# Program data structure to cmd_server #
########################################

################################################
# Program data structure to AGC Control class #
################################################
Link.Talise.GetAgcCtrlRegisters(gen_AGC)
gen_AGC.agcPeakWaitTime = 2
gen_AGC.agcRx1MaxGainIndex=255
gen_AGC.agcRx1MinGainIndex=195
gen_AGC.agcRx2MaxGainIndex=255
gen_AGC.agcRx2MinGainIndex=195
gen_AGC.agcGainUpdateCounter_us = 500
gen_AGC.agcRx1AttackDelay = 0
gen_AGC.agcRx2AttackDelay = 0
gen_AGC.agcSlowLoopSettlingDelay = 16
gen_AGC.agcLowThreshPreventGain = 1
gen_AGC.agcChangeGainIfThreshHigh = 1
gen_AGC.agcPeakThreshGainControlMode=0
gen_AGC.agcResetOnRxon=0
gen_AGC.agcEnableSyncPulseForGainCounter=0
gen_AGC.agcEnableFastRecoveryLoop=0
Link.Talise.InitAgcCtrlRegisters(gen_AGC)

########################################
# Program data structure to peakclass #
########################################
Link.Talise.GetAgcPeakRegisters(Peak_AGC)
```

```
Peak_AGC.agcUnderRangeLowInterval_ns=4000
Peak_AGC.agcUnderRangeMidInterval=2
Peak_AGC.agcUnderRangeHighInterval=4
Peak_AGC.apdHighThresh=41
Peak_AGC.apdLowThresh=26
Peak_AGC.apdUpperThreshPeakExceededCnt=6
Peak_AGC.apdLowerThreshPeakExceededCnt=3
Peak_AGC.apdGainStepAttack=4
Peak_AGC.apdGainStepRecovery=2
Peak_AGC.enableHb2Overload=1
Peak_AGC.hb2OverloadDurationCnt=1
Peak_AGC.hb2OverloadThreshCnt=1
Peak_AGC.hb2HighThresh=203
Peak_AGC.hb2UnderRangeLowThresh=80
Peak_AGC.hb2UnderRangeMidThresh=100
Peak_AGC.hb2UnderRangeHighThresh=128
Peak_AGC.hb2UpperThreshPeakExceededCnt=6
Peak_AGC.hb2LowerThreshPeakExceededCnt=3
Peak_AGC.hb2GainStepHighRecovery=2
Peak_AGC.hb2GainStepMidRecovery=4
Peak_AGC.hb2GainStepLowRecovery=8
Peak_AGC.hb2GainStepAttack=4
Peak_AGC.hb2OverloadPowerMode=0
Peak_AGC.hb2OvrgSel=0
Peak_AGC.hb2ThreshConfig=3

Link.Talise.InitAgcPeakRegisters(Peak_AGC)

########################################
# Program data structure to powerclass #
########################################
Power_AGC.powerEnableMeasurement=1
Power_AGC.powerUseRfirOut=1
Power_AGC.powerUseBBDC2=0
Power_AGC.underRangeHighPowerThresh=0x0E
Power_AGC.underRangeLowPowerThresh=0x02
Power_AGC.underRangeHighPowerGainStepRecovery=0x02
Power_AGC.underRangeLowPowerGainStepRecovery=0x04
Power_AGC.powerMeasurementDuration=0x05
Power_AGC.rx1TddPowerMeasDuration=0x05
Power_AGC.rx1TddPowerMeasDelay=0x01
Power_AGC.rx2TddPowerMeasDuration=0x05
Power_AGC.rx2TddPowerMeasDelay=0x01
Power_AGC.upper0PowerThresh=0x0A
Power_AGC.upper1PowerThresh=0x02
Power_AGC.powerLogShift=1
```

```
Link.Talise.InitAgcPowerRegisters(Power_AGC)

################################################
# Program data structure to Talise, enable AGC #
################################################
Link.Talise.SetupRxAgc()

Link.Talise.SetRxGainControlMode(AgcSlowMode)

#Disconnect from the Zynq Platform
if(Connect == 1):
        Link.hw.Disconnect()
```

GAIN COMPENSATION, FLOATING POINT FORMATTER, AND SLICER

The user has the option to enable gain compensation. In gain compensation mode, the digital gain block compensates for the analog front-end attenuation. The cumulative gain across the device is 0 dB, for example, if 5 dB of analog attenuation is applied at the front end of the device, 5 dB of digital gain is applied. This cumulative gain ensures that the digital data is representative of the rms power of the signal at the receiver input port. Any internal front-end attenuation changes in the device to prevent the ADC from overloading are transparent to the BBP. This means that AGC can be used to react quickly to incoming blockers without the need for the BBP to track the current gain index to determine the gain setting of the device for received signal strength measurements.

The digital gain block is controlled by the gain table and a compensated gain table is required to operate in this mode. This type of gain table has a unique front-end attenuator setting with a corresponding amount of digital gain that is programmed at each index of the table. Note that the default gain table supplied with the API is not designed for gain compensation.

Gain compensation can be used in AGC or MGC mode. The maximum amount of gain compensation is 41.95 dB, which allows compensation of the internal analog attenuator and any external gain component (for example, a digital step attenuator (DSA) or low noise amplifier (LNA)). Considering an ADC with a 16-bit output, large amounts of digital gain increase the bit width of the path. Figure 91 shows a block diagram of the gain compensation portion of the receiver chain and shows the locations of the various blocks.

There are a number of modes that these blocks can be configured in.

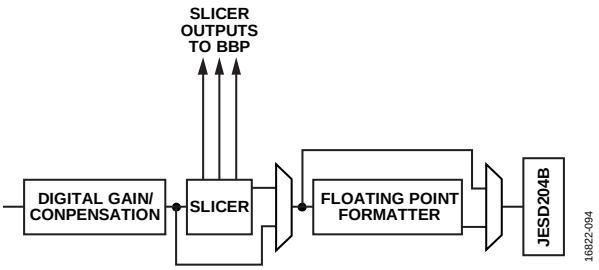

Figure 91. Gain Compensation, Floating Point Formatter, and Slicer Section of the Receiver Datapath

Mode 1—No Digital Gain Compensation

The device is configured to this mode by default. In this Mode 1, the digital gain block is not used for gain compensation. Instead, the digital gain block can be utilized to apply small amounts of digital gain/attenuation to provide consistent gain steps in a gain table. The premise is that because the analog attenuator does not have consistent stops in dB terms across its range, the digital gain block can be utilized to even out the steps for consistency (the default table utilizes the digital gain block to provide consistent 0.5 dB steps).

Neither the slicer nor the floating point formatter block is utilized. Because no significant amounts of gain compensation are applied, there is no bit width expansion of the digital signal. The signal is provided to the JESD204B port, which in turn sends the signal to the BBP in 12-bit, 16-bit, or 24-bit format.

Mode 2—Digital Gain Compensation with Slicer GPIO Outputs

In Mode 2, gain compensation is used with the device loaded with a gain table that compensates for the analog front-end attenuation applied. Considering 16-bit data at the input to the digital compensation block, as more digital gain is applied, the bit width of the signal is increased. With every 6 dB of gain, the bit width increases by 1. Figure 92 showns this effect with grey boxes indicating the valid (used) bits in each case.

Figure 92. Bit Width of Received Signal for Increasing Gain Compensation

The slicer takes (or slices) 16 MSBs of the data from each sample, regardless of input signal. The slicer moves based on the digital gain applied (for example, the MSBs are the MSBs of the largest permissible signal). The slicker then indicates its position, or the number of LSBs omitted from the sliced window over slicer outputs, as shown in Figure 93. These slicer outputs are outputted on GPIOs. When no digital gain is applied, the slicer takes Bits[D15:D0], passes these bits to the JESD204B block, and outputs Position 0 on the slicer output to the BBP. When digital gain is being applied, and the digital gain is less than 6 dB, the slicer selects Bits[D16:D1], passes these to the JESD204B block, and 1 is output on each of the GPIOs.

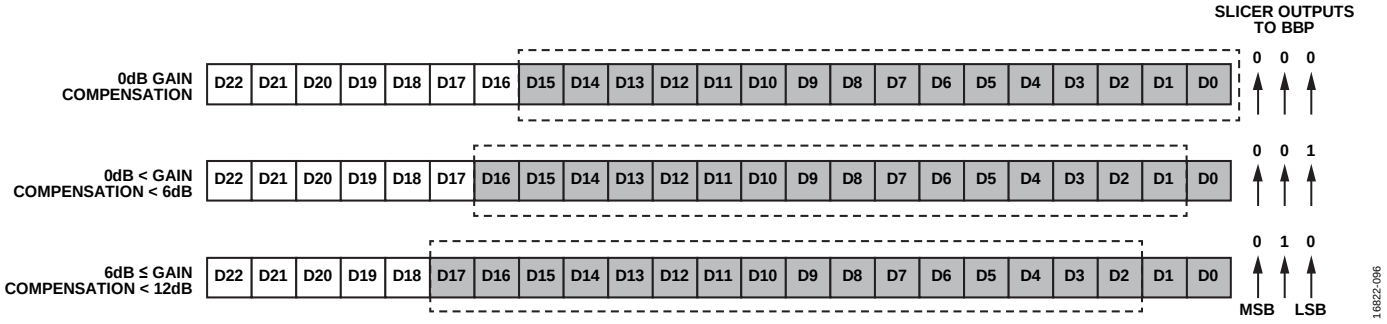

Figure 93. Slicer Bit Selection with Digital Gain

The BBP receives these 16 bits and uses the slicer output to scale the power of the received signal to determine the power at the input to the device (or at the input to an external gain element, if considered part of the digital gain compensation).

The slicer position vs. digital gain is described in Table 67 and the GPIOs used to indicate the Receiver 1 and Receiver 2 slicer positions are shown in Table 68, along with the GPIOs used to output the slicer position. The GPIOs must be enabled as outputs and configured for slicer output mode (see the GPIO Configuration section).

Table 67. Slicer GPIO Output vs. Digital Gain Compensation

Digital Gain Compensation (dB)	Slicer Position (Value Output on GPIO)
0	0
0 < DIG_GAIN < 6	1
6 ≤ DIG_GAIN < 12	2
12 ≤ DIG_GAIN < 18	3
18 ≤ DIG_GAIN < 24	4
24 ≤ DIG_GAIN < 30	5
30 ≤ DIG_GAIN < 36	6
36 ≤ DIG_GAIN ≤ 41.95	7

Table 68. GPIOs Used for Slicer Output Mode

	GPIO		
Receiver	MSB	Middle Position	LSB
Rx1	GPIO10	GPIO9	GPIO8
Rx2	GPIO14	GPIO13	GPIO12

Mode 3—Digital Gain Compensation with Embedded Slicer Position

Mode 3 is similar to Mode 2 because the slicer is used to select the 16 MSBs based on the amount of digital gain used by the currently selected gain index in the gain table. However, in this mode, the GPIO slicer outputs are not used. Instead, the slicer position (or number of trailing LSBs) are encoded in the data. There are a number of permissible ways that the trailing LSBs can be configured, which is controlled by the `intEmbeddedBits` parameter. The options are to place the slicer setting as 1 bit on both I and Q, or as 2 bits on both I and Q. These bits can be placed at the MSBs or at the LSBs. Table 69 shows the various modes that can be selected by the intEmbeddedBits function.

Table 69. intEmbeddedBits_t Parameter Description

intEmbeddedBits Parameter	Description
TAL_EMBED_1_SLICERBIT_AT_MSB	Embeds 1 slicer bit each on I and Q at the MSB position. See Figure 94.
TAL_EMBED_1_SLICERBIT_AT_LSB	Embeds 1 slicer bit each on I and Q at the LSB position. See Figure 95.
TAL_EMBED_2_SLICERBITS_AT_MSB	Embeds 2 slicer bits each on I and Q at the MSB positions. See Figure 96.
TAL_EMBED_2_SLICERBITS_AT_LSB	Embeds 2 slicer bits each on I and Q at the LSB position. See Figure 97.

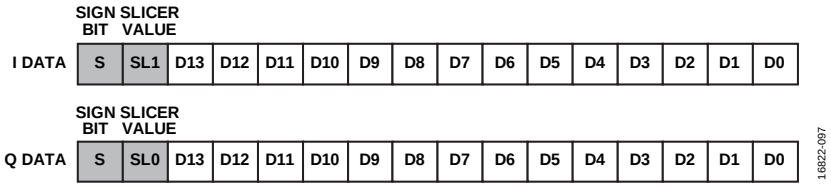

Figure 94. Encoding of Slicer Information as Control Bits (intembeddedbits = TAL_EMBED_1_SLICERBIT_AT_MSB)

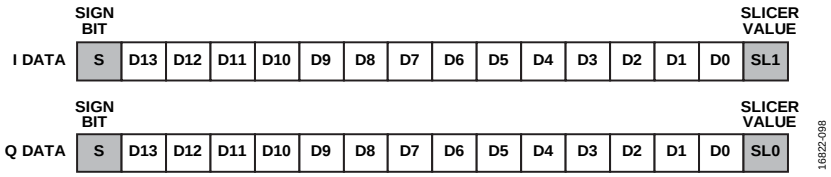

Figure 95. Encoding of Slicer Information as Control bIts (intEmbeddedBits = TAL_EMBED_1_SLICERBIT_AT_LSB)

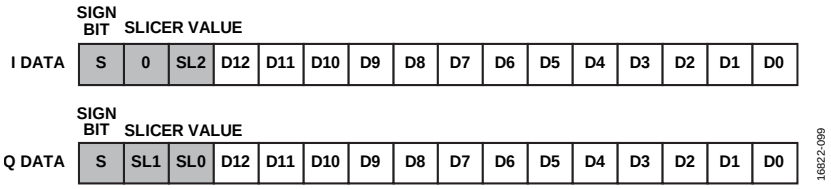

Figure 96. Encoding of Slicer Information as Control Bits (intEmbeddedBits = TAL_EMBED_2_SLICERBITS_AT_MSB)

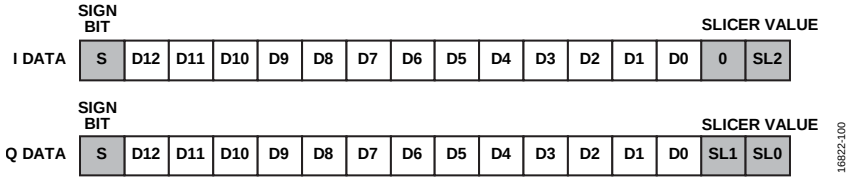

Figure 97. Encoding of Slicer Information as Control Bits (intEmbeddedBits = TAL_EMBED_2_SLICERBITS_AT_LSB)

Mode 4—Digital Gain Compensation and Slicer Input

In Mode 4, the user controls the slicer position. In Mode 2 and Mode 3, the slicer can be viewed as an attenuator, which reduces the signal level by 6 dB with each step, such that the signal can be sent across the JESD204B link. Mode 4 operates similarly; however, the position (amount of attenuation) is controlled externally, and the step sizes are no longer 6 dB. The valid step sizes are between 1 dB and 4 dB, and these sizes are controlled by the `extPinStepSize` parameter with the `TAL_EXTSLICER_STEPSIZE_1DB`, `TAL_EXTSLICER_STEPSIZE_2DB`, `TAL_EXTSLICER_STEPSIZE_3DB`, and `TAL_EXTSLICER_STEPSIZE_4DB` functions.

The slicer has three input pins (see Figure 91). The valid options are shown in Table 70. The value of these pins and the chosen step size set the level of slicer attenuation applied to the data before transmission across the JESD204B link, as shown in the following equation:

Slicer Attenuation = Slicer Input Pin Values × extPinStepSize

For example, if the value on the slicer input pins is 0'b111, and the step size is 2 dB, the slicer applies 14 dB (7 × 2 dB) of attenuation to the data.

Table 70. rx1GpioSelect and rx2GpioSelect description

Parameter Name	Pin Assignment	GPIOs		
		MSB	Middle Position	LSB
rx1GpioSelect	TAL_EXTSLICER_RX1_GPIO0_1_2	GPIO_2	GPIO_1	GPIO_0
	TAL_EXTSLICER_RX1_GPIO5_6_7	GPIO_7	GPIO_6	GPIO_5
	TAL_EXTSLICER_RX1_GPIO8_9_10	GPIO_10	GPIO_9	GPIO_8
rx2GpioSelect	TAL_EXTSLICER_RX2_GPIO11_12_13	GPIO_13	GPIO_12	GPIO_11
	TAL_EXTSLICER_RX2_GPIO5_6_7	GPIO_7	GPIO_6	GPIO_5

Mode 5—Digital Gain Compensation and Floating Point Formatting

The floating point formatter offers an alternative way of encoding the digitally compensated data onto the JESD204B link. In this mode, the data is converted to the IEEE 754 standard, half precision, floating point format (Binary 16). A slight loss in resolution occurs when using the floating point formatter, though resolution is distributed such that smaller numbers have higher resolution.

In Binary 16 floating point format, the number is composed on a sign bit (S), an exponent (E), and a significand (T). There are a number of options in terms of the number of bits that can be assigned to the exponent. More bits in the exponent result in a higher range, and can allow more digital compensation to be represented, and more bits in the significand provides higher resolution. The available options for the floating point formatter are as follows:

- 5-bit exponent, 10-bit significand
- 4-bit exponent, 11-bit significand
- 3-bit exponent, 12-bit significand
- 2-bit exponent, 13-bit significand

It is also possible to pack the data in the following formats (as shown in Figure 98):

- Sign, exponent, significand
- Sign, significand, exponent

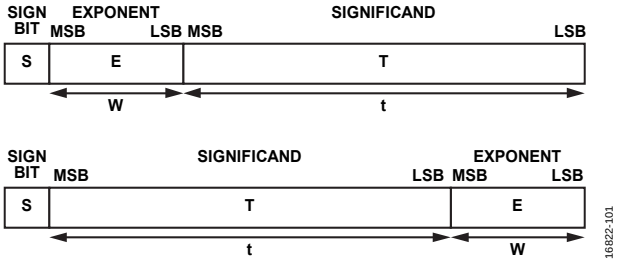

Figure 98. Floating Point Number Representation

In Figure 98, S is the sign bit, E is the value of the exponent, T is the value of the significand, w is the bit width of the exponent, and t is the bit width of the significand.

When the encoded floating point data is received, the user breaks up the Binary 16 number into its constituent parts. For the purposes of this explanation, consider a 3-bit exponent. In IEEE 754, the maximum exponent (0'b111 in this case) is reserved for the not a number value (NaN). The minimum exponent (0'b000) is used for a signed zero (E = 0, T = 0) and subnormal numbers (E = 0, T ≠ 0).

To decode a received floating point sample, use the following equations:

If E = 0 and T = 0,

$$Value = 0$$

If E = 0 and T ≠ 0,

$$Value = (-1)^{S} \times 2^{(E - Bias + 1)} \times (0 + 2^{1-p} \times T)$$

where:
Bias is used to convert the positive Binary values to exponents which allow for values both less than and greater than the full-scale of the ADC.
p is the precision of the mode ($p = t + 1$, because there are t significand bits coupled with a sign bit).

If E ≠ 0,

$$Value = (-1)^{S} \times 2^{(E - Bias)} \times (1 + 2^{(1-p)} \times T)$$

Table 71 provides the values to use in these equations for the various IEEE 754 supported modes.

Table 71. Floating Point Formatter, Supported IEEE 754 Modes

Exponent Bit Width (w)	Significand Bit Width (t)	Precision (p)	Bias
5	10	11	15
4	11	12	7
3	12	13	3
2	13	14	1

Figure 99 shows how the values of a waveform are encoded in floating point format. In this case, the maximum exponent (E bias) is 3, which means that data up to 24 dBFS of the ADC can be represented. When the signal reduces, the exponent required to represent each waveform value differs. This concept is different to the slicer, which bit shifts the data solely based on the applied digital attenuation and has a constant value for a constant digital gain. In this example, the floating point formatter interprets each value separately, after the digital gain compensation. Given the fixed precision of the significand and the sign bit, Figure 99 shows that there is higher resolution at lower signal levels than at higher signal levels, which preserves the SNR when the received signal strength is low.

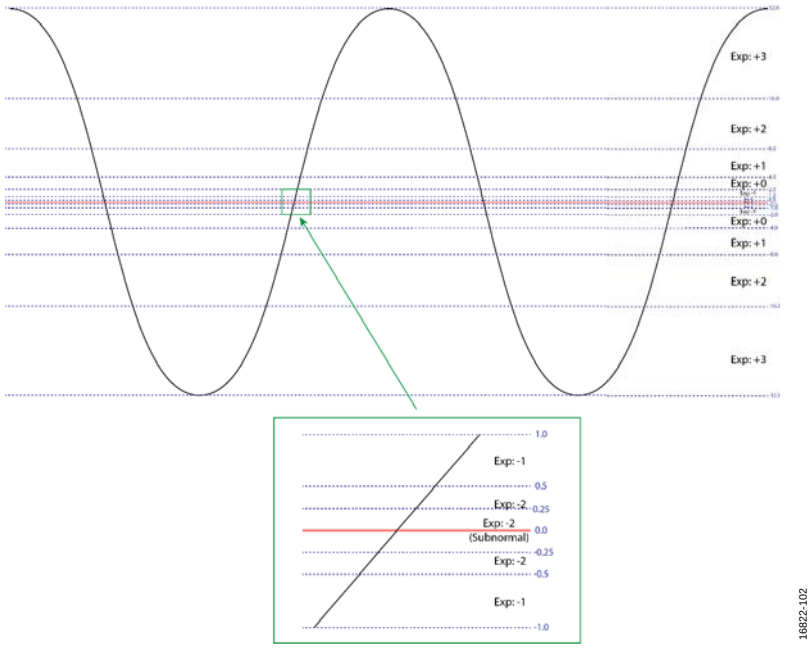

Figure 99. Visualization of the Floating Point Formatter Values

The floating point formatter also supports non-IEEE 754 modes, referred to as Analog Devices modes, where the largest exponent is not used to express NaN, in accordance with IEEE 754. It is unnecessary for the device to encode NaN because none of the data values can be NaN, and using this extra exponent value increases the largest value representable for a given exponent bit width.

Table 72. Exponent Bit Widths of IEEE 754 Modes and ADI Modes

Exponent Bit Width (w)	IEE 754 Mode Exponent Range (After Unbiasing)	Analog Devices Mode Exponent Range (After Unbiasing)
5	+15 to −14	+16 to −14
4	+7 to −6	+8 to −6
3	+3 to −2	+4 to −2
2	+1 to −1	+2 to −1

In the default floating point formatting format, the leading format is inferred and not encoded (for normal numbers). It is possible to enable a format where the leading one is encoded and stored in the MSB of the significand, which reduces the precision of the values.

If the user knows that the range of attenuation that is required for the worst case blocker (and therefore the digital gain required to compensate for the worst case blocker) exceeds the correction range allowed by the exponent width chosen, it is also possible to enable a fixed digital attenuation (from 6 dB to 42 dB) before the floating point formatter to ensure that the signal never exceeds the maximum range that can be encoded.

Receiver Data Format Data Structure

The configuration parameters for the floating point formatter and slicer are set up in the `taliseRxDataFormat_t` data structure (see Table 73 to Table 82).

Table 73. taliseRxDataFormat_t Data Structure Definition, formatSelect Parameter

Parameter Name	Data Type	formatSelect Comments	Format Comments
formatSelect	taliseDataFormattingModes_t	TAL_GAIN_COMPENSATION_DISABLED	No gain compensation (Mode 1)
		TAL_GAIN_WITH_FLOATING POINT	Gain compensation and floating point formatter enabled (Mode 5)
		TAL_GAIN_WITH_INTSLICER_NOGPIO	Gain compensation and slicer bits embedded on JESD204B signal (Mode 3)
		TAL_GAIN_WITH_INTSLICER	Gain compensation and slicer bits outputted on GPIOs (Mode 2)
		TAL_GAIN_WITH_EXTERNAL_SLICER	Gain compensation and slicer position inputted from GPIOs (Mode 4)
		For use in floating point mode; sets the round mode for the significand; settings are defined in the IEEE 754 specification, consult Section 4.3 in IEEE 754-2008	Gain compensation and slicer position inputted from GPIOs (Mode 4)

Table 74. taliseRxDataFormat_t Data Structure Definition, fpRoundMode Parameter

Parameter Name	Data Type	fpRoundMode Comments	Floating Point Rounding Mode
fpRoundMode	taliseFpRoundModes_t	TAL_ROUND_TO_EVEN	Floating point ties to an even value
		TAL_ROUNDTOWARDS_POSITIVE	Round floating point toward the positive direction
		TAL_ROUNDTOWARDS_NEGATIVE	Round floating point toward the negative direction
		TAL_ROUNDTOWARDS_ZERO	Round floating point toward the zero direction
		For use in floating point mode; sets the format of the 16-bit output on the JESD204B interface	Round floating point toward the zero direction

Table 75. taliseRxDataFormat_t Data Structure Definition, fpDataFormat and fpEncodeNan Parameters

Parameter Name	Data Type	fpDataFormat Comments	Floating Point Data Format
fpDataFormat	uint8	t 0 1 For use in floating point formatter mode; if set to 1, the floating point formatter reserves the highest value of the exponent for NaN to be compatible with the IEEE 754 specification; setting this parameter to 0 increases the range of the exponent by 1 (Analog Devices mode)	Not applicable {Sign, exponent, significand} {Sign, significand, exponent} {Sign, significand, exponent}
fpEncodeNan	uint8_t	Not applicable	For use in floating point formatter mode; used to indicate the number of exponent bits in the floating point number according to the following settings shown in Table 76

Table 76. taliseRxDataFormat_t Data Structure Definition, fpNumExpBits and fpHideLeadingOne Parameters

Parameter Name	Data Type	fpNumExpBits Comments	No. of Exponent Bits	No. of Significand Bits	No. of Sign Bits
fpNumExpBits	taliseFpExponentModes_t	0	2	Not applicable	Not applicable
		1	3	13	1
		2	4	12	1
		3	5	11	1
		For use in floating point formatter mode; setting to 1 hides the leading one in the significand to be compatible to the IEEE 754 specification (IEEE mode); clearing causes the leading one to be at the MSB of the significand	5	10	1
fpHideLeadingOne	uint8_t	Not applicable	For use in floating point formatter mode; attenuates integer data on Rx1 when floating point mode enabled; attenuation values for individual settings are shown in Table 77	Not applicable	Not applicable

Table 77. taliseRxDataFormat_t Data Structure Definition, fpRx1Atten Parameter

Parameter Name	Data Type	fpRx1Atten Comments	Attenuation (dB)
fpRx1Atten	taliseFpAttenSteps_t	TAL_FPATTEN_0DB	0
		TAL_FPATTEN_MINUS6DB	−6
		TAL_FPATTEN_MINUS12DB	−12
		TAL_FPATTEN_MINUS18DB	−18
		TAL_FPATTEN_24DB	+24
		TAL_FPATTEN_18DB	+18
		TAL_FPATTEN_12DB	+12
		TAL_FPATTEN_6DB	+6
		For use in floating point formatter mode; attenuates integer data on Rx1 when floating point mode enabled; attenuation values for individual settings are shown in Table 78	+6

Table 78. taliseRxDataFormat_t Data Structure Definition, FpRx2Atten Parameter

Parameter Name	Data Type	fpRx2Atten Comments	Attenuation (dB)
fpRx2Atten	taliseFpAttenSteps_t	TAL_FPATTEN_0DB	0
		TAL_FPATTEN_MINUS6DB	−6
		TAL_FPATTEN_MINUS12DB	−12
		TAL_FPATTEN_MINUS18DB	−18
		TAL_FPATTEN_24DB	24
		TAL_FPATTEN_18DB	18
		TAL_FPATTEN_12DB	12
		TAL_FPATTEN_6DB	6
		For use in slicer modes; sets the integer number of embedded slicer bits to embed in Rx data sample and bit position to embed them (see the Mode 3—Digital Gain Compensation with Embedded Slicer Position section)	6

Table 79. taliseRxDataFormat_t Data Structure Definition, intEmbeddedBits Parameter

Parameter Name	Data Type	intEmbeddedBits Comments	Slicer Bit Embedded Position in Data Frame
intEmbeddedBits	taliseEmbeddedBits_t	0	Disabled all embedded slicer bits
		1	Embeds 1 slicer bit on I, 1 slicer bit on Q and the MSB position
		2	Embeds 1 slicer bit on I and 1 slicer bit on Q and the LSB position
		3	Embeds 2 slicer bits on I and 2 slicer bits on Q and the MSB position
		4	Embeds 2 slicer bits on I and 2 slicer bits on Q and the LSB position
		Sets the integer sample resolution selecting either 12, 16, or 24 bits of data with either twos complement or signed magnitude	Embeds 2 slicer bits on I and 2 slicer bits on Q and the LSB position

Table 80. taliseRxDataFormat_t Data Structure Definition, intSampleResolution Parameter

Parameter Name	Data Type	intSampleResolution Comments	Resolution of Integer Sample
intSampleResolution	taliseIntSampleResolution_t	0	12-bit resolution with twos complement
		1	12-bit resolution with signed magnitude
		2	16-bit resolution with twos complement (default)
		3	16-bit resolution with signed magnitude
		4	24-bit resolution with twos complement
		5	24-bit resolution with signed magnitude
		For use in slicer modes; used in gain compensation with external slicer control (Mode 4); sets the slicer step value that is used with this external control mechanism	24-bit resolution with signed magnitude

Table 81. taliseRxDataFormat_t Data Structure Definition, extPinStepSize Parameter

Parameter Name	Data Type	extPinStepSize Comments	Slicer Step Size (dB)
extPinStepSize	taliseGainStepSize_t	TAL_EXTSLICER_STEPSIZE_1DB	1
		TAL_EXTSLICER_STEPSIZE_2DB	2
		TAL_EXTSLICER_STEPSIZE_3DB	3
		TAL_EXTSLICER_STEPSIZE_4DB	4
		For use in slicer Mode 4; selects which GPIO pins are used for the slicer to output its position on (Mode 2) or to control the slicer position with (Mode 4); see Table 68 for a full list of possible GPIO permutations	4

Table 82. taliseRxDataFormat_t Data Structure Definition, rx1GpioSelect, Rx2GpioSelect, externalLnaGain, and tempCompensationEnable Parameters

Parameter Name	Data Type	Comments
rx1GpioSelect	taliseRx1ExtSlicerGpioSelect_t	For use in slicer Mode 4; selects which GPIO pins are used for the slicer to output its position on (Mode 2), or to control the slicer position with (Mode 4); see Table 68 for a full list of possible GPIO permutations.
rx2GpioSelect	taliseRx2ExtSlicerGpioSelect_t	For use in dual band modes; not supported
externalLnaGain	uint8_t	Not supported
tempCompensationEnable	uint8_t	Not applicable

To configure the receiver data format, call the following API function:

```
TALISE_setRxDataFormat (taliseDevice_t *device, taliseRxDataFormat_t *rxDataFormat)
```

Note that for the slicer to properly use the GPIO, the GPIO pins must be configured appropriately during the GPIO setup. See the GPIO Slicer Features section for details.

RECEIVER DC OFFSET CALIBRATION

This section discusses the receiver and observation receiver dc offset correction algorithms within the transceiver. The device receivers implement a direct conversion receiver architecture. To minimize dc contributions from various sources within the transceiver, a dc offset correction algorithm minimizes dc offset in the analog circuitry and the output data that is sent to the BBP. The dc offset sources include LO self mixing and amplifier dc offsets within the datapath.

This section provides details on receiver dc offset correction circuitry, configuration options available to the customer, and API programming instructions.

RECEIVER DC OFFSET CORRECTION CIRCUITRY

The dc offset configuration occurs within the normal device initialization sequence. Additional API commands are not necessary to enable the dc offset correction. However, some API commands are provided to modify the dc offset correction behavior, if required.

The dc offset correction is a hardware algorithm that does not directly involve the Arm processor. The dc offset correction circuitry involves an RF (analog) dc offset correction and digital dc offset correction. The RF dc offset correction provides coarse dc offset correction to reduce dc offset levels in the analog circuitry. The digital dc offset correction cleans up residual dc offset detected prior to the JESD204B interface.

The observation point for RF and digital dc offset correction is at the end of the digital path after all digital filtering and the digital gain block. The observation point and the correction points for RF and digital dc offset is shown in the Figure 100.

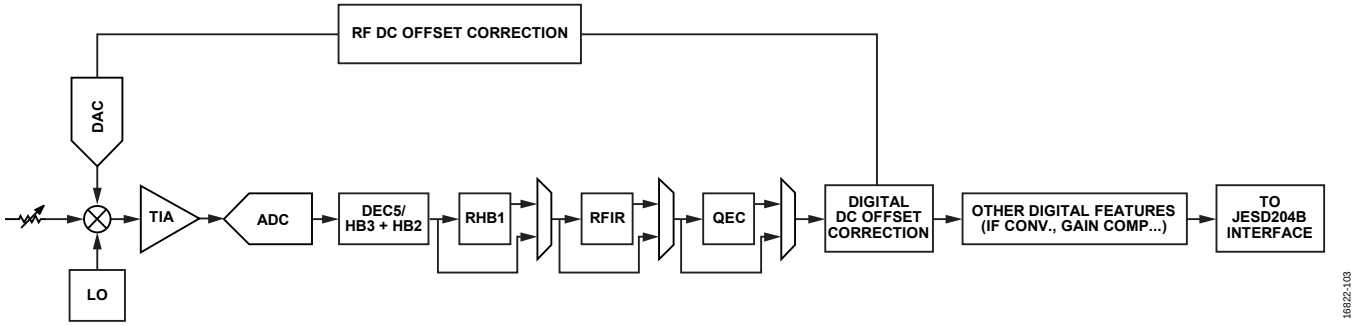

Figure 100. DC Offset Circuitry Within the Receiver Datapath

The RF dc offset correction and digital dc offset correction are enabled by default for all receiver and observation receiver channels. This default condition is not configurable through the API.

The digital dc offset correction is enabled on the receiver channels. The digital dc offset correction is not enabled for the observation receiver channels.

RF (Analog) DC Offset Correction

The RF (analog) dc offset correction circuitry is responsible for a coarse correction of the dc offset in the datapath. The observation point is at the end of the digital path, after all digital filtering and the digital gain block. This dc offset correction is applied via a dedicated correction DAC located prior to the TIA, which injects an offset voltage to cancel the dc offset that the algorithm observes.

RF DC Offset Initialization Calibration

The Arm processor is only used to start the dc offset initialization calibration and configure the register settings for the dc offset tracking calibrations. The dc offset initialization calibration generates an initial set of corrections to use at all gain indices. The correction value stored for each gain index automatically switches into the correction DAC when the gain index is changed, which implies that the hardware maintains a look-up table of RF dc offset corrections across gain indices. The dc offset initialization calibration provides a reasonable starting point for the RF dc offset tracking calibration.

If the dc offset initialization calibration is not performed, dc offset performance significantly underperforms the specifications found in the ADRV9008-1 and ADRV9009 data sheets.

RF DC Offset Tracking Calibration

The RF dc offset tracking calibration is a type of tracking calibration that maintains the dc offset performance during the operation of the device. Note that because the Arm processor is not used in data processing, dc offset configuration is not included in the Arm tracking calibration mask enumerations. The RF dc offset calibration updates its correction in response to changes in the estimated dc offset that is present in the datapath at regular intervals. Modifications to the RF dc offset tracking calibration behavior is not possible through the API.

The RF dc offset corrections are updated by adding or subtracting to/from the current correction word. A high level depiction of this update process is shown Figure 101.

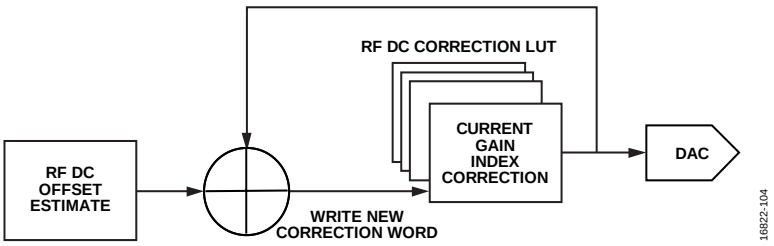

Figure 101. RF DC Offset Tracking Update Process

Corrections for each channel (Receiver 1, Receiver 2, Observation Receiver 1, and Observation Receiver 2) are stored independently of one another. Additionally, I/Q datapath corrections are independently stored because of potential differences in the inherent dc offset of the datapath.

Digital DC Offset Correction

The digital dc offset correction circuitry is responsible for reducing the residual dc offset that is uncorrected by the RF dc offset correction. The digital dc offset correction estimates the residual dc offset to generate an equal and opposite correction value to the dc condition present. This offset correction is added or subtracted into the digital datapath to further minimize dc energy. This offset correction is a tracking calibration that adapts the correction to the dc offset observed in the path.

Unlike the RF dc offset correction, the digital dc offset correction does not maintain a storage table for dc offset correction values. There is also no initialization calibration related to digital dc offset. If a gain change occurs that generates a different dc offset condition, the digital dc offset correction takes a finite period of time to converge to an optimal correction value. The time to convergence is set, in part, by the `mShift` value described in the mShift section.

The digital dc offset correction is enabled on the receiver channels. The digital dc offset correction is not enabled for the observation receiver channels.

mShift

Note that the digital dc offset correction effectively creates a narrow-band notch filter around the dc, which can lead to degradation of the subcarriers on or close to dc depending on the subcarrier spacing. If the default dc offset settings are causing this type of degradation, the API provides commands to adjust the filtering behavior near the dc.

The `mShift` parameter controls the frequency corner for the narrowband notch filter. The `mShift` parameter can be set with the `TALISE_setDigDcOffsetMShift()` command. Note that larger `mShift` values correspond to narrower notch filter bandwidth around the dc. A narrower notch filter corresponds to an increased convergence time for the digital dc offset calibration. The notch filter bandwidth is typically set on the order of 10s of kHz.

The Arm processor selects an `mShift` value depending on the profile sample rate. If this selected `mShift` value is not desired, call the `TALISE_setDigDcOffsetMShift()` command before the dc offset initialization calibration occurs to change the `mShift` value. See the DC Offset API Functions section for more information.

DC Offset API Functions

The following section describes the dc offset API commands available in the API.

TALISE_setDigDcOffsetMShift()

This function sets the receiver/observation receiver channel digital dc offset convergence time (`mShift`). The function is as follows:

```
uint32_t TALISE_setDigDcOffsetMShift(taliseDevice_t *  device, taliseDcOffsetChannels_t
channel, uint8_t  mShift)
```

This function allows the BBIC to adjust the digital dc offset convergence time. This value (`mShift`) affects the corner frequency of the notch filter that is used to filter the dc offset out of the receiver/observation receiver receive signal. The `mShift` value applies to all receiver channels or all observation receiver channels, but the receiver and observation receiver can be set individually.

In software versions older than Arm 4.0, the Arm initialization calibration for dc offset overrides the `mShift` setting. The BBIC calls this function after running the initialization calibrations to ensure that the Arm processor does not overwrite a custom `mShift` setting.

In software versions of Arm 4.0 or newer, the Arm bootup sets a default calculated value for the `mShift` based on the selected profile sample rates. The BBIC can set the `mShift` any time after the Arm boots up (before or after Arm initialization calibrations). The Arm dc offset initialization function saves and restores the custom `mShift` value if the Arm processor must change the `mShift` value.

Digital dc offset tracking is not enabled on the observation receiver channels. Applying this command to the observation receiver channel does not have a meaningful effect.

Parameters include the following:

- `*device` is a pointer to the device data structure.
- `channel` is a receiver channel select. Refer to Table 83 for a `taliseDcOffsetChannels_t` enumeration definition.
- `mShift` is a value to set for the given channel (valid value range is 8 to 20).

Table 83. Enumerated Values for taliseDcOffsetChannels_t data type.

Enum Name (taliseDcOffsetChannels_t)	Value	Enumerator Description
TAL_DC_OFFSET_RX_CHN	0x0	Select Rx channel
TAL_DC_OFFSET_ORX_CHN	0x1	Select ORx channel (if selected, this command has no meaningful effect)

TALISE_getDigDcOffsetMShift()

This function retrieves the receiver/observation receiver channel digital dc offset convergence time (`mShift`). The function is as follows:

```
uint32_t TALISE_getDigDcOffsetMShift(taliseDevice_t* device, taliseDcOffsetChannels_t channel,
uint8_t* mShift)
```

This function allows the BBIC to read back the digital dc offset convergence time. This value (`mShift`) affects the corner frequency of the notch filter that is used to filter the dc offset out of the receiver/observation receiver receive signal. The `mShift` value applies to all receiver channels or all observation receiver channels, but the receiver and the observation receiver can be set individually.

In software versions older than Arm 4.0, the Arm initialization calibration for dc offset overrides the `mShift` setting. The BBIC calls this function after running the initialization calibrations to ensure that the Arm processor does not overwrite a custom `mShift` setting.

In software versions of Arm 4.0 or newer, the Arm bootup sets a default calculated value for the `mShift` based on the selected profile sample rates. The BBIC can set the `mShift` any time after the Arm boots up (before or after the Arm initialization calibrations). The Arm dc offset initialization function saves and restores the custom `mShift` value if the Arm processor must change the `mShift` value.

Parameters include the following:

- `*device` is a pointer to the device data structure.
- `channel` is the digital dc offset channel to read `mShift` value for.
- `*mShift` is a pointer to store current `mShift` for the requested channel.

QEC, CALIBRATION, AND ARM CONFIGURATION

The device comes with a built in Arm processor that is tasked with performing some initial calibrations of the signal paths of the device, as well as maintaining QEC and LO leakage performance during device operation through tracking algorithms. It is useful to refer to the System Control section and the Use Cases section when reviewing this section of the reference manual.

ARM STATE MACHINE OVERVIEW

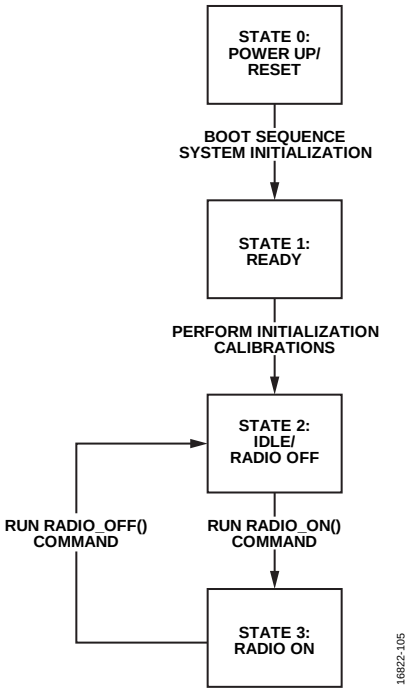

Figure 102. Arm State Machine

State 0

When the Arm core is powered up, the Arm processor moves into power-up/reset state. At this point, an Arm image is loaded. See the Loading the Arm Processor section for details. When the Arm image is loaded, the Arm can be enabled and begins its boot sequence.

State 1

When the Arm is successfully booted, it enters the ready state. In this state, the Arm processor can receive configuration settings or commands (instructions), for example, performing the initialization calibrations of the device.

State 2

After the initial calibrations are performed, the Arm processor enters the idle state. In this state, the Arm processor can receive configuration settings, for example, which tracking calibrations must be enabled.

State 3

When the required tracking calibrations are enabled, a `Radio_On()` command is provided to the Arm, which moves the processor into State 3. In this state, the Arm scheduler is active, and the Arm runs tracking calibrations when the necessary signal chains are available. The RF paths of the device are also made available for use.

LOADING THE ARM PROCESSOR

When the device is powered-up or reset, it is necessary for the Arm image to be loaded to the device (this is towards the end of the initialization process, see an initialization script for further details). Prior to loading the Arm image, the Arm core is reset and prepares to receive its image with the following function:

```
TALISE_initArm(taliseDevice_t *device, taliseInit_t *init)
```

Parameters include the following:

- `*device` is the structure pointer to the data structure.
- `*init` is a point to the initialization data structure.

After this function is run, the Arm image is loaded with the following function:

```
TALISE_loadArmFromBinary(taliseDevice_t *device, uint8_t *binary, uint32_t count)
```

Parameters include the following:

- `*binary` is a pointer to the byte array containing the Arm program memory bytes.
- `count` is the number of bytes in this binary array.

The Arm image is provided in the resources folder of the GUI install. Note that there is a separate Arm image for each device version. The image for the ADRV9008-1 is **TaliseRxArmFirmware.bin**, the ADRV9008-2 image is **TaliseTxArmFirmware.bin**, and the ADRV9009 image is **TaliseTDDArmFirmware.bin**.

When the Arm image is loaded, the Arm automatically begins its boot sequence. As part of the boot sequence, the Arm calculates a checksum for the image that has been loaded. The following API function verifies the Arm load has been completed successfully:

```
TALISE_verifyArmChecksum(taliseDevice_t *device)
```

This function ensures that the boot sequence has completed before reading back the calculated checksum from the relevant Arm memory location. The function compares this checksum to the precalculated checksum that is embedded in the Arm image. A successful load is verified when the checksums are equal.

ADRV9008-1, ADRV9008-2, AND ADRV9009 INITIAL CALIBRATIONS

The Arm processor in the device is tasked with scheduling/performing initial calibrations to optimize the performance of the signal paths prior to device operation. These calibrations are called by the following API function:

```
TALISE_runInitCals(taliseDevice_t *device, uint32_t calMask)
```

Parameters include the following:

- `calMask` is a 32-bit mask that informs the Arm processor of which calibrations to run.

Table 84 shows the bit assignments of the calibration mask. The Arm processor runs the selected initial calibration for each enabled channel.

Table 84. Calibration Mask Bit Assignments

Calibration Mask Bit(s)	Corresponding Enumerator	Calibration	Description
D0	TX_BB_FILTER	Tx baseband filter calibration	Tunes the corner frequency of the Tx baseband filter.
D1	ADC_TUNER	ADC tuner calibration	Configures the ADC for the required profile bandwidth.
D2	TIA_3DB_CORNER	Rx TIA filter calibration	Tunes the corner frequency of the Rx TIA filter.
D3	DC_OFFSET	Rx dc offset calibration	Corrects for dc offset within the Rx chain.
D4	TX_ATTENUATION_DELAY	Tx attenuation delay	Calculates the path delay between the Tx analog and Tx digital attenuation blocks. This delay is then used to offset the onset of Tx analog and Tx digital attenuations relative to each other to compensate for the path delay between these blocks.
D5	RX_GAIN_DELAY	Rx gain delay	Calculates the path delay between the Rx analog and Rx digital attenuation blocks. This delay is then used to offset the onset of Rx analog and Rx digital attenuations relative to each other to compensate for the path delay between these blocks.

Calibration Mask Bit(s)	Corresponding Enumerator	Calibration	Description
D6	FLASH_CAL	ADC flash calibration	Optimally configures the ADC flash converters.
D7	PATH_DELAY	Path delay calibration	Computes the Tx to loopback path delay that is required for the Tx QEC and Tx LO leakage algorithms.
D8	TX_LO_LEAKAGE_INTERNAL	Tx LO leakage initial calibration	Performs an initial LO leakage calibration for the Tx path and utilizes the Tx path and the internal loopback path (see Figure 108).
D9	TX_LO_LEAKAGE_EXTERNAL	Tx LO leakage external initial calibration	Performs an initial external LO leakage calibration for the Tx path and utilizes the Tx path, a required external loopback path, and the ORx path (see Figure 109).
D10	TX_QEC_INIT	Tx QEC initial calibration	Performs an initial QEC calibration for the Tx path and utilizes the Tx path and an internal loopback path (see Figure 108).
D11	LOOPBACK_RX_LO_DELAY	Loopback ORx LO delay	Performs an LO delay calibration for the loopback path.
D12	LOOPBACK_RX_RX_QEC_INIT	Loopback Rx QEC initial calibration	Performs an initial QEC calibration for the Rx path.
D13	RX_LO_DELAY	Rx LO delay	Performs an LO delay calibration for the Rx path. Do not use this calibration.
D14	RX_QEC_INIT	Rx QEC initial calibration	Performs an initial QEC calibration for the Rx path.
D15	RX_PHASE_CORRECTION	Rx phase correction	Performs a phase correction calibration for the Rx path attenuator.
D16	ORX_LO_DELAY	ORx LO delay	Performs an LO delay calibration for the ORx path. Do not use this calibration.
D17	ORX_QEC_INIT	ORx QEC initial calibration	Performs an initial QEC calibration for the ORx path.
D18	TX_DAC	Tx DAC initial calibration	Performs a calibration of the Tx DAC.
D19	ADC_STITCHING	ADC stitching initial calibration	Used for wideband ORx modes where the output of two ADCs are stitched together to form a single quadrature channel. In this mode, a total of four ADCs are used to produce the final I/Q outputs. For a description of this, see the Observation Receiver section of this document.
[D20:D31]		Not used	Not applicable.

The calibration mask can be created using a bit map from Table 84, or by utilizing the appropriate enums. For example, the following function enables the ADC tuner and ADC flash calibration in a calibration mask to be transferred to the `TALISE_runInitCals()` function:

```
unit32_t initCalMask = ADC_TUNER | FLASH_CAL;
```

These calibrations follow a specific order. The Arm processor proceeds through these calibrations in this sequential order. The user must wait for these routines to complete before continuing the configuration of the device. Use the following function to verify that these initial calibrations have been completed by the Arm processor:

```
TALISE_waitInitCals(taliseDevice_t *device, uint32_t timeoutMs, uint8_t *errorFlag)
```

Parameters include the following:

- `timeoutMs` is the time in ms that the function must wait for the calibrations to complete before returning an error.
- `*errorFlag` is a 3-bit flag that indicates an Arm error occurred when running the initial calibrations. Refer to the Initialization Calibration Errors section for details.

This function implements a blocking wait until the initial calibrations have been completed. The following is an alternative function that can be used to determine if the initial calibrations are still running:

```
TALISE_checkInitCalComplete(taliseDevice_t *device, uint8_t *areCalsRunning, uint8_t *errorFlag)
```

Parameters include the following:

- `areCalsRunning` is a value to indicate if calibrations are still running (0 = initial calibrations have completed, 1 = initial calibrations are still running).
- `*errorFlag` is the same value as the value that is returned in `waitInitCals`. See the Initialization Calibration Errors section for details.

This alternative function allows the user to avoid a blocking wait and use the wait time for other system functionality. The user must call this function until the function reports that the initial calibrations are complete. During this time, this function is the only communication to the device until the initial calibrations have completed. The user must only proceed with the device configuration after the initial calibrations are complete.

Note that there are requirements on a system level for these initialization calibrations to perform successfully. See the System Considerations for Initial Calibrations section for details.

ADRV9008-1, ADRV9008-2, AND ADRV9009 TRACKING CALIBRATIONS

The Arm processor ensures that QEC and LO leakage (and HD2 for GSM applications) corrections are optimal throughout device operation, for example, over time, attenuation, and temperature. The processor achieves optimal correction performance by performing calibrations at regular intervals. These calibrations are referred to as tracking calibrations and utilize normal traffic data to update the path correction coefficients.

The following function enables the tracking calibrations in the Arm processor:

```
TALISE_enableTrackingCals(taliseDevice_t *device, uint32_t enableMask)
```

Parameters: `enableMask` is a 10-bit mask that informs the ARM processor which calibrations to run.

Table 85 shows the bit assignments of the enable mask. The following function is also an equivalent function to read the tracking calibrations that are enabled, which uses the same mask as the preceding function:

```
TALISE_getEnabledTrackingCals(taliseDevice_t *device, uint32_t *enableMask)
```

Table 85. Tracking Calibrations Enable Mask Bit Assignments

Calibration Mask Bit	Function
D0	Rx1 QEC tracking
D1	Rx2 QEC tracking
D2	ORx1 QEC tracking
D3	ORx2 QEC tracking
D4	Tx1 LO leakage tracking
D5	Tx2 LO leakage tracking
D6	Tx1 QEC tracking
D7	Tx2 QEC tracking
D8	Rx1 HD2 tracking
D9	Rx2 HD2 tracking

Run the `TALISE_enableTrackingCals()` function before the Arm is moved to the radio on state. Do not run the `TALISE_enableTrackingCals()` function when the device is operational (Arm is in the radio on state). The following function can be used to suspend or resume tracking calibrations when the device is in the radio on state:

```
TALISE_setAllTrackCalState(taliseDevice_t *device, uint32_t calSubsetMask, uint32_t
resumeCalMask)
```

Parameters include the following:

- `calSubsetMask` is a mask that indicates which calibrations the `resumeCalMask` must control.
- `resumeCalMask` is a mask of the calibrations to be paused or resumed.

The bit assignments for the calibration mask are shown in Table 86.

Table 86. Transmitter Tracking Calibration Pause/Resume Bit Assignments

Calibration Mask Bit	Function
D0	Unused
D1	Unused
D2	Unused
D3	Unused
D4	Tx1 LO leakage tracking
D5	Tx2 LO leakage tracking
D6	Tx1 QEC tracking
D7	Tx2 QEC tracking

If the corresponding bit of a calibration is set to 1 in the `calSubsetMask`, this calibration can be paused or resumed using the `resumeCalMask` function. If the corresponding bit is set to 0, the `resumeCalMask` cannot control the pausing/resuming of the calibration. If the corresponding bit of a calibration is set to 1 in the `resumeCalMask`, the calibration resumes. If the corresponding bit is set to 0, the calibration is paused.

Note that only calibrations that have initially been enabled when using the `TALISE_enableTrackingCals()` prior to moving the device into the radio on state can be paused and resumed using this function. It is not possible to enable a tracking calibration that was not initially enabled at the last entry into the radio on state.

To get the current status of the calibration mask (paused/enabled), use the following function:

```
TALISE_getAllTrackCalState(taliseDevice_t *device, uint32_t *resumeCalMask)
```

Parameter: `resumeCalMask` is a bit mask composed as shown in Table 85, with a 1 indicating that the corresponding calibration is enabled and a 0 indicating the calibration is paused.

The Arm processor schedules the tracking calibrations. No user input is required to initiate a tracking calibration. The Arm processor schedules its calibrations based on the periodicity required for each calibration. Transmit tracking calibrations are only run at times when the user advises that the observation receiver path is available to the Arm processor for calibrations. See the ADRV9008-1, ADRV9008-2, and ADRV9009 Tracking Calibration Scheduler section and the System Control section for details.

The receiver and observation receiver channels also have dc correction tracking, which is active when these channels are being utilized. This calibration is not an Arm-based calibration.

ADRV9008-1, ADRV9008-2, AND ADRV9009 TRACKING CALIBRATION SCHEDULER

The Arm processor schedules the tracking calibrations based on the periodicity required for each calibration. No user input is required to initiate a tracking calibration. Receive calibrations are only run when the receive chains are enabled. Transmit tracking calibrations also require the user to assign the observation receiver path to the Arm processor for calibrations for a specified proportion of time to allow the transmitter data to be observed.

When the device is initialized, the Arm processor enters the idle/radio off state. When the Arm processor is in idle/radio off state, the device is not transmitting/receiving data. The Arm processor must be in the radio on state with the tracking calibrations enabled for the device to transmit/receive data. See Figure 102 for an overview of the Arm state machine.

In the radio off state, the scheduler is not active. The signal chains are powered down, and the device is not receiving or transmitting data.

In the radio on state, the scheduler is active, and tracking calibrations are run. The signal chains are available for use (see the System Control and Use Cases sections for details).

The following functions advise the Arm processor to move to the radio off and radio on states:

```
TALISE_radioOn(taliseDevice_t *device)
TALISE_radioOff(taliseDevice_t *device)
```

When the Arm processor moves the state machine into the radio on state, the tracking calibration scheduler is initialized. The required tracking calibrations must be specified prior to calling the `TALISE_radioOn()` function.

Use the following function to determine which state the Arm processor is in:

`TALISE_getRadioState(taliseDevice_t *device, uint32_t *radioStatus)`

In this function, `*radioStatus` indicates the current Arm processor state as indicated by the values in Table 87.

Table 87. Radio Status and Corresponding Arm Functions

Radio Status	Arm Function
0	Power-up/reset
1	Ready
2	Radio off
3	Radio on
>3	Arm error, check profile configuration

When the device is the radio on state, the Arm scheduler performs the tracking calibrations on a periodic basis, ensuring that the correction values are optimal. For each tracking calibration enabled in the tracking calibration mask, a corresponding calibration task is initiated when the Arm processor is moved into the radio on state, as shown in Figure 103.

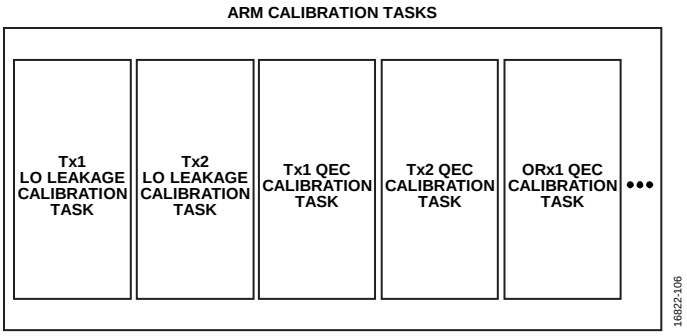

Figure 103. Calibration Tasks Run in the Arm Processor

Each calibration task follows the same sequence of processes, and is responsible for indicating to the scheduler when a task must be run through its own pending bit, as shown in Figure 104. This bit is periodically set by the calibration task when the calibration timer expires.

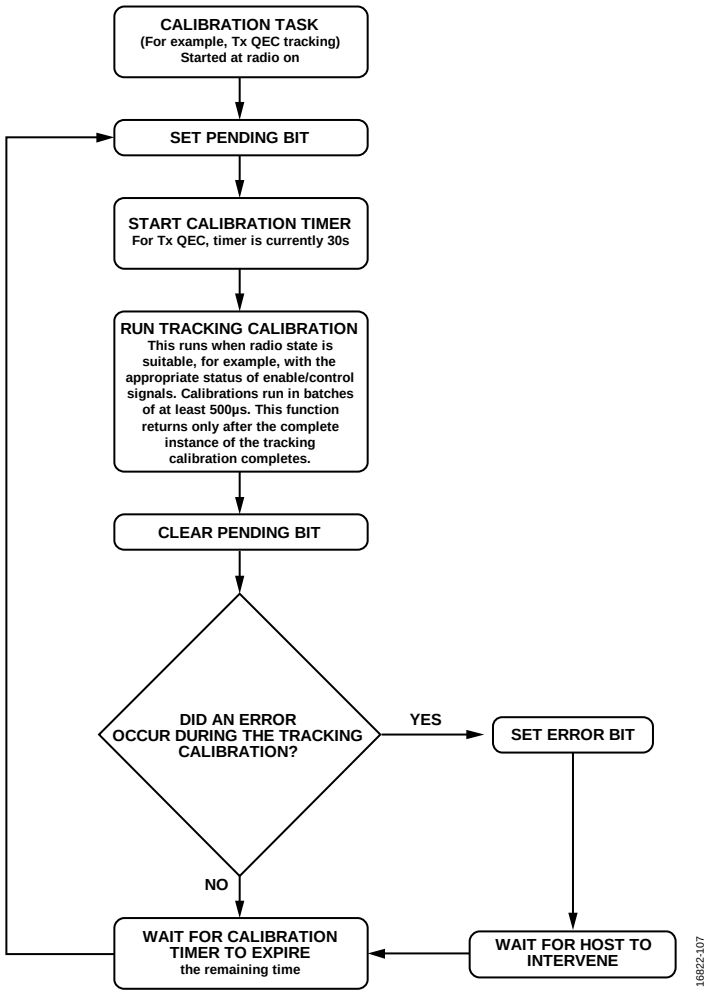

Figure 104. Single Process Scheduling Flow Diagram

The pending bits can be readback from the device. See the Tracking Calibration Monitoring section for details.

The scheduler runs each calibration when the corresponding pending bit is set; however, more than one calibration task can be pending at any one time.

The scheduler determines which calibration task to run at any time based on three conditions: pending bits, priority, and availability of the required paths.

Condition 1, Pending Bits

The scheduler reads the pending bits and determines which calibrations are requesting to run.

Condition 2, Priority

Each calibration task is given its own priority level. The calibration of the highest priority is given preference (highest priority being 1). The order of the priorities is shown in Table 88.

Table 88. Priority Levels of the Calibration Tasks

Priority Level	Calibration Task
1	Tx LO leakage
2	Tx QEC
3	ORx QEC
4	Rx QEC

Note that there is no set priority between the individual channels calibrations, for example, Transmitter 1 LO leakage and Transmitter 2 LO leakage. For calibration tasks of the same priority, the scheduler prioritizes the calibration task that was completed the longest time ago.

Condition 3, Availability of Required Paths

The scheduler also determines if the calibration task can be performed. For example, as shown in Figure 104, the transmitter QEC task needs the transmitter to be enabled and the observation receiver to be assigned to the Arm calibrations. If both conditions are not true, the calibration cannot be run. The scheduler determines these conditions and if the calibration cannot run, the scheduler continues through the priority list to find a calibration that is pending and can be run, for example, Receiver 1 QEC. See the System Considerations for Tracking Calibrations section for details.

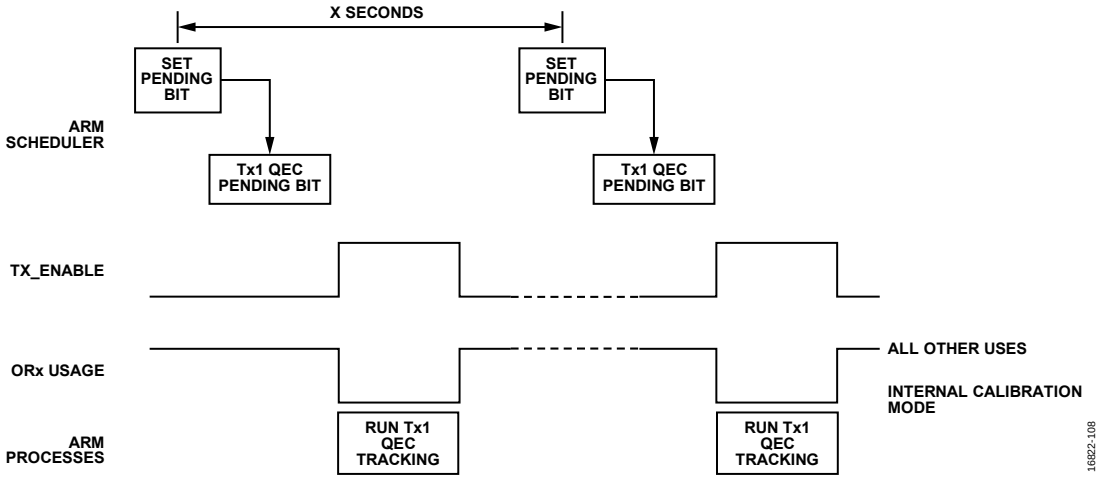

Figure 105. Arm Scheduler Operation

The scheduler runs the tracking calibrations in batches, as shown in Figure 104. Transmitter tracking calibrations typically require in the 10s of milliseconds of transmitter data observation before making an update to the correction parameters, and it is recognized that the user may not provide sufficient time in a single instance for the tracking calibration to complete. Because of this, the scheduler performs calibrations in batches, where the transmitter data can be observed in chunks of 500 µs. When sufficient batches of a tracking calibration are run, the algorithm then computes the correction based on the observed data across all the batches. The pending bit is only cleared after the correction parameters have been updated, as shown in Figure 104.

This batch operation means that when a calibration is pending and is selected by the scheduler to be run (based on the 3 conditions), the scheduler initiates a batch to observe the transmitter data for a duration of 500 µs. When this batch is complete, the scheduler determines which calibrations can be run next. If the same calibration cannot continue to run, for example, in TDD mode, if the path to be calibrated is no longer active, or if a higher priority calibration is pending, the scheduler waits for the next opportunity before taking another batch of data.

Note that if a tracking calibration batch has begun but the observation is disrupted (for example, if the transmitter or receiver path is disabled, or if the observation path has been reacquired by the user for DPD captures) before 500 µs has completed, whatever observation has been made up to this point is discarded. Discarding the observation does not affect the algorithm; the scheduler waits for another batch as normal. When assigning the use of the observation receiver path for tracking calibrations, do so in slots of at least 500 µs or multiples of 500 µs.

There are no additional requirements on the exact period of tracking calibration batches that must be maintained. The user determines the structure of the observation receiver path assignment to fit around the observation receiver path requirements of the user (for example, DPD and VSWR). The tracking calibration period can be supplied in one full section, or in batches of 500 µs spread across the 6 sec in a nonperiodic fashion. The Arm processor never takes control of the assignment of the observation receiver path and is reliant on the user to assign the observation receiver for calibrations. If the user fails to provide the assignment of the observation receiver path, the calibrations do not run.

The scheduler sets the pending bits of each calibration in a periodic fashion. Each calibration is immediately pending when entering the radio on state and the counters for each calibration are enabled. Use the following function to set a calibration as pending immediately and restart the periodic counter:

```
TALISE_rescheduleTrackingCal(taliseDevice_t *device, taliseTrackingCalibrations_t trackingCal)
```

In this function, `trackingCal` is an enum indicating the calibration to be rescheduled as indicated by Table 89:

Table 89. TALISE_rescheduleTrackingCal() trackingCal Mask Defintions

Enumerator	Rescheduled Calibration
TAL_TRACK_RX1_QEC	Rx1 QEC tracking calibration
TAL_TRACK_RX2_QEC	Rx2 QEC tracking calibration
TAL_TRACK_ORX1_QEC	ORx1 QEC tracking calibration
TAL_TRACK_ORX2_QEC	ORx2 QEC tracking calibration
TAL_TRACK_TX1_LOL	Tx1 LO leakage tracking calibration
TAL_TRACK_TX2_LOL	Tx2 LO leakage tracking calibration
TAL_TRACK_TX1_QEC	Tx1 QEC tracking calibration
TAL_TRACK_TX2_QEC	Tx2 QEC tracking calibration
TAL_TRACK_RX1_HD2	Rx1 HD2 tracking calibration
TAL_TRACK_RX2_HD2	Rx2 HD2 tracking calibration
TAL_TRACK_ALL	All tracking calibrations

SYSTEM CONSIDERATIONS FOR ARM CALIBRATIONS

This section describes the considerations necessary from a system perspective for the Arm processor to run its calibrations, for example, input/output path conditions for initial calibrations and GPIO status for tracking calibrations.

System Considerations for Initial Calibrations

Figure 106 to Figure 109 are used to show how the device is configured for notable calibrations with external system requirements, for example, the QEC and LO leakage calibrations. A brief explanation of the calibration is provided in Figure 106 to Figure 109. Note that as the Arm processor performs each of the calibrations, the processor is tasked with configuring the device and enabling/disabling paths. This configuration does not require user input.

The user must ensure that external conditions are met, such as turning off the power amplifier for all calibrations other than the external LO leakage initialization calibration, or properly terminating the receiver for an Rx QEC initialization calibration.

ADRV9008-1 and ADRV9009 Receiver QEC Initial Calibration

The receiver QEC initialization calibration algorithm is utilized to improve the receiver path QEC performance. The receiver QEC calibration sweeps a number of internally generated test tones across the band, measures quadrature performance, and calculates correction coefficients.

The input port must be isolated from incoming signals. The calibration tones appear on the receiver pins. The calibration tones must be prevented from reaching the antenna by properly terminating the receiver port.

Note that the auxiliary PLL is used to generate the tones of the receiver QEC initialization calibration (CalPLL in Figure 106). Operation of the auxiliary PLL must be limited to 6.061 GHz. For RF LO frequencies close to the maximum frequency, 6 GHz, the frequency limit of the auxiliary PLL must be taken into consideration. The auxiliary PLL generates tones in the RF frequency by sweeping across the baseband bandwidth. The range that the auxiliary PLL sweeps the tones across is RF LO frequency – baseband bandwidth/2 to RF LO frequency + baseband bandwidth/2. A 200 MHz profile at an RF LO frequency of 6 GHz causes a conflict because the auxiliary PLL must generate tones up to 6.1 GHz. In this case, the recommendation is to change the RF LO frequency for the initialization calibration so that the maximum frequency generated by the auxiliary PLL is 6.061 GHz (RF LO frequency + baseband bandwidth/2 ≤ 6.061 GHz). The RF LO frequency can be returned to the required setting before operation.

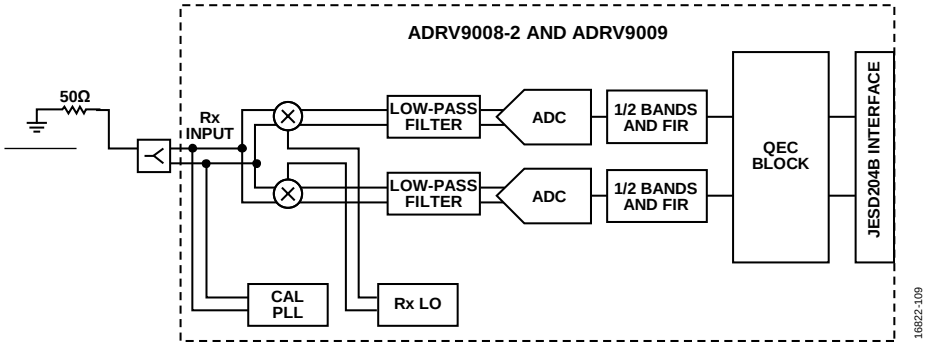

Figure 106. Receiver QEC Initial Calibration System Configuration (JESD204B Interface Not Active)

ADRV9008-2 and ADRV9009 Observation Receiver QEC Initial Calibration

The observation receiver QEC initialization calibration algorithm is utilized to improve the observation receiver path QEC performance. The algorithm itself is a replica of the receiver QEC initialization calibration. Because the ADRV9009 shares the baseband section for the receiver and observation receiver, it is necessary to run the receiver and observation receiver initialization calibrations, because the input pins and mixer front ends are different between the receiver and observation receiver modes.

The observation receiver QEC calibration sweeps a number of internally generated test tones across the band, measures quadrature performance, and calculates correction coefficients.

It is a system requirement for optimum performance and lower calibration duration, the observation receiver QEC initialization calibration must be run at attenuations between 0 dB and 5 dB. The observation receiver input must be isolated from incoming signals and be properly terminated into a 50 Ω load when the calibration is running. The calibration tones appear on the receiver pins and must be prevented from reaching the antenna.

Note that the auxiliary PLL generates the tones of the observation receiver QEC initialization calibration (shown as CalPLL Figure 107). The auxiliary PLL frequency operation must be limited to 6.061 GHz. For RF LO frequencies close to the maximum frequency of 6 GHz, the frequency limit of the auxiliary PLL must be taken into consideration. The auxiliary PLL generates tones in RF frequency, sweeping across the baseband bandwidth. The range that the auxiliary PLL sweeps across the tones is RF LO frequency – baseband bandwidth/2 to RF LO frequency + baseband bandwidth/2. A 200 MHz profile at an RF LO of 6 GHz causes a conflict, because the auxiliary PLL must generate tones up to 6.061 GHz. In this case, the recommendation is to change the RF LO frequency for the initialization calibration so that the maximum frequency generated by the auxiliary PLL is 6.061 GHz (RF LO frequency + baseband bandwidth/2 ≤ 6.061 GHz). The RF LO frequency can be returned to the required setting before operation

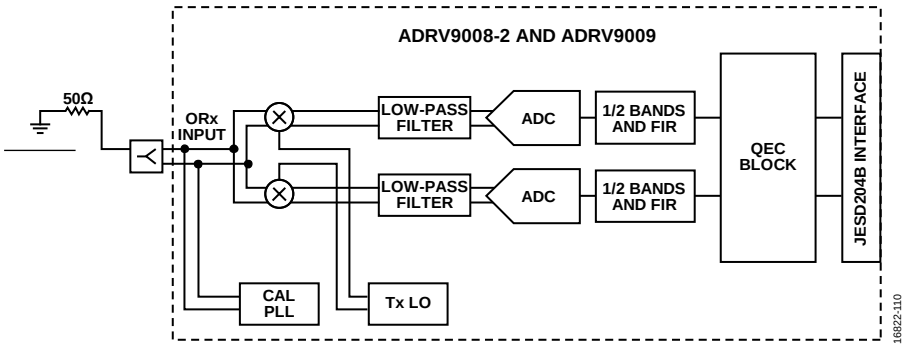

Figure 107. Observation Receiver QEC Initial Calibration System Configuration (JESD204B Interface Not Active)

ADRV9008-2 and ADRV9009 Internal Transmitter LO Leakage and Transmitter QEC Initial Calibrations

The transmitter LO Leakage and transmitter QEC initial calibrations utilize the loopback path (feedback path) and the baseband section of the observation receiver path to calculate the initial correction factors. During these calibrations, test signals (tones and wide-band signals) are output. These appear at the transmitter output, and so it is important that the power amplifier at the output of the device be switched off. Both calibrations sweep through a series of attenuation values, creating a table of initial calibration values. During operation and at the application of a new transmitter attenuation setting, the corresponding QEC and LO leakage correction values are applied to the transmitter channel by the Arm processor. The device configuration for this calibration is shown in Figure 108.

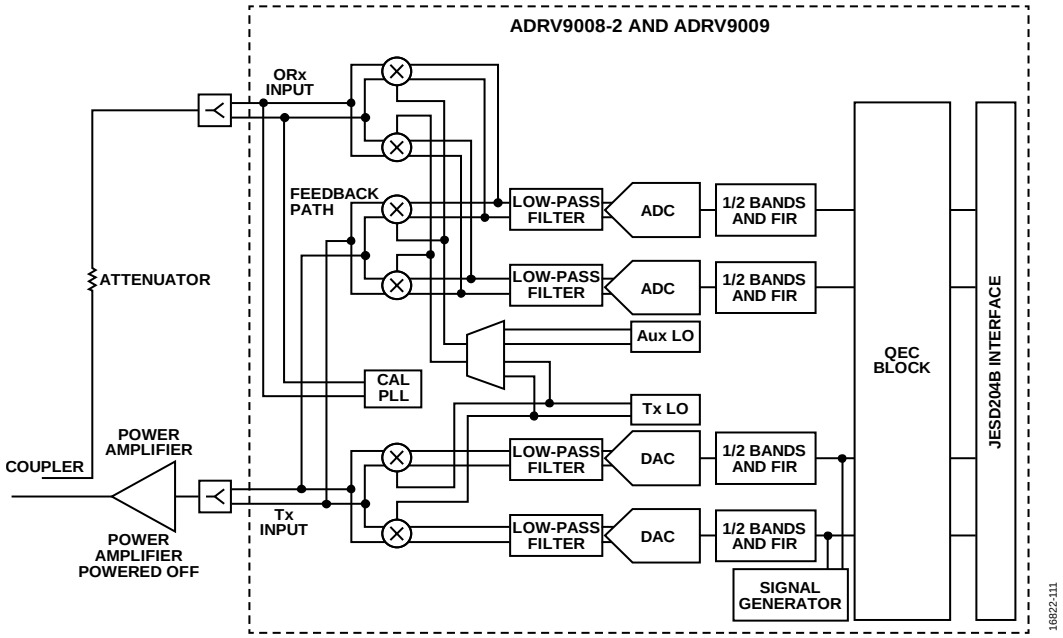

Figure 108. Device Path Configuration for Transmitter LO Leakage andTransmitter QEC Initialization Calibrations (Attenuator, Coupler, Power Amplifier, Observation Receiver Input, Calibration PLL, and JESD204B Interface Not Active)

It is a system requirement that the power amplifier in the transmitter path is powered off during these calibrations.

DAC Boost Mode

If the system performance requirement for the transmitter LO leakage is marginal in terms of the typical performance of the device, it is possible for the user to increase the full-scale signal output of the transmitter DAC. This is a 3 dB boost, and in this mode, a further 3 dB margin is applied between the output signal and the LO leakage. There is a reduction in linearity performance in this mode; therefore, the setting is a tradeoff based on the system requirements of the user. Use the following function to enable DAC boost mode:

```
TALISE_setDacFullScale(taliseDevice_t *device, taliseDacFullScale_t dacFullScale)
```

In this function, `dacFullScale` is a parameter that selects the required DAC full-scale mode, as described in Table 90.

Table 90. Definition of taliseDacFullScale_t

Mode	Enumerator
No DAC Full-Scale Boost	TAL_DACFS_0DB
Full-Scale DAC Boost	TAL_DACFS_3DB

The function to enable DAC boost mode must be run before the Arm is booted up.

ADRV9008-2 and ADRV9009 External Transmitter LO Leakage Initial Calibration

The external LO leakage initialization calibration requires that the power amplifier be enabled so that a full external loop is made between the transmitter outputs and the observation receiver inputs. The purpose of this calibration is to obtain a good estimate (gain/phase) of the external loop channel conditions prior to operation. The device configuration is shown in Figure 109. The calibration utilizes a pseudorandom noise signal to estimate the channel conditions. This noise signal is a broadband signal with a nominal level of −78 dBFS out of the DAC.

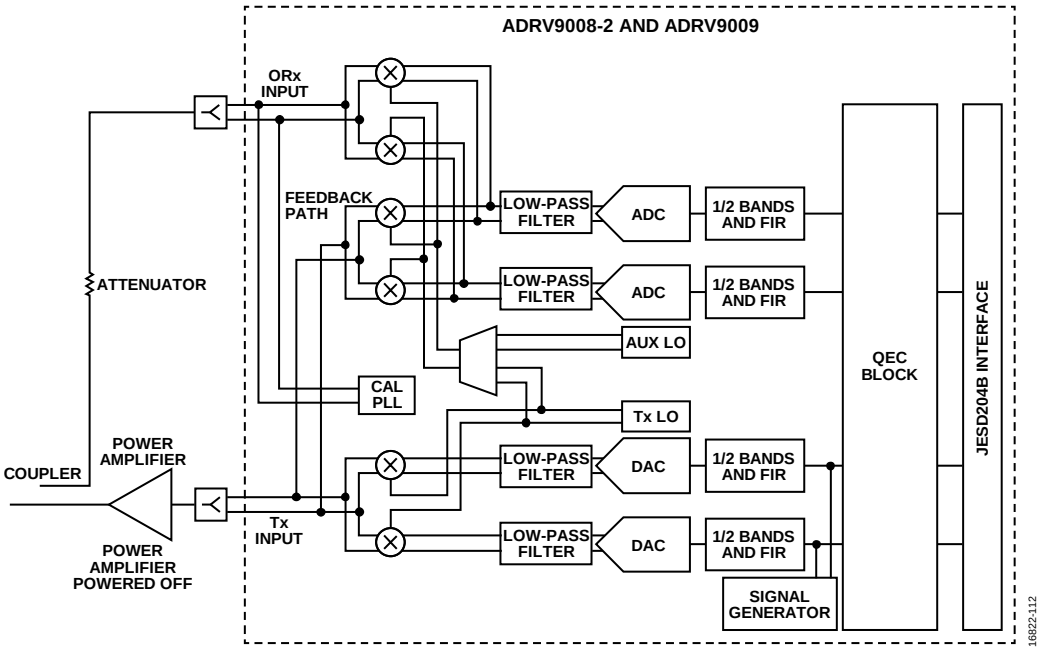

Figure 109. External LO Leakage System Configuration (Feedbac Path, Calibration PLL, and JESD204B Interface Not Active)

It is important that a suitable attenuator be chosen between the power amplifier output and the observation receiver input to prevent the transmitter data from saturating the observation receiver input. This attenuator is also necessary from the perspective of DPD operation. The full-scale input of the observation receiver path is −13 dBm (with a 0 dB attenuation setting) for a single tone input.

It is a system requirement that the output of the transmitter channel to be calibrated is routed to the utilized observation receiver path for the calibration signal to be observed. The device must be configured prior to the calibration to indicate which transmitter is routed back to which observation receiver. For optimal LO leakage performance when operating above 4 GHz, only the Observation Receiver 1 input must be used for both Transmitter 1 and Transmitter 2 (use an external switch to accomplish this).

To advise the device which transmitter is routed to which observation receiver input, use the following function:

```
TALISE_setTxToOrxMapping(taliseDevice_t *device, uint8_t txCalEnable, taliseTxToOrxMapping_t
oRx1Map, taliseTxToOrxMapping_t oRx2Map)
```

Table 91 outlines the parameters of the `setTxToOrxMapping` function.

Table 91. setTxToOrxMapping Parameter Descriptions

Parameter	Value	Description
txCalEnable	0	Internal calibrations are disabled.
	1	Internal calibrations are enabled.
oRx1Map	TAL_MAP_NONE	No Tx routed to ORx1.
	TAL_MAP_TX1_ORX	Tx1 is routed to ORx1.
	TAL_MAP_TX2_ORX	Tx2 is routed to ORx1.
oRx2Map	TAL_MAP_NONE	No Tx routed to ORx2.
	TAL_MAP_TX1_ORX	Tx1 is routed to ORx2.
	TAL_MAP_TX2_ORX	Tx2 is routed to ORx2.

Skipping External LO Leakage Initialization

It is possible to avoid running the external LO leakage algorithm during the initialization of the device. In this case, the LO leakage algorithm does not have a valid estimate of the external channel when transmission begins. The algorithm tries to learn the channel as quickly as possible, using the time provided to observe the external path to learn the channel. During this channel learning time, correction factors are not updated by the tracking algorithm and performance can be worse than when an external LO leakage initialization is run. When channel learning is completed, normal tracking resumes and the algorithm running every 6 sec on each enabled transmitter channel, updating the correction coefficients as necessary. If the external LO leakage initialization is skipped, it is expected that more time is assigned to the transmitter calibrations when the radio is first enabled so that the LO leakage can converge faster. With the standard 10% assignment of time for transmitter calibrations, the LO leakage can take a number of seconds to converge.

If the fastest time to optimal LO leakage performance at the first transmission of user data is required, run the external LO leakage initialization. Running the external LO leakage initialization is especially important for LO frequencies above 3 GHz, where the base performance of the internal LO leakage initialization calibration is not as good as the performance at frequencies less than 3 GHz. For frequencies greater than 3 GHz, it is recommended to use the external LO leakage initialization calibration.

Running Initialization Calibrations

Terminate the receiver inputs for the ADRV9008-1, as shown in Figure 106. All relevant calibrations in the calibration mask can be set when calling `Talise_runInitCals()` (see the latest GUI script for current advice on recommended calibrations for the ADRV9008-1). In all cases, the Arm processor runs the enabled calibrations in the calibration mask for each enabled channel. It is not necessary to call `Talise_runInitCals()` separately for Receiver 1 and Receiver 2 in the case where both receivers are used.

The following serves as a timeline of how to run the initialization calibrations for the ADRV9008-1 device in a single-receiver or dual-receiver configuration.

1. It is assumed that the device has been fully configured up to an including the JESD204B initialization.
2. Turn off the power amplifier and isolate the receiver inputs. This can be done by default on startup.
3. Run initial calibrations for all required calibrations except for the external LO leakage calibration. API command: `TALISE_runInitCals(taliseDevice_t *device, uint32_t calMask)`
4. Wait until these calibrations have been completed. API command: `TALISE_waitInitCals(taliseDevice_t *device, uint32_t timeoutMs, uint8_t *errorFlag)`

For the ADRV9008-2, turn off the power amplifiers, as shown in Figure 108, and the terminate the observation receiver inputs, as shown in Figure 107. With this configuation, it is possible to run all relevant calibrations by setting the corresponding bits in the calibration mask. Ensure that the external LO leakage initialization calibration Bit D9 is omitted, and then call `Talise_runInitCals()`. In all cases, the Arm processor runs the enabled calibrations in the calibration mask for each enabled channel. It is not necessary to call `Talise_runInitCals()` separately for Transmitter 1 and Transmitter 2 in the case where both transmitters are used.

The external LO leakage initialization calibration is omitted because it requires the power amplifier to be on, and requires the observation receiver to be connected to the power amplifier output through the external loopback path. The current configuration of the external path is advised to the Arm processor through the `setTxToOrxMapping()` API function. When one observation is shared between two transmitter aths, as shown in Figure 110, the mapping must be set such that Transmitter 1 is being fed back to Observation Receiver 1, the calibration is run, then the mapping is changed such that Transmitter 2 is being fed back to Observation Receiver 1, and the calibration is run again. The external LO leakage calibration differs from the other internal calibrations in that it does not always run the calibration for each path consecutively. Instead, when the calibration is called, it verifies which path is connected to which observation receiver input, and then runs the call for that path only.

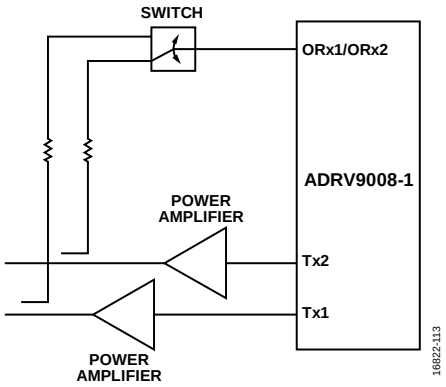

Figure 110. External Observation Receiver Switch for a Two Transmitter Configuration

The following is a timeline of how the initialization calibrations must be run for the ADRV9008-2 in a two-transmitter, one-observation receiver configuration for the 200 MHz or 450 MHz transmitter, 450 MHz observation receiver use case:

The following list describes the initialization calibration procedural steps for the ADRV9008-2. It is assumed that the device has been fully configured up to and including the JESD204B initialization (see GUI configuration procedure for an example of the initialization procedure):

1. Turn off the power amplifiers and isolate the observation receiver inputs. This can be done by default on startup.
2. Run initialization calibrations for all required calibrations except for the external LO leakage calibration. API command: `TALISE_runInitCals(taliseDevice_t *device, uint32_t calMask)`
3. Wait until these calibrations are complete. API command: `TALISE_waitInitCals(taliseDevice_t *device, uint32_t timeoutMs, uint8_t *errorFlag)`.
4. Turn on Power Amplifier 1 and close switches to connect Transmitter 1 to Observation Receiver 1.
5. Advise the device of the current transmitter to observation receiver connection. API command: `TALISE_setTxToOrxMapping(taliseDevice_t *device, uint8_t txCalEnable, taliseTxToOrxMapping_t oRx1Map, taliseTxToOrxMapping_t oRx2Map)`.
6. In this case, set `txCalEnable` to 1, define `oRx1Map` with the enum `TAL_MAP_TX1_ORx`, and define `oRx2Map` with the enum `TAL_MAP_NONE`.
7. Run an external LO leakage initial calibration on Transmitter 1. API command: `TALISE_runInitCals(taliseDevice_t *device, uint32_t calMask)`. In this case, the `calMask` is 0x200.
8. Wait until this calibration is complete. API command: `TALISE_waitInitCals(taliseDevice_t *device, uint32_t timeoutMs, uint8_t *errorFlag)`.
9. Turn on Power Amplifier 2 and close switches to connect Transmitter 2 to Observation Receiver 1.
10. Advise the device of the current transmitter to observation receiver connection. API command: `TALISE_setTxToOrxMapping(taliseDevice_t *device, uint8_t txCalEnable, taliseTxToOrxMapping_t oRx1Map, taliseTxToOrxMapping_t oRx2Map)`.
11. In this case, set `txCalEnable` should be set to 1, define `oRx1Map` with the enum `TAL_MAP_TX2_ORx`, and define `oRx2Map` with the enum `TAL_MAP_NONE`.
12. Run an external LO leakage initial calibration on Transmitter 2. API command: `TALISE_runInitCals(taliseDevice_t *device, uint32_t calMask)`. In this case, the `calMask` is 0x200.
13. Wait until this calibration is complete. API command: `TALISE_waitInitCals(taliseDevice_t *device, uint32_t timeoutMs, uint8_t *errorFlag)`.

For the ADRV9009, turn off the power amplifiers, as shown Figure 108, and terminate the observation receiver inputs, as shown in Figure 107. With this configuration, it is possible to run all relevant calibrations by setting the corresponding bits in the calibration mask. Ensure that the external LO leakage initialization calibration bit D9 is omitted and then call `Talise_runInitCals()`. In all cases, the Arm processor runs the enabled calibrations in the calibration mask for each enabled channel. It is not necessary to call `Talise_runInitCals()` separately for Transmitter 1 and Transmitter 2 in the case where both transmitters are used.

The following is a timeline of how to run the initialization calibrations for the ADRV9009 in a two receiver, two transmitter, one observation receiver configuration:

1. It is assumed that the device has been fully configured up to an including the JESD204B initialization (see GUI configuration procedure for an example of the initialization procedure).
2. Turn off the power amplifiers and isolate the receiver and observation receiver inputs. This can be done by default on startup.
3. Run initialization calibrations for all required calibrations except for the external LO leakage calibration. API command: `TALISE_runInitCals(taliseDevice_t *device, uint32_t calMask)`.
4. For Step 3 to Step 11, see corresponding steps in the previous list within this section.

System Considerations for Tracking Calibrations

This section describes the operation of tracking calibrations. Figure 111 to Figure 121 show the device configuration for each calibration. When the Arm processor performs each of the calibrations, it is tasked with configuring the whether the feedback path or observation receiver input is selected, as shown in Figure 111 to Figure 121. No user input is required in this regard. However, for external LO leakage tracking, the user must ensure that the feedback path is available to use.

During tracking calibrations, it can be required that the GPIO and enable pins must have many milliseconds of observation to calculate an update; however, the Arm processor splits this time up into batches so that observations do not need to be continuous. These batches are observations of 500 µs in duration. The receiver/observation receiver tracking algorithms run when the channels are in normal use, and these algorithms use the data in the channel to calculate updates to the correction coefficients. The transmitter correction algorithms utilize the observation receiver path when run, and feed back transmission data for observation to calculate updates to the correction coefficients. This means that the observation receiver path must be time shared with other uses of the observation receiver path, for example, DPD and VSWR.

The time requirement for the tracking calibrations is for the observation receiver to be assigned to the tracking calibrations for 10% of transmitter operation time, and this 10% time period must occur when the transmitter is transmitting data. In a TDD scenario, where the transmitter only accounts for 50% of the time, 20% of the transmitter operation time must be assigned to tracking calibrations.

For a tracking calibration to successfully make an observation, it must be assigned at least 500 µs in any one instance. If more time is assigned at an instance, the time period must be in multiples of the 500 µs.

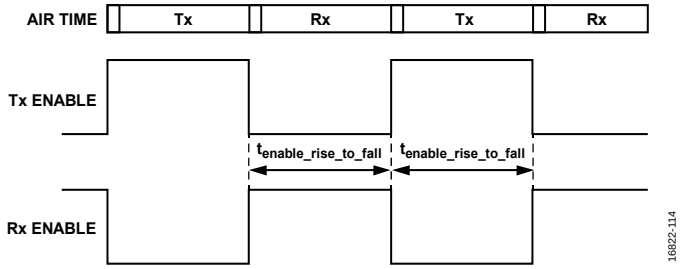

Figure 111. Transmitter and Receiver Enable Signals Timing

Table 92. Receiver and Transmitter Timing for Tracking Calibrations

Symbol	Description	Minimum Duration for Calibration
$t_{ENABLE\_RISE\_TO\_FALL}$	Tx/Rx enable rising edge to enable falling edge; enable signal width high	500 µs
$t_{ENABLE\_FALL\_TO\_RISE}$	Tx/Rx enable falling edge to enable rising edge; enable signal width low	500 µs

Note that although Table 92 indicates that the minimum duration for a transmitter period is 500 µs, this does not mean that durations cannot be less than this for some special transmitter subframes. If duration periods less than 500 µs occur, the tracking algorithms discard any observations made during these shorter time periods and do not utilize these periods to calculate the next correction update.

ADRV9008-1 and ADRV9009 Receiver QEC Tracking Calibration

The receiver QEC tracking algorithm improves the receiver path QEC performance during operation. The tracking algorithm utilizes normal traffic data to calculate updated corrected coefficients and runs continuously when the receivers are active.

It is a system requirement that receiver channels are enabled, for example, in TDD mode, receiver QEC tracking only runs during receiver periods. If only one channel is enabled, the receiver QEC only runs on this channel.

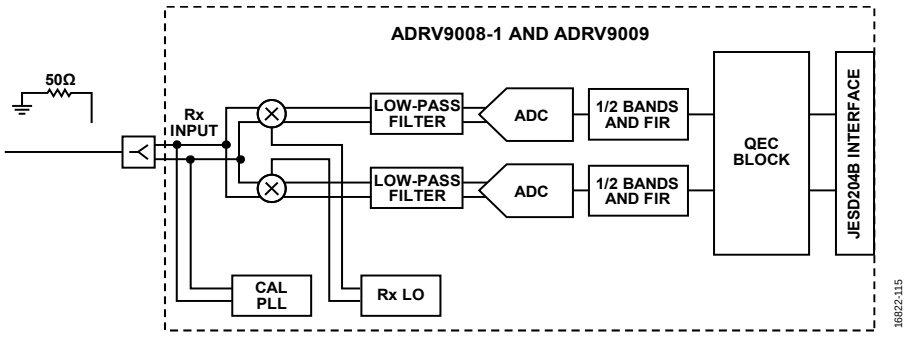

Figure 112. Receiver QEC Tracking (50 Ω Load and Calibration PLL Not Active)

Figure 113 is a timing diagram that shows when the receiver QEC tracking calibration can run in TDD mode. In frequency division duplex (FDD) modes, receiver enable is always high. Receiver enable refers to the enable signal of Receiver 1 and/or Receiver 2.

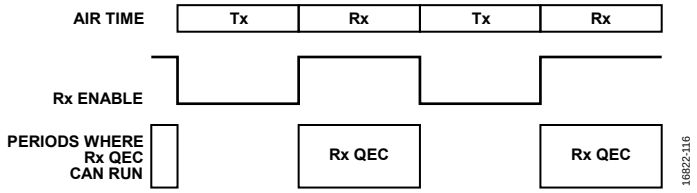

Figure 113. Possible Periods to Run Receiver QEC Tracking Calibration in TDD Mode

ADRV9008-1 and ADRV9009 Receiver HD2 Tracking Calibration

The receiver HD2 tracking algorithm improves the receiver path HD2 performance during operation, required for GSM use cases. In GSM use cases, use a low IF configuration should in the receive path. See the GSM Use Cases section for details. GSM reception is only supported with a receiver profile of 200 MHz with an I/Q rate of 245.76 MSPS, or the equivalent low IF receiver profile of 100 MHz, where the receive chain is configured as per the receiver profile of 200 MHz up to the IF conversion stage, before the carrier is shifted and the data rate is down-converted to a lower rate. This means that the HD2 algorithm is only supported with these profiles and must not be enabled in other profiles.

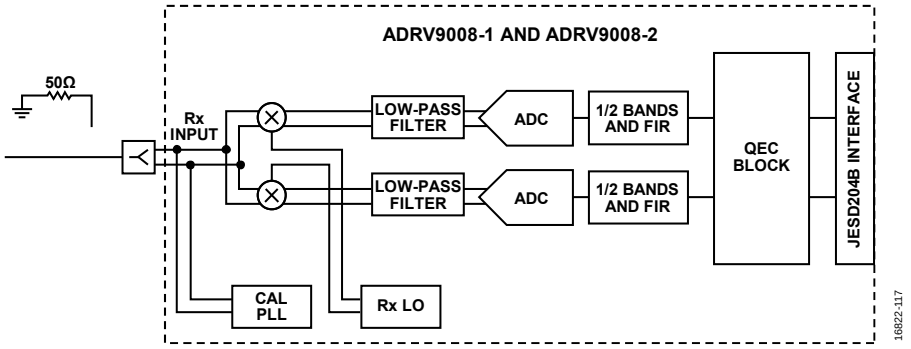

Figure 114. Receiver HD2 Tracking (50 Ω Load and Calibration PLL Not Active)

Figure 115 shows a use case where a 75 MHz multicarrier GSM signal is placed on the lower half of the spectrum. The receiver HD2 algorithm is designed to monitor the −5 MHz to −50 MHz frequency range. The algorithm detects any signals in this range and corrects the HD2 of these signals on the same side of the spectrum as the carrier. The range of observation is limited because the HD2 of signals outside of this range fall out of band. Similarly, if the multicarrier GSM signal was placed in the upper half of the spectrum, the HD2 correction algorithm observes between the 5 MHz and 50 MHz frequency range and corrects the HD2 on the same side of the signals observed.

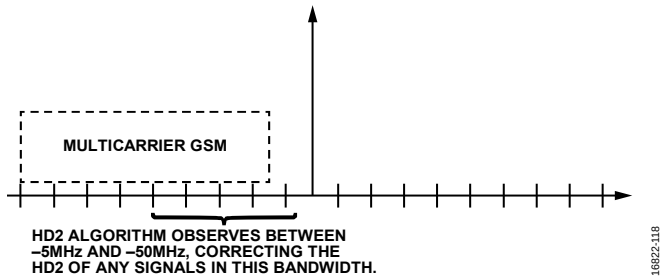

Figure 115. GSM Use Case and HD2 Correction Showing a Negative Offset of the Carrier from the LO

The following API is used to indicate which carrier placement is being used:

`TALISE_setRxHd2Config(taliseDevice_t *device, taliseRxHd2Config_t *hd2CalConfig)`

Parameter: hd2CalConfig is a taliseRxHd2Config_t structure containing one integer member, posSideBandSel. The valid settings of posSideBandSel are described in Table 93.

Table 93. Permissible Setting Values for posSideBandSel

posSideBandSel Setting Value	Description
0	Multicarrier GSM placed on the lower side of complex spectrum. Rx HD2 corrects for HD2 components on the lower side of the spectrum.
1	Multicarrier GSM placed on the upper side of the complex spectrum. Rx HD2 corrects for HD2 components on the upper side of the spectrum.

The receiver HD2 tracking algorithm utilizes normal traffic data to calculate updated corrected coefficients and runs continuously when the receivers are active.

It is a system requirement that the receiver channels are enabled for the receiver HD2 to run.

Figure 116 is a timing diagram that shows when the receiver HD2 can run based on the receiver enable signal.

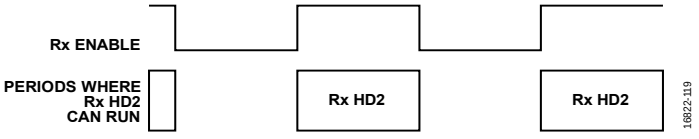

Figure 116. Time Periods to Run Receiver HD2 Based on Receiver Enable Signal

ADRV9008-2 and ADRV9009 Observation Receiver QEC Tracking Calibration

The observation receiver QEC tracking algorithm improves the observation receiver path QEC performance during operation and utilizes normal traffic data (for example, DPD capture data) to calculate updated corrected coefficients. The algorithm runs continuously in the background when the observation receiver is active.

It is a system requirement that the receiver channels are enabled, for example, in TDD mode, the observation receiver QEC tracking only runs during observation receiver periods. If only one channel is enabled, the observation receiver QEC only runs on this channel.

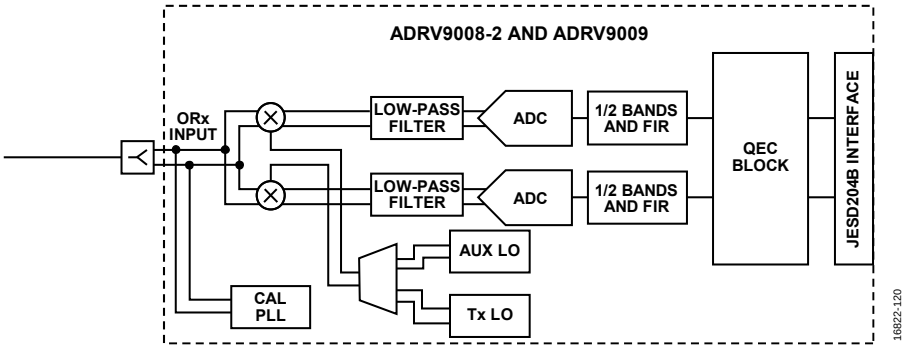

Figure 117. Observation Receiver QEC Tracking (Calibration PLL Not Active)

Figure 118 is a timing diagram that shows when the observation receiver QEC tracking calibration can be run in TDD mode. In FDD modes, observation receive enable is high at all times. Observation receiver enable refers to the enable signal of Observation Receiver 1 and/or Observation Receiver 2.

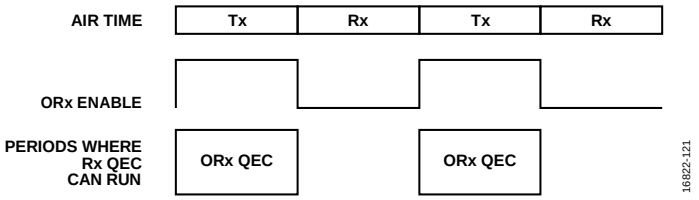

Figure 118. Possible Periods to Run Observation Receiver QEC Tracking Calibration in TDD Mode

ADRV9008-2 and ADRV9009 Transmitter QEC Tracking Calibration

The transmitter QEC tracking is an online calibration that is run to improve the QEC performance using transmit data. The calibration utilizes the loopback (feedback) path for operation. Therefore, the transmitter QEC tracking must be interleaved with normal DPD captures (or channel sniffing functions) that utilize the observation receiver path. This tracking determines optimal coefficients for the current gain setting, updating the table stored during the transmitter QEC initialization to ensure that this table has the best values for the current operating conditions. Figure 119 shows the device configuration for the transmitter QEC tracking calibration.

It is a system requirement that the transmitter channel(s) are enabled. To run the calibration the observation receiver path must be available for the Arm processor to use (observation receiver enable low), for example, not required by the user for DPD (or VSWR) captures.

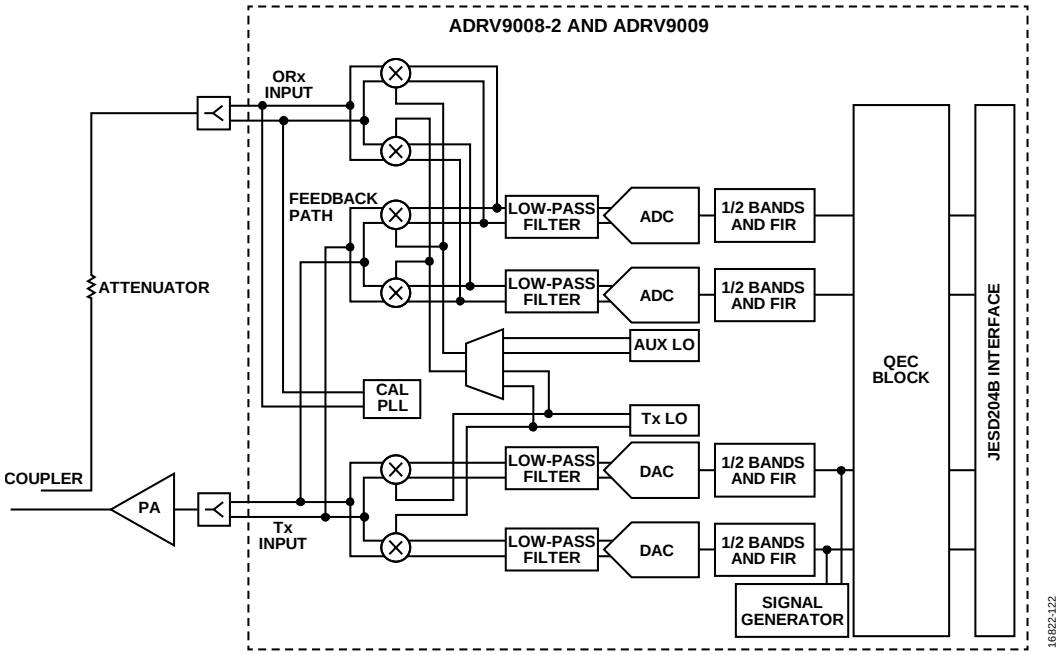

Figure 119. Transmitter QEC Tracking Calibration Configuration (Observation Receiver Input, Calibration PLL, and Signal Generator Not Active)

Figure 120 is a timing diagram that shows periods where the tracking calibration can run in TDD mode. In FDD modes, transmit enable is high at all times. Transmit enable refers to the enable signal of Transmitter 1 and/or Transmitter 2. Observation receive enable refers to either Observation Receiver 1 or Observation Receiver 2, depending on which is being used.

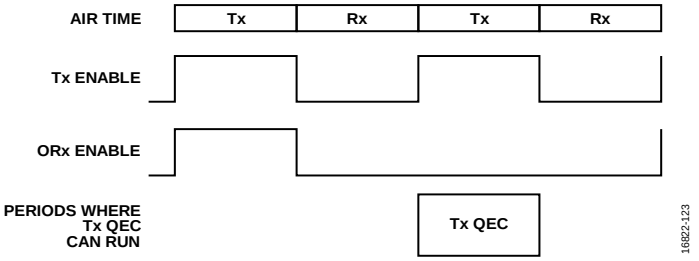

Figure 120. Possible Periods to Run Transmitter QEC Tracking Calibration in TDD Mode

Note that the transmitter QEC tracking uses an offset LO frequency on the feedback path during tracking, which ensures that the quadrature errors of the transmitter path are not aligned with those of the observation receiver path. This frequency is set to ((Primary Transmitter Bandwidth/4) + 5 MHz). Continuous wave tones placed at ± this offset frequency, or set at twice this offset frequency show reduced QEC performance. Modulated signals centered at these frequencies do not have reduced performance.

ADRV9008-2 and ADRV9009 Transmitter LO Leakage Tracking Calibration

The transmitter LO leakage tracking calibration uses an external DPD path between the transmitter output and observation input to measure LO leakage and calculate correction factors. the transmitter LO leakage tracking calibration is run when user data is being transmitted, with the power amplifier operational). For this calibration, the auxiliary LO is used in the observation receiver path to offset the transmitter LO leakage from the observation receiver LO. Therefore, this calibration must be interleaved with the observation captures of the user, which utilize the transmitter LO to drive the observation receiver path. Figure 121 shows the device configuration for the transmitter LO leakage tracking calibration with the transmitter output looped back to the observation receiver, whichever observation receiver input is in use in the system.

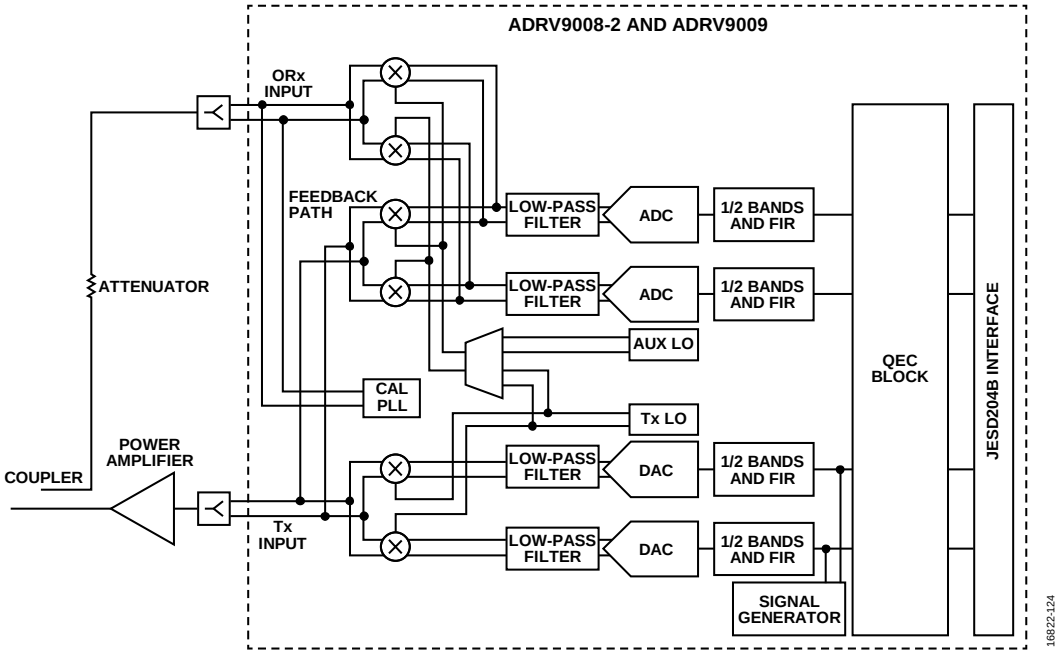

Figure 121. Transmitter LO Leakage Tracking Configuration (Feedback Path, Calibration PLL, and Signal Generator Not Active)

It is a system requirement that the transmitter channel(s) are be enabled. The observation receiver path must be available for the Arm processor to use (observation receiver enable low), for example, not required by the user for DPD (or VSWR) captures. The observation receiver path must be connected to the appropriate transmitter to be calibrated, and the Arm processor must be advised of which transmitter output has a connection to which observation receiver. See the Arm GPIO Pins section for details.

The user must ensure that the appropriate external feedback path is available when the Arm processor is given access to the observation receiver path to perform transmitter tracking calibrations.

Note that the external LO leakage tracking calibration utilizes an estimate of the external channel (gain/phase rotation) to calculate the correction coefficients. This channel estimate is updated over time during tracking on transmitter data, and any phase/gain drift over time/temperature can be tracked out. Sudden changes in the phase/gain of the external path can result in reduced performance, until, for example, the algorithm that tracks the channel changes out.

By default, the algorithm acquires 67% of the new channel estimate in 200 sec. This slow update rate is chosen because, typically, the external channel changes slowly over time. To get an optimal estimate of the external channel in a shorter period of time, the external channel estimate can be reset using the following API function:

`TALISE_resetExtTxLolChannel(taliseDevice_t *device, taliseTxChannels_t channelSel)`

Parameter: `channelSel` is the channel for which the external LO leakage channel estimate must be reset as shown in Table 94.

Table 94. ChannelSel for TALISE_resetExtTxLolChannel()

Channels	Enumerator
Tx1	TAL_TX1
Tx2	TAL_TX2
Tx1 and Tx2	TAL_TX1TX2

When the LO leakage algorithm is reset, it goes into a channel learning mode. In this mode, the channel is learned as quickly as possible, and the default schedule is not obeyed, if the user provides more time for the algorithm to observe the external path, the algorithm takes this time to make the observations necessary to learn the channel. The correction is not updated during this time, rather, it is frozen with the values that were being applied before the API call was issued. After the channel is learned, further instances of transmitter LO leakage tracking updates the LO leakage correction coefficients.

Note that the LO leakage algorithm only begins to learn the channel when the scheduler sets the pending bit. Resetting the external channel does not reschedule the calibration; therefore, when the user resets the external channel, use the `TALISE_rescheduleTrackingCal()` function to reschedule the calibration immediately after the external channel is reset.

If the sudden changes are large enough, the external channel estimate must be reset using the `TALISE_rescheduleTrackingCal()` command above, but this reset is at it is at the discretion of the user at other times. The external channel must be reset is if the LO frequency of the device has been changed or if the gain and phase have suddenly changed by the corresponding gain error and maximum phase error values shown in Table 95.

Table 95. Table of Gain Error vs. Max Phase Error

Gain Error (dB)	Maximum Phase Error (Degrees)
−3	69.26949155
−2.5	67.97895638
−2	66.59898696
−1.5	65.12136412
−1	63.53663696
−0.5	61.83382241
0	60
0.5	58.0197531
1	55.87437871
1.5	53.54073591
2	50.98950693
2.5	48.18245508
3	45.0678624

ARM GPIO PINS

The Arm processor scheduler must know which feedback is currently available so to schedule the correct transmitter LO leakage tracking calibration, for example, if Transmitter 1 is fed back to Observation Receiver 1 at a given time, run the Transmitter 1 LO leakage tracking calibration. The scheduler keeps track of how this feedback path(s) varies over time to ensure that the correct transmit LO leakage tracking is run at any given time.

The Arm processor assigns two signals to each observation receiver to determine if a feedback path exists between a transmitter output and an observation receiver input. The possible assigned signals are ORX1_TX_SEL0, ORX1_TX_SEL1, ORX2_TX_SEL0, and ORX2_TX_SEL1. Table 96 and Table 97 show the mapping of the transmitter to the observation receiver according to these signals.

Table 96. Observation Receiver Select Bits for Observation Receiver 1

ORX1_TX_SEL1 Signal Setting	ORX1_TX_SEL0 Signal Setting	Description
0	0	No Tx routed to ORx1; external LO leakage tracking calibration not run
0	1	No Tx routed to ORx1; external LO leakage tracking calibration not run
1	0	Tx1 is routed to ORx1
1	1	Tx2 is routed to ORx1

Table 97. Observation Receiver Select Bits for Observation Receiver 2

ORX2_TX_SEL1 Signal Setting	ORX2_TX_SEL0 Signal Setting	Description
0	0	No Tx routed to ORx2; external LO leakage tracking calibration not run
0	1	No Tx routed to ORx2; external LO leakage tracking calibration not run
1	0	Tx1 is routed to ORx2
1	1	Tx2 is routed to ORx2

These signals can be controlled either through the SPI (signal level toggled internally) or through the GPIOs. To control the signals with the SPI, use the `TALISE_setTxToOrxMapping()` function. This function indicates the transmitter output that is fed back to the Observation Receiver 1 and Observation Receiver 2 input (see Table 96 and Table 97). This function is applicable in the case where fixed feedback paths exist, for example, if Transmitter 1 is permanently routed to Observation Receiver 1 (no external switches), and Transmitter 2 is permanently routed to Observation Receiver 2.

Note that for observation receiver bandwidths that are greater than 200 MHz (I/Q rate at 245.76 MSPS), it is not possible to switch between observation receivers (only one can be used).

Alternatively, up to 4 GPIOs can be used (two per observation receiver used). These GPIOs can be selected from GPIO_0 to GPIO_15. GPIOs are typically utilized because of a real-time system requirement. The ORX_TX_SEL0 signal is used to select between Transmitter 1 and Transmitter 2. The ORX_TX_SEL1 signal is used to indicate if a feedback path from either transmitter exists, which is utilized if multiple feedback paths exist, for example, forward paths for DPD/VSWR and reverse paths for VSWR. The LO leakage algorithm must see a consistent feedback path, and so ORX_TX_SEL1 is used to block the tracking algorithm when an alternate feedback is present at the input of the observation receiver, such as a reverse path.

In scenarios where only two options are allowable, for example, Transmitter 1 to Observation Receiver 1 and Transmitter 2 to Observation Receiver 1, it is possible to have the ORX_TX_SEL0 signal controlled through a GPIO for dynamic operation and to have the ORX_TX_SEL1 signal in a fixed position, and to configure this signal through the SPI.

To advise the Arm scheduler of which pins to monitor for ORX_TX_SELx signals, use the following function:

```
TALISE_ setArmGpioPins(taliseDevice_t *device, taliseArmGpioConfig_t *armGpio)
```

In this function, `*armGPIO` is a pointer to a `taliseArmGpioConfig` structure.

taliseArmGpioConfig Structure

The `taliseArmGpioConfig` structure contains all the settings for the Arm GPIO pins. Table 98 describes the data fields of this structure.

Table 98. taliseArmGpioConfig Structure Parameters

Type	Data Field	Description
taliseArmGpioPinSettings_t	orx1TxSel0Pin	A structure containing the settings for the GPIO assigned to the ORX1_TX_SEL0 signal.
taliseArmGpioPinSettings_t	orx1TxSel1Pin	A structure containing the settings for the GPIO assigned to the ORX1_TX_SEL1 signal.
taliseArmGpioPinSettings_t	orx2TxSel0Pin	A structure containing the settings for the GPIO assigned to the ORX2_TX_SEL0 signal.
taliseArmGpioPinSettings_t	orx2TxSel1Pin	A structure containing the settings for the GPIO assigned to the ORX2_TX_SEL1 signal.
taliseArmGpioPinSettings_t	enTxTrackingCals	A structure containing the settings for the GPIO assigned to the enTxTrackingCals. This pin is not supported by the Arm processor. If defined in the taliseArmGpioPinSettings_t structure, assign the pin to GPIO_INVALID and set enable to 0.

taliseArmGpioPinSettings Structure

The `taliseArmGpioPinSettings` data structure holds the pin assignments, polarity, and pin enable for each of the Arm GPIO pins (see Table 99).

Table 99. taliseArmGpioPinSettings Parameters

Type	Data Field	Permissible Values	Description
taliseGpioSel_t	gpioPinSel	TAL_GPIO_00 TAL_GPIO_01 TAL_GPIO_02 TAL_GPIO_03 TAL_GPIO_04 TAL_GPIO_05 TAL_GPIO_06 TAL_GPIO_07 TAL_GPIO_08 TAL_GPIO_09 TAL_GPIO_10 TAL_GPIO_11 TAL_GPIO_12 TAL_GPIO_13 TAL_GPIO_14 TAL_GPIO_15 TAL_GPIO_16 TAL_GPIO_17 TAL_GPIO_18 TAL_GPIO_INVALID	An enumerator that contains all possible GPIO pin assignments. TAL_GPIO_INVALID is intended for use with pins, such as the GPIO assigned to enTxTrackingCals confirm that are not being used.
unit8_t	Polarity	0: normal polarity 1: reversed polarity	An unsigned integer indicating whether to invert GPIO polarity or not. If inverted, then this inverts the truth tables shown in Table 96 and Table 97.
uint8_t	Enable	0: signal set by Arm command 1: signal set by GPIO pin	

To utilize GPIO_0 as the ORX1_TX_SEL0 signal pin, a `taliseArmGpioPinSettings` structure is assigned to the `orx1TxSel0Pin` data field of a `taliseArmGpioStructure`. The data fields are as follows: `gpioPinSel` = GPIO_0, `polarity` = 0, and `enable` = 1.

The `taliseArmGpioStructure` is completed with similar structures for each of the data fields, before being transferred to the `TALISE_setArmGpioPins()` function.

Figure 122 shows an example of a TDD use case with Observation Receiver 1 being utilized as the observation path. There are two forward paths used for DPD and transmitter external LO leakage. There are also two reverse paths for VSWR. With four permutations being controlled in real-time, ORX1_TX_SEL0 and ORX1_TX_SEL1 are controlled by GPIOs.

To enable the Arm GPIOs in this configuration, the `taliseArmGpioConfig` structure is created as follows (assuming GPIO_10 and GPIO_11 are used for the Arm GPIOs):

```
static taliseArmGpioConfig_t armGpio =
{
              .orx1TxSel0Pin
              {
                     .gpioPinSel = TAL_GPIO_10
                     .polarity = 0
                     .enable=1
              }
              .orx1TxSel1Pin
              {
                     .gpioPinSel = TAL_GPIO_11
                     .polarity = 0
                     .enable=1
              }
              .orx2TxSel0Pin
              {
                     .gpioPinSel = TAL_GPIO_INVALID
                     .polarity = 0
                     .enable=0
              }
              .orx2TxSel1Pin
              {
                     .gpioPinSel = TAL_GPIO_INVALID
                     .polarity = 0
                     .enable=0
              }
              .enTxTrackingCals
              {
                     .gpioPinSel = TAL_GPIO_INVALID
                     .polarity = 0
                     .enable=0
              }
}
```

Figure 122 shows an example of a TDD use case where Observation Receiver 1 is used as the observation path. In this case, there are only two forward paths that are used for DPD and transmitter external LO leakage calibrations. Two permutations are controlled in real-time, which means that only the ORX1_TX_SEL0 signal must be controlled by a GPIO.

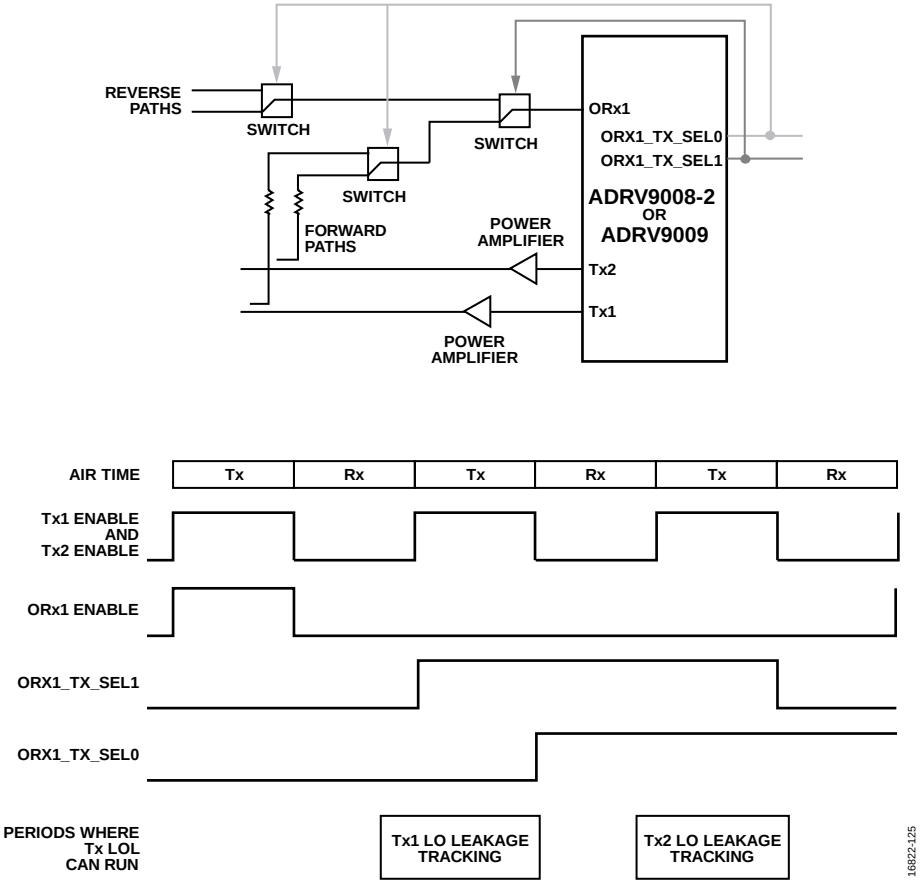

Figure 122. Possible Periods to Run the Transmitter LO Leakage Tracking Calibration in 2-Feedback Path to a 10-Receiver TDD Use Case

The ORX1_TX_SEL1 signal must be fixed through the SPI. To do this, call the following function prior to calling the `TALISE_setArmGpioPins()` function:

```
TALISE_setTxToOrxMapping(&taldevice, TAL_MAP_TX1_ORX, TAL_MAP_NONE)
```

To enable the Arm GPIOs in this configuration, the `taliseArmGpioConfig` structure is created as follows (assuming that GPIO_10 is used):

```
static taliseArmGpioConfig_t armGpio =
{
            .orx1TxSel0Pin
            {
                  .gpioPinSel = TAL_GPIO_10
                  .polarity = 0
                  .enable=1
            }
            .orx1TxSel1Pin
            {
                  .gpioPinSel = TAL_GPIO_INVALID
                  .polarity = 0
                  .enable=0
            }
```

```
                .orx2TxSel0Pin
                {
                        .gpioPinSel = TAL_GPIO_INVALID
                        .polarity = 0
                        .enable=0
                }
                .orx2TxSel1Pin
                {
                        .gpioPinSel = TAL_GPIO_INVALID
                        .polarity = 0
                        .enable=0
                }
                .enTxTrackingCals
                {
                        .gpioPinSel = TAL_GPIO_INVALID
                        .polarity = 0
                        .enable=0
                }
}
```

Figure 123 shows an example of a TDD use case where Transmitter 1 is fed back to Observation Receiver 1, and Transmitter 2 is fed back to Observation Receiver 2. In this case there are two feedback paths for two transmitters. Because the feedback paths are fixed, there is no need to control any of the ORX_TX_SEL signals in real-time. To configure this use case, call the following function during the initialization sequence:

```
TALISE_setTxToOrxMapping(&taldevice, TAL_MAP_TX1_ORX, TAL_MAP_TX2_ORX)
```

Note that for observation receiver bandwidths that are greater than 200 MHz (I/Q rate at 245.76 MSPS), it is not possible to switch between observation receivers (only one can used).

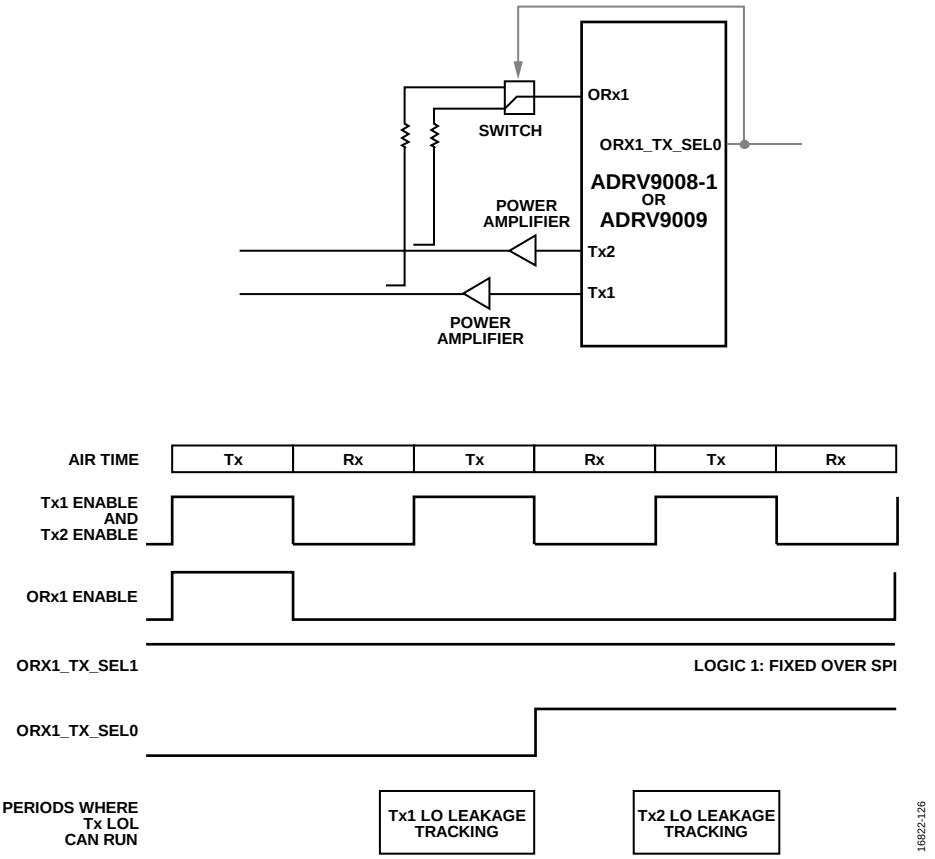

Figure 123. Possible Periods to Run Transmitter LO Leakage Tracking Calibration Two Receiver, Two Observation Receiver TDD Use Case

INITIALIZATION CALIBRATION ERRORS

This section describes the procedure for determining the error that occurs in the event of running of the initial calibrations. If an error occurs during an initial calibration, isolate the cause of the issue through the following description of error codes. Then, reinitialize the device with whatever necessary change made to the initialization procedure.

For example, if the external LO leakage initial calibration is run without the external feedback path complete, the calibration advises that it was unable to observe the transmitter channel, and that the calibration was unsuccessful. This error can be due to an external switch being in an incorrect position.

The following commands can be used to verify that these initial calibrations have been completed by the Arm processor, as well as to return error information from the initialization calibrations:

```
TALISE_waitInitCals(taliseDevice_t *device, uint32_t timeoutMs, uint8_t *errorFlag)
TALISE_checkInitCalComplete(taliseDevice_t *device, uint8_t *areCalsRunning, uint8_t *errorFlag)
```

In these commands, `*errorFlag` is used to indicate if an error has occurred. The returned error flag values are defined in Table 100.

Table 100. Error Flag Definitions Returned From waitInitCals()

errorFlag	Description
0x00	Command completed successfully.
0x01	Reserved.
0x02	Command not allowed in the radio on state. The calibrations are not run. The device must not be in transmit/receive state when initial calibrations are called.
0x03	Reserved.
0x04	Reserved.
0x05	RF PLL frequencies are not set prior to running initial calibrations. Calibrations were not run.
0x06	Initialization sequence interrupted by an abort command.
0x07	Calibration error.

If the error flag returns as 0x07, a calibration error has occurred. To determine which initial calibration is the source of the error, the following API function can be called:

```
TALISE_getInitCalStatus(taliseDevice_t *device, uint32_t *calsSincePowerUp, uint32_t
*calsLastRun, uint32_t *calsMinimum, uint8_t *initErrCal, uint8_t *initErrCode)
```

Parameters:

- `calsDoneLifetime` is a bit mask indicating all the initialization calibrations that have been run since the Arm was booted. For the definition of the bit mask, see Table 84.
- `calsDoneLastRun` is a bit mask indicating the specific calibrations that were run on the last call to `TALISE_runInitCals()`. For the definition of the bit mask, see Table 84.
- `calsMinimum` is a bit mask indicating the set calibrations that must be performed before the Arm allows the user to move the processor into the radio on state. For the definition of the bit mask, see Table 84.
- `initErrCal` is the code that indicates which calibration returned and error, if any, during initialization. See Table 101 for the mapping between value and type of calibration.}
- `initErrCode` is the exact error code returned by the calibration, if any has occurred, during `TALISE_runInitCals()`. See Table 102 to Table 111 for details of the possible errors returned.

Table 101. Mapping Between intErrCal and Failed Calibration

initErrCal	Calibration
0x00	Tx baseband filter calibration
0x01	ADC tuner calibration
0x02	Rx TIA filter calibration
0x03	Rx dc offset calibration
0x04	Tx attenuation delay
0x05	Rx gain delay
0x06	ADC flash calibration
0x07	Path delay calibration
0x08	Tx LO leakage initial calibration

initErrCal	Calibration
0x09	Tx LO leakage external initial calibration
0x0A	Tx QEC initial calibration
0x0B	Loopback ORx LO delay
0x0C	Loopback RxQEC initial calibration
0x0D	Rx LO delay
0x0E	Rx QEC initial calibration
0x0F	Not applicable
0x10	ORx LO delay
0x11	ORx QEC initial calibration
0x12	Tx DAC calibration
0x13	ADC stitching calibration

Transmitter Baseband Filter Calibration Errors

Table 102 describes the initial error flags returned during transmitter baseband filter calibration.

Table 102. initErrCodes Error Flags for Transmitter Baseband Filter Calibration

initErrCode Flag	Description
0	No error
1	Reserved
2	Calibration timed out

ADC Tuner Calibration Errors

Table 103 describes the initial error flags returned during ADC tuner calibration.

Table 103. initErrCodes Error Flags for ADC Tuner Calibration

initErrCode Flag	Description
0	No error
1	Calibration timed out

Receiver TIA Calibration Errors

Table 104 describes the initial error flags returned during receiver TIA calibration.

Table 104. initErrCodes Error Flags for Receiver TIA Calibration

initErrCode Flag	Description
0	No error
1	Error configuring PLL, ORx
2	Error during TIA Calibration, ORx
3	Error configuring PLL, Rx
4	Error during TIA calibration, Rx

Receiver DC Offset Calibration Errors

Table 105 describes the initial error flags returned during receiver dc offset calibration.

Table 105. initErrCodes Error Flags for Receiver DC Offset Calibration

initErrCode Flag	Description
0	No error
1	Calibration timed out, Rx
2	Calibration timed out, ORX
3	Calibration timed out, receive path loopback

ADC Flash Calibration Errors

Table 106 describes the initial error flags returned during ADC Flash calibration.

Table 106. initErrCodes Error Flags for ADC Flash Calibration

initErrCode Flag	Description
0	No error
1	Calibration aborted
2	Calibration timed out
3	No channel is selected
4	Rx is disabled
5	ORx is disabled

Path Delay Calibration Errors

Table 107 describes the initial error flags returned during path delay calibration.

Table 107. initErrCodes Error Flags for Path Delay Calibration

initErrCode Flag	Description
0	No error
1	Rx is disabled
2	Tx is disabled
3	Data captured timed out due to hardware setup
4	Data capture aborted

Transmitter LO Leakage Calibration (Internal and External) Errors

Table 108 describes the initial error flags returned during transmitter LO leakage calibration.

Table 108. initErrCodes Error Flags for Transmitter LO Leakage Calibration

initErrCode Flag	Description
0	No error
1	Reserved
2	Tx is disabled
3	Path delay not present (invalid)
4	Initial calibration not completed
5	Internal loopback tracking disabled
6	Data capture timed out due to hardware setup
7	Reserved
8	No Tx to ORx mapping

Receiver, Observation Receiver, and Loopback LO Delay Calibration Errors

Table 109 describes the initial error flags returned during the receiver, observation receiver, and loopback LO delay calibration.

Table 109. initErrCodes Error Flags for Receiver, Observation Receiver, and Loopback LO Delay Calibration

initErrCode Flag	Description
0	No error
1	Rx is disabled
2	Tx is disabled
3	CalPLL error
4	Reserved
5	Reserved
6	Reserved
7	Batch time too small

Receiver QEC Initial Calibration Errors

Table 110 describes the initial error flags returned during the receiver QEC initial calibration.

Table 110. initErrCodes Error Flags for Receiver QEC Initial Calibration

initErrCode	Description
0	No error
1	Rx is disabled
2	Tx is disabled
3	CalPLL error
4	Settling time error
5	Reserved
6	Reserved
7	Reserved
8	Batch time too small

Transmitter QEC Initial Calibration Errors

Table 111 describes the initial error flags returned during transmitter QEC calibration.

Table 111. initErrCodes Error Flags for Transmitter QEC Calibration

initErrCode Flag	Description
0	No error
1	Reserved
2	Tx is disabled
3	No path delay present

TRACKING CALIBRATION MONITORING

During operation, the Arm processor can be monitored with a variety of API functions. This section describes the information that is available from the Arm processor.

System Exception Monitoring Using the GP_INTERRUPT Pin

The GP_INTERRUPT pin alerts the user when errors occur within the device. This is a single pin that advises numerous potential errors, such as Arm errors, PLL unlocking events, and JESD204B errors. The functionality and configuration of the GP_INTERRUPT pin is covered in the General-Purpose Interrupt Operation section, which advises how to mask only certain events to trigger the GP_INTERRUPT pin.

This section also advises how to determine which error event has occurred. If the source of the error is an Arm error, reset and reinitialize the device.

Pending Calibrations and Determining Errors in Tracking Calibrations

The following function can be used to determine which calibrations are pending and if one of the tracking calibrations has returned an error:

```
TALISE_getPendingTrackingCals(taliseDevice_t *device, uint32_t *pendingCalMask)
```

In this function, `pendingCalMask` is the returned mask that advises if a calibration is pending or has returned an error as indicated in Table 112.

Table 112. pendingCalMask Bit Descriptions

pendingCalMask Bit	Description
D0	Rx1 QEC tracking pending bit
D1	Rx1 QEC tracking error bit
D2]	Rx2 QEC tracking pending bit
D3	Rx2 QEC tracking error bit
D4	ORx1 QEC tracking pending bit
D5	ORx1 QEC tracking error bit
D6	ORx2 QEC tracking pending bit
D7	ORx2 QEC tracking error bit
D8	Tx1 LO leakage tracking pending bit

pendingCalMask Bit	Description
D9	Tx1 LO leakage tracking error bit
D10	Tx2 LO leakage tracking pending bit
D11	Tx2 LO leakage tracking error bit
D12	Tx1 QEC tracking pending bit
D13	Tx1 QEC tracking error bit
D14	Tx2 QEC tracking pending bit
D15	Tx2 QEC tracking error bit

In the event of an error occurring during one of the calibrations, clear and reschedule the error using the `TALISE_rescheduleTrackingCal()` function.

Tracking Calibration Status Monitoring

Tracking calibration status monitoring that monitors how many times the calibration has run since inception, and what error the tracking calibration has returned.

Each tracking calibration also has its own API to return its current status, as follows:

```
TALISE_getTxLolStatus(taliseDevice_t *device, taliseTxChannels_t channelSel, taliseTxLolStatus_t *txLolStatus)
TALISE_getTxQecStatus(taliseDevice_t *device, taliseTxChannels_t channelSel, taliseTxQecStatus_t *txQecStatus)
TALISE_getRxQecStatus(taliseDevice_t *device, taliseRxChannels_t channelSel, taliseRxQecStatus_t *rxQecStatus)
TALISE_getOrxQecStatus(taliseDevice_t *device, taliseObsRxChannels_t channelSel, taliseOrxQecStatus_t *orxQecStatus)
TALISE_getRxHd2Status(taliseDevice_t *device, taliseRxChannels_t channelSel, taliseRxHd2Status_t *rxHd2Status)
```

The `channelSel` parameter indicates which channel status to return. It is only possible to read back one channel at a time. For this reason, only certain elements of the `taliseTxChannels_t`, `taliseRxChannels_t`, and `taliseObsRxChannels_t` data structures are applicable (see Table 113).

Table 113. Applicable Enumerators for Specified channelSel Tracking Calibration Status Functions

Enumerator	taliseTxChannels_t Channels	taliseRxChannels_t Channels	taliseObsRxChannels_t Channels
TAL_TX1	Tx1	Not applicable	Not applicable
TAL_TX2	Tx2	Not applicable	Not applicable
TAL_RX1	Not applicable	Rx1	Not applicable
TAL_RX2	Not applicable	Rx2	Not applicable
TAL_ORX1	Not applicable	Not applicable	ORx1
TAL_ORX2	Not applicable	Not applicable	ORx2

The individual status types (`taliseTxLolStatus_t`, `taliseTxQecStatus_t`, `taliseRxQecStatus_t`, `taliseOrxQecStatus_t`, and `taliseRxHd2Status_t`) are all equivalent and are composed of the parameters described in Table 114.

Table 114. Tracking Calibration Status Type Definitions

Type	Data Field	Description
uint32	errorCode	The returned error code from the calibration algorithm. 0 indicates no error.
uint32	percentComplete	The percent of the required data collected for the current instance of the tracking calibration. Range of field: 0 to 100.
uint32	Metric (see Table 115)	A metric is provided that can provide debug information of operation of the algorithm. The name and measurement used for this metric differs per calibration. See Table 115 for metric descriptions.
uint32	iterCount	A counter that updates each time a calibration has completed.
uint32	updateCount	A counter that updates each time a calibration updates the correction being applied in the correction hardware.

Table 115. Calibration Status Metric Descriptions

Calibration Status	Metric Name	Description
taliseTxLolStatus_t	varianceMetric	A metric related to the inverse of the variance of the measured LO leakage. Higher values are indicative of lower variance, which can indicate better performance (dependent on signal conditions). This metric can be useful in debug environments if questions of performance exist.
taliseTxQecStatus_t	correctionMetric	A metric related to the phase and gain adjustments made by the QEC algorithm. Smaller adjustments can be indicative of better performance (dependent on signal conditions). This metric can be useful in debug environments if questions of performance exist.
taliseRxQecStatus_t	selfCheckIrrDb	An estimate of the current QEC performance reported as an image rejection ratio (in dB). A readback of 80 advises that the current estimated QEC performance is 80 dBc. The performance is measured as the power weighted average QEC performance across the entire band. It is measured periodically after the correction block. Before the first calculation, this metric reads back 0xFFFFFFFF − 1. This metric can be useful in debug environments if questions of performance exist.
taliseOrxQecStatus_t	selfCheckIrrDb	An estimate of the current QEC performance reported as an image rejection ratio (in dB). A readback of 80 advises that the current estimated QEC performance is 80 dBc. The performance is measured as the power weighted average QEC performance across the entire band. It is measured periodically after the correction block. Before the first calculation, this metric reads back 0xFFFFFFFF − 1. This metric can be useful in debug environments if questions of performance exist.
taliseRxHd2Status_t	confidenceLevel	The confidence level on the correction coefficient applied to cancel HD2. Until the calibration makes its first observation, this metric reads back 0. Otherwise, the confidence level is between 1 and 7, where 7 indicates the highest confidence. This metric can be useful in debug environments if questions of performance exist.

Transmitter QEC Tracking Calibration Errors

Table 116 describes the error flags that are returned during transmitter QEC tracking calibration.

Table 116. Transmitter QEC Tracking Calibration Error Flags

initErrCode Flag	Description
0	No error
0x30001	Data capture failed
0x30002	Tx is disabled
0x30003	Initial calibration not performed
0x30004	Numerically controlled oscillator failed to lock
0x30005	Batch time too small

Receiver/Observation Receiver QEC Tracking Calibration Errors

Table 117 describes the error flags that are returned during receiver/observation receiver QEC tracking calibrations.

Table 117. Receiver/Observation Receiver QEC Tracking Calibration Error Flags

initErrCode Flag	Description
0	No error
1	Reserved
2	Data capture error
3	Batch time too small

Receiver HD2 Tracking Calibration Errors

Table 118 describes the error flags that are returned during receiver HD2 tracking calibrations.

Table 118. Receiver HD2 Tracking Calibration Error Flags

initErrCode Flag	Description
0	No error
1	No Rx selected

READING THE ARM VERSION

When the Arm processor is booted up, it is possible to read back the Arm version using the following function:

```
TALISE_getArmVersion(taliseDevice_t *device, uint8_t *majorVer, uint8 minorVer, unit8_t *rcVer)
```

Parameters include the following:

- `majorVer` is the major version of the Arm build.
- `minorVer` is the minor version of the Arm build.
- `rcVer` is the release candidate version (build number).

Each Arm build has a unique combination of these versions.

PERFORMING AN ARM MEMORY DUMP

As noted in the General-Purpose Interrupt Operation section of this user guide, the Arm processor uses the GP_INTERRUPT pin to advise if the processor detects an error. At this stage, perform an Arm memory dump, and then provide this dump to Analog Devices for diagnostics. There is no API written to perform a full Arm memory dump because the API is written to be file system agnostic.

Example code is supplied in the Example Code for Performing an Arm Memory Dump Operation section. This code reads the Arm memory and writes the binary byte data directly to a binary file. Note that an exception is forced if an exception has not already occurred. When an exception occurs, important diagnostic information is stored in the Arm memory. In the event of the Arm being dumped for debug in situations where an exception has not occurred, this code calls an exception such that this diagnostic information is stored before the Arm memory is dumped.

Example Code for Performing an Arm Memory Dump Operation

```
/// <summary>
        /// Reads the ARM Memory and writes the binary byte array directly to a binary file.  Fi
rst 114688 bytes are program memory followed
        /// by 81920 bytes of data memory.  The binaryFilename is opened before reading the ARM
memory to verify that the filepath is has valid write access
        /// before reading ARM memory.  A file IO exception will be thrown if write access is no
t valid for the binaryFilename path.
        /// </summary>
        /// <param name="binaryFilename">File path to save the binary data.  Make sure you have
write access to the location.</param>
        /// <exception cref="InvalidOperationException">Thrown if TCPIP is not connected</except
ion>
        public void DumpArmMemory(string binaryFilename)
        {
            if (this.dllScriptAction != AdiCommandServerClient.DllScriptActions.ExecuteOnly)
            {
                Logging.LogApi("Link.Talise.DumpArmMemory", binaryFilename);
                if (this.dllScriptAction == AdiCommandServerClient.DllScriptActions.LogOnly)
                {
                    return;
                }
            }
            if (this.hw.Connected)
            {
                const UInt32 armExceptionAddr = 0x0101BFF0;

                //Write in BINARY FILE format
                String filename = binaryFilename;
                System.IO.FileStream fileStream = new System.IO.FileStream(filename, System.IO.F
ileMode.Create, System.IO.FileAccess.Write);

                byte[] programMem = new byte[114688];
                byte[] dataMem = new byte[81920];

                //Check if exception has occurred
                byte[] exceptionArray = new byte[4];
                this.ReadArmMem(armExceptionAddr, 4, 0, ref exceptionArray);
                UInt32 exceptionValue = (UInt32)(exceptionArray[0] | (exceptionArray[1] << 8) |
(exceptionArray[2] << 16) | (exceptionArray[3] << 24));

                byte armException = 0;
                //TODO: read GP Interrupt status [4] to see if ARM interrupt occurred


                if (exceptionValue == 0)
                {
                    //Force an exception during ARM MEM dump for more useful information
                    byte armMailboxReady = 0;
```

```
                    this.ReadEventStatus(WaitEvent.ARMBUSY, ref armMailboxReady);
                    if (armMailboxReady == 1)
                    {
                        this.SendArmCommand(0x0A, new byte[] { 0x69 }, 1);

                        System.Diagnostics.Stopwatch stopWatch = new System.Diagnostics.Stopwatc
h();
                        stopWatch.Start();
                        while (exceptionValue == 0)
                        {
                            //TODO: add call to get GP Interrupt status [4]
                            //armException = GP INt status[4]
                            this.ReadArmMem(armExceptionAddr, 4, 0, ref exceptionArray);
                            exceptionValue = (UInt32)(exceptionArray[0] | (exceptionArray[1] <<
8) | (exceptionArray[2] << 16) | (exceptionArray[3] << 24));

                            //timeout to break while loop
                            if (stopWatch.ElapsedMilliseconds > 5000)
                            {
                                break;
                            }
                        }
                        armException = 0;
                        stopWatch.Stop();
                    }
                }

                this.ReadArmMem(0x01000000, programMem.Length, 0, ref programMem);
                this.ReadArmMem(0x20000000, dataMem.Length, 0, ref dataMem);

                if (armException == 0)
                { //if we forced an exception, clear the exception so the ARM will continue to r
un.
                    //TODO: Write x13C1[0] = 0 to clear forced ARM exception
                    this.WriteArmMem(armExceptionAddr, 4, new byte[] { 0, 0, 0, 0 });
                }

                fileStream.Write(programMem, 0, programMem.Length);
                fileStream.Write(dataMem, 0, dataMem.Length);

                fileStream.Close();
            }
            else
            {
                throw new InvalidOperationException("No Hardware Connection");
            }
        }
```

FILTER CONFIGURATION

This section describes the digital filters within the devices, and provides a description of each of the filters in terms of filter coefficients and positions within the signal chain. The API structures are also described in this section and an example profile specific configuration is provided for each of the signal chains, as well as a description of the API functions that are used to configure the filters.

RECEIVER SIGNAL PATH

The ADRV9008-1 and ADRV9009 have independent signal paths for the Receiver 1 and Receiver 2 ports. Each receiver signal path consists of separate I/Q mixers that feed into programmable TIAs serving as low-pass filters in the analog datapath. The signals are then converted by the Σ-Δ ADCs, and filtered in half-band decimation stages and the programmable finite impulse response filter (RFIR). The fixed coefficient half-band filters (RHB1, RHB2, RHB3, and DEC5) and the RFIR are designed to prevent data wrapping and overrange conditions.

The IF conversion stage the receiver with the ability to frequency shift or upsample/downsample digital data. Configurations supported include real IF (real valued baseband data) configuration and low IF (complex data) configuration.

Figure 124 shows the signal path for the Receiver 1 and Receiver 2 signal chain. The 90° block, LO generator, IADC, QADC, QEC correction, dc correction, and digital gain blocks are not discussed in this reference manual.

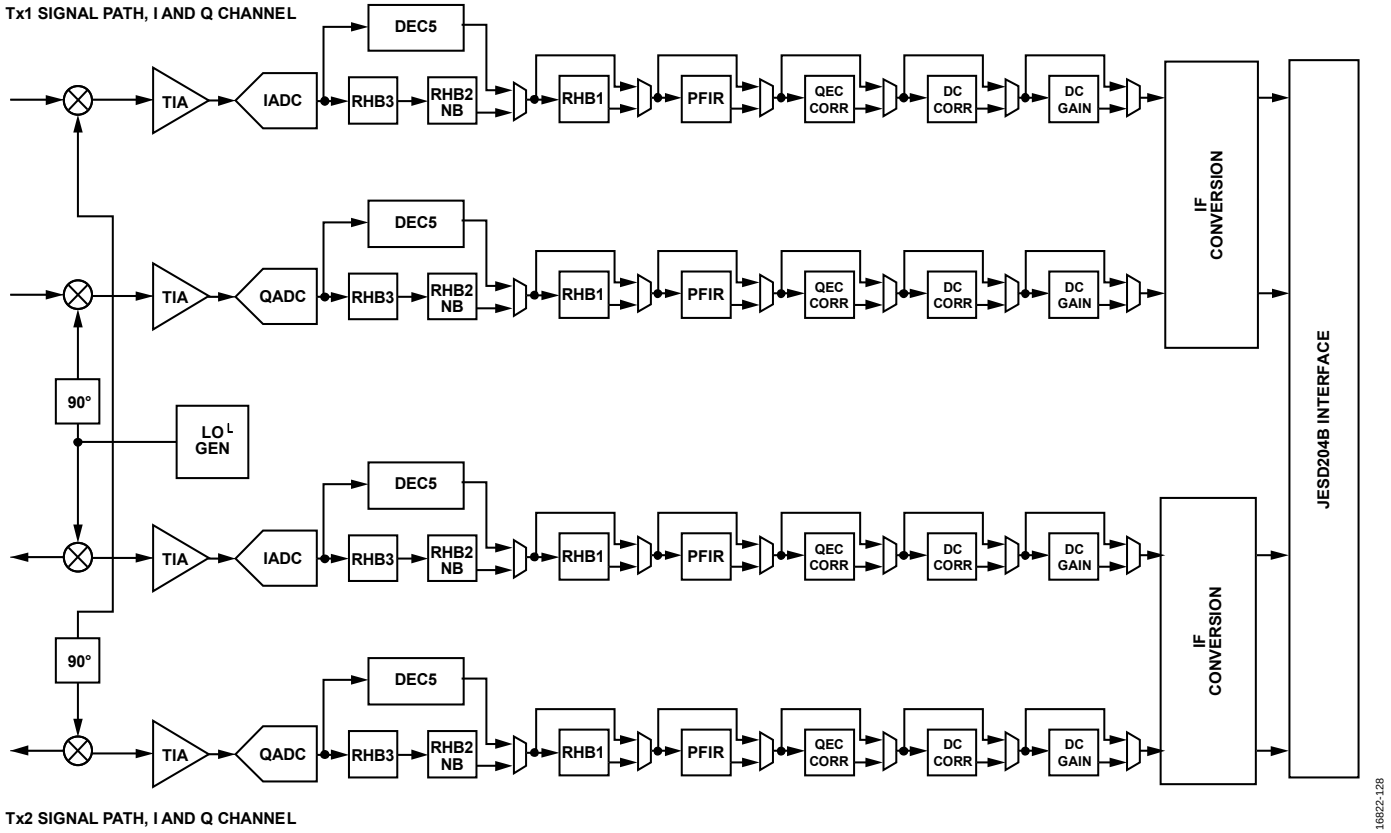

Figure 124. Receiver 1 and Receiver 2 Signal Paths

RECEIVER TRANSIMPEDANCE AMPLIFIER (TIA)

The receiver transimpedance amplifier is a low-pass filter with a single, real pole frequency response. The ADRV9008-1 and ADRV9009 support bandwidths up to 200 MHz and each TIA supports a pass-band bandwidth of 100 MHz on the I and Q paths. The TIA is calibrated during device initialization to ensure a consistent frequency corner across all devices. The TIA 3 dB bandwidth is set within the device data structure and is profile dependent. Roll-off within the receiver pass band is compensated by the RFIR to ensure a maximally flat pass band frequency response.

RECEIVE DEC5

Either the DEC5 filter or the combination of RHB3 and RHB2 is used in the receiver digital path. The DEC5 filter decimates by a factor of 5. The DEC5 filter coefficients are as follows:

[0.002197, 0.004272, 0.006836, 0.008789, 0.008545, 0.003418, −0.004639, −0.015381, −0.025512, −0.029785, −0.022461, −0.002441, 0.03125, 0.074707, 0.119141, 0.155396, 0.176758, 0.176758, 0.155396, 0.119141, 0.074707, 0.03125, −0.002441, −0.022461, −0.029785, −0.025512, −0.015381, −0.004639, 0.003418, 0.008545, 0.008789, 0.006836, 0.004272, 0.002197]

RECEIVE HALF-BAND 3 (RHB3) FILTER

The RHB3 filter is a fixed, coefficient decimating filter that decimates by a factor of 2. The RHB3 coefficients are as follows.

[−0.01874, −0.04218, 0.050476, 0.293884, 0.439636, 0.293884, 0.050476, −0.04218, −0.01874]

RECEIVE HALF-BAND 2, NARROW-BAND (RHB2) FILTER

The RHB2 narrow-band filter is a fixed, coefficient decimating filter that decimates by a factor of 2. The RHB2 coefficients are as follows.

[0.003174, 0, −0.01239, 0, 0.03418, 0, −0.08551, 0, 0.310913, 0.5, 0.310913, 0, −0.08551, 0, 0.03418, 0, −0.01239, 0, 0.003174]

RECEIVE HALF-BAND 1 (RHB1) FILTER

The RHB1 filter is a fixed, coefficient decimating filter that can decimate by a factor of 2, or it can be bypassed. The RHB1 coefficients are as follows:

[−0.000122, 0, 0.000244, 0, −0.000488, 0, 0.000854, 0, −0.001221, 0, 0.001831, 0, −0.002502, 0, 0.003479, 0, −0.004700, 0, 0.006287, 0, −0.008179, 0, 0.010620, 0, −0.013611, 0, 0.017578, 0, −0.022766, 0, 0.030029, 0, −0.040955, 0, 0.059998, 0, −0.103027, 0, 0.313721, 0.493652, 0.313721, 0, −0.103027, 0, 0.059998, 0, −0.040955, 0, 0.030029, 0, −0.022766, 0, 0.017578, 0, −0.013611, 0, 0.010620, 0, −0.008179, 0, 0.006287, 0, −0.004700, 0, 0.003479, 0, −0.002502, 0, 0.001831, 0, −0.001221, 0, 0.000854, 0, −0.000488, 0, 0.000244, 0, −0.000122]

RECEIVER FINITE IMPULSE RESPONSE (RFIR) FILTER

The programmable RFIR filter acts as a decimating filter that can decimate by a factor of 1, 2, or 4, or it can be bypassed. The RFIR is used to compensate for the roll-off of the analog TIA low-pass filter. The RFIR can use either 24, 48, or 72 filter taps.

The maximum number of taps is limited by the FIR clock rate (data processing clock, DPCLK). The maximum DPCLK clock rate is 500 MHz. The DPCLK clock rate is the ADC clock rate divided by 4 or 5. The ADC clock rate is divided by 4 when using the HB2 and HB3 filters, and is divided by 5 when using the DEC5 filter. The DPCLK clock rate affects the maximum number of RFIR filter taps that can be used, as shown in the following equation.

$$\textit{Maximum Number of RFIR Filter Taps} = (\textit{DPCLK Clock Rate} \div \textit{Receiver I/Q Data Rate}) \times 24$$

The RFIR also has programmable gain setting of +6 dB, 0 dB, −6 dB, or −12 dB.

RECEIVER IF CONVERSION

The IF conversion stage provides the user with the ability to change how the received data is presented to the JESD204B port. Figure 125 shows a block diagram of the IF conversion stage. There are two parallel paths where data can be processed, referred to as Band A and Band B. In the circuitry of each band, there are two mixer stages, allowing upshifting or downshifting, interpolation and decimation stages, and a half-band filter with a pass-band of 0.4 × sample rate. The coefficients of the half-band filter in this IF conversion stage are as follows:

[-9.1553×10^{-5}, 0, 2.4414×10^{-4}, 0, -5.7983×10^{-4}, 0, 0.0012, 0, −0.0023, 0, 0.0040, 0, −0.0065, 0, 0.0103, 0, −0.0157, 0, 0.0236, 0, −0.0357, 0, 0.0563, 0, −0.1015, 0, 0.3168, 0.5000, 0.3168, 0, −0.1015, 0, 0.0563, 0, −0.0357, 0, 0.0236, 0, −0.0157, 0, 0.0103, 0, −0.0065, 0, 0.0040, 0, −0.0023, 0, 0.0012, 0, -5.7983×10^{-4}, 0, 2.4414×10^{-4}, 0, -9.1553×10^{-5}]

IF Conversion Use Cases

The following use cases provide an example of the types of functionality supported by this block.

In the complex low IF to zero IF use case, the received signal is offset from LO such that the entire signal of interest is on one side of the LO. The Band A, Numerically Controlled Oscillator 1, is used to downshift the signal such that it is centered at 0 Hz. There is a half-band filter and decimate by 2 stage, that decreases the bandwidth and the I/Q rate if used. This decimation reduces the number of JESD204B lanes required, or the rate that the lanes need to be run at.

Figure 126 shows the IF conversion stage configuration for a receiver profile of 100 MHz with an I/Q rate of 122.8MHz, which is configured for reception of a total of 200 MHz RF bandwidth to receive an offset multicarrier GSM signal. For a +75 MHz bandwidth, multicarrier GSM signal, the center frequency is +52.5 MHz offset from the LO, so that the band occupies from ±15 MHz to ± 90 MHz. The receiver then uses the IF conversion stage to shift the signal so that it is centered at about 0 Hz, filters with the half-band filter, and decimates the output by two, so that the I/Q rate that is sent over the JESD204B is 122.88 MSPS.

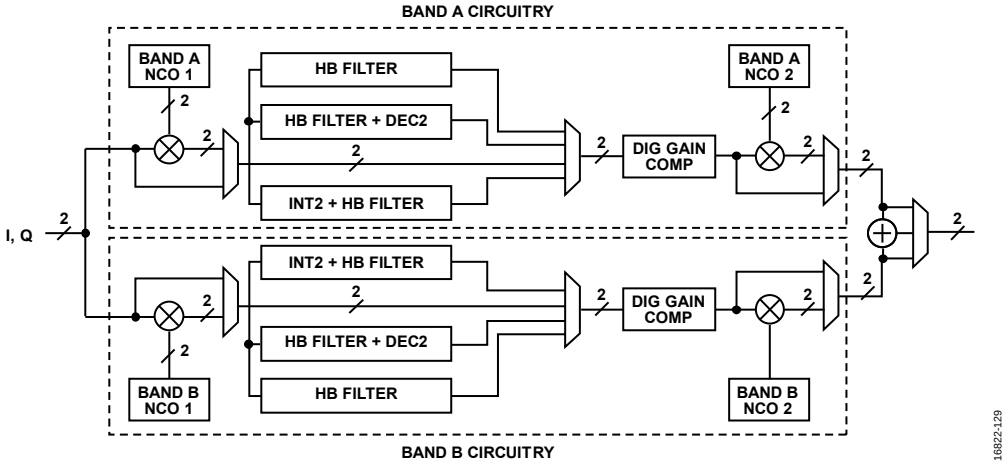

Figure 125. Block Diagram of the IF Conversion Stage (All Circuitry is Implemented in Quadrature as Indicated)

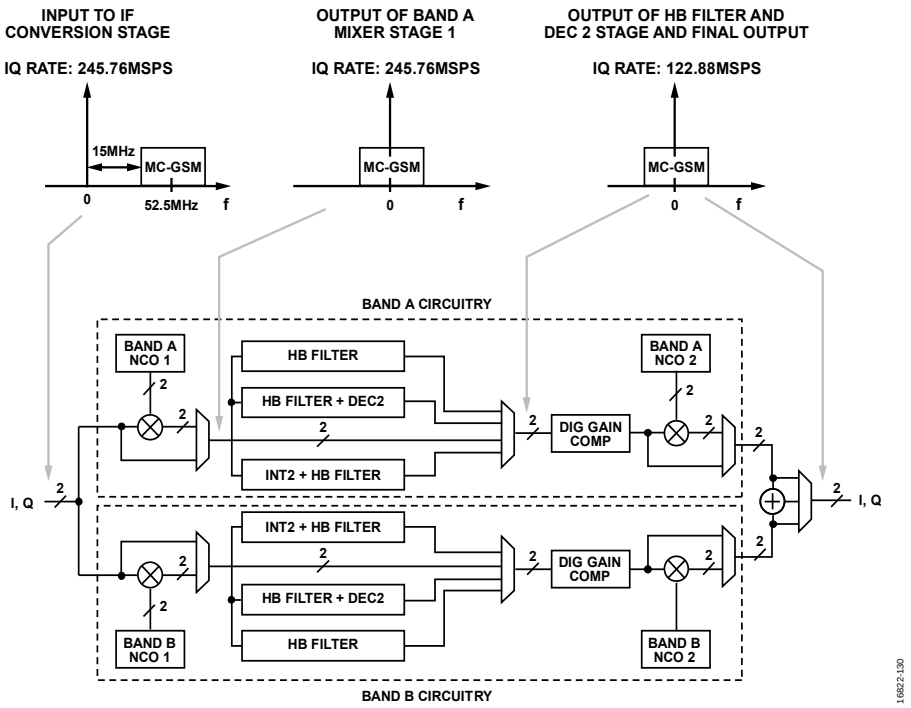

Figure 126. Block Diagram of the IF Conversion Stage in Zero IF, Multicarrier GSM Configuration

RECEIVER SIGNAL PATH EXAMPLE

The TTES provides an example depicting how the baseband filtering stages are used in profile configurations for a signal pathway. In this example, the receiver profile of 200 MHz with an I/Q rate of 245.76 MHz profile is selected for the receiver channels.

Figure 127 shows the filter configuration for this profile. The signal rate shown after the RFIR block is equal to the I/Q rate of the profile.

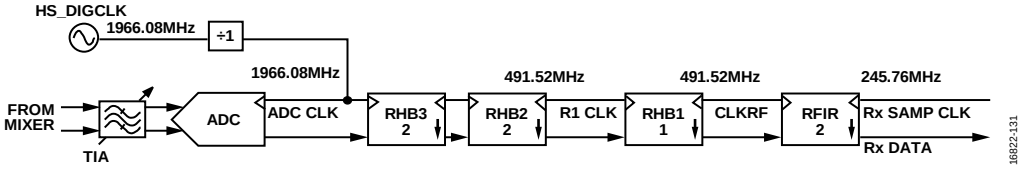

Figure 127. Filter Configuration for the Receiver Profile of 200 MHz with an I/Q Rate of 245.76 MHz

A graphed frequency response of the TIA, digital filters, ADC transfer function, and the composite response from dc to the sampling rate of the ADC is available in the TTES in the **Rx Summary** tab, as shown in Figure 128.

Tx FILTER RESPONSE

PROFILE PASS BAND (MHz)	3dB CORNER (MHz)	DC GAIN (dB)	PASS-BAND RIPPLE (dB)
100	118.08	–0.002	0.011

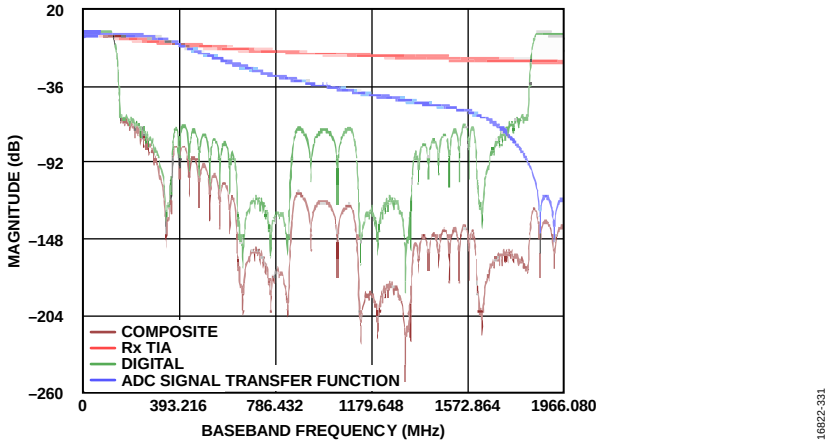

Figure 128. Receiver Filter Responses

An examination of the profile pass band frequency shows that the receiver TIA 3 dB setting slightly attenuates information within the pass band. This analog attenuation is compensated by the digital filter response to obtain a maximally flat pass band for this profile. A zoomed in view of the pass band is shown in Figure 129.

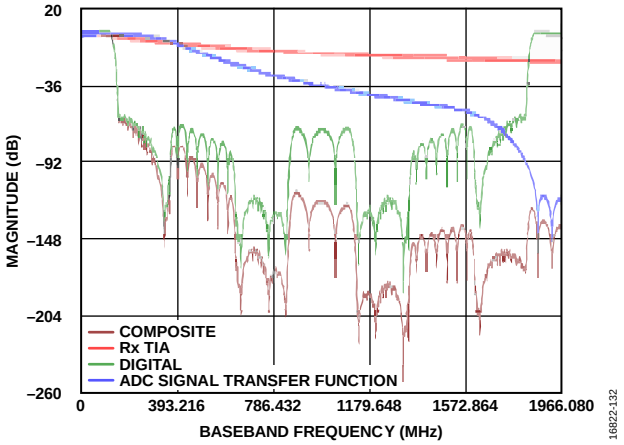

Figure 129. Examination of the Pass Band Frequency Response of a Receiver Profile of 200 MHz with an I/Q Rate of 245.76 MHz Profile

Receiver Filter API Structure

The filter configuration is stored in the `taliseRxProfile_t` structure. This structure is stored within the `taliseRxSettings_t` structure, which is stored in the `taliseInit_t` structure and contains the parameters described in Table 119.

Table 119. taliseRxProfile_t Structure Parameters

Type	Data Field	Permissible Values	Description
taliseFir_t	rxFIR	See Table 120.	See Table 120.
uint8_t	rxFirDecimation	1, 2, and 4	RFIR decimation setting.
unit8_t	rxDec5Decimation	4, 5	Setting to use either the DEC5 or HB3 and HB2 in the ORx path. 5: Use DEC5. 4: Use HB3 and HB2.
uint8_t	rhb1Decimation	1, 2	Rx HB1 decimation setting. 1 means bypass, 2 means in use.
uint32_t	rxOutputRate_kHz	46,080 to 307,200	I/Q data rate (to the input of the JESD204B block).
uint32_t	rfBandwidth_Hz	20,000,000 to 200,000,000	The RF bandwidth specified in Hz.
uint32_t	rxBbf3dBCorner_kHz	20,000 to 200,000	The baseband frequency 3 dB corner frequency specified in kHz.
uint16_t	rxAdcProfile	Valid ADC arrays are provided for specific use cases through the GUI software init.c files or Python scripts.	A 42-element array that provides the profile for the loop filter of the Σ-Δ ADC.
taliseRxDdc_t	rxDdcMode	See Table 121.	See Table 121.
taliseRxNcoShifterCfg_t	rxNcoShifterCfg	See Table 122.	See Table 122.

The permissible values provided in Table 120 are based on the currently defined profiles/use cases. A given profile has a specific combination of these values.

Receiver FIR

The `taliseFir_t` structure is contained within the `taliseRxProfile_t` structure and contains the parameters described in Table 120.

Table 120. taliseFir_t rxFir Structure Parameters

Type	Data Field	Permissible Values	Description
int8_t	gain_dB	+6, 0, −6, −12	The setting (in dB) for the gain block within the RFIR.
uint8_t	numFirCoefs	24, 48, 72	Number of taps to be used in the RFIR.
int16_t *	coefs	Not applicable	A pointer to an array of filter coefficients.

The receiver FIR is specified in signed coefficients from +32,767 to −32,768. The gain block allows more flexibility when designing a digital filter. For example, a FIR can be designed with a gain of 6 dB in the pass band, and then this block can be set to a gain of −6 dB to give an overall gain of 0 dB in the pass band. The gain of the filter coefficients (Σ FIR coefficients) can be calculated as follows:

$$DC\ Gain = \frac{\Sigma\ FIR\ Coefficients}{2^{15} - 1}$$

Receiver Digital Downconverter (DDC) Mode

The receiver DDC mode is defined within the `taliseRxProfile_t` structure as an enumerated type from the `taliseRxDdc_t` type definition. The permissible values are shown in Table 121.

Table 121. Permissible Settings of Receiver DDC Mode

Permissible Enumerator Values	Description
TAL_RXDDC_BYPASS	In this mode, the half-band filter and interpolation/decimation stages are bypassed.
TAL_RXDDC_FILTERONLY	In this mode, the half-band filter stage is used, but the interpolation and decimation stages are bypassed.
TAL_RXDDC_INT2	In this mode, the interpolate by 2 and half-band filter stages are utilized.
TAL_RXDDC_DEC2	In this mode, the half-band filter and decimate by 2 stages are utilized.
TAL_RXDDC_BYPASS_REALIF	In this mode, the half-band filter and interpolation/decimation stages are bypassed. At the input to the JESD204B core, Q data is dropped.
TAL_RXDDC_FILTERONLY_REALIF	In this mode, the half-band filter stage is used, but the interpolation and decimation stages are bypassed. At the input to the JESD204B core, Q data is dropped.
TAL_RXDDC_INT2_REALIF	In this mode, the interpolate by 2 and half-band filter stages are utilized. At the input to the JESD204B core, Q data is dropped.
TAL_RXDDC_DEC2_REALIF	In this mode, the half-band filter and decimate by 2 stages are utilized. At the input to the JESD204B code, Q data is dropped.

Receiver NCO Shifter Configuration

The `taliseRxNcoShifterCfg_t` structure is contained within the `taliseRxProfile_t` structure. It contains the settings of the NCO stages of Band A and Band B, as well as the bandwidth and baseband center frequency of the desired signal(s). This allows the API to ensure that the IF conversion stage has been correctly setup, and that the signal(s) post NCO shifting is falling within the bandwidth provided by the I/Q rate being utilized, and the pass band bandwidth of the half-band filter if utilized.

Table 122. Description of the taliseRxNcoShifterCfg_t structure

Type	Data Field	Description
uint32_t	bandAInputBandwidth_kHz	The bandwidth of the received signal being processed in Band A, specified in kHz.
int32_t	bandAInputCenterFreq_kHz	The center frequency, in terms of baseband frequencies, of the received signal being process in Band A, specified in kHz.
int32_t	bandANco1Freq_kHz	The frequency shift to be provided by NCO1 of Band A, specified in kHz. Positive values shift the spectrum up in frequency; negative values shift the spectrum down in frequency.
int32_t	bandANco2Freq_kHz	The frequency shift to be provided by NCO2 of Band A, specified in kHz. Positive values shift the spectrum up in frequency; negative values shift the spectrum down in frequency.
uint32_t	bandBInputBandwidth_kHz	The bandwidth of the received signal being processed in Band B, specified in kHz.
int32_t	bandBInputCenterFreq_kHz	The center frequency, in terms of baseband frequencies, of the received signal being process in Band B, specified in kHz.
int32_t	bandBNco1Freq_kHz	The frequency shift to be provided by NCO1of Band B, specified in kHz. Positive values shift the spectrum up in frequency; negative values shift the spectrum down in frequency.
int32_t	bandBNco2Freq_kHz	The frequency shift to be provided by NCO2 of Band B, specified in kHz. Positive values shift the spectrum up in frequency; negative values shift the spectrum down in frequency.

Note that dual-band mode is selected when the input bandwidths of Band A and Band B are both specified, for example, are non-zero. In nondual band modes, only specify the settings for Band A, and leave Band B with zero settings. If the NCO stages of both Band A and Band B are not to be used, provide zero settings for all variables in the `taliseRxNcoShifterCfg_t` structure.

TRANSMITTER SIGNAL PATH

The ADRV9008-2 and ADRV9009 transmitters have independent signal paths for the Transmitter 1 and Transmitter 2 channels. Data is input to the transmitter signal path via the JESD204B, high speed, serial data interface at the I/Q data rate of the transmitter profile. The serial data is converted to parallel format through the JESD204B deframer into I and Q components, is processed through digital filtering and signal correction stages, and input to the IDACs and QDACs.

The DAC output is filtered by the transmitter low-pass filter, and input to the upconversion mixer. The I and Q paths are identical to one another. Overranging is detected in the transmitter digital signal path at each stage and limited to the maximum code value to prevent data wrapping. A block diagram of the Transmitter 1 and Transmitter 2 signal paths is shown in Figure 130. The 90° block, LO generator (LO GEN), IADC, QADC, QEC correction, dc correction, and digital gain (DIG GAIN) blocks are not discussed in this reference manual.

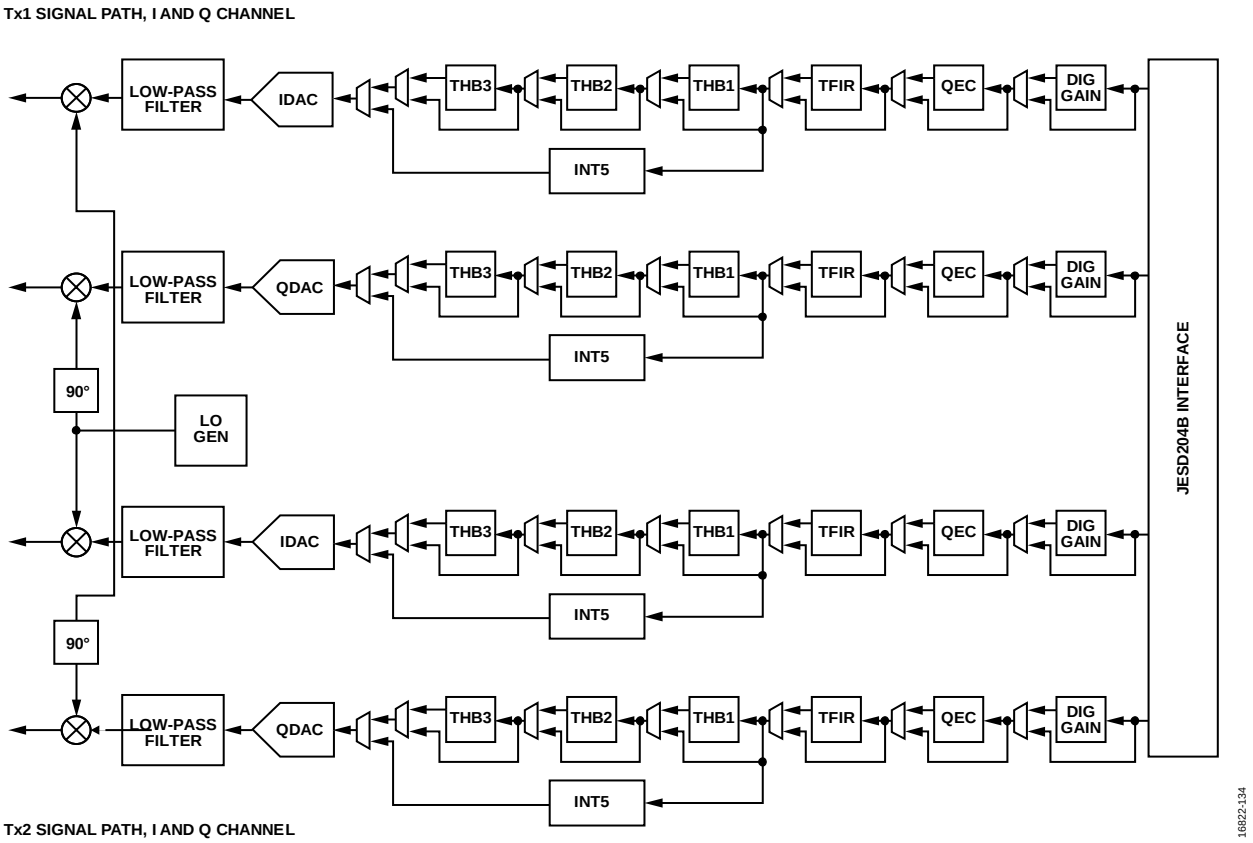

Figure 130. Transmitter 1 and Transmitter 2 Signal Path Block Diagram

Low-Pass Filter

The low-pass filter is an analog, second-order, Butterworth, low-pass filter with an adjustable 3 dB corner. The transmitter chains can support pass band bandwidths of up to 225 MHz each on I and Q. The low-pass filter is calibrated during device initialization, resulting in a consistent frequency corner across all devices. The low-pass filter bandwidth is set within the device data structure, and is profile dependent. Roll-off within the analog low-pass filter pass band is compensated by the TFIR to ensure a maximally flat, pass band frequency response.

INT5 Filter

Either the INT5, or any combination of the THB3, THB2, and THB1 filters is used in the transmitter digital path. The INT5 filter interpolates by a factor of 5. The INT5 filter coefficients are as follows:

[+0.002930, +0.029053, −0.029297, +0.031250, −0.012207, −0.005859, −0.056641, +0.051514, −0.055664, +0.025391, +0.020996, +0.081299, −0.057617, +0.072510, −0.045166, −0.047607, −0.095947, +0.030518, −0.071289, +0.068604, +0.093994, +0.113770, +0.030762, +0.055420, −0.103760, −0.185791, −0.185303, −0.136963, −0.037354, +0.227051, +0.518555, +0.717285, +0.928467, +1.019287, +0.928467, +0.717285, +0.518555, +0.227051, −0.037354, −0.136963, −0.185303, −0.185791, −0.103760, +0.055420, +0.030762, +0.113770, +0.093994, +0.068604, −0.071289, +0.030518, −0.095947, −0.047607, −0.045166, +0.072510, −0.057617, +0.081299, +0.020996, +0.025391, −0.055664, +0.051514, −0.056641, −0.005859, −0.012207, +0.031250, −0.029297, +0.029053, +0.002930]

Transmit Half-Band 3 (THB3) Filter

The THB3 filter is a fixed coefficient, half-band, interpolating filter that can interpolate by a factor of 2 or can be bypassed. The coefficients are as follows:

[0.125, 0.5, 0.75, 0.5, 0.125]

Transmit Half-Band 2 (THB2) Filter

The THB2 filter is a fixed coefficient, half-band, interpolating filter that can interpolate by a factor of 2 or can be bypassed. The coefficients are as follows:

[−0.082031, 0, +0.582031, +1, +0.582031, 0, −0.082031]

Transmit Half-Band 1 (THB1)

The THB1 filter is a fixed coefficient, half-band, interpolating filter that can interpolate by a factor of 2, or can be bypassed. The coefficients are as follows:

[−0.002319, 0, +0.003601, 0, −0.004059, 0, +0.004120, 0, −0.006439, 0, +0.009613, 0, −0.012024, 0, +0.014404, 0, −0.018738, 0, +0.024292, 0, −0.030060, 0, +0.037354, 0, −0.048157, 0, +0.062927, 0, −0.084351, 0, +0.122284, 0, −0.209564, 0, +0.635925, +1.000000, +0.635925, 0, −0.209564, 0, +0.122284, 0, −0.084351, 0, +0.062927, 0, −0.048157, 0, +0.037354, 0, −0.030060, 0, +0.024292, 0, −0.018738, 0, +0.014404, 0, −0.012024, 0, +0.009613, 0, −0.006439, 0, +0.004120, 0, −0.004059, 0, +0.003601, 0, −0.002319]

Transmitter Finite Impulse Response (TFIR) Filter

The programmable TFIR filter acts as an interpolating filter in the transmitter path. The TFIR can interpolate by a factor of 1, 2, or 4, or it can be bypassed. The TFIR is used to compensate for roll-off caused by the post DAC, analog low-pass filter. The TFIR has a configurable number of taps: 20, 40, 60, or 80 taps can be used.

The maximum number of taps is limited by the TFIR clock rate, which is derived from the data processing clock, DPCLK. The maximum DPCLK clock rate is 500 MHz. The DPCLK clock rate is the high speed digital clock, HSDIG_CLK, divided by 4 or 5, depending on the HSDIG_CLK divider setting. The DPCLK clock rate affects the maximum number of TFIR filter taps that can be used, as shown in the following equation:

$$Maximum\ Number\ of\ Transmitter\ FIR\ Filter\ Taps = (DPCLK\ Clock\ Rate/Transmitter\ I/Q\ Data\ Rate) \times 20$$

The TFIR also has a programmable gain setting of + 6 dB, 0 dB, −6 dB, or −12 dB.

Transmitter Signal Path Example

The TTES provides an example that shows how the baseband filtering stages are used in profile configurations for a signal datapath. In this example, the transmitter profile of 200 MHz or 450 MHz with an I/Q rate of 491.52 MHz is selected for the transmitter channels. In this profile naming convention, 200 refers to the primary signal bandwidth and 450 refers to the DPD synthesis bandwidth.

Figure 131 shows the filter configuration for this profile. The signal rate after the TFIR block is equal to the I/Q rate of the profile.

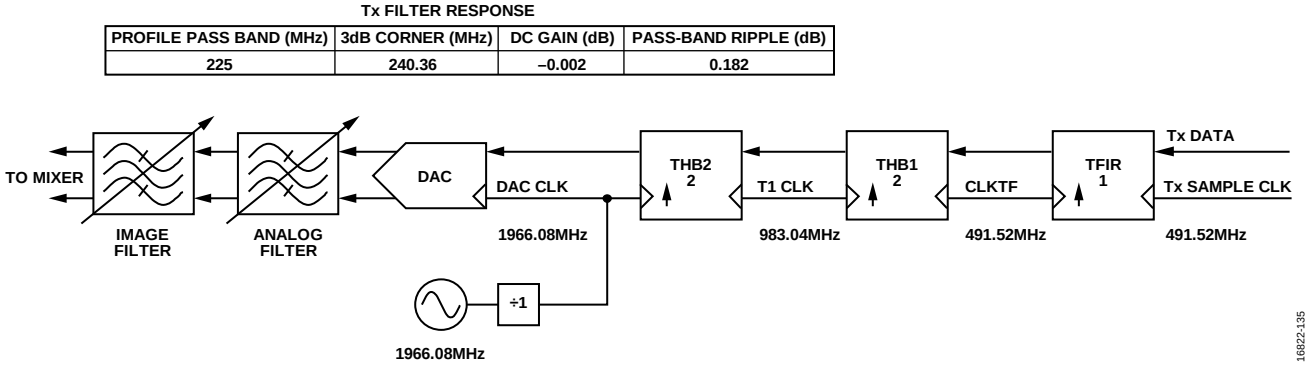

Figure 131. Filter Configuration for the Transmitter Profile of 200 MHz/450 MHz with an I/Q Rate of 491.52 MHz

The **Tx Summary** tab of the software also shows the frequency response of the digital filters, analog filters, DAC sinc response, and the composite response of the signal chain. The responses are plotted from dc to the DAC clock rate, as shown in Figure 132.

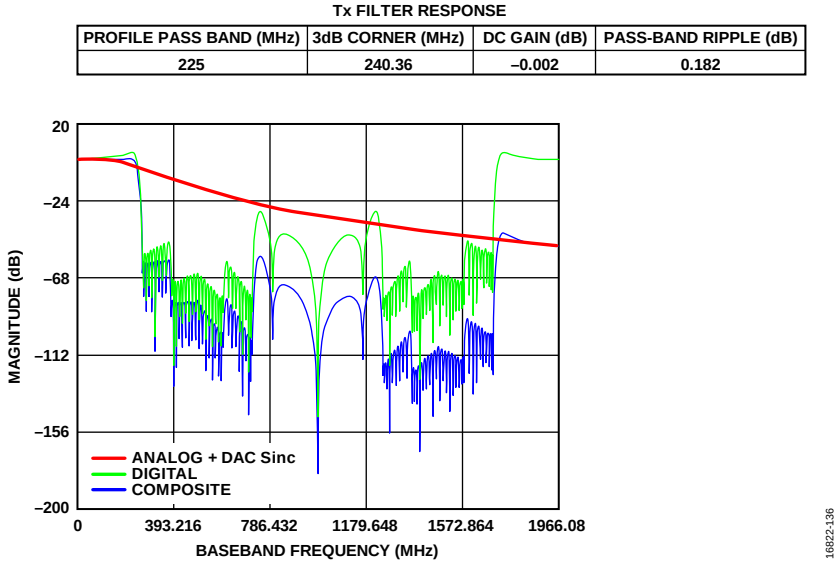

Figure 132. Transmitter Filter Responses

An examination of the profile pass band in Figure 133 shows that the analog response slightly attenuates information within the profile pass band. This analog attenuation is compensated by the digital filter response to obtain a maximally flat pass band for this profile. The primary signal bandwidth is restricted to 200 MHz, which equates to 100 MHz on each I and Q channel. There is minimal compensation required by digital filters within this bandwidth.

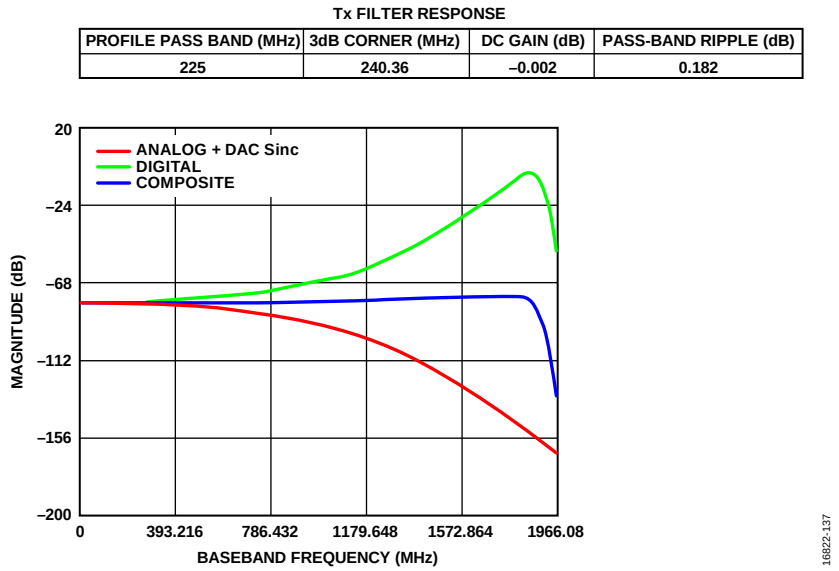

Figure 133. Examination of the Pass Band Frequency Response of the Transmitter Profile of 200 MHz/450 MHz with an I/Q Rate of 491.52 MHz

Transmitter Filter API Structure

The transmitter filter configuration is stored in the `taliseTxProfile_t` structure. This structure is stored within the `taliseTxSettings_t` structure, which is stored in the `taliseInit_t` initialization structure. The parameters for the transmitter filter configuration (`taliseTxProfile_t`) are described in Table 123.

Table 123. taliseTxProfile_t Structure Parameters

Type	Data Field	Permissible Values	Description
uint8_t	dacDiv	1, 2	DAC clock, divide by 1 or 2
taliseFir_t	txFIR	See Table 124	See Table 124
uint8_t	txFirInterpolation	1, 2, 4	Tx FIR Interpolation setting
uint8_t	thb1Interpolation	1, 2	Tx HB1 interpolation setting; 1 = bypass, 2 = in use
uint8_t	thb2Interpolation	1, 2	Tx HB2 interpolation setting; 1 = bypass, 2 = in use
uint8_t	thb3Interpolation	1, 2	Tx HB3 interpolation setting; 1 = bypass, 2 = in use
uint8_t	txInt5Interpolation	1,5	Tx INT5 interpolation setting; 1 = bypass, 5 = in use
uint32_t	txInputRate_kHz	122,880 to 491,520, based on currently defined use cases	I/Q data rate at the input to the TFIR; specified in kHz
uint32_t	primarySigBandwidth_Hz	25,000,000 to 200,000,000	Primary signal bandwidth; specified in Hz
uint32_t	rfBandwidth_Hz	56,000,000 to 450,000,000	RF bandwidth; specified in Hz
uint32_t	txDac3dBCorner_kHz	187,000 to 450,000	DAC 3 dB corner; specified in kHz
uint32_t	txBbf3dBCorner_kHz	50,000 to 225,000	Baseband filter 3 dB corner frequency; specified in kHz
uint16_t	loopBackAdcProfile	Valid ADC arrays are provided for specific use cases through the init.c files or Python scripts	A 42-element array; provides the profile for the loop filter of the Σ-Δ ADC

The permissible values provided in Table 124 are based on the currently defined profiles/use cases. A given profile has a specific combination of these values.

The `taliseFir_t structure` is contained within the `taliseTxProfile_t` structure. It contains the following parameters:

Table 124. taliseFir_t Structure

Type	Data Field	Permissible Values	Description
int8_t	gain_dB	+6, 0, −6, −12	The setting (in dB) for the gain block within the Tx FIR.
uint8_t	numFirCoefs	20, 40, 60, 80	Number of taps to be used in the Tx FIR.
int16_t *	coefs	Not applicable	A pointer to an array of filter coefficients

The transmitter FIR is specified in signed coefficients from +32,767 to −32,768. The gain block allows for more flexibility when designing a digital filter. For example, a FIR can be designed with 6 dB gain in the pass band, and then this block can be set to −6 dB gain to give an overall 0 dB gain in the pass band. The equation to calculate the gain of the filter coefficients is shown in the Receiver FIR section.

OBSERVATION RECEIVERS SIGNAL PATH

The ADRV9008-2 and the ADRV9009 feature two observation receivers (Observation Receiver 1 and Observation Receiver 2) that can be used to capture data for DPD algorithms. The observation receiver can serve as an external loopback path to loop back the output of a power amplifier, provided that the input level to the observation receiver is below the full-scale level of the ADC.

The ADRV9008-2 and the ADRV9009 Observation Receiver 1 and Observation Receiver 2 channels have separate I/Q mixers. These mixers are identical to the mixers of the ADRV9008-1, with the exception that the observation mixers include an LO multiplexer. The LO multiplexer allows either the RF PLL or the auxiliary PLL to provide the LO signal source for the Observation Receiver 1 and Observation Receiver 2 mixers.

The mixer feeds into a programmable TIA that serves as a low-pass filter in the analog datapath. The signal is converted by the Σ-Δ ADC and is filtered in half-band decimation stages and the programmable RFIR. The fixed coefficient half-band filters (RHB1, RHB2, RHB3, and DEC5) and the RFIR prevent data wrapping and overrange conditions.

The observation receiver signal path allows an ADC stitching mode, which allows the Observation Receiver 1 and Observation Receiver 2 digital datapaths to be combined, creating larger observation receiver bandwidths. Bandwidths of 450 MHz can be achieved by operating in ADC stitching mode. In this use case, the ADCs are provided with the same signal, for example, if Observation Receiver 1 is selected, this input is digitized by all 4 ADCs.

The IF conversion stage provides the ability frequency shift or upsample/downsample digital data. Configurations supported include real IF (real valued baseband data) configuration and low IF (complex data) configuration.

Figure 134 shows the signal path for the Observation Receiver 1 and Observation Receiver 2 signal chain.

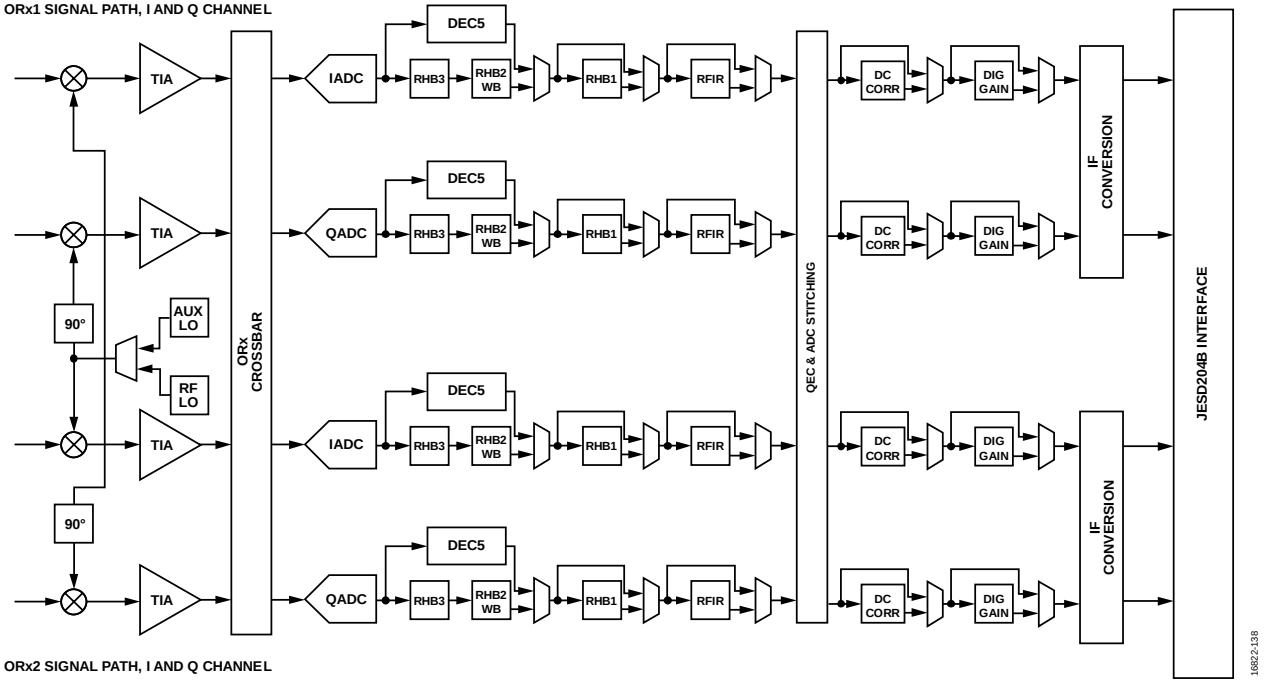

Figure 134. Observation Receiver 1 and Observation Receiver 2 Signal Path

Observation Receiver TIA

The observation receiver TIA is a low-pass filter with a single, real pole, frequency response. The TIA can support pass band bandwidths up to 225 MHz each for both I and Q. The TIA is calibrated during device initialization, which ensures a consistent frequency corner across all devices. The TIA 3 dB bandwidth is set within the device data structure and is profile dependent. Roll-off within the observation receiver pass band is compensated by the RFIR to ensure a maximally flat pass band frequency response.

Observation Receiver DEC5 Filter

Either the observation receiver DEC5 filter, or the combination of the observation receiverRHB3 and observation receiver RHB2 is used in the observation receiver digital path. The DEC5 filter decimates by a factor of 5. The DEC5 filter coefficients are as follows:

[+0.002197, +0.004272, +0.006836, +0.008789, +0.008545, +0.003418, −0.004639, −0.015381, −0.025512, −0.029785, −0.022461, −0.002441, +0.03125, +0.074707, +0.119141, +0.155396, +0.176758, +0.176758, +0.155396, +0.119141, +0.074707, +0.03125, −0.002441, −0.022461, −0.029785, −0.025512, −0.015381, −0.004639, +0.003418, +0.008545, +0.008789, +0.006836, +0.004272, +0.002197]

Observation Receiver RHB3 Filter

The observation receiver RHB3 filter is a fixed coefficient decimating filter that decimates by a factor of 2. The observation receiver RHB3 filter coefficients are as follows:

[−0.01874, −0.04218, +0.050476, +0.293884, +0.439636, +0.293884, +0.050476, −0.04218, −0.01874]

Observation Receiver Wideband RHB2 Filter

The observation receiver wideband RHB2 filter is a fixed coefficient decimating filter that decimates by a factor of 2. The observation receiver wideband RHB2 filter coefficients are as follows:

[+0.001404, 0, −0.00134, 0, +0.002014, 0, −0.00281, 0, +0.003845, 0, −0.00519, 0, +0.006775, 0, −0.00873, 0, +0.01123, 0, −0.01428, 0, +0.01825, 0, −0.0235, 0, +0.030823, 0, −0.04181, 0, +0.061035, 0, −0.10449, 0, +0.317749,+ 0.5, +0.317749, 0, −0.10449, 0, +0.061035, 0, −0.04181, +0, 0.030822, 0, −0.0235, 0, +0.01825, 0, −0.01428, 0, +0.01123, 0, −0.00873, 0, +0.006775, 0, −0.00519, 0, +0.003845, 0, −0.00281, 0, +0.002014, 0, −0.00134, 0, +0.001404]

Observation Receiver RHB1 Filter

The observation receiver RHB1 filter is a fixed coefficient decimating filter that can decimate by a factor of 2, or can be bypassed. The observation receiver RHB1 filter coefficients are as follows:

[−0.000122, 0, +0.000244, 0, −0.000488, 0, +0.000854, 0, −0.001221, 0, +0.001831, 0, −0.002502, 0, +0.003479, 0, −0.004700, 0, +0.006287, 0, −0.008179, 0, +0.010620, 0, −0.013611, 0, +0.017578, 0, −0.022766, 0, +0.030029, 0, −0.040955, 0, +0.059998, 0, −0.103027, 0, +0.313721, +0.493652, +0.313721, 0, −0.103027, 0, +0.059998, 0, −0.040955, 0, +0.030029, 0, −0.022766, 0, +0.017578, 0, −0.013611, 0, +0.010620, 0, −0.008179, 0, +0.006287, 0, −0.004700, 0, +0.003479, 0, −0.002502, 0, +0.001831, 0, −0.001221, 0, +0.000854, 0, −0.000488, 0, +0.000244, 0, −0.000122]

Observation Receiver RFIR

The programmable observation receiver RFIR filter acts as a decimating filter. that can decimate by a factor of 1, 2, or 4, or it can be bypassed. The RFIR is used to compensate for the roll-off of the analog TIA low-pass filter and can use either 24, 48, or 72 filter taps.

The maximum number of taps is limited by the FIR clock rate, which is derived from the DPCLK. The maximum DPCLK clock rate is 500 MHz. The DPCLK clock rate is the ADC clock rate divided by 4 or 5. Divide by 4 when using the HB2 and HB3 filters and divide by 5 when using the DEC5 filter. The DPCLK clock rate affects the maximum number of RFIR filter taps that can be used, as shown in the following equation:

$$\textit{Maximum Number of Observation Receiver RFIR Filter Taps} = (\textit{DPCLK Clock Rate}/\textit{Observation Receiver I/Q Data Rate}) \times 24$$

The observation receiver RFIR also has programmable gain setting of +6 dB, 0 dB, −6 dB, or −12 dB.

Observation Receiver IF Conversion

The IF conversion stage allows for frequency shifting of the baseband digital data.

Observation Receiver Signal Path Example

The TTES provides an example that shows how the baseband filtering stages are used in profile configurations for a signal pathway. In this example, the observation receiver profile of 450 MHz with an I/Q rate of 491.52 MHz is selected for the observation receiver channels.

Figure 135 shows the filter configuration for the observation receiver profile of 450 MHz with an I/Q rate of 491.52 MHz. The signal rate shown after the RFIR block is equal to the I/Q rate of the profile.

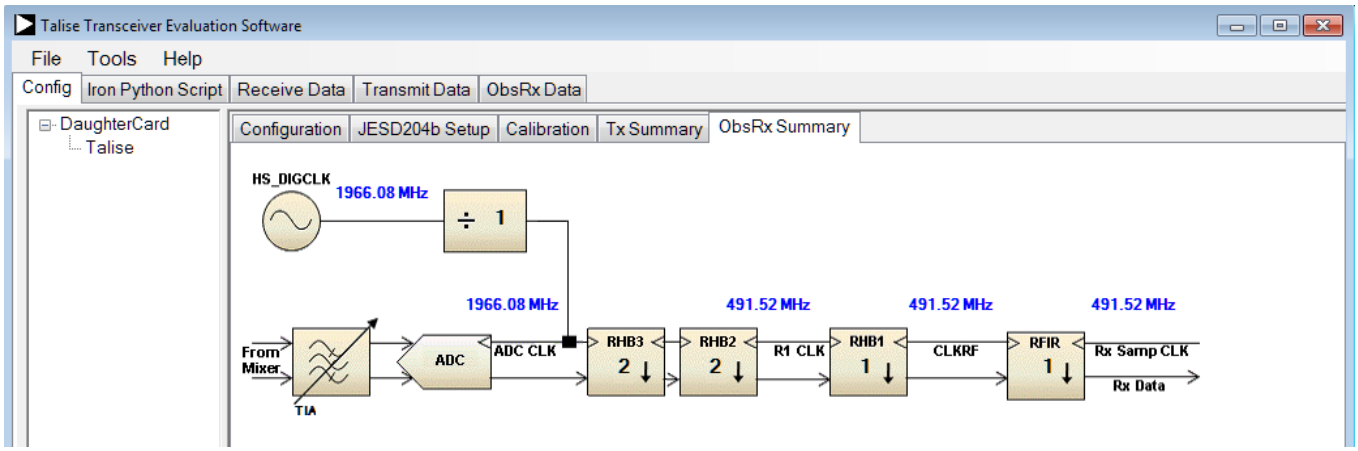

Figure 135. Filter Configuration for the Observation Receiver Profile of 450 MHz with an I/Q Rate of 491.52 MHz

Observation Receiver Filter API Structure

The observation receiver filter configuration is stored in the `taliseORxProfile_t` structure within the `taliseObsRxSettings_t` structure, which is stored in the `taliseInit_t` initialization structure. The `taliseORxProfile_` contains the following parameters:

Table 125. taliseORxProfile_t Structure Parameters

Type	Data Field	Permissible Values	Description
taliseFir_t	rxFIR	See Table 126.	See Table 126.
uint8_t	rxFirDecimation	1, 2, 4	RFIR decimation setting.
unit8_t	rxDec5Decimation	4, 5	Setting to use either the DEC5 or HB3 and HB2 in the ORx path. 5 = use DEC5. 4 = use HB3 and HB2.
uint8_t	rhb1Decimation	1, 2	Rx HB1 DECIMATION setting. 1 = bypass, 2 = in use.
uint32_t	orxOutputRate_kHz	46,080 to 307,200	I/Q data rate to the input of the JESD204B block.
uint32_t	rfBandwidth_Hz	20000000 to 200000000	RF bandwidth, specified in Hz.
uint32_t	rxBbf3dBCorner_kHz	20000 to 200000	Baseband filter 3 dB corner frequency, specified in kHz.
uint16_t	orxLowPassAdcProfile	Valid ADC arrays provided for specific use cases through the init.c files or Python scripts.	42-element array that provides the profile for the loop filter of the ADC pair in low-pass mode.
uint16_t	orxBandPassAdcProfile	Valid ADC arrays provided for specific use cases through the init.c files or Python scripts.	42-element array that provides the profile for the loop filter of the ADC pair in band-pass mode.
taliseRxDdc_t	orxDdcMode	TAL_ORXDDC_DISABLED. The digital downconversion (or IF conversion) functionality for ORx usage is not supported in the current software.	Digital downconversion mode. Device defaults to zero IF mode, where the RF LO is the center frequency of the digital data that is output from the device.
int16_t	orxMergeFilter	Valid merge filter arrays provided for specific use cases through the init.c files or Python scripts.	12-element array used in ADC stitching modes, used to merge the low-pass and band-pass ADCs to obtain the wideband ADC transfer function.

The permissible values provided in Table 126 are based on the currently defined profiles/use cases. A given profile has a specific combination of these values.

The `taliseFir_t` structure is contained within the `taliseRxProfile_t` structure and contains the parameters shown in Table 126.

Table 126: taliseFir_t Structure

Type	Data Field	Permissible Values
int8_t	gain_dB	+6 dB, 0 dB, −6 dB, −12 dB
uint8_t	numFirCoefs	24, 48, 72
int16_t *	coefs	A pointer to an array of filter coefficients

The receiver RFIR is specified in signed coefficients from +32,767 to −32,768. The gain block allows more flexibility when designing a digital filter. For example, an RFIR can be designed with 6 dB gain in the pass band, and then this block can be set to −6 dB gain to give an overall 0 dB gain in the pass band. The gain of the filter coefficients can be calculated with the equation shown in the Receiver FIR section.

FILTER CONFIGURATION API FUNCTIONS

The digital filters are configured in the `TALISE_initialize()` API function. This function utilizes the `taliseRxProfile_t` and `taliseTxProfile_t` structures specified prior to initialization. As part of the filter configuration, the FIRs are programmed into memory within the device, utilizing the `TALISE_programFir()` function. The analog filters (receiver TIA and transmitter low-pass filter) are calibrated in order to achieve consistent pass band bandwidths. This is done after the Arm processor is initialized during the `TALISE_runInitCals()` function call.

OBSERVATION RECEIVER

This section describes the configuration and operation of the observation receivers integrated into the ADRV9008-2 and ADRV9009 devices. Each device has two observation receiver inputs, Observation Receiver 1 and Observation Receiver 2. Either of these observation receivers can be used at any time.

If the observation receiver inputs are not used and the input connections are grounded on the printed circuit board (PCB), disable the unused inputs in the configuration to prevent calibration failures.

To achieve the maximum supported bandwidth (up to 450 MHz for the observation receiver paths), the device uses four ADCs concurrently, two for I data and two for Q data. Figure 136 shows the two observation receiver inputs, separate mixers, and TIA stages. Each I and Q signal can be multiplexed to two Σ-Δ ADCs, which digitize the signal and process this in baseband. One ADC is configured in a low-pass configuration and the other ADC is configured in a band-pass configuration. A stitching algorithm is then used to align and merge the data from the concurrently running ADCs, with the merging having the effect of using the low frequency information from the low-pass ADC, and the high frequency information from the band-pass ADC.

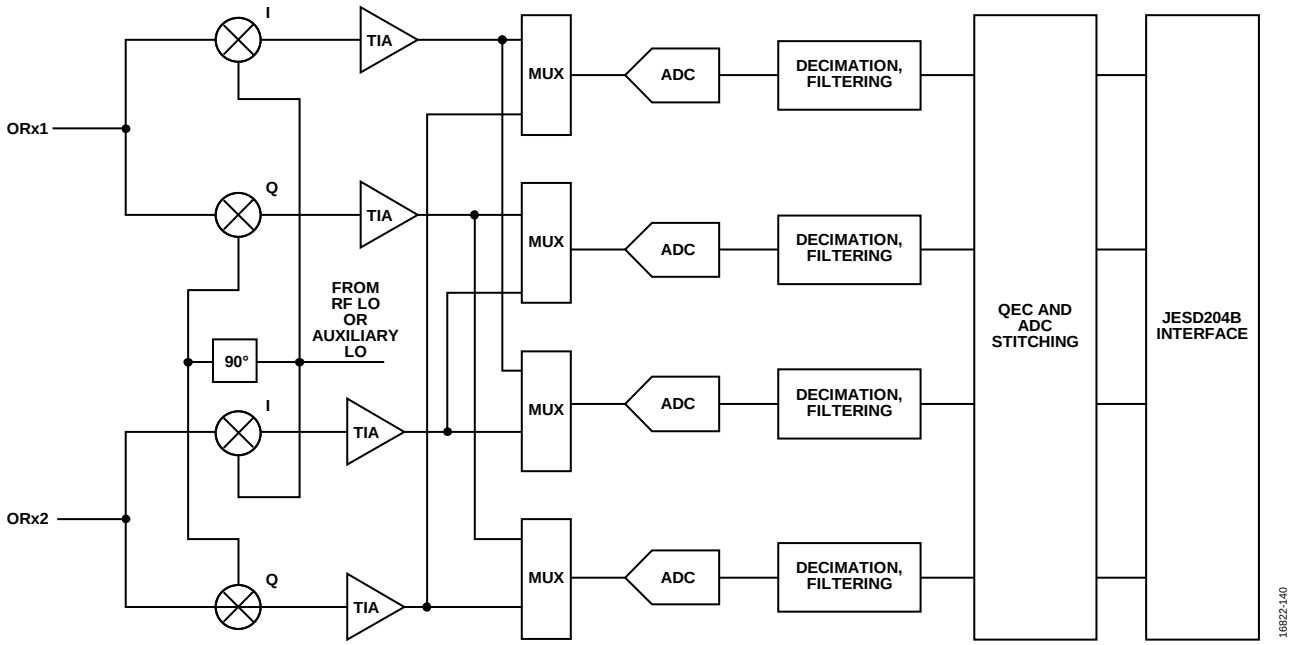

Figure 136. Observation Receiver Signal Chains, Showing Use Case (Signal Flow) for Observation Receiver 1 and Observation Receiver 2

Note that in the current version (Version 3) of the firmware, it is not possible to switch between observation receiver profiles greater than an observation receiver bandwidth of 200 MHz with an I/Q rate of 307.2 MSPS).

OBSERVATION RECEIVER API STRUCTURE

The observation receiver settings structure, `taliseObsRxSettings_t`, is contained within the device structure (`device -> ObsRx`). The `taliseObsRxSettings_t` structure contains the following data fields described in Table 127.

Table 127. taliseObsRxSettings_t Structure Parameters

Type	Data Field	Description
taliseFramerSel_t	framerSel	This is used to select the desired JESD204B Framer A or Framer B to be used by the ORx path. For more details, see the JESD204B Interface section.
taliseObsRxChannel_t	obsRxChannel	The ORx channel selected.
taliseObsRxLoSource_t	obsRxLoSource	This selects the desired LO source for the ORx path to use, either the RF PLL or auxiliary PLL.
taliseORxGainControl_t	orxGainCtrl	Structure that stores the settings of the ORx gain control. For more details, see the Gain Control Modes section of this document.
taliseRxProfile_t	orxProfile	Contains the settings of the filters and digital data chain in the ORx path.

OBSERVATION CHANNEL CONTROL

See the System Control and Use Cases sections for details.

GPIO CONFIGURATION

To advise the Arm processor of which pins to monitor for RX_ENABLE, ORX_ENABLE, TX_ENABLE, ORX1_TX_SEL0, ORX1_TX_SEL1, ORX2_TX_SEL0, and/or ORX2_TX_SEL1, refer to Arm GPIO Pins section of the QEC, Calibration, and Arm Configuration section.

The device has several GPIOs that can be used for a variety of control or monitoring functions. The device has 19 low voltage GPIO pins that are designated GPIO_0 through GPIO_18. The Logic 1 voltage for the low voltage GPIO pins is determined by the VDD_INTERFACE pin supply. The VDD_INTERFACE supply can be set between 1.8 V and 2.5 V. The device also provides twelve 3.3 V GPIO pins that are designated GPIO_3P3_0 to GPIO_3P3_11. The Logic 1 voltage for the 3.3 V GPIO pins is determined by the VDDA_3P3 supply pin. Ten of the twelve 3.3 V GPIO pins can be set as output pins for the auxiliary DAC signals. The AUXDAC pin mapping is described in the targeted device data sheet. Drive strength and other specifications are described in the targeted device data sheet.

Descriptions of pin operations related to real-time pin control of the transmitter/receiver/observation receiver state is provided in the System Control section and the Use Cases section. The observation receiver enable signals can be controlled and assigned to GPIO pins in the ADRV9008-2 and the ADRV9009.

The GPIO pins can be used as real-time status signals that provide device status information from the device to the BBP when the GPIO pins are configured as outputs. When set as inputs, the GPIO pins can be used as real-time control signals that can alter the state of the device. The API functions related to GPIO configuration give the user the ability to configure pins as inputs or outputs and assign functionality to specific pins. This section describes the GPIO signals and their behavior in detail.

Figure 137 shows a high level block diagram of the GPIO pins.

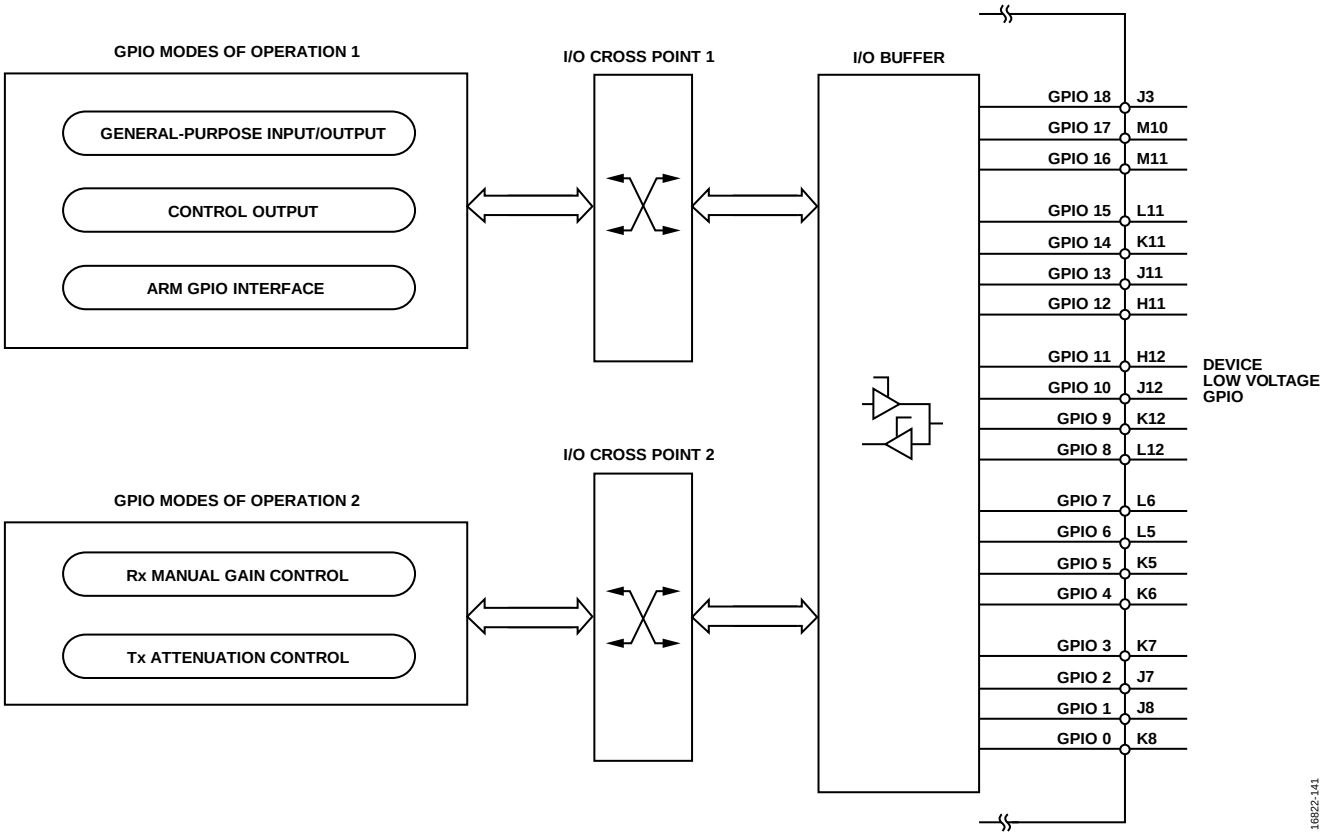

Figure 137. Low Voltage GPIO Feature Overview

LOW VOLTAGE GPIO OPERATION

The low voltage GPIO pins support a wide number of configuration options. This section describes setting up the low voltage GPIO pins for monitoring internal signals, manual toggling of GPIO pins, Arm processor interaction, and the slicer pins. In configuring the GPIO, the two major factors to consider are the GPIO output enable control and the GPIO source control.

The output enable signal determines the direction of the pin with respect to the GPIO pins. If a pin is set as an output, the GPIO buffer is configured as an output.

The GPIO source control determines the functionality of the pin. The GPIO source control is assigned in groups of four. This means that GPIO_0 to GPIO_3 share a single source control, GPIO_4 to GPIO_7 share a single source control, and so on. There are four types of GPIO pin functions that are described by the `taliseGpioMode_t` enumerator type. Table 128 describes the enumerators for different GPIO modes.

Table 128. taliseGpioMode_t Enumerators for Low Voltage GPIO Modes

taliseGpioMode_t Enumerator Name	Enumerator Value	Description
GPIO_MONITOR_MODE	0	This mode allows a choice of debug signals to output from the device to monitor the state of the device.
GPIO_BITBANG_MODE	3	Manual mode. An API function sets the output pin levels, and another API function can read the input level.
GPIO_ARM_OUT_MODE	9	This mode allows communication to or from the internal Arm processor on the GPIO pins.
GPIO_SLICER_OUT_MODE	10	This mode allows the slicer to output data over specific GPIO pins. Slicer functionality is not complete in the API.

The source control is configured by the `TALISE_setGpioSourceCtrl()` function. This function can be called any time after initialization. A complementary readback command, `TALISE_getGpioSourceCtrl()`, returns the source control programmed to the device.

A separate enumerator is provided in the API that represents each low voltage GPIO pin. This enumerator is designated `taliseGpioPinSel_t` (see Table 129).

Table 129. taliseGpioPinSel_t Enumerations for Low Voltage GPIO Pins

taliseGpioPinSel_t Enumerator Name	Enumerator Value	Pin Number
TAL_GPIO_00	0	GPIO_0
TAL_GPIO_01	1	GPIO_1
TAL_GPIO_02	2	GPIO_2
TAL_GPIO_03	3	GPIO_3
TAL_GPIO_04	4	GPIO_4
TAL_GPIO_05	5	GPIO_5
TAL_GPIO_06	6	GPIO_6
TAL_GPIO_07	7	GPIO_7
TAL_GPIO_08	8	GPIO_8
TAL_GPIO_09	9	GPIO_9
TAL_GPIO_10	10	GPIO_10
TAL_GPIO_11	11	GPIO_11
TAL_GPIO_12	12	GPIO_12
TAL_GPIO_13	13	GPIO_13
TAL_GPIO_14	14	GPIO_14
TAL_GPIO_15	15	GPIO_15
TAL_GPIO_16	16	GPIO_16
TAL_GPIO_17	17	GPIO_17
TAL_GPIO_18	18	GPIO_18

Functionality for receiver manual gain control (MGC) is supported.

GPIO MONITOR MODE OUTPUT

The GPIO monitor mode provides internal device information over GPIO pins that can be read back in real time by the baseband processor. Some convenient features include real-time monitoring of overload counters related to AGC operation and PLL lock status indicators. The GPIO monitor modes generate an 8-bit word that can be expressed over GPIO_0 through GPIO_7, GPIO_8 to GPIO_15. Bit 4 through Bit 6 of the word can be sent to GPIO_16 to GPIO_18.

Parameters passed by the `TALISE_setGpioSourceCtrl()`, `TALISE_setGpioOe()`, and `TALISE_setGpioMonitorOut()` functions determine what monitor signals are mapped to specific GPIO pins. The GPIO source control must be set to GPIO_MONITOR_MODE with output mode enabled for the desired monitor pins.

Figure 138 shows the internal GPIO configuration when in the GPIO_MONITOR_MODE source control for all nibble groups.

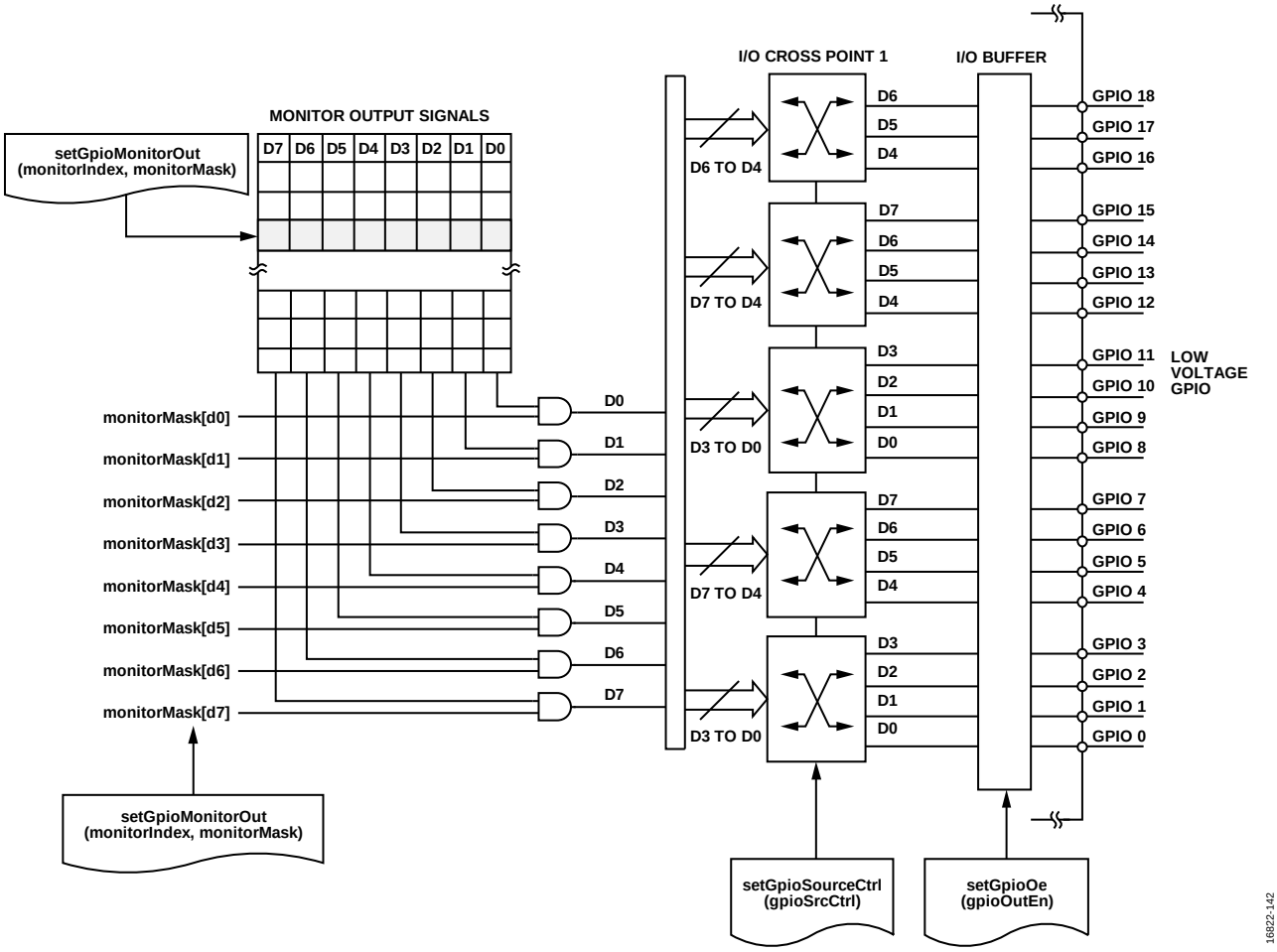

Figure 138. GPIO Hardware Configuration in GPIO_MONITOR_OUT_MODE Source Control

Figure 138 shows that the `TALISE_setGpioMonitorOut()` parameter monitor index sets the monitor mode for the device. The 8-bit monitor mode and 8-bit monitor mask parameter are passed bitwise through an AND gate and distributed to the GPIO pins that are set in the source control for GPIO_MONITOR_MODE and configured as outputs.

Table 130 describes the available monitor modes.

Table 130. GPIO Monitor Modes for Low Voltage GPIO Pins

monitorIndex [D7:D0]	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x01	AGC_MONITOR_0	Rx2 Gain Change	Rx1 Gain Change	Rx2 APD High Threshold Counter Exceeded	Rx1 APD High Threshold Counter Exceeded	Rx2 HB2 High Threshold Counter Exceeded	Rx1 HB2 High Threshold Counter Exceeded	Rx2 HB2 Low Threshold Counter Exceeded	Rx1 HB2 Low Threshold Counter Exceeded
0x02	AGC_MONITOR_1	Rx2 Gain Change	Rx1 Gain Change	Rx2 APD High Threshold Exceeded	Rx1 APD High Threshold Exceeded	Rx2 HB2 High Threshold Exceeded	Rx1 HB2 High Threshold Exceeded	Not applicable	
0x03	AGC_MONITOR_2	Not applicable		Rx2 APD High Threshold Exceeded	Rx1 APD High Threshold Exceeded	Rx2 HB2 High Threshold Exceeded	Rx1 HB2 High Threshold Exceeded	Not applicable	
0x04	AGC_MONITOR_3	Rx1 Low Power Threshold Exceeded	Rx1 High Power Threshold Exceeded	Rx1 AGC Gain Update Counter Expiry	Not applicable		Rx1 Gain Change	Rx1 Gain Increment	Rx1 Gain Decrement
0x05	AGC_MONITOR_4	Rx2 Low Power Threshold Exceeded	Rx2 High Power Threshold Exceeded	Rx2 AGC Gain Update Counter Expiry	Not applicable		Rx2 Gain Change	Rx2 Gain Increment	Rx2 Gain Decrement
0x06	AGC_MONITOR_5	Rx2 Gain Increment	Rx1 Gain Increment	Rx2 Gain Decrement	Rx1 Gain Decrement	Not applicable			
0x07	AGC_MONITOR_6	Rx2 APD High Threshold Counter Exceeded	Rx1 APD High Threshold Counter Exceeded	Rx2 HB2 High Threshold Counter Exceeded	Rx1 HB2 High Threshold Counter Exceeded	Rx2 AGC Gain Update Counter Expiry	Rx1 AGC Gain Update Counter Expiry	Rx2 Gain Change	Rx1 Gain Change
0x08	AGC_MONITOR_7	Rx2 RSSI Decimated Power Ready	Rx1 RSSI Decimated Power Ready	Rx2 AGC Gain Update Counter Expiry	Rx1 AGC Gain Update Counter Expiry	Rx2 APD High Threshold Counter Exceeded	Rx1 APD High Threshold Counter Exceeded	Rx2 HB2 High Threshold Counter Exceeded	Rx1 HB2 High Threshold Counter Exceeded
0x09	AGC_MONITOR_8	Rx1 Gain Index [7:0]							
0x0A	AGC_MONITOR_9	Rx2 Gain Index [7:0]							
0x0B	AGC_MONITOR_10	Rx1 Gain Index [3:0]				Rx1 Gain Index [3:0]			
0x0C	PLL lock monitor	RF Synthesizer RF PLL Lock	CLK Synthesizer Lock	AUX Synthesizer RF PLL Lock	Not applicable				
0x2C	AGC_OVERLOAD_MONITOR_CH1	Not applicable	Rx1 APD Low Threshold Exceeded	Rx1 APD High Threshold Exceeded	Rx1 HB2 Interval 1 Low Threshold Overflow	Rx1 HB2 Interval 0 Low Threshold Overflow	Rx1 HB2 Low Threshold Overflow	Rx1 HB2 IP3 High Threshold Overflow	Rx1 HB2 High Thresh Overflow
0x2D	AGC_OVERLOAD_MONITOR_CH2	Not applicable	Rx2 APD Low Threshold Exceeded	Rx2 APD High Threshold Exceeded	Rx2 HB2 Interval 1 Low Threshold Overflow	Rx2 HB2 Interval 0 Low Threshold Overflow	Rx2 HB2 Low Threshold Overflow	Rx2 HB2 IP3 High Threshold Overflow	Rx2 HB2 High Threshold Overflow
0x2E	AGC_OVERLOAD_COUNTER_MONITOR_CH1	Rx1 AGC Gain Update Counter Expiry	Rx1 APD Low Threshold Counter Exceeded	Rx1 APD High Threshold Counter Exceeded	Rx1 HB2 Interval 1 Low Threshold Counter Exceeded	Rx1 HB2 Interval 0 Low Threshold Counter Exceeded	Rx1 HB2 Low Threshold Counter Exceeded	Rx1 HB2 IP3 High Threshold Counter Exceeded	Rx1 HB2 High Threshold Counter Exceeded
0x2F	AGC_OVERLOAD_COUNTER_MONITOR_CH2	Rx2 AGC Gain Update Counter Expiry	Rx2 APD Low Threshold Counter Exceeded	Rx2 APD High Threshold Counter Exceeded	Rx2 HB2 Interval 1 Low Threshold Counter Exceeded	Rx2 HB2 Interval 0 Low Threshold Counter Exceeded	Rx2 HB2 Low Threshold Counter Exceeded	Rx2 HB2 IP3 High Threshold Counter Exceeded	Rx2 HB2 High Threshold Counter Exceeded

GPIO BITBANG MODE

The GPIO bitbang mode allows the user to configure GPIO pins as inputs or outputs where the device can read back or set pin voltage levels. This mode is also referred to as manual mode. If a GPIO pin is configured as an input, the user can read back the voltage level present at the input. The voltage readback is either 0 or 1 (must be connected to ground or VDD_INTERFACE). If a GPIO pin is configured as an output, the user can set a binary voltage on the pin (must be connected to ground or VDD_INTERFACE).

Figure 139 shows the operation of the device GPIO when all source control is set in the `GPIO_BITBANG_MODE` enumerators (see Table 128).

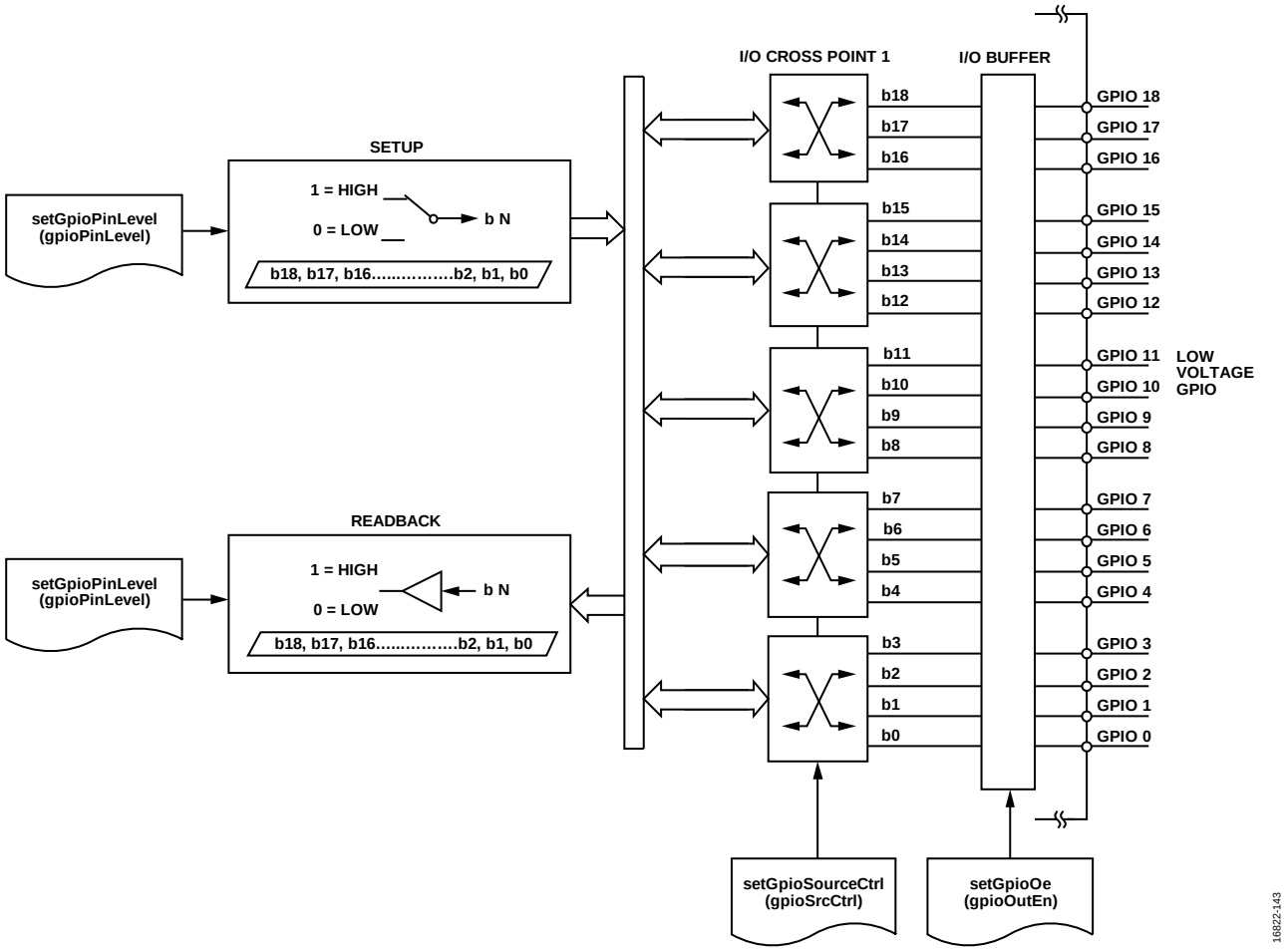

Figure 139. GPIO Hardware Configuration for GPIO_BITBANG_MODE Source Control

If a GPIO nibble group of four pins are set as outputs and the nibble group source control for the pins are set to `GPIO_BITBANG_MODE`, the command to set the output voltage on the GPIO pins is `TALISE_setGpioPinLevel()`. This command takes an input parameter specifying a voltage level for each pin, however, the pin must be set up properly for the voltage to appear on the pin. The expected level of the GPIO pins can be read back from the device with the `TALISE_getSetGpioPinLevel()` command.

If a GPIO nibble group of four pins are set as inputs and the nibble group source control for the pins are set to `GPIO_BITBANG_MODE`, the command to read back the input voltage on the GPIO pins is `TALISE_getGpioPinLevel()`. This function returns a 32-bit word where each bit corresponds to the level detected on the GPIO pin.

GPIO ARM OUTPUT OPERATION

See the System Control section for information related to programming GPIOs relevant to the Arm processor.

GPIO SLICER FEATURES

Due to the potential for values expressed over the JESD204B interface to exceed the range of a 16-bit twos complement number in applications that use gain compensation, a floating point formatter or slicer is used to overcome this limitation. The floating point formatter does not require GPIO connections to the BBP. However, support for floating point computation is required in the BBP. The slicer requires a 3-bit slicer word input or output on GPIO to determine how the BBP shifts the data sent over the JESD204B interface. The slicer can also be configured in an external control mode where the BBP sends a 3-bit slicer word to the device to shift the data, prior to transmission over the JESD204B interface. See the Gain Control Modes section for details on the full functionality of the slicer and floating point formatter. This section focuses on programming the GPIO pins for use with the slicer.

The slicer has two modes of operation. One mode is an internal slicer control mode, where the slicer outputs the amount of attenuation applied in the datapath (in 6 dB/LSB steps) over a 3-bit GPIO word. The other mode is an external slicer control mode, where the 3-bit GPIO word on the slicer input pins indicates a coefficient (from 0 to 7) to an attenuation factor (1 dB, 2 dB, 3 dB, or 4 dB).

In the internal slicer control mode, the GPIO pins that can be used are fixed. The 4-pin nibble groupings must be set to the `GPIO_SLICER_OUT_MODE` source control (see Table 131).

Table 131. Internal Slicer Control Mode

Signal	Internal Slicer Output Pin
RX1_SLICER_POSITION[2:0]	TAL_GPIO_10 to TAL_GPIO_08
RX2_SLICER_POSITION[2:0]	TAL_GPIO_14 to TAL_GPIO_12

In external slicer control mode, the GPIO pins that can be used are programmable. The pin groupings that can be used are described in Table 132. Pin groupings are described by the enumerator types `taliseRx1ExtSlicerGpioSelect_t` and `taliseRx2ExtSlicerGpioSelect_t`.

Table 132. External Slicer Control Mode

Signal	External Slicer Input Pin
RX1_SLICER_POSITION[2:0]	TAL_GPIO_02 to TAL_GPIO_00, TAL_GPIO_07 to TAL_GPIO_05, and TAL_GPIO_10 to TAL_GPIO_08
RX2_SLICER_POSITION[2:0]	TAL_GPIO_07 to TAL_GPIO_05 and TAL_GPIO_13 to TAL_GPIO_11

The slicer, whether in internal or external control mode, is configured by executing the `TALISE_setRxDataFormat()` function. This function also sets up the receiver data format (integer or floating point mode) and can enable gain compensation.

To retrieve the slicer position, use the `TALISE_getSlicerPosition()` function. The slicer position is only needed for integer 12-bit and 16-bit formats.

GPIO FOR RECEIVER MANUAL GAIN CONTROL MODE PIN CONTROL

If the receiver is in MGC mode of operation, there are two methods to change the current gain index. The first method is through a SPI based command, as performed in the `TALISE_setRxManualGain()` function, which changes the current gain index to a new value as indicated by the input parameter to the command. The second method is through the low voltage GPIO pins, which can enable more precise control over the timing of the gain change by assigning separate GPIO pins for gain index increment and gain index decrement control. The gain is changed when a pulse is detected on the increment or decrement pin. The gain index increment step and gain index decrement step are programmable from 0 to 7, corresponding to a number of gain table indices from 1 to 8. The pin assignments are also programmable.

The MGC pin control is enabled by the `TALISE_setRxGainCtrlPin()` function. This command takes a device data structure parameter, a receiver channel select parameter, and a data structure type (`taliseRxGainCtrlPin_t`) containing the settings for receiver MGC pin control, particular to the receiver channel select parameter. The `taliseRxGainCtrlPin_t` structure is described in Table 133.

Table 133. Data Structure Description for taliseRxGainCtrlPin_t Parameters

Data Type	taliseRxGainCtrlPin_t Structure Member	Description
uint8_t	incStep	Increment in gain index applied when the increment gain pin is pulsed. A value of 0 to 7 applies a step size of 1 to 8. This parameter is common between Rx1 and Rx2.
uint8_t	decStep	Decrement in gain index applied when the decrement gain pin is pulsed. A value of 0 to 7 applies a step size of 1 to 8. This parameter is common between Rx1 and Rx2.
taliseGpioPinSel_t	rxGainIncPin	GPIO used for the increment gain input. Rx1 uses GPIO_0 or GPIO_10. Rx2 uses GPIO_3 or GPIO_13.
taliseGpioPinSel_t	rxGainDecPin	GPIO used for the decrement gain input. Rx1 uses GPIO_1 or GPIO_11. Rx2 uses GPIO_4 or GPIO_14.
uint8_t	enable	Enable (1) or disable (0) the gain pin control.

TRANSMITTER ATTENUATION CONTROL, SPI2 PORT

The device uses the primary SPI port for nearly all SPI transactions needed during operation but also features a secondary SPI port, SPI2, that can be used as a communication interface. The SPI2 port has access to a limited set of device registers that are related to transmitter attenuation control, observation receiver MGC, or current receiver gain settings (read only). Refer to Table 135. The SPI2 port uses a fixed set of low voltage GPIO pins, from GPIO_0 to GPIO_3. Table 134 describes the pin mapping.

Table 134. SPI2 Port Function Mapping on Low Voltage GPIO Pins

GPIO Pin	SPI2 Functionality	Pin Direction	Description
GPIO_0	SPI_DIN_2/SPI_DOUT_2	Input/output	SPI Data Input 2/SPI Data Output 2; depends on 3-wire or 4-wire operation
GPIO_1	SPI_DOUT_2	Output	SPI Data Output 2
GPIO_2	SPI_CLK_2	Input	SPI Clock 2
GPIO_3	SPI_CSB_2	Input	Chip Select 2

The SPI2 port is set up with the `TALISE_setSpi2Enable()` function. This function transfers a pointer to the device data structure, an enable/disable parameter, and a level sensitive GPIO that can be used to switch between two distinct attenuation states. SPI2 uses the same SPI configuration as the primary SPI port: LSB/MSB first, 3-wire or 4-wire mode. See the Low Voltage GPIO API Functions section.

The SPI2 port features a level sensitive GPIO pin to switch between two distinct attenuation states. The user can program a 10-bit attenuation word into registers designated TXx_ ATTENUATION_S1 (State 1) and TXx_ATTENUATION_S2 (State 2). When the GPIO voltage is low, the Transmitter 1 and Transmitter 2 channels are held at the attenuation condition of State 1. When the GPIO voltage is high, the Transmitter 1 and Transmitter 2 channels are held at the attenuation condition of State 2. The GPIO pins that can be selected (within the `taliseSpi2TxAttenGpioSel_t` structure) are assigned to the `TAL_SPI2_TXATTEN_GPIO4`, `TAL_SPI2_TXATTEN_GPIO8`, `TAL_SPI2_TXATTEN_GPIO14`, or `TAL_SPI2_TXATTEN_GPIO_DISABLE` functions.

After the two different attenuations are set in State 1 and State 2, the desired attenuation state is selected using a GPIO. Only update the attenuation state that is not currently selected because updates to the selected attenuation state take immediate effect when the LSBs of the attenuation value are written. Typically, it is recommended to synchronize the attenuation change of both Transmitter 1 and Transmitter 2 so that the Transmitter 1 and Transmitter 2 states that are not currently in use are written to, and then the GPIO is toggled to simultaneously apply the new attenuation value to both transmitters.

Table 135. SPI2 Register Map

Address	Register Name	Bit Number								Description
		7	6	5	4	3	2	1	0	
0x2E9	TX1_ATTENUATION_S1_MSB	Not used						TX1_ATTENUATION_S1[9:8]		Tx1 State 1 MSBs
0x2EA	TX1_ATTENUATION_S1_LSB	TX1_ATTENUATION_s1[7:0]								Tx1 State 1 LSBs
0x2EB	TX1_ATTENUATION_S2_MSB	Not used						TX1_ATTENUATION_S2[9:8]		Tx1 State 2 MSBs
0x2EC	TX1_ATTENUATION_S2_LSB	TX1_ATTENUATION_S2[7:0]								Tx1 State 2 LSBs
0x2ED	TX2_ATTENUATION_S1_MSB	Not used						TX2_ATTENUATION_S1[9:8]		Tx2 State 1 MSBs
0x2EE	TX2_ATTENUATION_S1_LSB	TX2_ATTENUATION_S1[7:0]								Tx2 State 1 LSBs
0x2EF	TX2_ATTENUATION_S2_MSB	Not used						TX2_ATTENUATION_S2[9:8]		Tx2 State 2 MSBs
0x2F0	TX2_ATTENUATION_S2_LSB	TX2_ATTENUATION_S2[7:0]								Tx2 State 2 LSBs
0x2F2	TX1_ATTENUATION_READBACK_LSB	TX1_ATTENUATION_READBACK[7:0]								Tx1 readback LSBs

Address	Register Name	Bit Number								Description
		7	6	5	4	3	2	1	0	
0x2F3	TX1_ATTENUATION_READBACK_MSB	Not used						TX1_ATTENUATION_READBACK[9:8]		Tx1 readback MSBs
0x2F4	TX2_ATTENUATION_READBACK_LSB	TX2_ATTENUATION_READBACK[7:0]								Tx2 readback LSBs
0x2F5	TX2_ATTENUATION_READBACK_MSB	Not used						TX2_ATTENUATION_READBACK[9:8]		Tx2 readback MSBs
0x2F6	ORX1_GAIN_INDEX	ORX1_GAIN_INDEX								ORx1 gain index
0x2F7	ORX2_GAIN_INDEX	ORX2_GAIN_INDEX								ORx2 gain index
0x2F8	RX1_GAIN_INDEX_READBACK	RX1_GAIN_INDEX_READBACK								Rx1 gain index, read only
0x2F9	RX2_GAIN_INDEX_READBACK	RX2_GAIN_INDEX_READBACK								Rx2 gain index, read only

LOW VOLTAGE GPIO API FUNCTIONS

This section summarizes all API functions related to configuration of the low voltage GPIO functionality.

TALISE_setArmGpioPins()

This function instructs the Arm of which GPIO pins to use for TDD pin control and loads the `taliseArmGpioConfig_t` structure settings to the Arm processor. The function is as follows:

```
uint32_t TALISE_setArmGpioPins(taliseDevice_t *device, taliseArmGpioConfig_t *armGpio)
```

The device can control any of these related signals by sending an Arm command, or the Arm signals can be routed to GPIO pins. Each signal in the `taliseArmGpioConfig_t` structure has an enable member. If the enable member is set, the Arm processor expects the BBP to drive that signal on the specified GPIO pin. The signals can be intermixed, some on GPIO pins and some set by the Arm command.

The BBP calls this function after loading the Arm processor `TALISE_loadArmFromBinary()` function call. If the BBP wishes to change the GPIO assignments, this function can be called again to change the configuration when the device is in the radio off state. This function also sets the GPIO pin direction for any GPIO pins that are enabled in this function but does not modify the GPIO source control parameter.

Preconditions: this function can be called after loading the Arm binary. The device must be in the radio off state.

Parameters include the following:

- `*device` is a structure pointer to the data structure.
- `*armGpio` is a structure to a pointer that describes which GPIO pins and settings to use for each possible Arm GPIO signal.

TALISE_setGpioOe()

This function sets the low voltage GPIO direction given by the transferred parameter. The direction can be either output or input per pin. The `gpioUsedMask` parameter allows the function to only affect the GPIO pins of interest. The function is as follows:

```
uint32_t TALISE_setGpioOe(taliseDevice_t *device, uint32_t gpioOutEn)
```

Preconditions: execute `TALISE_initialize()`.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `gpioOutEn` is a valid range is from 0 (all low voltage GPIO are inputs) to 0x07FFFF (all low voltage GPIO are outputs). Each bit corresponds to a GPIO pin, for example, `gpioOutEn`, Bit D0, corresponds to GPIO_0, gpioOutEn, Bit D1, corresponds to GPIO_1, and so on. If a particular bit is set to 1, the corresponding GPIO pin is set as an output.

TALISE_getGpioOe()

This function receives the low voltage GPIO direction currently set in the device. The direction can be either output or input per pin. The return `gpioOutEn` parameter returns one bit per GPIO pin. A return value of 1 is the output from the device, and a return value of 0 is the input into the device.

```
uint32_t TALISE_getGpioOe(taliseDevice_t *device, uint32_t *gpioOutEn)
```

Preconditions: execute `TALISE_initialize()`.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `*gpioOutEn` is a valid range is from 0 (all low voltage GPIOs are inputs) to 0x07FFFF (all low voltage GPIOs are outputs). Each bit corresponds to a GPIO pin.

TALISE_setGpioSourceCtrl()

This function sets the GPIO output source for different GPIO functionality and only affects the GPIO pins that have their output enable direction set to output. Each set of four GPIO pins can be assigned to a GPIO source, and each GPIO nibble (four pins) must share that GPIO output source. The `taliseGpioMode_t` enumerator can be bit shifted and bitwise OR'ed together to create the value for the `gpioSrcCtrl` parameter. The function is as follows:

```
uint32_t TALISE_setGpioSourceCtrl(taliseDevice_t *device, uint32_t gpioSrcCtrl)
```

Preconditions: execute `TALISE_initialize()`.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `gpioSrcCtrl` is a nibble-based source control. This is a 32-bit value containing five nibbles that set the output source control for each set of four GPIO pins. Each nibble group has a value equal to the `taliseGpioMode_t` enumerator values.

TALISE_getGpioSourceCtrl()

This function reads the GPIO output source for each set of four low voltage GPIO pins. The function is as follows:

```
uint32_t TALISE_getGpioSourceCtrl(taliseDevice_t *device, uint32_t *gpioSrcCtrl)
```

Preconditions: execute `TALISE_initialize()`.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `*gpioSrcCtrl` is a nibble-based source control. This is a 32-bit value containing five nibbles that set the output source control for each set of four GPIO pins. Each nibble group has a value equal to the `taliseGpioMode_t` enumerator values.

TALISE_setGpioPinLevel()

This function sets the low voltage GPIO output pin levels and only affects the GPIO pins that have their output enable direction set to output and that have the correct source control for the nibbles in `GPIO_BITBANG_MODE`. The function is as follows:

```
uint32_t TALISE_setGpioPinLevel(taliseDevice_t *device, uint32_t gpioPinLevel)
```

Preconditions: execute `TALISE_initialize()`. The GPIO pin levels do not change unless the desired GPIO pins are set in `GPIO_BITBANG_MODE` source control and are set to output enable.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `gpioPinLevel` is a parameter that corresponds each bit to a GPIO pin. 0 = low output, and 1 = high output voltage.

TALISE_getGpioPinLevel()

This function reads the low voltage GPIO pin levels and returns their contents in a single, 32-bit word. The GPIO pins that are set to be inputs in bitbang mode read back and return in the `gpioPinLevel` parameter. The return value is one bit per pin. GPIO_0 returns on Bit 0 of the `gpioPinLevel` parameter. A logic low level returns a 0, and a logic high level returns a 1. The function is as follows:

```
uint32_t TALISE_getGpioPinLevel(taliseDevice_t *device, uint32_t *gpioPinLevel)
```

Preconditions: execute `TALISE_initialize()`. The GPIO pin levels are not read back properly unless the desired GPIO pins are set to `GPIO_BITBANG_MODE` source control and set to input enable.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `*gpioPinLevel` is a pointer to a `uint32_t` variable that returns the GPIO pin levels that are read back on the pins that are assigned as inputs. Each bit corresponds to a GPIO pin. 0 = low output, and 1 = high output voltage.

TALISE_getGpioSetLevel()

This function reads the GPIO pin output levels for bitbang mode to drive the pins out. The function is as follows:

```
uint32_t TALISE_getGpioSetLevel(taliseDevice_t *device, uint32_t *gpioPinSetLevel)
```

Preconditions: execute `TALISE_initialize()`.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `*gpioPinSetLevel` is a pointer to a single `uint32_t` variable that returns the level that is set to output each output GPIO pin. Each bit corresponds to a GPIO pin (one bit per pin).

TALISE_setGpioMonitorOut()

This function configures the monitor output function for the GPIOs. The function is as follows:

```
uint32_t TALISE_setGpioMonitorOut(taliseDevice_t *device, uint8_t monitorIndex, uint8_t
monitorMask)
```

The monitor outputs allow visibility to some internal signals. Each monitor index outputs a set of eight signals. To output these signals on the low voltage GPIO_10 to GPIO_0 pins, set the desired GPIO_18 to GPIO_0 pin direction, and then set the GPIO nibble source control to allow the monitor signals to route to a set of four GPIO pins. If the GPIO_18 to GPIO_16 nibble source is set to monitor outputs, monitor output signals, Bits[6:4] (as shown in Table 130), are routed to GPIO_18 to GPIO_16. When the nibble source is set to monitor the outputs for GPIO_15 to GPIO_0, the monitor output signals, Bits[7:0] are routed to GPIO_07 to GPIO_0, and the monitor output signals, Bits[7:0] are also routed to GPIO_15 to GPIO_8.

Preconditions: execute `TALISE_initialize()`. Pins must be set to `GPIO_MONITOR_OUT` source control and output enable.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `monitorIndex` is the index that outputs a set of eight monitor outputs. See Table 130.
- `monitorMask` is a mask that indicates which outputs are sent along the assigned GPIO pins. Set to 0xFF to enable all monitor signals.

TALISE_getGpioMonitorOut()

This function reads the GPIO monitor index and monitor mask from the device. The function is as follows:

```
uint32_t TALISE_getGpioMonitorOut(taliseDevice_t *device, uint8_t *monitorIndex, uint8_t
*monitorMask)
```

Preconditions: execute `TALISE_initialize()`.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `*monitorIndex` is a pointer to a single `uint8_t` variable, which returns the current monitor signal selection index.
- `*monitorMask` is a pointer to a single `uint8_t` variable, which returns the monitor out signal masking. One bit is returned per monitor output signal.

TALISE_setRxGainCtrlPin ()

This function configures the gain step size and the GPIO inputs for receiver MGC. A high pulse on the `rxGainIncPin` in pin control mode increments the gain by the value set in the `incStep` parameter. A high pulse on the `rxGainDecPin` in pin control mode decrements the gain by the value set in the `decStep` parameter. The function is as follows:

```
uint32_t TALISE_setRxGainCtrlPin(taliseDevice_t *device, taliseRxChannels_t rxChannel,
taliseRxGainCtrlPin_t *rxGainCtrlPin)
```

Preconditions: execute `TALISE_initialize()`.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `rxChannel` is the `taliseRxChannels_t` enumerator type to select the Receiver 1 or Receiver 2 channel for programming.
- `*rxGainCtrlPin` is a pointer to the `taliseRxGainCtrlPin_t` structure that configures the receiver MGC.

TALISE_getRxGainCtrlPin ()

This API function returns the configuration (gain steps and GPIO inputs) for receiver MGC. The function is as follows:

```
uint32_t TALISE_getRxGainCtrlPin(taliseDevice_t *device, taliseRxChannels_t rxChannel,
taliseRxGainCtrlPin_t *rxGainCtrlPin)
```

Preconditions: execute `TALISE_initialize()`.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `rxChannel` is the `taliseRxChannels_t` enumerator type to select the Receiver 1 or Receiver 2 channel for programming.
- `*rxGainCtrlPin` is a pointer to the `taliseRxGainCtrlPin_t` structure that configures the manual receiver gain pin control.

TALISE_setSpi2Enable()

This function enables or disables the SPI2 port on the device. The function is as follows:

```
uint32_t TALISE_setSpi2Enable(taliseDevice_t *device, uint8_t spi2Enable,
taliseSpi2TxAttenGpioSel_t spi2TxAttenGpioSel)
```

The device can enable a second SPI port on the low voltage GPIO_3 to GPIO_0 pins. This SPI port allows read or write access to a limited set of transmitter attenuation and receiver gain index registers.

The SPI2 port uses the same configuration that is programmed for SPI, which includes LSB/MSB first, 4-wire mode, streaming and address increment.

The transmitter attenuation control includes a unique feature where the SPI register value can be set and does not update to the transmitter until a GPIO pin is toggled. The GPIO pin is level sensitive and selects the transmitter attenuation that is programmed in either the TXx_ATTENUATION_S1 or TXx_ATTENUATION_S2 bit fields of the second SPI registers. The GPIO pin that is used to switch between the two transmitter attenuation settings is user selectable using the enumerator in the function parameter.

For readback of the transmitter attenuation SPI registers, write to the desired register to force the value to be updated before reading the register back across the SPI.

Preconditions: execute `TALISE_initialize()`.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `spi2Enable` is the enable = 1 or disable = 0 SPI2 protocol on the device.
- `spi2TxAttenGpioSel` sets up the GPIO that is used to select between two transmitter attenuation values that are programmed through the SPI2 port. This GPIO is only used if SPI2 is enabled.

TALISE_getSpi2Enable()

This function receives the current status of the SPI2 port configuration on the device. The function is as follows:

```
uint32_t TALISE_getSpi2Enable(taliseDevice_t *device, uint8_t spi2Enable,
taliseSpi2TxAttenGpioSel_t spi2TxAttenGpioSel)
```

Preconditions: execute `TALISE_initialize()`.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `spi2Enable` is the enable = 1 or disable= 0 SPI2 protocol on the device.
- `spi2TxAttenGpioSel` sets up the GPIO that is used to select between two transmitter attenuation values programmed through the SPI2 port. This GPIO is only used if the SPI2 port is enabled.

GENERAL-PURPOSE INTERRUPT OPERATION

The general-purpose interrupt pin (GP_INTERRUPT) can alert the BBP that a significant event regarding device operation has occurred. These events include the unlocking of PLLs, stream processor errors, as well as other system errors that can occur. The GP_INTERRUPT pin is configured by the `TALISE_setGpIntMask()` function after device initialization. When a rising edge is detected on the GP_INTERRUPT pin, the BBP calls the `TALISE_getGpIntStatus()` function to determine the source of the interrupt.

Table 136 lists the available general-purpose interrupt sources. These sources are described with enumerators of the data type `taliseGpIntMask_t`.

Table 136. taliseGpIntMask_t Enumerations and Descriptions

taliseGpIntMask_t Enumerator Name	Enumerator Value	Description
TAL_GP_MASK_STREAM_ERROR	0x1000	Stream processor error.
TAL_GP_MASK_ARM_CALIBRATION_ERROR	0x0800	ARM calibration error.
TAL_GP_MASK_ARM_SYSTEM_ERROR	0x0400	ARM system error.
TAL_GP_MASK_ARM_FORCE_INTERRUPT	0x0200	ARM forced interrupt event.
TAL_GP_MASK_WATCHDOG_TIMEOUT	0x0100	ARM watchdog timer timeout.
TAL_GP_MASK_PA_PROTECTION_TX2_ERROR	0x0080	Tx2 power amplifier protection error.
TAL_GP_MASK_PA_PROTECTION_TX2_ERROR	0x0040	Tx2 power amplifier protection error.
TAL_GP_MASK_JESD_DEFRMER_IRQ	0x0020	JESD204B deframer IRQ error.
TAL_GP_MASK_JESD_FRAMER_IRQ	0x0010	JESD204B framer IRQ error.
TAL_GP_MASK_CLK_SYNTH_LOCK	0x0008	Device clock PLL unlock error. This bit is not sticky and tracks to the current PLL status.
TAL_GP_MASK_AUX_SYNTH_LOCK	0x0004	Auxiliary PLL unlock error. This bit is not sticky and tracks to the current PLL status.
TAL_GP_MASK_RF_SYNTH_LOCK	0x0002	RF PLL unlock error. This bit is not sticky and tracks to the current PLL status.

The GP_INTERRUPT pin represents a logical OR of the enabled GP_INTERRUPT mask sources. It is not necessary to enable all of the interrupt sources.

GP_INTERRUPT Handler

When the GP_INTERRUPT pin is asserted, the attempt to determine what caused the GP_INTERRUPT assertion through two different functions: `TALISE_getGpIntStatus()` and `TALISE_gpIntHandler()`.

The `TALISE_getGpIntStatus()` function can be used if the user only wants to determine what error condition caused the GP_INTERRUPT assertion. With this command, no further actions or information is acquired.

The `TALISE_gpIntHandler()` function can be used if the user wants to determine the GP_INTERRUPT source, clear the error, if possible, and then receive further diagnositic information, if requested. The third parameter passed in this command is a pointer to a `taliseGpIntInformation_t` data structure type. If this parameter is null, no diagnostic information is returned. The `taliseGpIntInformation_t` data structure is described in Table 137.

Table 137. taliseGpIntInformation_t Data Structure Parameters

Data Type	taliseGpIntInformation _t Struct Member	Description
uint8_t	data[8]	All GP_INTERRUPT sources. See Table 138.
taliseFramerSel_t	framer	Interrupting framer, only valid for framer sources.
taliseDeframerSel_t	deframer	Interrupting deframer, only valid for deframer sources
int32_t	deframerInputsMask	Interrupting deframer input mask (one bit per deframer input), only valid for deframer sources (valid for 0x0 to 0xF). deframerInputsMask is the deframer lane after the deframer lane crossbar swapping (lane input of the deframer).

The first parameter in the `taliseGpIntInformation_t` data structure is an 8-element array of `uint8_t`. This array can be decoded using the Table 138.

Table 138. Definitions for the 8-Element Array (data[8]) in the taliseGpIntInformation_t Data Structure

Bit Number	Array Element	Description
Data 0	0	Stream Rx1 enable signal falling edge error
	1	Stream Tx2 enable signal falling edge error
	2	Stream ORx2 enable signal rising edge error
	3	Stream ORx1 enable signal rising edge error
	4	Stream Rx2 enable signal rising edge error
	5	Stream Tx2 enable signal rising edge error
	6	Stream Rx1 enable signal rising edge error
	7	Stream Tx2 enable signal rising edge error
Data 1	0	Stream Loopback 2 enable signal falling edge error
	1	Stream Loopback 2 enable signal rising edge error
	2	Stream Loopback 2 enable signal rising edge error
	3	Stream Loopback 1 enable signal rising edge error
	4	Stream ORx2 enable signal falling edge error
	5	Stream ORx1 enable signalfalling edge error
	6	Stream Rx2 enable signal falling edge error
	7	Stream Tx2 enable signal falling edge error.
Data 2	0	Stream GPIO3 signal falling edge error
	1	Stream GPIO2 signalfalling edge error
	2	Stream GPIO1 signal falling edge error
	3	Stream GPIO0 signal falling edge error
	4	Stream GPIO3 signal rising edge error
	5	Stream GPIO2 signal rising edge error
	6	Stream GPIO1 signal rising edge error
	7	Stream GPIO0 signal rising edge error
Data 3	0	Stream ORx2 low to Rx2 high signal stream error
	1	Stream ORx1 low to ORx1 high signal stream error
	2	Stream Rx2 low to ORx2 high signal stream error
	3	Stream Rx1 low to ORx1 high signal stream error
	4	Stream; erroneous completion of pin mode stream for GP_IRQ signal rising edge
	5	Deframer A, bad disparity error
	6	Deframer A, not in table error
	7	Deframer A, unexpected k error
Data 4	0	Deframer A, interlane deskew
	1	Deframer A, initial lane sync
	2	Deframer A, good checksum
	3	Deframer A, frame sync
	4	Deframer A, code group sync
	5	Reserved
	6	Deframer A, pointers out of alignment
	7	Deframer A, SYSREF misalignment to LMFC

Bit Number	Array Element	Description
Data 5	0	Deframer B, bad disparity error
	1	Deframer B, not in table error
	2	Deframer B, unexpected k error
	3	Deframer B, interlane deskew
	4	Deframer B, initial lane sync
	5	Deframer B, good checksum
	6	Deframer B, frame sync
	7	Deframer B, code group sync
Data 6	0	Reserved
	1	Deframer B, pointers out of alignment
	2	Deframer B, SYSREF misalignment to LMFC
	3	Framer A, asynchronous FIFO pointer offset error
	4	Framer A, misalignment to current LMFC error
	5	Framer B asynchronous FIFO pointer offset error
	6	Framer B, misalignment to current LMFC error
	7	Power amplifier protection error for Tx2
Data 7	0	Power amplifier protection error for Tx2
	1	Arm calibration error
	2	Clock PLL lock detect reset
	3	Auxiliary PLL lock detect reset
	4	RF PLL lock detect reset
	5	Reserved
	6	Reserved
	7	Reserved

When using the `TALISE_gpIntHandler()` function, use the `uint32_t` return variable to determine the appropriate recovery mechanism.

GP_INTERRUPT PIN API FUNCTIONS

This section describes the API functions that pertain to the GP_INTERRUPT pin functionality.

TALISE_setGpIntMask()

This function can be called any time after device initialization. This command determines which GP_INTERRUPT sources can assert the GP_INTERRUPT pin. The function is as follows:

```
TALISE_setGpIntMask(taliseDevice_t *device, uint16_t gpIntMask)
```

Precondition: this function can be called any time after device initialization.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `mask` is a 16-bit word that indicates which interrupt sources are allowed to assert the GP_INTERRUPT pin.

TALISE_getGpIntStatus()

When the BBIC detects a rising edge on the GP_INTERRUPT pin, this function allows the BBIC to determine the source of the interrupt. The value returned in the status parameter shows one or more sources for the interrupt based on the `taliseGpIntMask_t` structure. The function is as follows:

```
TALISE_getGpIntStatus(taliseDevice_t *device, uint16_t *gpIntStatus)
```

The PLL unlock bits are not sticky. These bits follow the current status of the PLLs. If the PLL relocks, the status bit clears. The GP_INTERRUPT pin is the logical OR of all the sources. When all the status bits are low, the GP_INTERRUPT pin is low. The status word that is read back shows the current value for all interrupt sources, even if the sources are disabled by the mask. The GP_INTERRUPT pin only asserts for the enabled sources.

Precondition: this function can be called any time after device initialization and the `TALISE_setGpIntMask()` function has been executed.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `*gpIntStatus` is a 16-bit word that indicates which interrupt sources are currently asserted. Use the `taliseGpIntMask_t` structure to determine the source of the interrupt.

TALISE_getGpIntHandler()

This function is called whenever the BBIC detects a GP_INTERRUPT pin assertion to find the source and clear it. The function is as follows:

```
TALISE_getGpIntHandler(taliseDevice_t *device, uint32_t *gpIntStatus, taliseGpIntInformation_t
*gpIntDiag)
```

When the BBIC detects a rising edge on the GP_INTERRUPT pin, this function provides the BBIC with a simplified way to determine the GP_INTERRUPT source, clear it if possible, and receive a recovery action.

The PLL unlock bits are not sticky and follow the current status of the PLLs. If the PLL relocks, the status bit clears. The GP_INTERRUPT pin is the logical OR of all the sources. When all status bits are low, the GP_INTERRUPT pin is low. The status word readback shows the current value for all interrupt sources, even if the sources are disabled by the mask. The GP_INTERRUPT pin only asserts for the enabled sources.

Precondition: this function can be called any time after device initialization, and the `TALISE_setGpIntMask()` function has been executed.

Parameters include the following:

- `*device` is a pointer to the data structure.
- `*gpIntStatus` is a 32-bit word that indicates which interrupt sources are currently asserted. Use `taliseGpIntMask_t` to determine the source of the interrupt.
- `*gpIntDiag` is a pointer to a diagnostic structure that returns more specific error information from the GP_INTERRUPT source. If the pointer is NULL, no diagnostic information is returned.

3.3 V GPIO OPERATION

The device features twelve, 3.3 V capable GPIOs that can be configured for numerous functions. Similar to the low voltage GPIO pins, the 3.3 V GPIO pins can be used for monitoring or controlling external devices. However, not all functions of the 3.3 V GPIOs and the low voltage GPIOs are interchangeable.

The physical pins that are used to control the 3.3 V GPIO pins are the same physical pins that are used to control the AUXDACs (see Table 140). It is important to note that an AUXDAC function is given priority over a 3.3 V GPIO function that is assigned to the same GPIO.

Specific operation modes for the 3.3 V GPIO include level translate and inverted level translate mode, manual control of the 3.3 V GPIO logic level (also known as bitbang mode), and gain table external element control.

3.3 V GPIO Overview

The 3.3 V GPIO pins can be configured as input or output pins on a per pin basis. To configure the 3.3 V GPIO pins for input or output mode, use the `TALISE_setGpio3v3Oe()` function.

If a 3.3 V GPIO pin is configured as an input pin, no further action is necessary. The only input function available on the 3.3 V GPIO pin is to obtain the logic level of a 3.3 V input pin (1 or 0).

If a 3.3 V GPIO pin is configured as an output pin, it is necessary to set the source control for the pin. The source control determines the functionality of a group of output pins. Source control is assigned in nibble groups of four pins. There are three total nibble groups for the twelve 3.3 V GPIO pins. Each nibble group can have a different assignment. If required, the user can set a pin within a nibble group to input mode or to enable the AUXDAC. In this case, the nibble group assignment is ignored, and the function assigned to the individual GPIO is executed.

To set the source control, use the `TALISE_setGpio3v3SourceCtrl()` function. The source control assignments are described by the `taliseGpio3v3Mode_t` enumerations in Table 139.

Table 139. talise3v3GpioMode_t Enumerations for 3.3V GPIO Modes

taliseGpio3v3Mode_t Enumerator	Enumerator Value	Description
TAL_GPIO3V3_LEVELTRANSLATE_MODE	1	Level translate mode. Signal level on low voltage GPIO pins are level shifted to 3.3 V.
TAL_GPIO3V3_INVLEVELTRANSLATE_MODE	2	Inverted level translate mode. Inverse of signal levels on low voltage GPIO pins are outputs.
TAL_GPIO3V3_BITBANG_MODE	3	Manual control mode. When enabled, use TALISE_setGpio3v3PinLevel() to control the logic output level of a pin.
TAL_GPIO3V3_EXTATTEN_LUT_MODE	4	This mode configures specific 3.3 V GPIO pins to output the 4-bit, external attenuator control word for the selected gain index. Rx1 uses GPIO_3P3_3 to GPIO_3P3_0. Rx2 uses GPIO_3P3_7 to GPIO_3P3_4.

Setup and configuration of the 3.3 V GPIOs can be performed after initialization of the device.

3.3 V GPIO, Level Translate Mode

The 3.3 V GPIO level translate mode translates digital logic input signals from the low voltage GPIO pins to 3.3 V logic levels. The inputs and outputs for this function operate at the logic levels of each pin interface, specifically, at the logic level of the VDD_INTERFACE pin for low voltage GPIO pins, and at the logic level of the VDDA_3P3 pin for the 3.3 V GPIOs. This control is unidirectional from the low voltage GPIOs to the 3.3 V GPIOs. The device is capable of straightforward or inverted level translation.

Figure 140 shows the operation of the level translation block.

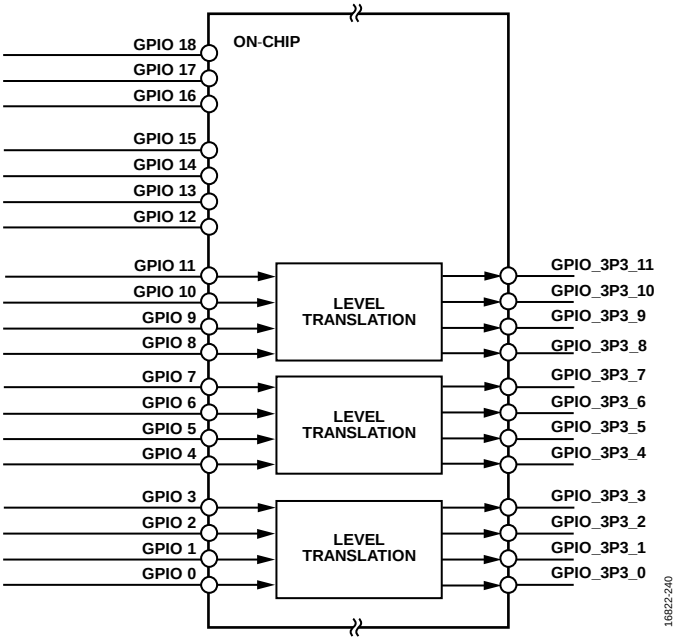

Figure 140. Level Translation Mode Between the Low Voltage GPIOs and 3.3 V GPIOs

To enable level translation mode, take the following steps after device initialization:

1. Set the desired low voltage GPIO pins to input mode with the `TALISE_setGpio0e()` function.
2. Set the desired 3.3 V GPIO pins to output mode with the `TALISE_setGpio3v30e()` function.
3. Set the source control to the desired level translation scheme.

3.3 V GPIO, Bitbang Mode

The 3.3 V GPIO bitbang mode allows the user to configure the 3.3 V GPIO pins as inputs or outputs where the device can read back or set pin logic levels. This mode is also referred to as manual mode. If a GPIO pin is configured as an input, the user can read back the logic level present at the input. The logic readback is either 0 or 1 (must be connected to ground or VDDA_3P3). If a GPIO pin is configured as an output, the user is able to set a logic level on the pin (must be connected to ground or VDDA_3P3).

Figure 141 shows the operation of the 3.3V GPIOs in bitbang mode.

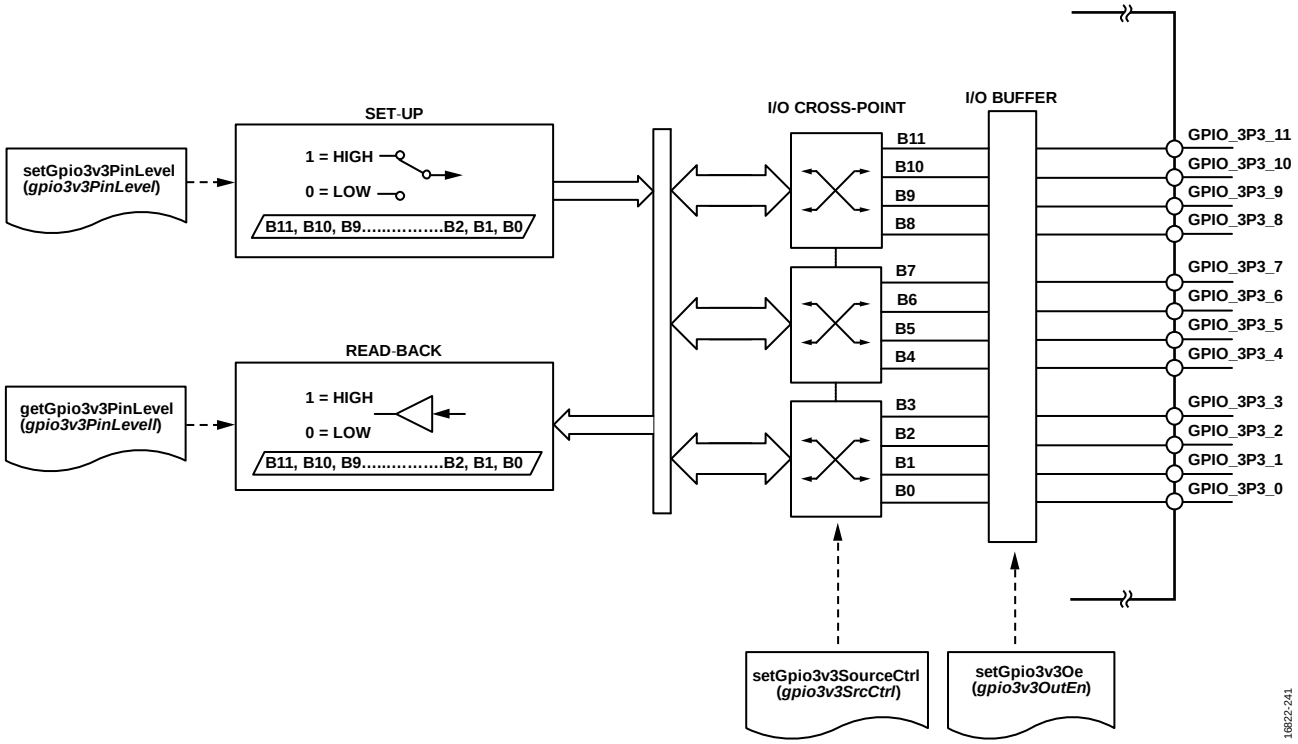

Figure 141. 3.3 V GPIO Hardware Configuration for Bitbang Mode

The `TALISE_setGpio3v3PinLevel()` function sets the output logic level on the GPIO pins is. This command requires an input parameter that specifies a logic level for each pin. However, the pin must be set to output mode and bitbang source control for the desired logic level to appear on the pin. To read back the expected output logic level of the GPIOs, use the `TALISE_getSet3v3GpioPinLevel()` function.

Use the `TALISE_getGpio3v3PinLevel()` function to read back the input logic level on the 3.3 V GPIOs. This function returns a 16-bit word where each bit corresponds to the input logic level detected on the GPIOs.

3.3 V GPIO, Gain Table External Element Control

The gain table, as discussed in the Receiver Gain Control section, includes a 4-bit, external element control column. This control column can be used to control an external, digitally stepped attenuator (DSA), or the column can be used for low noise amplifier (LNA) bypass control. This feature is convenient for users that require external elements to the transceiver to change gain based on the gain table setting within the transceiver, particularly in AGC scenarios.

To enable the gain table external element control, take the following steps after device initialization:

1. Load a gain table with non-zero values in the external element control column.
2. Set the 3.3 V GPIOs to output mode with the `TALISE_setGpio3v3Oe()` function.
3. Set the 3.3 V GPIO source control to `TAL_GPIO3V3_EXTATTEN_LUT_MODE` with the `TALISE_setGpio3v3SourceCtrl()` function.

After these steps are performed, the following 3.3V GPIO pins express the value of the external element control column for a specific receiver: Receiver 1 uses GPIO_3P3_3 to GPIO_3P3_0, and Receiver 2 uses GPIO_3P3_7 to GPIO_3P3_4.

The external element control values expressed on these pins depends on the gain index setting and the gain table. See the Receiver Gain Control for details.

API Functions for 3.3 V GPIOs

This section summarizes the API functions for 3.3 V GPIO function configuration.

TALISE_setGpio3v3Oe ()

This function sets the 3.3 V GPIO as an input or an output. Each GPIO can only be an input or an output (unidirectional) set by a transferred parameter. The `gpioUsedMask` parameter allows this function to only affect the GPIO pins of interest. The function is as follows:

```
uint32_t TALISE_setGpio3v3Oe(taliseDevice_t *device, uint16_t gpio3v3OutEn, uint16_t
gpio3v3UsedMask)
```

Parameters include the following:

- `*device` is a structure pointer to the device data structure.
- `gpioOutEn` = 1, the corresponding pin is configured as an output. If `gpioOutEn` = 0, the corresponding pin is configured as an input.
- `gpioUsedMask` is a mask used to control which OE bits are set/cleared. If this mask bit = 1, that bit is modified by the `gpioOutEn` bit.

TALISE_getGpio3v3Oe ()

This function retrieves the GPIO direction currently set in the device. The direction can be either output or input per pin. The return `gpioOutEn` function parameter returns one bit per GPIO pin. 1 = output, and 0 = input. The function is as follows:

```
uint32_t TALISE_getGpio3v3Oe(taliseDevice_t *device, uint16_t *gpio3v3OutEn)
```

Parameters include the following:

- `*device` is a structure pointer to the device data structure.
- `*gpioOutEn` = 1, the corresponding pin is configured as an output. If `*gpioOutEn` = 0, the corresponding pin is configured as an input.

TALISE_setGpio3v3SourceCtrl ()

This function sets the 3.3 V GPIO output source for different GPIO functionalities. This function only affects the GPIO pins that are set as outputs. Each GPIO nibble (four pins) can be assigned to a GPIO source and must share that same GPIO output source. The `taliseGpio3v3Mode_t` structure can be bit shifted and bitwise OR-ed together to create the value for the `gpioSrcCtrl` parameter. The function is as follows:

```
uint32_t TALISE_setGpio3v3SourceCtrl(taliseDevice_t *device, uint16_t gpio3v3SrcCtrl)
```

Parameters include the following:

- `*device` is a structure pointer to the device data structure.
- `gpio3v3SrcCtrl` is a nibble-based source control. This is a 12-bit value containing three nibbles that set the output source control for each set of four GPIO pins.

TALISE_getGpio3v3SourceCtrl ()

This function reads the 3.3 V GPIO output source for different GPIO functionalities. The function is as follows:

```
uint32_t TALISE_getGpio3v3SourceCtrl(taliseDevice_t *device, uint16_t *gpio3v3SrcCtrl)
```

Parameters include the following:

- `*device` is a structure pointer to the device data structure.
- `*gpio3v3SrcCtrl` is a pointer to readback word for nibble-based source control. This is a 12-bit value containing three nibbles that set the output source control for each set of four GPIO pins.

TALISE_setGpio3v3PinLevel ()

This function sets the 3.3 V GPIO output level. This function only affects the GPIOs that are set as outputs and that have the proper source control set for the nibbles in `TAL_GPIO3V3_BITBANG_MODE`.

```
uint32_t TALISE_setGpio3v3PinLevel(taliseDevice_t *device, uint16_t gpio3v3PinLevel)
```

Parameters include the following:

- `*device` is a structure pointer to the device data structure.
- `gpio3v3PinLevel` is the bit returned per GPIO pin and indicates the level to output for each GPIO pin. 0 = low output, and 1 = high output.

TALISE_getGpio3v3PinLevel ()

This function reads the 3.3 V GPIO pin level. The GPIO pins that are set as inputs read back and are returned in the `gpioPinLevel` parameter. The return value is one bit per pin. GPIO_3P3_0 returns on Bit 0 of the `gpioPinLevel` parameter. A logic low level returns a 0, and a logic high level returns a 1. The function is as follows:

```
uint32_t TALISE_getGpio3v3PinLevel(taliseDevice_t *device, uint16_t *gpio3v3PinLevel)
```

Parameters include the following:

- `*device` is a structure pointer to the device data structure.
- `*gpio3v3PinLevel` is a pointer to the readback word. One bit is returned per GPIO pin, and the return is the level to be output for each GPIO. 0 = low input, and 1 = high input.

TALISE_getGpio3v3SetLevel ()

This function reads the GPIO pin output levels for bitbang mode. This function allows the readback of the value that the GPIO output pins are set to so that the pins can be driven out. The function is as follows:

```
uint32_t TALISE_getGpio3v3SetLevel(taliseDevice_t *device, uint16_t *gpio3v3PinSetLevel)
```

Parameters include the following:

- `*device` is a structure pointer to the device data structure.
- `*gpio3v3PinLevel` is a pointer to the readback word. One bit is read back per GPIO pin, and the return is the level to be output for each GPIO. 0 = low input, and 1 = high input.

AUXILIARY CONVERTERS AND TEMPERATURE SENSOR

The integrated transceiver contains integrated auxiliary data converters including auxiliary DACs and auxiliary ADCs. The device also supports a diode-based temperature sensor that can provide the current temperature of the transceiver. These features are included to simplify control tasks for the BBIC, take static measurements during operation, and provide flexibility that can be used across multiple applications without adding external components. This section outlines the operation of these features along with the API functions that are required to configure the circuitry.

AUXILARY DAC (AUXDAC)

The are 12 independent AUXDACs integrated onto the device that is operating on the VDDA_3P3 pin supply domain. AUXDAC_0 to AUXDAC_9 are 10-bit current steering DACs that can support an aggregate 12 bits with a programmable reference voltage select. AUXDAC_10 and AUXDAC_11 are true, 12-bit DACs. The AUXDAC output mapping to 3.3 V GPIO pins is described in Table 140.

Table 140. AUXDAC Mapping to 3.3 V GPIO Pins

AUXDAC Number	3.3 V GPIO	Resolution Supported
AUXDAC_0	GPIO_3P3_10	10-bit, 11-bit subrange, 12-bit subrange
AUXDAC_1	GPIO_3P3_8	10-bit, 11-bit subrange, 12-bit subrange
AUXDAC_2	GPIO_3P3_7	10-bit, 11-bit subrange, 12-bit subrange
AUXDAC_3	GPIO_3P3_11	10-bit, 11-bit subrange, 12-bit subrange
AUXDAC_4	GPIO_3P3_0	10-bit, 11-bit subrange, 12-bit subrange
AUXDAC_5	GPIO_3P3_1	10-bit, 11-bit subrange, 12-bit subrange
AUXDAC_6	GPIO_3P3_4	10-bit, 11-bit subrange, 12-bit subrange
AUXDAC_7	GPIO_3P3_5	10-bit, 11-bit subrange, 12-bit subrange
AUXDAC_8	GPIO_3P3_6	10-bit, 11-bit subrange, 12-bit subrange
AUXDAC_9	GPIO_3P3_9	10-bit, 11-bit subrange, 12-bit subrange
AUXDAC_10	GPIO_3P3_2	12-bit
AUXDAC_11	GPIO_3P3_3	12-bit

Note that although the ten 10-bit, current steering DACs are true 10-bit DACs, the API can refer to the resolution of the DAC as effectively an 11-bit or a 12-bit DAC. The effective 11-bit or 12-bit DAC uses the 10-bit DAC but has configurable voltage references (1 V, 1.5 V, 2 V, and 2.5 V) and high slope (11-bit) and low slope (12-bit) options that allow the 10-bit DAC to span a wider effective range. The voltage reference and slope are not adjustable in the 12-bit mode because this mode is designed to span the range from 0 V to 3.3 V. In all modes using the 10-bit DAC, the accepted input codes range from 0 to 1023.

The 10-bit, current steering DACs for AUXDAC_0 to AUXDAC_9 have programmable voltage reference points and a programmable voltage resolution per LSB input. Figure 142 shows the 10-bit AUXDACs. The 10-bit input code to each AUXDAC is independent to other AUXDACs. To ensure stability for AUXDAC_0 to AUXDAC_9, a 100 nF bypass capacitor is required at the respective 3.3 V GPIO pin.

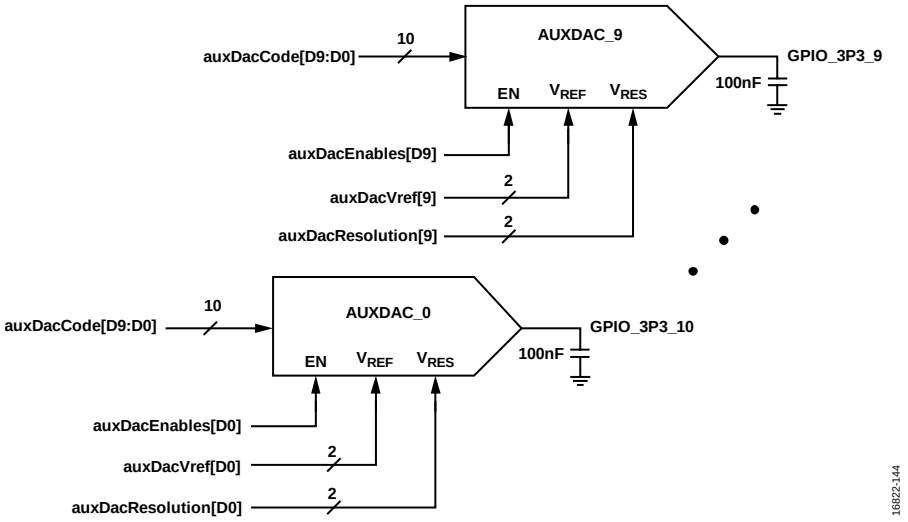

Figure 142. Auxiliary DAC Schematic for 10-Bit Auxiliary DACs

The true 12-bit DAC for AUXDAC_10 and AUXDAC_11 does not feature a programmable voltage reference or programmable voltage resolution. However, the 12-bit AUXDAC supports a full 12-bit resolution without subranging. The following diagram illustrates the 12-bit AUXDACs. Use a bypass capacitor of less than 100 pF for stability.

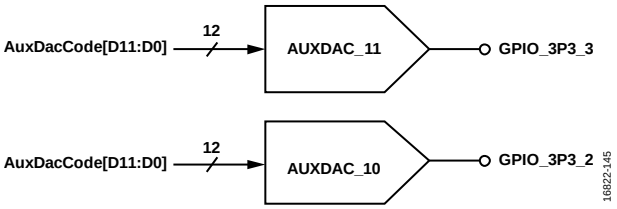

Figure 143. AUXDAC schematic for 12-bit AUXDACs

The AUXDACs are designed to be used in feedback loop operations. For example, an AUXDAC can be used to generate a voltage supply used to control a voltage controlled crystal oscillator (VCXO) voltage input. For such control system use cases, the absolute value of the voltage output is not critical, but the voltage steps must be 12-bit, accurate, and monotonic. The feedback of the servo loop assists to regulate the voltage input to the desired level.

Configuration of the AUXDACs is achieved with the `TALISE_setupAuxDacs()` function. Setting the values for each AUXDAC can also be performed in this function. Alternatively, where the AUXDAC value must change after AUXDAC configuration, the `TALISE_writeAuxDac()` function can be used. The `TALISE_writeAuxDac()` function allows the user to set a new input code to a selected AUXDAC.

AUXDAC Voltage Transfer Functions

The 10-bit, 11-bit subrange, 12-bit subrange, and true 12-bit AUXDAC ideal voltage transfer functions can be calculated with equations described in this section. Note that gain and offset variations are not described by these equations.

The 10-bit AUXDAC, subranged 11-bit AUXDAC, and subranged 12-bit AUXDAC ideal voltage transfer function can be described by the following equation.

$$V_{OUT}(Code) = (1 + 0.5 \times AuxDacVref[1:0]) + \frac{0.00143136\,(1 + lowres)(1.094 \times Code - 511)}{1 + AuxDACStepFactor[1:0]} \quad (2)$$

Where:
auxDacVref[1:0] = 0, 1, 2, or 3. These correspond to a reference voltage (V_{REF}) of 1.0 V, 1.5 V, 2.0 V, or 2.5 V, respectively.
lowres = 0 or 1. *lowres* = 1 when the AUXDAC resolution is set to 10-bit, and *lowres* = 0 otherwise.
auxDACStepFactor[1:0] = 0 or 1. *auxDACStepFactor* = 1 when the AUXDAC resolution is set to 12-bit, and *auxDACStepFactor* = 0 otherwise.
Code = {0, 1, … 1023}. This variable corresponds to the AUXDAC input code value.

Figure 144 to Figure 147 show the idealized transfer functions for the 10-bit, 11-bit subrange, and 12-bit subrange AUXDACs. The items in the legends of these figures denote the data structure configured reference voltage level and resolution of the AUXDAC. These plots are calculated with Equation 2.

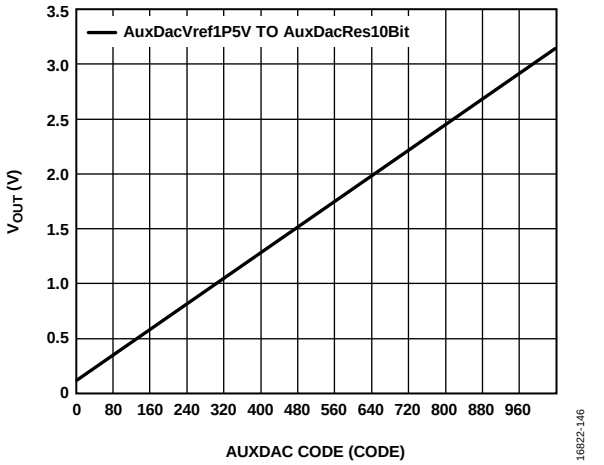

Figure 144. Ideal Voltage Transfer Function for 10-Bit AUXDAC

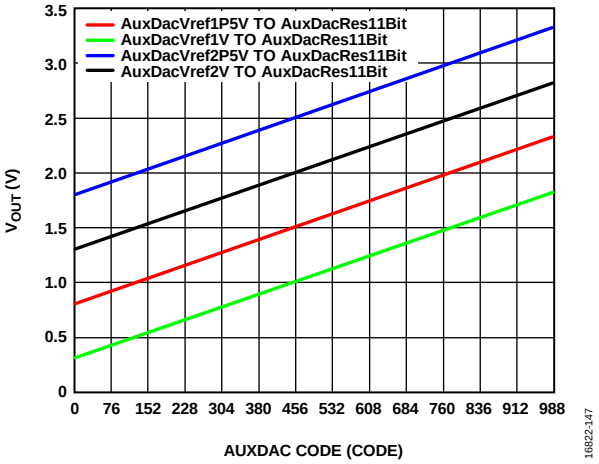

Figure 145. Ideal Voltage Transfer Function for 11-Bit (Subrange) AUXDAC

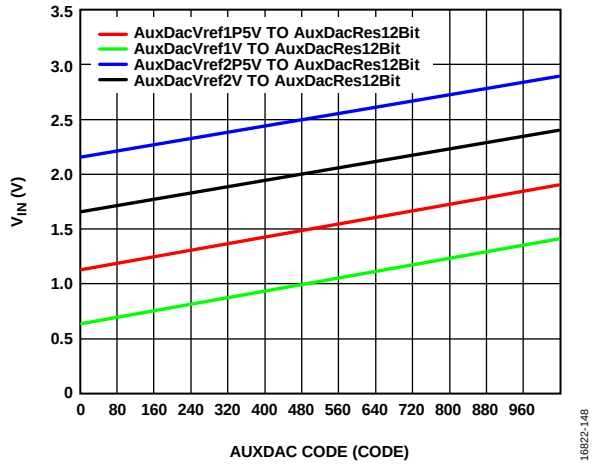

Figure 146. Ideal Voltage Transfer Function for 12-Bit (Subrange) AUXDAC

The true, 12-bit AUXDAC ideal voltage transfer function can be represented by the following equation:

$$V_{OUT}(Code) = (Code \times 3.3\ V)/4095$$

where:
V_{OUT} is the output voltage.
Code = {0, 1, … 4095}. This variable corresponds to the AUXDAC input code value.

The ideal voltage transfer function for the true, 12-bit AUXDAC is shown in Figure 147. Note that there are only two true, 12-bit AUXDACs available.

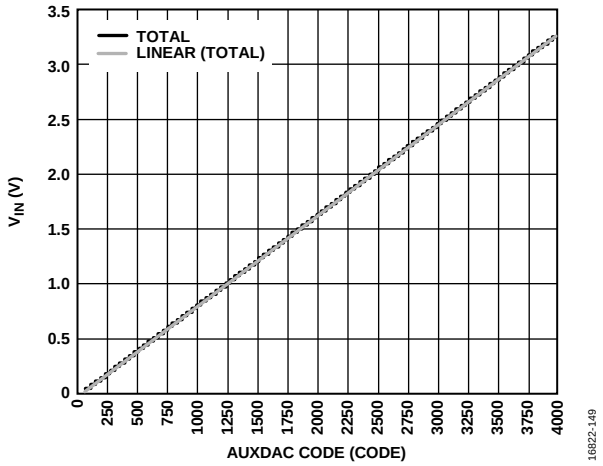

Figure 147. Ideal Voltage Transfer Function for True, 12-Bit AUXDAC

Data Structures and Enumerations for AUXDAC Programming

This section describes the data structures and enumerations that pertain to AUXDAC programming.

taliseAuxDac_t Data Structure

The `taliseAuxDac_t` data structure stores the configuration settings for all AUXDACs. The configuration settings stored in this structure are described Table 141.

Table 141. taliseAuxDac_t Data Structure Parameters

Data Type	taliseAuxDac_t Structure Member	Description
uint16_t	auxDacEnables	AUXDAC enable bit for each DAC, where the first 10 bits correspond to the 10-bit DACs and the consecutive 2 bits enable the 12-bit DACs.
taliseAuxDacVref_t[10]	auxDacVref	AUXDAC voltage reference value for each of the 10-bit DACs. This is a 10 element array.
taliseAuxDacResolution_t[10]	auxDacResolution	AUXDAC slope (resolution of voltage change per AUXDAC code). Only applies to the 10-bit DACs. This is a 10 element array.
uint16_t[12]	auxDacValues	AUXDAC values for each 10-bit DAC correspond to the first 10 array elements, the next consecutive array elements correspond to the two true 12-bit AUXDACs.

This data structure is programmed to the device registers when the `TALISE_setupAuxDacs()` function is called.

taliseAuxDacResolution_t Enumeration

The enumeration `taliseAuxDacResolution_t` is specific to the 10-bit AUXDACs. The `taliseAuxDacResolution_t` parameter allows configuration of the change in voltage per LSB of the input DAC code. This enumerator is described in Table 142.

Table 142. taliseAuxDacResolution_t Enumeration Descriptions

taliseAuxDacResolution_t Enumerator Name	Enumerator Value	Description
TAL_AUXDACRES_10BIT	0	10-bit DAC resolution mode. In this mode, the voltage range is from 100 mV to 3 V. The reference voltage in this mode is 1.5 V.
TAL_AUXDACRES_11BIT	1	11-bit DAC resolution mode for a subset of the output voltage range centered around V_{REF}. The ΔmV/LSB is approximately 1.404 mV/LSB.
TAL_AUXDACRES_12BIT	2	12-bit DAC resolution mode for a subset of the output voltage range centered around V_{REF}. The ΔmV/LSB is approximately 0.702 mV/LSB. Select this enumerator for the subrange 12-bit DAC or the true 12-bit DAC.

taliseAuxDacVref_t Enumeration

The enumeration `taliseAuxDacVref_t` is specific to the 10-bit AUXDACs when using the 11-bit or 12-bit subranged modes described in Table 142. This parameter allows flexible configuration of the reference voltage to four different values. This enumerator is described Table 143. The `Vref` parameter cannot be selected in true 12-bit AUXDAC or 10-bit AUXDAC.

Table 143. taliseAuxDacVref_t Enumeration Descriptions

taliseAuxDacVref_t Enumerator Name	Enum Value	Description
TAL_AUXDACVREF_1V	0	10-bit AUXDAC reference voltage is 1 V.
TAL_AUXDACVREF_1P5V	1	10-bit AUXDAC reference voltage is 1.5 V.
TAL_AUXDACVREF_2V	2	10-bit AUXDAC reference voltage is 2 V.
TAL_AUXDACVREF_2P5V	3	10-bit AUXDAC reference voltage is 2.5 V.

API Functions for AUXDAC Programming

This section describes the API functions for programming the AUXDACs.

TALISE_setupAuxDacs()

This function sets up the 12 AUXDACs on the device. The function uses the configuration in the `taliseAuxDac_t` data structure function parameter to setup each of the 12 DACs. This function can be called any time after the `TALISE_initialize()` function is called to reconfigure, enable, or disable the different DAC outputs. The DACs are used in manual control mode. After calling this setup function, it is possible to change a specific DAC code by calling the `TALISE_writeAuxDac()` function. The function is as follows:

```
uint32_t TALISE_setupAuxDacs(taliseDevice_t *device, taliseAuxDac_t *auxDac)
```

The AUXDAC outputs share the 3.3 V GPIO pins. When using an AUXDAC on a particular GPIO pin, ensure that the GPIO pin is set to be an input pin to tristate the GPIO pad driver.

Preconditions: complete device initialization.

Parameters include the following:

- `*device` is a pointer to the device data structure.
- `*auxDac` is a pointer to the `taliseAuxDac_t` data structure .

TALISE_writeAuxDacs()

This function writes the current auxiliary DAC code for a specific AUXDAC. The function is as follows:

```
uint32_t TALISE_writeAuxDac(taliseDevice_t *device, uint8_t auxDacIndex, uint16_t auxDacCode)
```

Preconditions: complete device initialization and call the `TALISE_setupAuxDacs()` function.

Parameters include the following:

- `*device` is a pointer to the device data structure.
- `auxDacIndex` selects the desired DAC to load the `auxDacCode` for AUXDAC_0 to AUXDAC_11. Values 0 to 9 correspond to the ten 10-bit DACs and values 10 and 11 corresponds to the two 12-bit DACs.
- `auxDacCode` is the DAC code to write to the selected AUXDAC. Sets the output voltage of the AUXDAC (valid code values are 0 to 1023 for `auxDacIndex` values 0 to 9), (valid code values are 0 to 4095 for `auxDacIndex` values 10 and 11).

AUXILIARY ADC (AUXADC)

The AUXADC on the device provides sixteem 3.3 V inputs for external analog-to-digital conversions. The AUXADC is a 12-bit converter and provides users with a high impedance input that can simplify board designs by potentially eliminating the need for additional external ADCs. The API currently supports the four AUXADC channels routed to a header on the evaluation board.

Calibration can be performed to obtain a full, 12-bit resolution for absolute measurements. If only relative and/or 7-bit uncorrected accuracy is required, the AUXADC can be used uncalibrated.

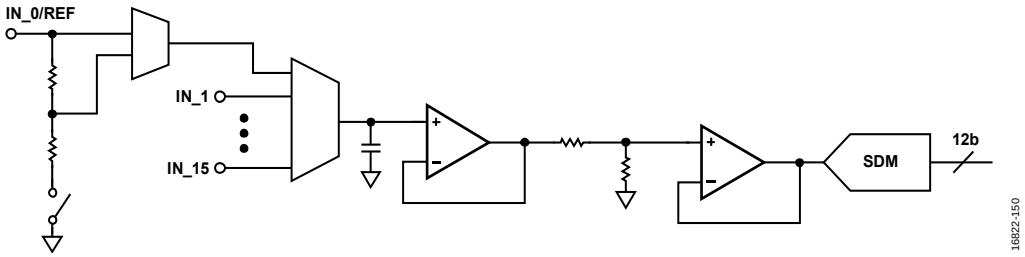

Figure 148. AUXADC Block Diagram

The true, 12-bit AUXADC ideal voltage transfer function can be represented by the following equation:

$$V_{IN} = Code \times 3.3\ V/4095$$

Where *Code* = {0, 1, … 4095}. This variable corresponds to the AUXADC result value.

The ideal voltage transfer function for the true, 12-bit AUXADC is shown in Figure 149. Each AUXADC is a true, 12-bit AUXADC.

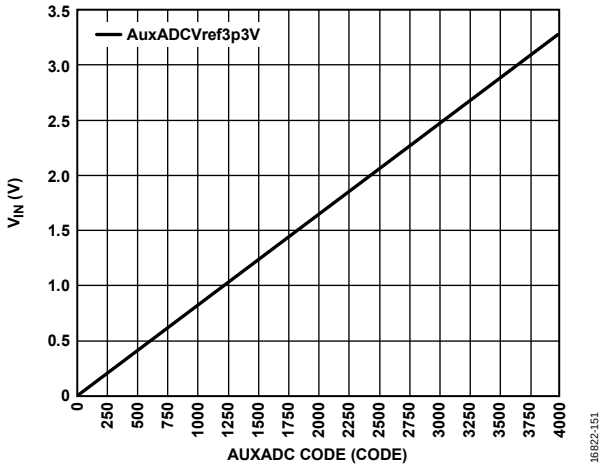

Figure 149. Ideal Voltage Transfer Function for 12-Bit AUXADC

AUXADC Calibration

The system offset and gain errors can be calibrated out by taking three measurements. From these measurements, offset, and gain error can be calculated, and a correction value can be determined. The first two measurments are made using the REF and REF/2 on the AUXADC input, IN_0. Slope (gain error) and offset can be calculated using these two points. A third measurement is then taken on the data input to which the gain and offset error correction values are applied. This results in a calibrated measurement result.

An external reference voltage can be supplied to AUXADC_0. An internal divider and mulitplexer allow this reference to be divided by two, which can be achieved by setting Bit 2 of SPI Register 0x1206. Make a measurement of REF/2. The following procedure describes the steps:

1. Make measurement of REF at AUXADC_0 and store as VO2.
2. Set Bit 2 at SPI Register 0x1206 to divide REF by 2.
3. Make measurement of REF/2 at AUXADC_0 and store as VO1.
4. Clear Bit 2 at SPI Register 0x1206.
5. Calculate the slope (m) with the following equation:

   $$m = (VO2 - VO1)/(REF - REF \div 2)$$

   where:
   VO2 is the REF voltage measurement.
   VO1 is the REF/2 voltage measurement.
   REF is the external reference voltage applied to AUXADC_0.
6. Calculate the offset with the following equation:

   $$b = VO2 - m \times REF$$

7. Use b and m to correct signal measurement with the following equation:

   $VO' = VO - b/m$

   where:
   VO' is the corrected signal measurement.
   VO is the signal measurement from the AUXADC.

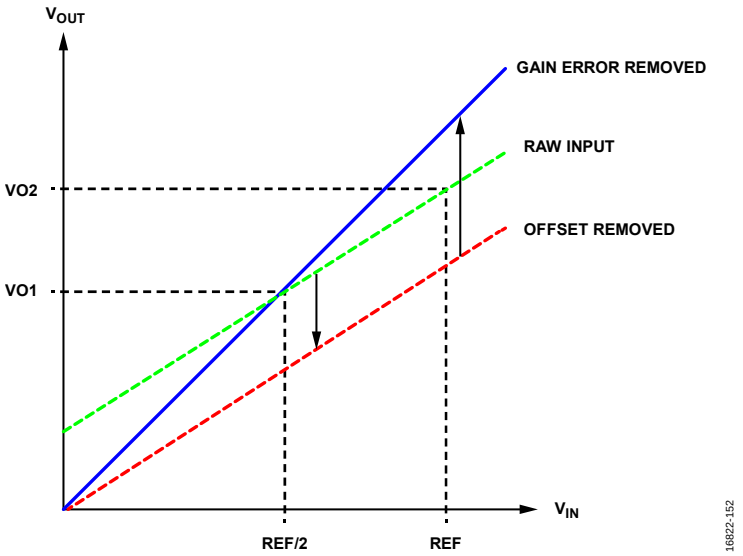

Figure 150. Offset and Gain Error Calibrations

Data Structures and Enumerations for AUXADC Programming

This section describes the data structures and enumerators for AUXADC programming.

taliseAuxAdcConfig_t Data Structure

The `taliseAuxAdcConfig_t` data structure stores the configuration settings for all AUXADCs. The configuration settings stored in this structure are described in Table 144.

Table 144. taliseAuxAdcConfig_t Data Structure Parameters

Data Type	taliseAuxAdcConfig_t Structure Member	Description
taliseAuxAdcChannels_t	auxAdcChannelSel	Selects the channel which is supposed to sample AUXADC input for analog-to-digital conversion.
taliseAuxAdcModes_t	auxAdcMode	Selects mode to latch and store conversion results.
uint16_t	numSamples	Number of analog-to-digital conversions to be performed in 1 to 1000 range.
uint16_t	samplingPeriod_us	Sampling interval in µs (minimum 15 µs). Valid only for nonpin mode. Ignored for pin mode.

taliseAuxAdcChannels_t Enumeration

The `taliseAuxAdcChannels_t` enumeration is specific to the 12-bit AUXADCs. This enumeration is described in Table 145.

Table 145. taliseAuxAdcChannels_t Enumeration Description

taliseAuxDacVref_t Enumerator Name	Enumerator Value	Description
TAL_AUXADC_CH0	0	Select AUXADC Channel 0 for sampling and conversion
TAL_AUXADC_CH1	1	Select AUXADC Channel 1 for sampling and conversion
TAL_AUXADC_CH2	2	Select AUXADC Channel 2 for sampling and conversion
TAL_AUXADC_CH3	3	Select AUXADC Channel 3 for sampling and conversion

taliseAuxAdcModes_t Enumeration

The `taliseAuxAdcChannels_t` enumeration is specific to the 12-bit AUXADCs. This enumeration is used to specify AUXADC sampling and conversion mode and is described in Table 146.

Table 146. taliseAuxAdcModes_t Enumeration Description

taliseAuxDacVref_t Enumerator Name	Enumerator Value	Description
TAL_AUXADC_NONPIN_MODE	0	Select AUXADC sampling and conversion in nonpin mode (Arm internal timer is used for sampling and conversion).
TAL_AUXADC_PIN_MODE	1	Select AUXADC sampling and conversion in pin mode (pulses on Arm GPIO input pins are used to schedule sampling and conversion).

taliseAuxAdcResult_t Data Structure

The `taliseAuxAdcResult_t` data structure stores the results of scheduled AUXADC conversions. The members stored in this structure are described in Table 147.

Table 147. taliseAuxAdcResult_t Data Structure Parameters

Data Type	taliseAuxAdcConfig_t Structure Member	Description
uint16_t	auxAdcCodeAvg	12-bit average of AUXADC analog-to-digital conversion samples.
uint16_t	numSamples	Number of samples averaged in auxAdcCodeAvg member.
uint8_t	completeIndicator	Flag to indicate if a scheduled AUXADC conversion is completed. 1 = conversion complete, and 0 = conversion incomplete.

Nonpin Mode AUXADC Conversions

The nonpin mode AUXADC conversions are initiated in the software. Prior to initiating an AUXADC conversion, the `AuxAdcConfig` parameter of the `taliseAuxAdcConfig_t` data structure must be set up. The AUXADC channel must be set, the `auxADCMode` must be set to `TAL_AUXADC_NONPIN_MODE`, and the number of conversions to be made and conversion period must be set. The `TALISE_startAuxAdc()` function initiates the nonpin mode conversions.

Results can be read back into the `taliseAuxAdcResult_t` data structure with the `TALISE_readAuxAdc()` function.

Pin Mode AUXADC Conversions

Pin mode AUXADC conversions are initiated by a pulse applied to a selected GPIO input pin. The `TALISE_setAuxAdcPinModeGpio()` function assigns a GPIO input to the AUXADC as a start signal. Passing a valid 1.8 V GPIO (GPIO_0 to GPIO_15) assigns the GPIO to start the AUXADC conversions, if the GPIO is not already assigned to another feature. Transferring the `TAL_GPIO_INVALID` command unassigns the GPIO pin that was previously assigned to the AUXADC. GPIO assignment must be made when the device is in idle state (`TALISE_RadioOff()`).

Prior to initiating an AUXADC conversion, the `AuxAdcConfig` parameter of the `taliseAuxAdcConfig_t` data structure must be setup. AUXADC channel must be set, `AuxADCMode` must be set to `TAL_AUXADC_PIN_MODE`, and the number of conversions to be made.

The `TALISE_startAuxAdc()` function allows the AUXADC conversions to be made when pulses are detected at the assigned GPIO pin. Be aware that after calling the `TALISE_startAuxADC()` function, the GPIO pulse are ignored for 150 µs. Additionally, after this, the pulses are ignored every 2 sec for a duration of 150 µs, is due to sharing the AUXADC with the internal temperature sensor. Temperature measurements take control of the AUXADC for 150 µs every 2 seconds.

Results can be read back into the `taliseAuxAdcResult_t` data structure with the `TALISE_readAuxAdc()` function.

API Functions for AUXADC Conversions

This section describes the API functions for programming the AUXADC conversions.

TALISE_getAuxAdcPinModeGpio()

This function returns the GPIO pin associated with the AUXADC start signal in the Arm processor for pin mode AUXADC operations. The function is as follows:

```
uint32_t TALISE_getAuxAdcPinModeGpio(taliseDevice_t *device, taliseGpioPinSel_t* pinModeGpio)
```

The only valid GPIO pin selects are GPIO_0 to GPIO_15. TAL_GPIO_INVALID is returned where no valid GPIO is assigned to the AUXADC start signal.

This function has no preconditions.

Parameters include the following:

- `*device` is a pointer to the device data structure.
- `* pinModeGpio` is a pointer to the `taliseGpioPinSel_t` enumerator type to the GPIO pin that is currently associated with the AUXADC that the start signal is written to.

TALISE_setAuxAdcPinModeGpio()

This function assigns the requested GPIO pin to the AUXADC start signal in the Arm processor The Arm processor expects a low (0) to high (1) pulse on the assigned GPIO pin to start the AUXADC conversion. Only GPIO_0 to GPIO_15 are valid GPIO pin selects. TAL_GPIO_INVALID is returned where no valid GPIO is assigned to the AUXADC start signal. The function is as follows:

```
uint32_t TALISE_getAuxAdcPinModeGpio(taliseDevice_t *device, taliseGpioPinSel_t pinModeGpio)
```

Preconditions: this function is only supported when the device is in an idle state. It is required to call `TALISE_RadioOff()` before setting up the GPIO pin for the AUXADC.

Parameters include the following:

- `*device` is a pointer to the device data structure.
- `pinModeGpio` is the GPIO pin assigned and unassigned from the AUXADC start signal in the Arm processor.

TALISE_startAuxAdc()

This function configures one of four 12-bit AUXADCs on the device for sampling and analog-to-digital conversion. This function uses the configuration in the `taliseAuxAdcConfig_t` data structure function parameter to set up the AUXADC channel to be used for external analog-to-digital conversion use. In nonpin mode, this function initiates the analog-to-digital conversions. In pin mode, this function enables the analog-to-digital conversion to be initiated by pulsing the selected GPIO from low to high. The function is as follows:

```
uint32_t TALISE_startAuxAdc (taliseDevice_t *device, taliseAuxAdcConfig_t* auxAdcConfig)
```

Preconditions: if pin mode is selected, GPIO pin must be assigned as the AUXADC start signal and cannot be used for another feature. The AUXADC must be configured via the `auxAdcConfig` data structure.

Parameters include the following:

- `*device` is a pointer to the device data structure.
- `*auxAdcConfig` is a pointer to the `auxAdcConfig` structure.

Note that temperature sensor measurements are prioritized over external AUXADC use. Temperature sensor measurements work in radio on and radio off modes.

TALISE_readAuxAdc()

This function reads the analog-to-digital conversion result from the Arm mailbox and updates the `auxAdcResult` data structure. The result is valid depending on the complete indicator field. The `CompleteIndicator` parameter in the `auxAdcResult_t` data structure reads back as 0 if the conversions are not complete and reads back as 1 if the conversions are complete. The function is as follows:

```
uint32_t TALISE_readAuxAdc(taliseDevice_t *device, taliseAuxAdcResult_t)
```

Preconditions: can be called any time after `TALISE_initialize()` and following a `TALISE_startAuxAdc()` call.

Parameters include the following:

- `*device` is a pointer to the device data structure.
- `*auxAdcResult` is a pointer to the `taliseAuxAdcResult_t` data structure where the result of the AUXADC conversion is written.

Note that this function works in radio on and radio off modes.

TEMPERATURE SENSOR

The temperature sensor provides the means to read back the current temperature as determined by the Arm processor. The temperature sensor reading is scaled to return the value in degrees Celsius.

API Functions for Temperature Sensor Readback

This section describes programming functions for the temperature sensor.

TALISE_getTemperature()

This function reads the temperature sensor of the device by requesting the latest temperature sensor value from the Arm processor. The temperature sensor value read back is scaled to return the temperature as degrees Celsius. The function is as follows:

```
uint32_t TALISE_getTemperature(taliseDevice_t *device, int16_t *temperatureDegC)
```

Preconditions: this function can be called after the device has been fully initialized any time during run-time operation but only after the Arm processor has been configured.

Parameters include the following:

- `*device` is a pointer to the device data structure.
- `temperatureDegC` is a pointer to a single `uint16_t` element that returns the current 12-bit temperature sensor value in degrees Celsius.

TRANSMITTER ATTENUATION

The device uses an accurate and efficient method of transmit power control (transmitter attenuation control) that involves a minimum interaction with the BBP. The transmitter attenuation can be set directly via the API or through an SPI2 mode, which enables real-time operation using a GPIO pin. For more information on SPI2, see the Transmitter Attenuation Control, SPI2 Port section. The transmitter attenuation control is implemented with the analog and digital gain in the transmitter signal chain. Gain settings are controlled via an internal lookup table. The attenuation controls are programmable and provide a range of 0 dB to 41.95 dB of attenuation. The attenuation step size is 0.05 dB, which results in 840 available attenuation settings. These attenuation settings are abstracted by the API for ease of use and efficient BBP implementation. See the targeted data sheet for the supported range of attenuation during device operation.

API FUNCTIONS FOR TRANSMITTER ATTENUATION

This section describes the API functions required to program the device for transmitter attenuation.

Talise.c Functions for Transmitter Atteunuation

The following function sets transmitter attenuation:

```
TALISE_setTxAttenuation(taliseDevice_t* device, taliseTxChannel_t txChannel,
        uint16_t txAttenuation_mdB)
```

Parameters include the following:

- `*device` is the structure pointer to the device data structure.
- `txChannel` is one of the two possible transmitter channels: Transmitter 1 or Transmitter 2. Only one transmitter can be set at a time.
- `txAttenuation_mdB` is the desired attenuation value expressed in mdB. The valid range for values of `txAttenuation_mdB` is 0 mdB to 41950 mdB. If `txAttenuation_mdB` is out of range, and invalid parameter error generates.

TRANSMITTER NCO INTERNAL SIGNAL SOURCE

A transmitter NCO test tone can be generated in the digital section of the device that is transmitted out the transmitter RF outputs. The NCO frequency can be set from –Transmitter Input Rate/2 to +Transmitter Input Rate/2. The transmitter attenuation is manually overridden when the `TALISE_enableTxNco()` function is enabled. Analog transmitter attenuation is set to 0 (maximum output power), and digital is set for an attuenation of 6 dB to prevent clipping any digital filters.

TRANSMITTER NCO API FUNCTIONS

Transmitter NCO Talise.c Functions

The transmitter NCO function is set by the following:

```
TALISE_enableTxNco(taliseDevice_t *device, taliseTxNcoTestToneCfg_t *txNcoTestToneCfg)


/**
*  \brief Data structure to hold ADRV900x Tx NCO test tone Configuration
*/
typedef struct
{
        uint8_t enable;                 /*!< 0 = Disable Tx NCO, 1 = Enable Tx NCO on both
transmitters */
        int32_t tx2ToneFreq_kHz;        /*!< Signed frequency in kHz of the desired Tx2 tone */
        int32_t tx2ToneFreq_kHz;        /*!< Signed frequency in kHz of the desired Tx2 tone */
} taliseTxNcoTestToneCfg_t;
```

MINIMUM SWITCHING TIMES FOR THE ADRV9008-1, ADRV9008-2, AND ADRV9009

This section provides analysis on retrieving the minimum switching time for the receiver, transmitter, and observation receiver enable signals for the ADRV9008-1, ADRV9008-2, and ADRV9009 devices. During switching (or turning enable signals on/off), the stream file is invoked, which governs how fast the channels can transition between different states. This stream file is needed to ensure fast and smooth transitions between the various radio states. Note that the stream file is automatically generated by the GUI based on configuration options and is not readable, nor modifiable, by the user. The goal of this section is to help the user to get an idea of the switching times needed on the baseband

ELEMENTAL TIMES FOR THE STREAM

During each state transition, the stream executes a series of commands. The time needed to execute each of these commands can be used to calculate the total time required for the transition to complete. This section discusses the time needed for the execution of each command.

The typical Arm clock ranges from 153.6 MHz to 245.76 MHz for any profile. Taking the worst case (153.6 MHz): 1 Arm clock cycle = 6.51 ns. For x Arm clock cycles, the wait command takes approximately (number of clock cycles) × 6.51 ns.

The wait command dominates the total time required for switching between different the radio states.

The switching time calculations in this section focus only on the wait times in each stream execution. All minimum switching times are discussed in Arm clock cycles.

MINIMUM SWITCHING TIMES FOR THE ADRV9008-1

For the following calculations, the buffer time, t_{BUFFER}, accommodates for the extra processing time needed to execute the internal writes done by the stream. The typical t_{BUFFER} value ranges from 40 ns to 200 ns. The user can select 200 ns as the value for t_{BUFFER} to accommodate for the worst case.

For the ADRV9008-1, the only possible switching transitions are from receiver high to receiver low and from receiver low to receiver high.

Receiver High to Receiver Low for the ADRV9008-1

For this case, the receiver low stream is executed. Calculate the time taken to execute the receiver low stream ($t_{RX\_LOW\_SINGLE}$) for a single channel with the following example equation:

$$t_{RX\_LOW\_SINGLE} = 40 \text{ Arm clock cycles}$$

For two channels, the $t_{RX\_LOW\_SINGLE}$ can be calculated with the following example equation:

$$2 \times t_{RX\_LOW\_SINGLE} = 80 \text{ Arm clock cycles} + t_{BUFFER}$$

Receiver Low to Receiver High for the ADRV9008-1

For this case, the receiver high stream is executed. The time taken to execute the receiver high stream ($t_{RX\_HIGH\_SINGLE}$) for a single channel can be calculated with the following example equation:

$$t_{RX\_HIGH\_SINGLE} = 258 \text{ Arm clock cycles}$$

For two channels, the $t_{RX\_HIGH\_SINGLE}$ can be calculated with the following example equation:

$$2 \times t_{RX\_HIGH\_SINGLE} = 516 \text{ Arm clock cycles} + t_{BUFFER}$$

Figure 151 shows the minimum time that elapses between the receiver enable signal going high and the receiver data being valid on the datapath, and shows the minimum time after which the receiver data becomes invalid when the receiver enable signal goes low.

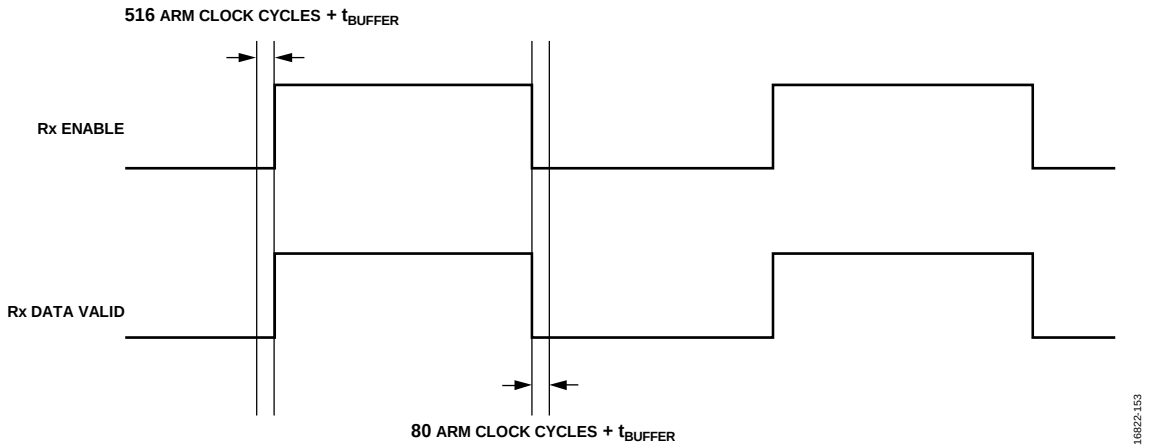

Figure 151. Minimum Switching Times for the ADRV9008-1 (Dual-Channel Mode)

MINIMUM SWITCHING TIMES FOR THE ADRV9008-2

For the ADRV9008-2, the possible switching transitions are from transmitter low to transmitter high, transmitter high to transmitter low, observation receiver low to observation receiver high, and observation receiver high to observation receiver low.

Transmitter Low to Transmitter High for the ADRV9008-2

For this case, the transmitter low stream is executed. The time taken to execute the transmitter low stream ($t_{TX\_LOW\_SINGLE}$) for a single channel can be calculated with the following example equation:

$$t_{TX\_LOW\_SINGLE} = (48 + 80 + 12) \text{ Arm clock cycles} = 140 \text{ Arm clock cycles}$$

For two channels, the $t_{TX\_LOW\_SINGLE}$ can be calculated with the following example equation:

$$2 \times t_{TX\_LOW\_SINGLE} = 280 \text{ Arm clock cycles} + t_{BUFFER}$$

Transmitter High to Transmitter Low for the ADRV9008-2

For this case, the transmitter high stream is executed. The time taken to execute the transmitter high stream ($t_{TX\_HIGH\_SINGLE}$) for a single channel can be calculated with the following example equation:

$$t_{TX\_HIGH\_SINGLE} = (80 + 8 + 80) \text{ Arm clock cycles} = 168 \text{ Arm clock cycles}$$

For two channels, the $t_{TX\_HIGH\_SINGLE}$ can be calculated with the following example equation:

$$2 \times t_{TX\_HIGH\_SINGLE} = 336 \text{ Arm clock cycles} + t_{BUFFER}$$

Figure 152 represents the minimum time between the transmitter enable going high and the transmitter data being valid on the datapath, and shows the minimum time after which the transmitter data becomes invalid when the transmitter enable goes low.

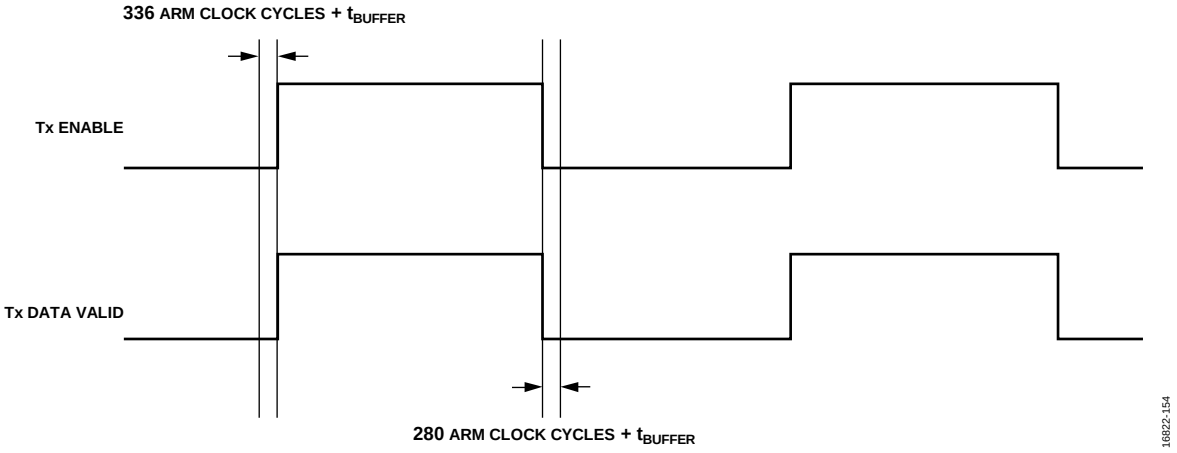

Figure 152. Minimum Switching Times for the ADRV9008-2 (Transmitter Only, Dual-Channel Mode)

Observation Receiver High to Observation Receiver Low for the ADRV9008-2

For this case, the observation receiver low stream is executed. The time taken to execute the observation receiver low ($t_{ORX\_LOW\_SINGLE}$) for a single channel can be calculated from the following example equation:

$$t_{ORX\_LOW\_SINGLE} = 64 \text{ Arm clock cycles} + t_{BUFFER}$$

Observation Receiver Low to Observation Receiver High for the ADRV9008-2

For this case, observation receiver high stream is executed. The time taken to execute the observation receiver high stream ($t_{ORX\_HIGH\_SINGLE}$) for a single channel can be calculated with the following example equation:

$$t_{ORX\_HIGH\_SINGLE} = (128 + 8 + 2) \text{ Arm clock cycles} + 2\ \mu s = 138 \text{ Arm clock cycles} + 2\ \mu s + t_{BUFFER}$$

Figure 153 shows the minimum time between the observation receiver enable going high and the observation receiver data being valid on the datapath, and shows the minimum time after which the observation receiver data becomes invalid when the observation receiver enable goes low.

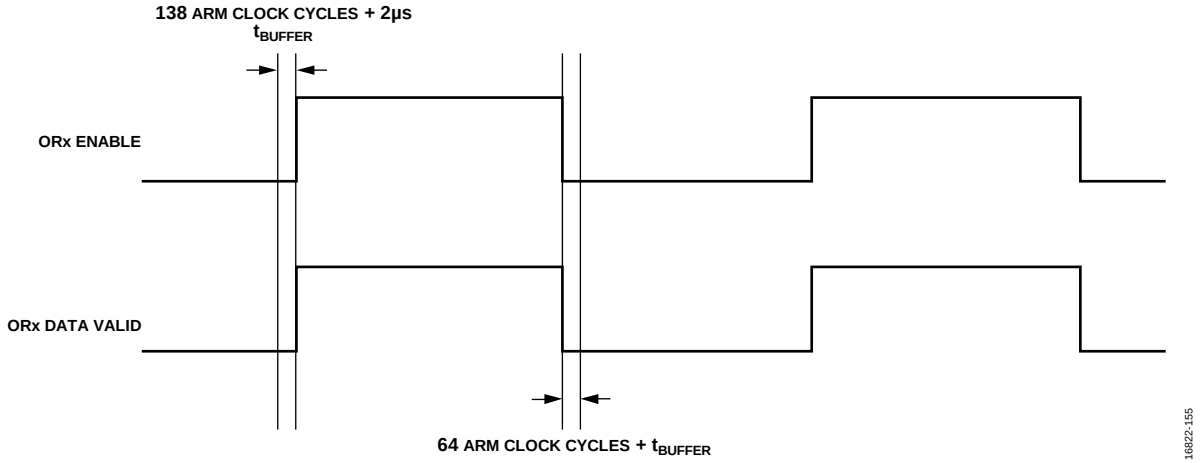

Figure 153. Minimum Switching Times for the ADRV9008-2 (Observation Receiver Only, Single-Channel)

MINIMUM SWITCHING TIMES FOR THE ADRV9009

For the ADRV9009, the possible transitions are receiver to transmitter/observation receiver to receiver, receiver to transmitter to receiver, and receiver to observation receiver to receiver.

Receiver to Transmitter/Observation Receiver to Receiver for the ADRV9009

This switching time case is broken down into two subcases: receiver to transmitter/observation receiver and transmitter/observation receiver to receiver.

Receiver to Transmitter/Observation Receiver Subcase for the ADRV9009

For this subcase, the streams execute a receiver low stream, a transmitter high stream, and an observation receiver high stream.

To calculate the minimum switching times for each of these streams for a single channel, use the following example equations:

$$t_{RX\_LOW\_SINGLE} = 40 \text{ Arm clock cycles}$$

$$t_{TX\_HIGH\_SINGLE} = 168 \text{ Arm clock cycles}$$

$$t_{ORX\_HIGH\_SINGLE} = 138 \text{ Arm clock cycles} + 2\ \mu s$$

To calculate the minimum switching times for each of these streams for two channels, use the following example equation:

$$2 \times (t_{RX\_LOW\_SINGLE} + t_{TX\_HIGH\_SINGLE}) + t_{ORX\_HIGH\_SINGLE} = 554 \text{ Arm clock cycles} + 2\ \mu s + t_{BUFFER}$$

It is important to note that the stream execution sequence for this case is Receiver 1 low stream to Receiver 2 low stream to Transmitter 1 high stream to Transmitter 2 high stream Observation Receiver 1/Observation Receiver 2 high stream.

Transmitter/Observation Receiver to Receiver Subcase for the ADRV9009

For this subcase, the streams execute a transmitter low stream, an observation receiver low stream, and a receiver high stream.

To calculate the minimum switching time for each of these streams for a single channel, use the following example equations:

$t_{TX\_LOW\_SINGLE}$ = 140 Arm clock cycles

$t_{ORX\_LOW\_SINGLE}$ = 64 Arm clock cycles

$t_{RX\_HIGH\_SINGLE}$ = 258 Arm clock cycles

To calculate the minimum switching times tor two channels, use the following example equation:

$2 \times (t_{TX\_LOW\_SINGLE} + t_{RX\_HIGH\_SINGLE}) + t_{ORX\_LOW\_SINGLE}$ = 860 Arm clock cycles + t_{BUFFER}

It is important to note that the stream execution sequence for this case is Transmitter 1 low stream to Transmitter 2 low stream to Observation Receiver 1/Observation Receiver 2 low stream to Receiver 1 high stream to Receiver 2 high stream.

Figure 154 shows the aggregation of these subcases.

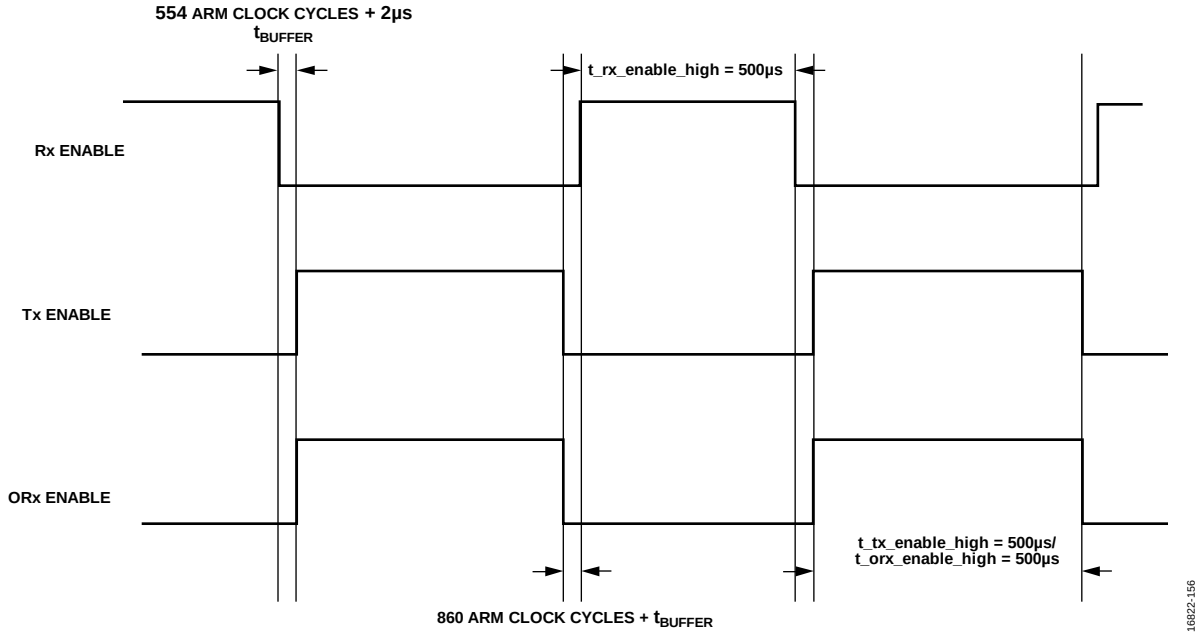

Figure 154. Minimum Switching Times for the ADRV9009 (Dual-Channel Receiver/Transmitter and Single-Channel Observation Receiver)

Figure 154 also shows the minimum time that the receiver/transmitter and observation receiver enable signals must stay high (approximately 500 µs) to successfully run tracking calibrations. See the System Considerations for Arm Calibrations section for details.

Receiver to Transmitter to Receiver for the ADRV9009

This switching time case is broken down into two subcases: receiver to transmitter and transmitter to receiver.

Receiver to Transmitter Subcase for the ADRV9009

For this subcase, the streams execute a receiver low stream and atransmitter high stream.

To calculate the minimum switching time for each of these streams for a single channel, use the following equations:

$t_{RX\_LOW\_SINGLE}$ = 40 Arm clock cycles

$t_{TX\_HIGH\_SINGLE}$ = 168 Arm clock cycles

To calculate the minimum time switching for two channels, use the following example equation:

$2 \times (t_{RX\_LOW\_SINGLE} + t_{TX\_HIGH\_SINGLE})$ = 416 Arm clock cycles + t_{BUFFER}

It is important to note that the stream execution sequence for this case is Receiver 1 low stream to Receiver 2 low stream to Transmitter 2 high stream to Transmitter 2 high stream.

Transmitter to Receiver Subcase for the ADRV9009

For this subcase, the streams execute a transmitter low stream and a receiver high stream.

To calculate the minimum switching time for each of these streams for a single channel, use the following equations:

$t_{TX\_LOW\_SINGLE}$ = 140 Arm clock cycles

$t_{RX\_HIGH\_SINGLE}$ = 258 Arm clock cycles

To calculate the minimum switching time for two channels, use the following example equation:

$2 \times (t_{TX\_LOW\_SINGLE} + t_{RX\_HIGH\_SINGLE})$ = 796 Arm clock cycles + t_{BUFFER}

It is important to note that the stream execution sequence for this case is Transmitter 1 low stream to Transmitter 2 low stream to Receiver 1 high stream to Receiver 2 high stream.

Figure 155 aggregates both the of these subcases.

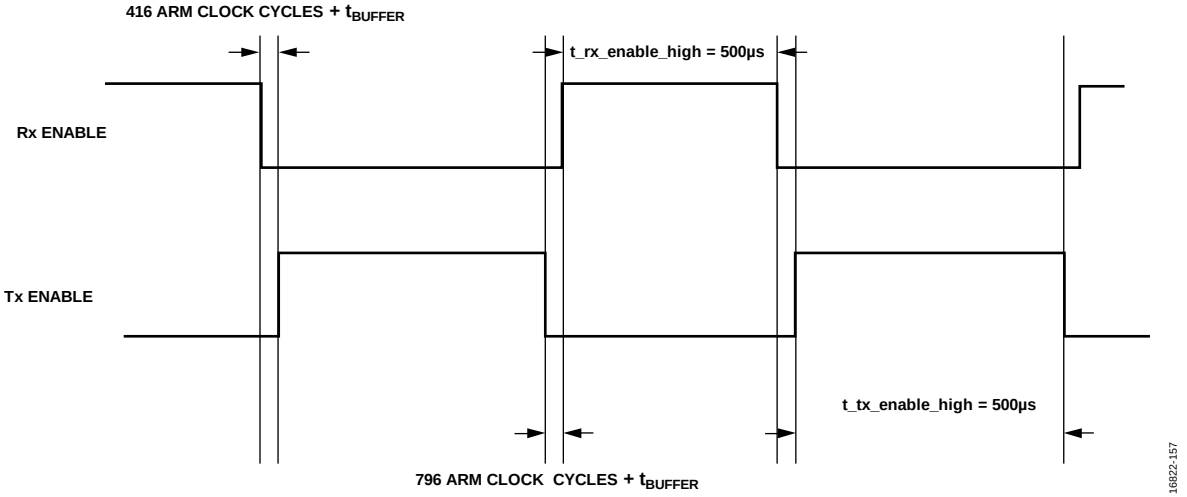

Figure 155. Minimum Switching TIMES for the ADRV9009 (Dual-Channel Receiver/Transmitter)

Figure 155 also shows the minimum time that the receiver and transmitter enable signals must stay high (approximately 500 µs) to successfully run tracking calibrations. See the System Considerations for Arm Calibrations section for details.

Receiver to Observation Receiver to Receiver for the ADRV9009

This switching time case is broken down into two subcases: receiver to observation receiver, and observation receiver to receiver.

Receiver to Observation Receiver Subcase for the ADRV9009

For this subcase, the streams execute a receiver low stream and an observation receiver high stream.

To calculate the minimum switching time for each of these streams for a single channel, use the following example equations:

$t_{RX\_LOW\_SINGLE}$ = 40 Arm clock cycles

$t_{ORX\_HIGH\_SINGLE}$ = 138 Arm clock cycles + 2 µs

To calculate the minimum switching time for two channels, use the following example equation:

$2 \times (t_{RX\_LOW\_SINGLE}) + t_{ORX\_HIGH\_SINGLE}$ = 218 Arm clock cycles + 2 µs + t_{BUFFER}

It is important to note that the stream execution sequence for this case is Receiver 1 low stream to Receiver 2 low stream to Observation Receiver 1/Observation Receiver 2 high stream.

Observation Receiver to Receiver Subcase for the ADRV9009

For this subcase, the streams execute an observation receiver low stream and a receiver high stream.

To calculate the minimum switching time for each of these streams for single channel, use the following example equations:

$t_{ORX\_LOW\_SINGLE}$ = 64 Arm clock cycles

$t_{RX\_HIGH\_SINGLE}$ = 258 Arm clock cycles

To calculate the minimum switching times for two channels, use the following example equation:

$$2 \times (t_{RX\_HIGH\_SINGLE}) + t_{ORX\_LOW\_SINGLE} = \text{Arm clock cycles} + t_{BUFFER}$$

It is important to note that the stream execution sequence for this case is Observation Receiver 1/Observation Receiver 2 low to Receiver 1 high stream to Receiver 2 high stream.

Figure 156 aggregates these subcases.

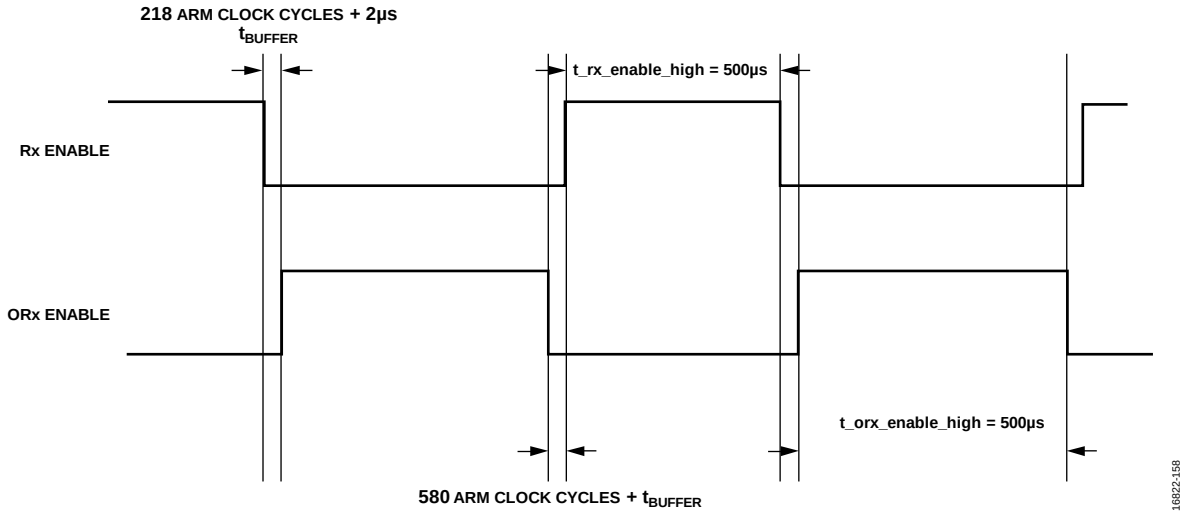

Figure 156. Minimum Switching Times for ADRV9009 (Dual-Channel Receiver and Single-Channel Observation Receiver)

Figure 156 also shows the minimum time that the receiver and observation receiver enable signals must stay high (approximately 500 µs) to successfully run tracking calibrations. See the System Considerations for Arm Calibrations section for details.