Application Report
SBAA288A–July 2018–Revised January 2019

TM4C1294 interface to ADS7142 software library

Peggy Liska and William Santos *Precision Analog-to-Digital Converters*

ABSTRACT

This application report describes how to communicate with the ADS7142 using the TM4C1294NCPDT as the host device. The communication is performed via inter-integrated circuit (I2C) protocol and the ADS7142 is enabled to operate in each of its functional modes. The accompanying software contains a function library allowing quick prototyping of ADS7142 setup and control. This framework is designed with the intention of minimal function modification required to operate the ADS7142 as a sensor monitor. The functional modes of the ADS7142 include manual mode, autonomous mode, and high precision mode. Additional features of the ADS7142 include sampling speed control, oscillator selection, and selection of input channel type (single-ended or pseudo-differential).

Contents

List of Figures

List of Tables

Trademarks

1 Introduction

Sensor monitoring technology is an integral piece of systems that seek to gather copious amounts of data and process that data in a meaningful way. Because of the differences in sensor monitoring techniques, the system designer must choose the best sensor monitor for a given application. In this application report the ADS7142 SAR ADC is paired with a TM4C1294NCPDT microcontroller (MCU) to provide accurate sampling data after the device is configured to operate in one of its functional modes.

2 Hardware

The TM4C1294NCPDT is a high performance MCU solution that combines complex integration with a suite of features. The device contains eight UART interfaces and ten I^2C modules with several clocking options. The ARM® Cortex®-M4F-Based MCU TM4C1294 Connected LaunchPad™ Evaluation Kit breaks out two of the MCU I^2C modules for use with standardized BoosterPack boards. Figure 1 shows a TM4C1294 LaunchPad™ development kit with connectors already in place.

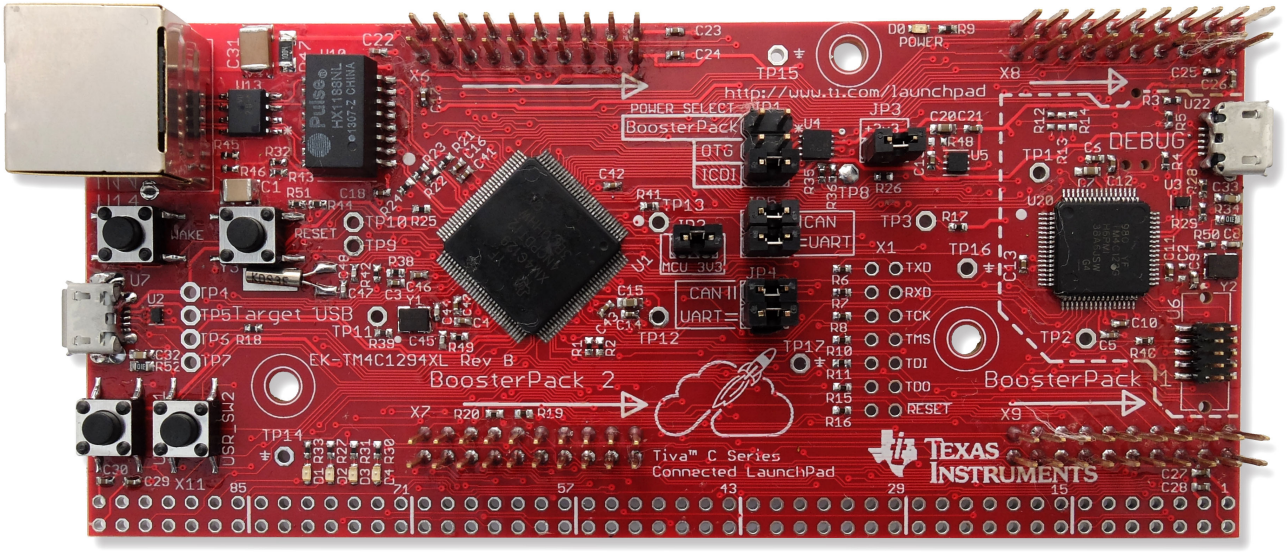


Figure 1. TM4C1294 Connected LaunchPad

The ADS7142 nanopower, dual-channel, programmable sensor monitor BoosterPack™ plug-in module (see Figure 2) hosts features specific to sensor monitoring. This module is ideally suited for a sensor node system architecture where one or two sensor outputs can be monitored as part of a central gateway, with the host MCU being the router to the cloud in an internet of things (IoT) application. The code examples provided in this application report can be used for software development on MCUs or digital processors other than the TM4C1294, allowing the designer maximum flexibility in choosing the best digital host for the application.

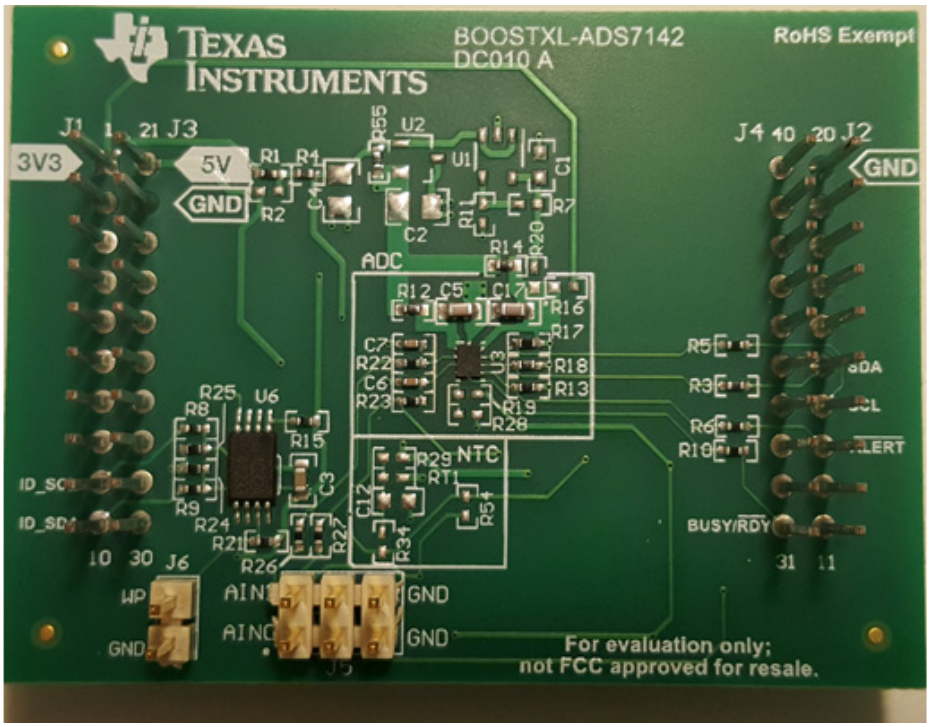


Figure 2. ADS7142 BoosterPack™

As shown in Figure 3, the ADS7142 BoosterPack™ connects to the TM4C1294 connected LaunchPad™.

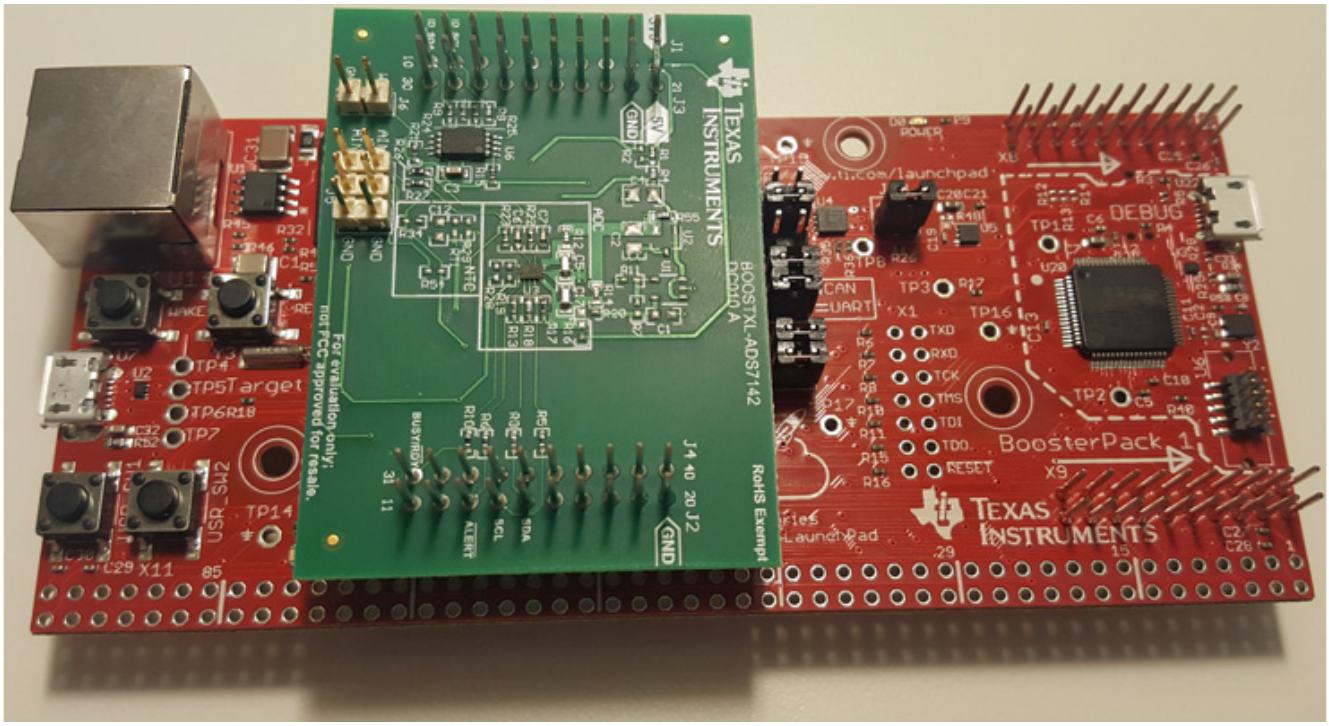


Figure 3. ADS7142 BoosterPack™ Connection to TM4C1294 LaunchPad™

3 Software

3.1 Header Files

To aid in interfacing the ADS7142 device, TI has produced a source code library that eliminates the need to write low-level interface functions that configure the core of the TM4C1294NCPDT. This library provides a boost in the development of a TM4C/ADS7142-based product, saving time and quickly progressing an applications project. This library is designed to be used with any TM4C device after appropriate software changes. An I^2C master can be implemented using one of ten modules within the TM4C1294. The library provides a number of header files that define digital masks to access each piece of hardware on the device.

The library has been implemented so that digital masks are interchangeable. If a piece of hardware must be accessed, simply find the correct global variable name for that mask and place that name as an argument in the desired function calls. The software designer must then be thorough in replacing all instances of these masks to reflect the desired access to a certain piece of hardware. There are header files specific to each of the hardware components.

Table 1 describes the hardware definition header files.

Table 1. Hardware Definition Files

Filename	Description
hw_i2c.h	This file contains macros used specifically to access the TM4C master and slave hardware, including register locations and commonly used masks for use with these registers.
hw_memmap.h	This file contains definitions specific to the TM4C device. Primarily, the port and pin masks used to access all of the device peripherals. Definitions for UART0-UART7, I2C0-I2C9, TIMER, and GPIO are included among others.
hw_types.h	This file contains helper macros for determining silicon revisions for TM4C devices. These macros are used by the driver library at *run-time* to create necessary conditional blocks that allow a single version of the Driverlib binary code to support multiple (all) Tiva silicon revisions.
hw_ints.h	This file contains the interrupt assignments on TIVA C series devices. Class interrupts and fault interrupts are included for user enable in firmware.
hw_gpio.h	This file contains the definitions and macros for the GPIO hardware. A set of register offset masks define the GPIO capabilities and features.
ADS7142RegisterMap.h	This file contains all of the register masks and value masks to interface and configure the ADS7142 using the host MCU. This file contains all of the command opcodes and function declarations used to develop the device functional modes main routines.

Hardware definition files are important for defining hardware attributes and mapping them to callable system variables in software. In order to enable communication with the target MCU, driver files must be implemented. Table 2 describes the driver library files to be used for communication with the TM4C1294.

Table 2. Driver Library Files

Filename	Description
gpio.h	This file contains values that can be passed to application programming interfaces (APIs) that configure the GPIO modules.
i2c.h	This file contains important I^2C Master Command masks used to perform I^2C transmission and receive. It also contains macros and APIs for advanced I^2C module features on the host MCU.
pin_map.h	This driver file contains the mapping of peripherals to pins for all the TIVA C series parts.
sysctl.h	This file contains system control directives for selection of clocks/pll and crystal frequencies as well as values to be used with control APIs. API prototypes for system control are also included in this file.
i2c.c	Driver file for the I^2C block. Contains low-level functions that allow the user to configure the I^2C modules on the core of the host device.

3.2 ADS7142 Device Functional Modes Overview

The ADS7142 contains three primary modes: manual mode, autonomous mode, and high precision mode. Each mode requires a set of register configurations before the device can operate in that mode. Table 3 describes the main code routines that place the device in each of its functional modes.

Table 3. ADS7142 Device Functional Modes Main Routines

Filename	Description
TM4C_ADS7142_Functions.c	Uses the low-level functions in i2c.c to configure the host TM4C MCU and develop the ADS7142 register configuration functions. The function prototypes in the ADS7142RegisterMap.h file are explicitly declared in this file.
ADS7142_ManualMode_CH0Scan.c	These routines configure the ADS7142 into Manual Mode (I^2C command mode) in both single channel and dual-channel configuration. In manual mode, 12-bit conversions are clocked out continuously.
ADS7142_ManualMode_CH1Scan.c	
ADS7142_ManualMode_AutoSequencing_CH0_CH1_Scan.c	
ADS7142_AutonomousMode_PreAlert_CH0Scan.c	These routines configure the ADS7142 into autonomous pre-alert mode in both single channel and dual-channel configuration. In pre-alert mode, the ADS7142 stores the sixteen conversions prior to the activation of the alert in a data buffer.
ADS7142_AutonomousMode_PreAlert_CH1Scan.c	
ADS7142_AutonomousMode_AutoSequencing_PreAlert_CH0_CH1Scan.c	
ADS7142_AutonomousMode_PostAlert_CH0Scan.c	These routines configure the ADS7142 into autonomous post-alert mode in both single-channel and dual-channel configuration. In post-alert mode, the ADS7142 stores the next 16 conversions after the alert is active.
ADS7142_AutonomousMode_PostAlert_CH1Scan.c	
ADS7142_AutonomousMode_AutoSequencing_PostAlert_CH0_CH1Scan.c	
ADS7142_AutonomousMode_StartBurst_CH0Scan.c	These routines configure the ADS7142 into autonomous start burst mode in both single channel and dual-channel configuration. In start burst mode, the device starts filling the data buffer upon starting channel sequencing: conversions stop once the data buffer is filled.
ADS7142_AutonomousMode_StartBurst_CH1Scan.c	
ADS7142_AutonomousMode_AutoSequencing_StartBurst_CH0_CH1Scan.c	
ADS7142_AutonomousMode_StopBurst_CH0Scan.c	These routines configure the ADS7142 into autonomous stop burst mode in both single-channel and dual-channel configuration. In stop burst mode, the device continues to fill the data buffer until the channel sequencing is aborted. Data buffer entries are continually overwritten until the sequence is aborted.
ADS7142_AutonomousMode_StopBurst_CH1Scan.c	
ADS7142_AutonomousMode_AutoSequencing_StopBurst_CH0_CH1Scan.c	
ADS7142_HighPrecisionMode_CH0Scan.c	These routines configure the ADS7142 into high precision mode in both single-channel and dual-channel configuration. In high precision mode, 16 12-bit conversions are stored in an accumulator. When the accumulator is filled, the 12-bit conversions are added to form one 16-bit reading.
ADS7142_HighPrecisionMode_CH1Scan.c	
ADS7142_HighPrecisionMode_AutoSequencing_CH0_CH1_Scan.c	

3.3 Software Functions

The main routines that place the ADS7142 into one of its functional modes call a set of functions that configure the registers of the device. Each function excluding *TM4C1294Init()* , *ADS7142Calibrate(), ADS7142Reset(),* and *ADS7142HighSpeedEnable()* is developed from pseudocode software flows in the TM4C1294 datasheet. Table 4 provides a brief description of these functions.

Table 4. Functions Used in Main

Function Name	Description
int TM4C1294Init (uint8_t bFast)	Configures the TM4C1294 as the target device for the compiler, sets the clocking of the device, and enables peripherals to be used in this application. The bFast variable sets the I^2C frequency (100kHz or 400kHz).
int ADS7142SingleRegisterWrite(uint8_t RegisterAddress, uint8_RegisterData)	Writes*RegisterData* to *RegisterAddress* in order to configure the ADS7142. The ADS7142 slave address is set by hardware on the BoosterPack™ and is defined in *ADS7142RegisterMap.h*.
*int ADS7142SingleRegisterRead(uint8_t RegisterAddress, uint32_t *read)*	Reads the data at RegisterAddress and places this data into the read variable for local use.
void TM4C1294_ArbitrationLost_ErrorService(void)	This function performs an I^2C SDA bus clear in the case that the host MCU loses arbitration. Arbitration loss in a single-master system is usually due to the slave device holding the bus at some undesirable state while the host attempts to perform some I^2C function. This function configures SCL as a GPIO and toggles SCL nine times, after which the slave releases the bus
int ADS7142Calibrate(void)	This function is called right after *TM4C1294Init()* to abort the present conversion sequence and calibrate any offset error out of the device.
int ADS7142Reset(void)	This function gives the user the option of a software reset of the ADS7142. This function is not called in the functional modes firmware because the device calibrates its own offset and I^2C address upon power-up.
int ADS7142HighSpeedEnable(uint8_t HighSpeedMask)	This function enables the host MCU and ADS7142 to communicate at high speed I^2C frequencies (1.7 MHz to 3.4 MHz).
int ADS7142DataRead_count(uint64_t SampleCount)	This function takes a sample count as a user input: the ADS7142 only samples this specified number of samples from the desired input channels. This function is used in high precision mode and autonomous mode to count 16 12-bit conversions before placing them in either the data buffer or channel accumulators.
int ADS7142DataRead_infinite(void)	Similar to *ADS7142DataRead_count() ,* this function samples data from the ADS7142 input channels. However, this function infinitely provides sample data as opposed to a specified number of data samples.
int ADS7142DataRead_autonomous(void)	This sampling function is specific to the pre-alert and post-alert autonomous modes of the ADS7142. Upon sampling, this function compares the digitized value of the sample to the digital window comparator settings for the high and low threshold alerts.

The development of the functions *ADS7142SingleRegisterWrite(), ADS7142SingleRegisterRead(), TM4C1294_ArbitrationLost_ErrorService,* and *ADS7142DataRead_infinite()* are crucial to effectively placing the ADS7142 in the user's desired functional mode. *ADS7142SingleDataRead_count()* and *ADS7142DataRead_autonomous()* are derivatives of *ADS7142DataRead_infinite().* Within each I^2C module of the master TM4C1294 there is an I2CMasterSlaveAddress (I2CMSA) register that contains the slave address of the device that the host must communicate with over I^2C. A low-level function named *MasterSlaveAddrSet()* in *i2c.c* is called to configure this address. When the slave address is properly configured, the first data byte can be put into the I2CMasterDataRegister (I2CMDR) via a call of the function *MasterDataPut()*. The I2CMasterDataRegister can be used to either write to or read from the slave device. A master command is then provided to the I2CMasterControlStatus (I2CMCS) register via function call *I2CMasterControl()* to send the data byte. A specific master command is used for multi-byte transmission.

The TM4C1294NCPDT datasheet provides a software flow that is used to develop the function *ADS7142SingleRegisterWrite()*. Figure 4 shows this flow.

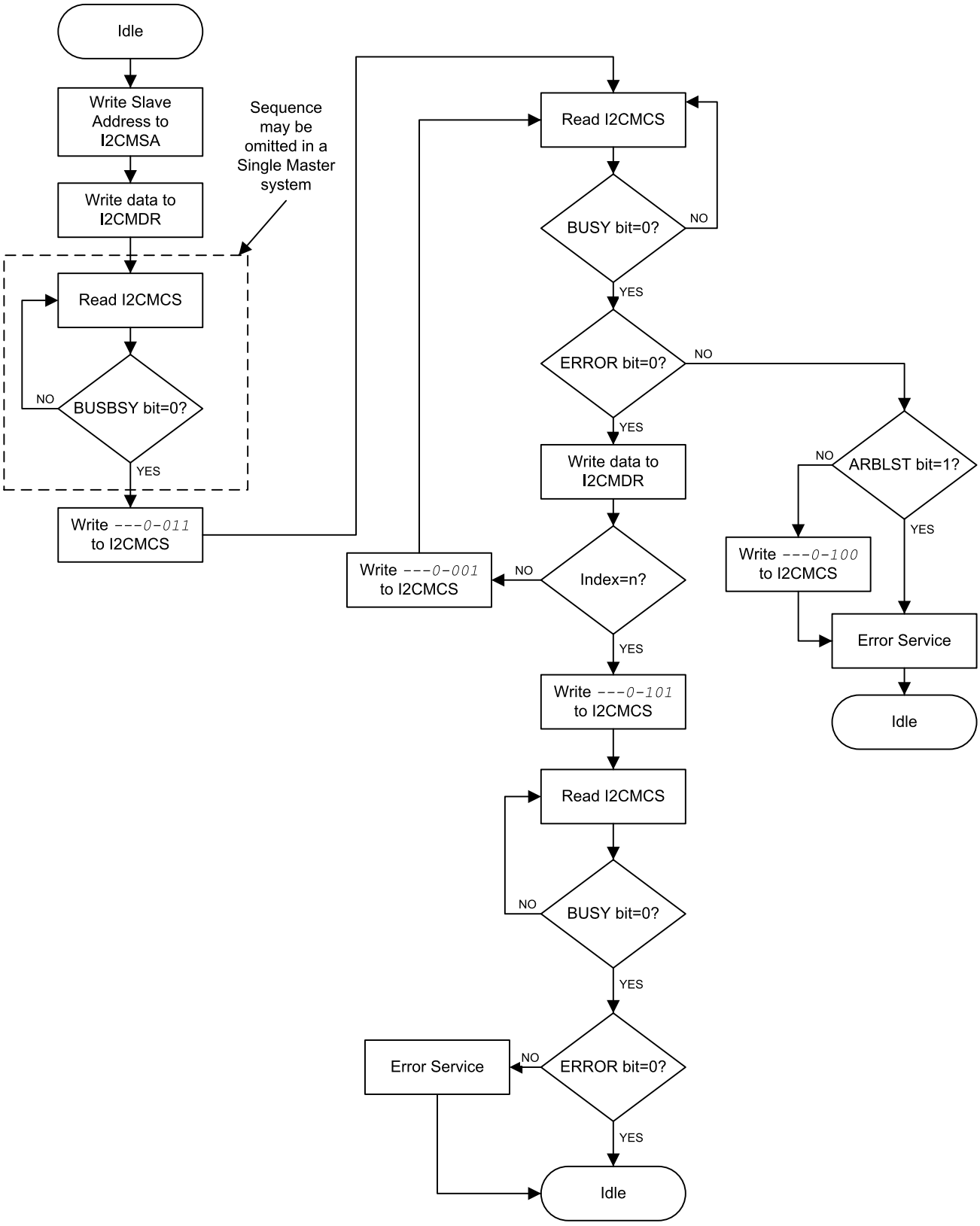


Figure 4. TM4C1294NCPDT Multi-Byte Transmit Flowchart

Figure 5 shows that to write to a single register in the ADS7142 device, the I²C master (TM4C1294) must transmit four bytes over I²C.

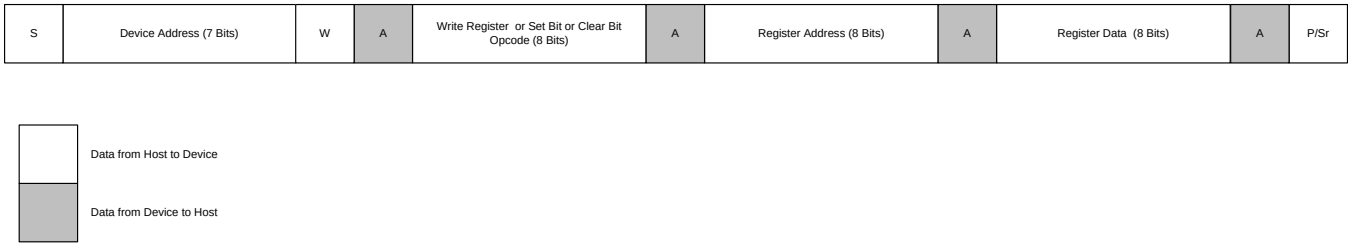


Figure 5. Writing a Single Register in the ADS7142 Over I²C

The *ADS7142SingleRegisterWrite()* function code is the following:

```
int
ADS7142SingleRegisterWrite (uint8_t RegisterAddress, uint8_t RegisterData)
{

    //
      //ADS7142SingleRegisterWrite writes data to an ADS7142 register address
      //ADS7142registermap.h contains all the device datasheet register map
      //register addresses and register values for configuration.
    //

    //
      // Tell the master module what address it will place on the bus when
      // communicating with the slave.  Set the address to ADS7142_I2C_ADDRESS
      // (as set in the slave module).  The receive parameter is set to false
      // which indicates the I2C Master is initiating a write to the slave.  If
      // the receive parameter is true, that would indicate that the I2C Master
      // is initiating reads from the slave.
    //

    I2CMasterSlaveAddrSet(I2C8_BASE, ADS7142_I2C_ADDRESS, false);

    //Place the first byte to be transmitted into the I2CMDR Register of the TM4C1294
    //The first byte to be transmitted following the SLAVE Address is the Single Register Write
opcode
    I2CMasterDataPut(I2C8_BASE, SINGLE_REG_WRITE);

    //Check the I2C Bus to ensure it is not busy (Read I2CMCS)
    //while(I2CMasterBusBusy(I2C8_BASE));

    //I2C Master Command for the Start BURST Send of 3 bytes
    I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_SEND_START);

    //Implement delay
    SysCtlDelay(100);

    //Read I2CMCS
    //Wait for the I2CMaster to finish transmitting before moving to next byte
    while(I2CMasterBusy(I2C8_BASE));

    //Check for errors in the I2C8 Module
    while (I2CMasterErr(I2C8_BASE))

    //Error branching
    {
       //Check for I2C Bus arbitration loss error condition
       if(I2CMasterErr(I2C8_BASE) == I2C_MASTER_ERR_ARB_LOST)

       {
```

```
            //Error Service
            TM4C1294_ArbitrationLost_ErrorService();

            //Return the error status
            return -1;

         }

         //Write I2C Master Command for error stop if the error
         //is not due to i2c bus arbitration loss
         else I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_SEND_ERROR_STOP);

     }

     //Place the next byte into I2CMDR, which is the ADS7142 register address for the desired data
write
     I2CMasterDataPut(I2C8_BASE, RegisterAddress);

     //I2C Master Command for continued BURST send of the next byte
     I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_SEND_CONT);

     //Implement Delay
     SysCtlDelay(100);

     //Read I2CMCS
     //Wait for the I2CMaster to finish transmitting before moving to next byte
     while(I2CMasterBusy(I2C8_BASE));

     //Check for errors in the I2C8 Module
     while (I2CMasterErr(I2C8_BASE))

     //Error branching
     {

         //Check for I2C Bus arbitration loss error condition
         if(I2CMasterErr(I2C8_BASE) == I2C_MASTER_ERR_ARB_LOST)

         {

            //Error Service for loss of bus arbitration
            TM4C1294_ArbitrationLost_ErrorService();

            //Return the error status
            return -1;

         }

         //Write I2C Master Command for receive error stop if the error
         //is not due to i2c bus arbitration loss
         else I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_SEND_ERROR_STOP);

     }

     //Place the final byte into I2CMDR, which is the register data to be placed in the desired
register address
     I2CMasterDataPut(I2C8_BASE, RegisterData);

     //I2C Master Command for the finished BURST send of the data
     I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_SEND_FINISH);

     //Implement delay
     SysCtlDelay(100);

     //Read I2CMCS (I2C Master Control/Status)
     //Wait for the I2C Master to finish transmitting
     while(I2CMasterBusy(I2C8_BASE));
```

```
    //Check the error flag in the I2C8 Module
    while (I2CMasterErr(I2C8_BASE));

    //Return no errors
    return 0;
}
```

To read a single register from the ADS7142, the desired register address must first be set through a series of I^2C writes. The *ADS7142SingleRegisterRead()* function first performs the multi-byte transmit software flow in Figure 4 to set the register that will be read from the ADS7142. Figure 6 shows the required I^2C writes prior to register read.

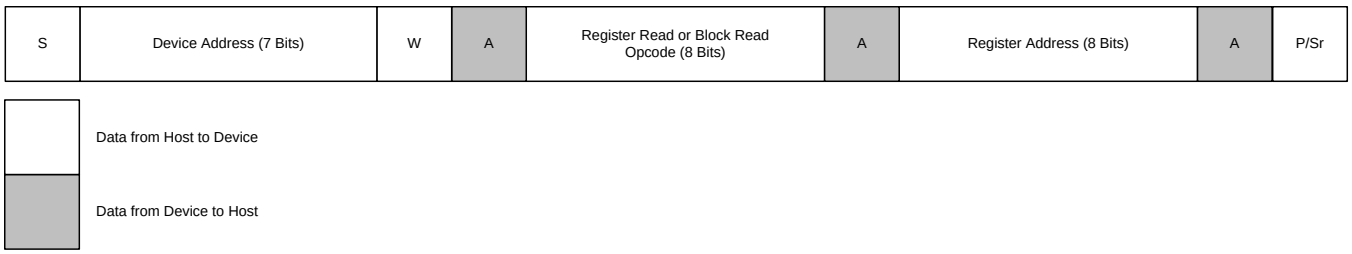


Figure 6. Setting Register Address for a Single Register Read From the ADS7142

When the required write operations are complete, the host MCU can read the register data. This operation is outlined as a master single byte receive software flow in the TM4C1294NCPDT datasheet. Figure 7 shows the TM4C1294NCPDT single byte receive software flow.

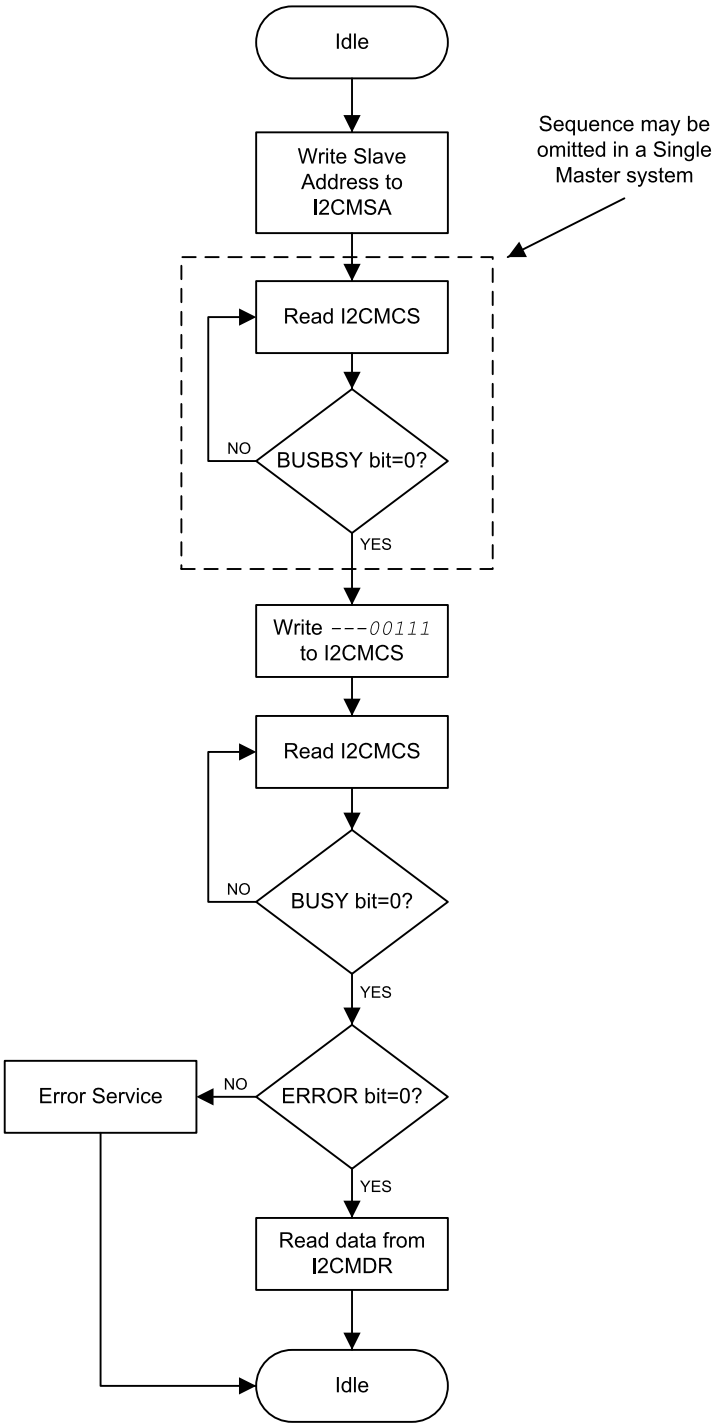


Figure 7. TM4C1294NCPDT Single-Byte Receive Flowchart

As Figure 8 shows, this software flow is also represented using I²C frames.

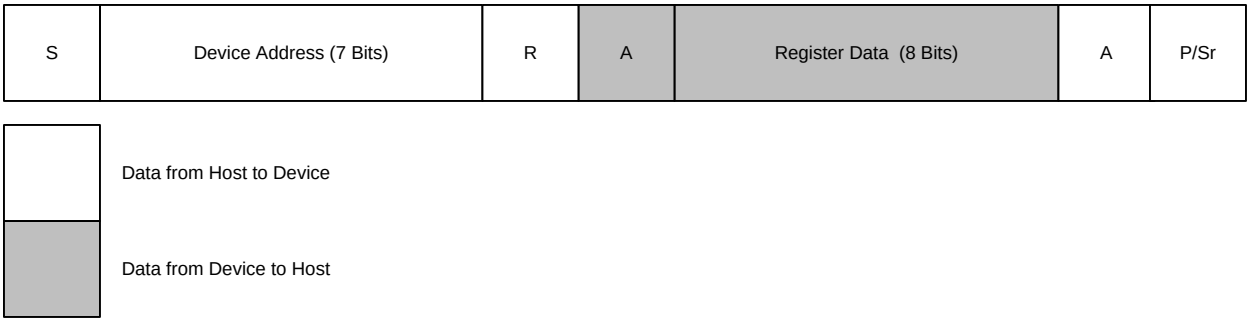


Figure 8. ADS7142 Single Register Read

The function code for the operations required to perform a single register read is the following:

```
int
ADS7142SingleRegisterRead(uint8_t RegisterAddress, uint32_t *read)
{
    //
      //ADS7142SingleRegisterRead reads data from a single register
      //in the ADS7142.
    //

    //
      // Tell the master module what address it will place on the bus when
      // communicating with the slave.  Set the address to ADS7142_I2C_ADDRESS
      // (as set in the slave module).  The receive parameter is set to false
      // which indicates the I2C Master is initiating a write to the slave.
      // To perform a register read, the Master must first transmit the desired I2C address for
communication.
      // Following the I2C address transmission, the single register read opcode will be
transmitted.
      // The receive parameter is then set to true for read of register contents.
    //


    I2CMasterSlaveAddrSet(I2C8_BASE, ADS7142_I2C_ADDRESS, false);

    //Place the Single Register Read opcode into the I2CMDR Register
    I2CMasterDataPut(I2C8_BASE, SINGLE_REG_READ);

    //Initiate the BURST Send of two data bytes
    I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_SEND_START);

    //Implement Delay
    SysCtlDelay(100);

    //Wait for the I2C Master to finish transmitting
    while(I2CMasterBusy(I2C8_BASE));

    //Check for errors in the I2C8 Module
    while (I2CMasterErr(I2C8_BASE))

    //Error branching
    {

       //Check for I2C Bus arbitration loss error condition
       if(I2CMasterErr(I2C8_BASE) == I2C_MASTER_ERR_ARB_LOST)

       {

           //Error service for loss of bus arbitration
```

```
            TM4C1294_ArbitrationLost_ErrorService();

            //Return the error status
            return -1;

         }

         //Write I2C Master Command for receive error stop if the error
         //is not due to i2c bus arbitration loss
         else I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_SEND_ERROR_STOP);

      }

      //Place the Register Address to be communicated with into the I2CMDR Register
      I2CMasterDataPut(I2C8_BASE,RegisterAddress);

      //I2C Master Command for finished Burst Send of the two bytes required for register read
      I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_SEND_FINISH);

      //Implement Delay
      SysCtlDelay(100);

      //Wait for the I2C Master to finish transmitting the data
      while(I2CMasterBusy(I2C8_BASE));

      //Check the error flag in the I2C8 Module
      while(I2CMasterErr(I2C8_BASE))

      {
      //Error service for address ACK or data ACK
      }

      //Set the receive parameter to true in order to receive data from the desired register address
      I2CMasterSlaveAddrSet(I2C8_BASE, ADS7142_I2C_ADDRESS, true);

      //Check the I2C Bus to ensure it is not busy
      while(I2CMasterBusBusy(I2C8_BASE));

      //I2C Master Command for the Single Byte Receive from the register address
      I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);

      //Implement Delay
      SysCtlDelay(100);

      //Wait for the I2C Master to finish transmitting the data
      while(I2CMasterBusy(I2C8_BASE));

      //Check the error flag in the I2C8 Module
      while(I2CMasterErr(I2C8_BASE))

      {
      //Error service for address ACK or data ACK
      }

      //Get the data placed into the I2CMDR Register from the ADS7142
      //Place data into the read flag that contains the data for local use
      read[0] = I2CMasterDataGet(I2C8_BASE);

      //Return no errors
      return 0;
   }
```

In order for conversion data to be read continuously from the ADS7142, the TM4C1294 must perform a multi-byte receive operation over I^2C. This software flow is the basis for the functions *ADS7142DataRead_infinite(), ADS7142DataRead_count(),* and *ADS7142DataRead_autonomous().* Figure 9 shows the pseudocode flow.

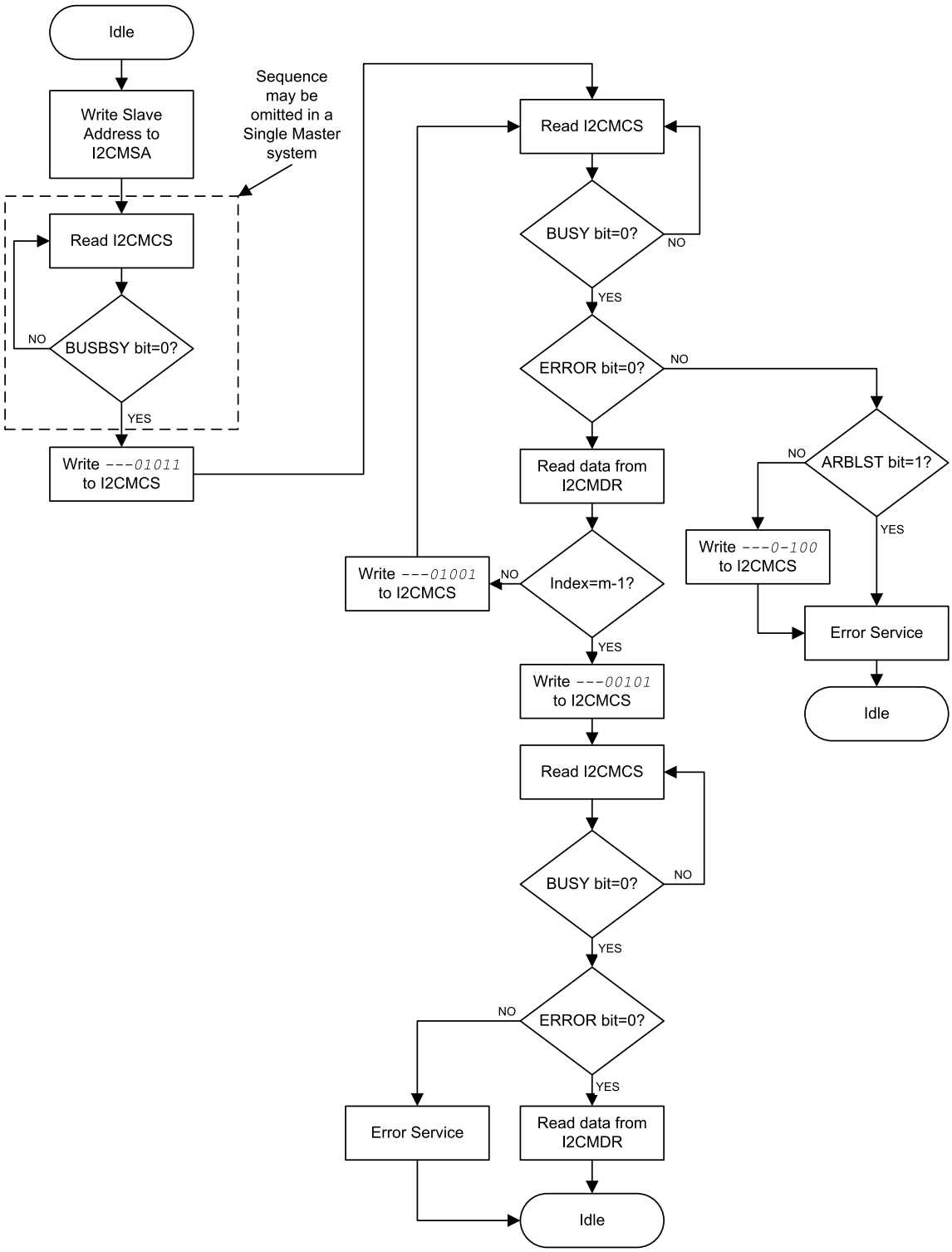


Figure 9. TM4C1294NCPDT Multi-Byte Receive Flowchart

Let's take a look at the development of *ADS7142DataRead_continuous()*. This function is only used in manual mode because in this mode, the ADS7142 is continuously outputting sample data from the desired input channels. Figure 10 shows the continuous read of sample bytes.

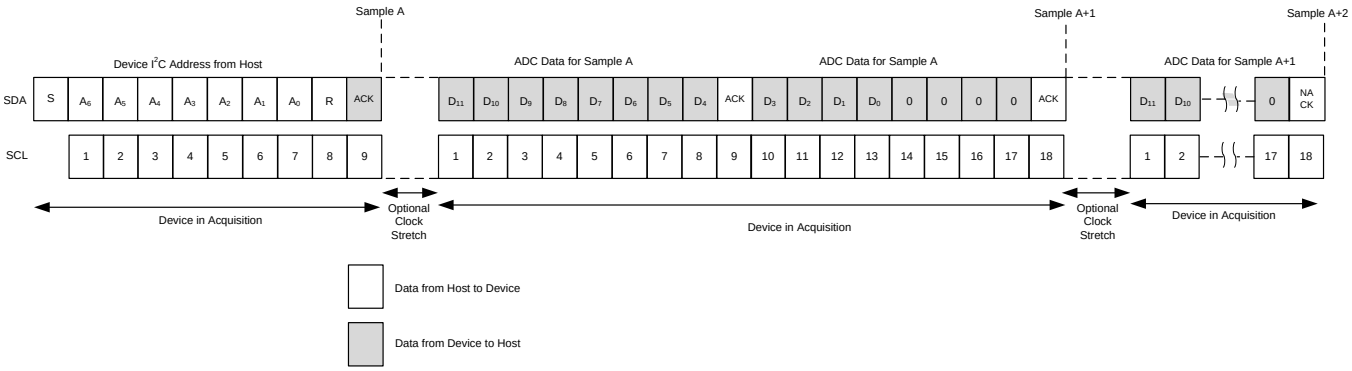


Figure 10. Reading ADS7142 Conversion Data in Manual Mode

The *ADS7142DataRead_continuous()* function code is the following:

```
int
ADS7142DataRead_continuous(void)
{

  //
    //ADS7142DataRead_continuous() continuously clocks out the data sampled by the ADS7142
    //once it is placed in manual mode.  The function provides
    //continuous SCL to clock out the data
  //

    //Provide Device Address and Read Bit to Start Conversions
    I2CMasterSlaveAddrSet(I2C8_BASE, ADS7142_I2C_ADDRESS, true);

    //Check the I2C Bus to ensure it is not busy
    //while(I2CMasterBusBusy(I2C8_BASE));

    //Write the Burst receive I2C Master Command to I2CMCS
    I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_RECEIVE_START);

    //Implement Delay
    SysCtlDelay(100);

    //Allow the Master to finish receiving the first byte
    while(I2CMasterBusy(I2C8_BASE));

    //Check for errors in the I2C8 Module
    while (I2CMasterErr(I2C8_BASE))

    //Error branching
    {


        //Check for I2C Bus arbitration loss error condition
        if (I2CMasterErr(I2C8_BASE) == I2C_MASTER_ERR_ARB_LOST)

        {

            //Error service for loss of bus arbitration
            TM4C1294_ArbitrationLost_ErrorService();

            //Return the error status
            return -1;

        }
```

```
    //Write I2C Master Command for receive error stop if the error
    //is not due to i2c bus arbitration loss
    else
    {

    I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP);

    {
    //Error service for address ACK or data ACK
    }

    }

 }


//Read data from I2CMDR
I2CMasterDataGet(I2C8_BASE);

//Provide Continuous SCL

while(1)

{

//Continue receiving the burst data
I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_RECEIVE_CONT);

//Implement Delay
SysCtlDelay(100);

//Allow the Master to finish receiving each byte
while(I2CMasterBusy(I2C8_BASE));

//Check for errors in the I2C8 Module
while (I2CMasterErr(I2C8_BASE))

//Error branching
{

    //Check for I2C Bus arbitration loss error condition
    if(I2CMasterErr(I2C8_BASE) == I2C_MASTER_ERR_ARB_LOST)

    {

        //Error service for loss of bus arbitration
        TM4C1294_ArbitrationLost_ErrorService();

        //Return the error status
        return -1;

    }

    //Write I2C Master Command for receive error stop if the error
    //is not due to i2c bus arbitration loss
    else
    {

        I2CMasterControl(I2C8_BASE, I2C_MASTER_CMD_BURST_SEND_ERROR_STOP);

        {
        //Error Service for address ACK or data ACK
        }

    }
```

```
     }

    //Receive data in I2CMDR
    I2CMasterDataGet(I2C8_BASE);

    }

    //Return no errors
    return 0;
}
```

The *ADS7142DataRead_count()* and *ADS7142DataRead_autonomous()* functions are discussed and presented in Section 5 for sake of coherence to the functional modes they are used in.

4 Using the Software

4.1 Prerequisites

To successfully compile the project, download and run the software described in this document, the following hardware and software is required:

- ARM Cortex-M4F-based MCU TM4C1294 connected LaunchPad™ evaluation kit
- ADS7142 nanopower, dual-channel, programmable sensor monitor BoosterPack™ plug-in module
- Code composer studio (CCS) integrated development environment (IDE) (version 7.4.0.00015)
- TivaWare (version 2.1.4.178)

4.2 Getting Started

Follow these steps to get the ADS7142 device functional modes firmware up and running:

1. Install Code Composer Studio version 7.4.0.00015.
2. In order to reach TivaWare, go to the MSP430Ware page and click on the *Start Development* button to download the MSP430Ware-cloud tool and the TI Resource Explorer will open in your browser.
3. On the left hand side of the screen under *Software* click on *TM4C ARM Cortex-M4F MCU - v:2.1.3.156*.
4. Under *Libraries* click on *Driver Library*.
5. Click on the *Download all* button on the right hand side of the screen to download both the user's guide and driverlib files.
6. After the driver files have been downloaded, move them to the workspace folder. This folder should be located in *C:///Users/workspacev7*.
7. Download the ADS7142 device functional modes firmware from ti.com, located under *Software*. Unzip the files into your working Code Composer Studio Directory. Your working directory is normally located under *C:///Users/workspacev7*.
8. Open Code Composer Studio and start a new project by clicking on *File → New → CCS Project*.
    a. Name the project as desired. At the top of this window, ensure that the Target device is selected to be the Tiva TM4C1294NCPDT.
    b. Right click on the project in the project explorer and click on *Add Files*.
    c. Select all of the files that were downloaded into the workspace from ti.com and click *Open*.
9. Right click the project name in the workspace window and click on *Properties*. A window should appear.
    a. Under *General options*, select the TIVA TM4C1294NCPDT. In the *Connection* drop down menu select the *Stellaris In-Circuit Debug Interface*.
    b. Under *Tool-Chain* select the appropriate device endianness *little* for the TM4C1294NCPDT. In the *Linker command file* field browse your directory for the .cmd file for your device. In this case, this file is *tm4c1294ncpdt.cmd*.

c. On the left hand side there is a side menu. Click on the expansion arrow next to *Build*. Click on *ARM Compiler*.
   i. Under *Include Options* click on the paper icon with a green plus to add an include directory. Add the directory in which the *TivaWare_C_Series-2.1.4.178* folder is located.
   ii. Under *Predefined Symbols* click on the paper icon with a green plus to add the appropriate predefined NAMEs for your target device. Add *TARGET_IS_TM4C129_RA0* and *PART_TM4C1294NCPDT* exactly as shown.
d. Click on *File Search Path*. Add a library that points to the *driverlib.lib file* within the *TivaWare_C_Series - 2.1.4.178* folder. This path should be *C://Users/workspacev7/TivaWare_C_Series-2.1.4.178/driverlib/ccs/Debug/driverlib.lib*.

10. Click *OK* to close the *Project Properties* window.
11. Select *Project → Build Project*.
12. Delete the *main.c* file that was created when the project was created.
13. Choose one device functional mode C file you would like to build. There should be three C files included in the project build: *TM4C1294_ADS7142_Functions.c, tm4c1294ncpdt_startup_ccs.c,* and your desired ADS7142 functional mode C file. The *ADS7142RegisterMap.h* file should also be included in the project build. Select all undesired functional mode C files and once highlighted, right click then click on *Exclude from build*.
14. Rebuild the project and only warnings should appear.
15. As illustrated in Figure 3, connect the ADS7142BoosterPack to the TM4C1294 target board.

4.3 Using the Library

The project explorer in the Code Composer Studio should look like Figure 11 when the software is downloaded. There should be numerous main routine files that are not included in the current build. This directory appearance is normal, as only one routine should run at a given time.

Ensure that only one of the functional mode main routine files is included in the project. Exclude all other functional modes from the build

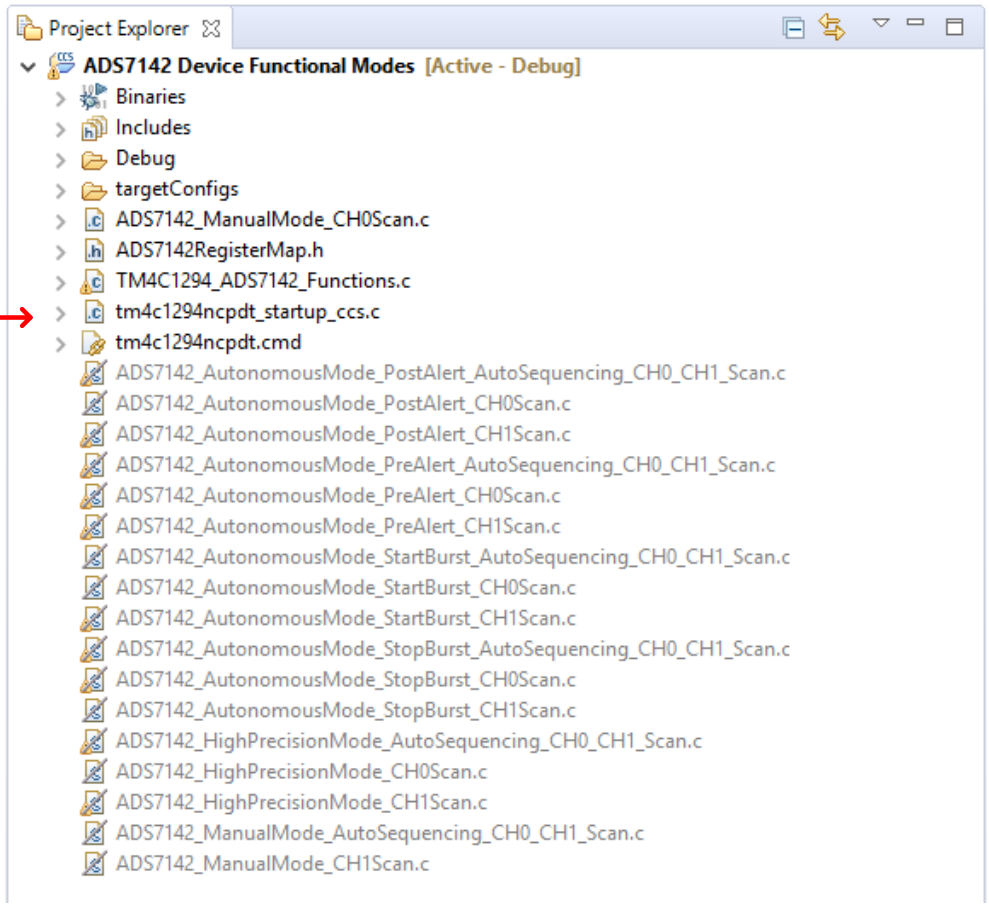
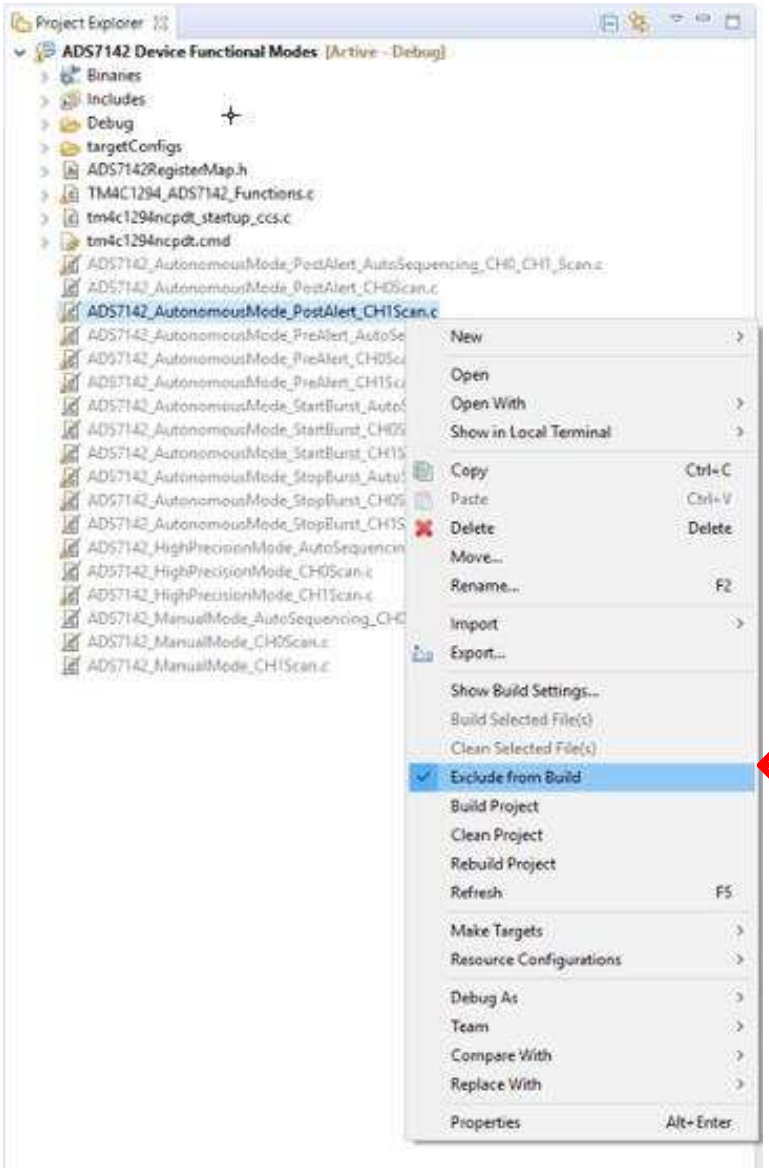


Figure 11. Project Explorer

Figure 12 shows how to change the current functional mode routine. The selected functional mode must be included in the current build before excluding the current one.



To include a selected file in the build, right click on it and the drop down menu will appear as shown. Uncheck “Exclude from Build” to have the file included in the build configuration.

Figure 12. Including a File in the Project Build

Figure 13 shows how to exclude and swap a file for the desired one.

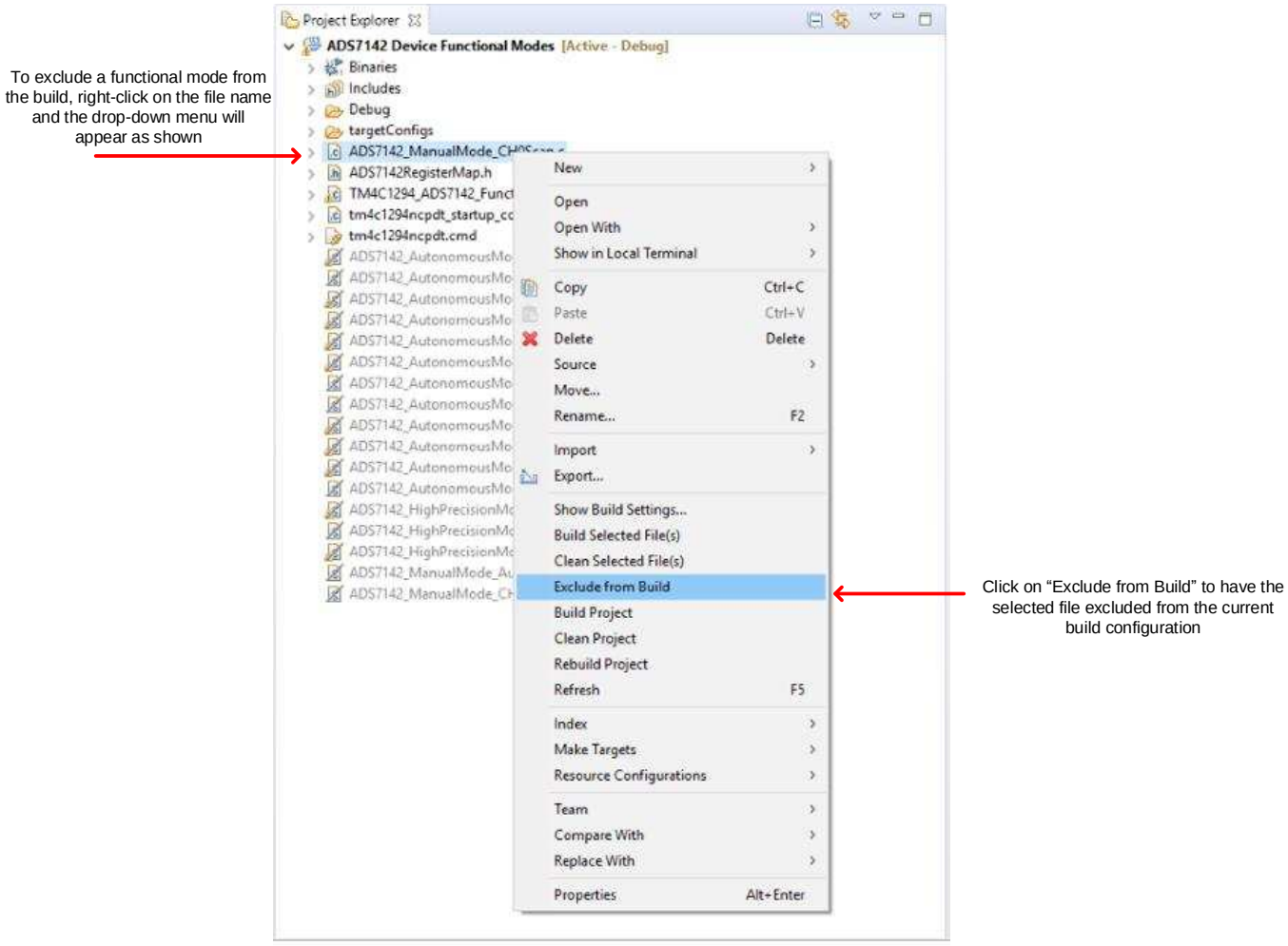


Figure 13. Excluding a File From the Project Build

When including the desired files for compilation, the user can build and run the project.

5 Main Routines and Test Data

When developing this firmware framework, Texas Instruments used a National Instruments VirtualBench to set two DC inputs to the ADS7142 for testing of dual-channel sampling capability. One input was set slightly above the middle of the ADS7142 full-scale range, and the other set one volt above this level. The reference on the ADS7142BoosterPack™ is set to 3.3 V so voltage inputs to the ADS7142 channels are limited to a maximum of 3.3 V. The output code equation in decimal for a SAR ADC is:

$$Output\ \ Code\ \ (decimal) = \left(\frac{V_{Input}}{V_{Ref}}\right) \times 2^{Resolution} \quad (1)$$

For convenient conversion of decimal output codes to hexadecimal output codes TI recommends using the Analog Engineer's Calculator. The calculator contains an *ADC Code to Voltage* converter under the *Data Converters* calculator. The logic analyzer used to decode the I^2C data provides outputs in hexadecimal form.

5.1 Manual Mode

At power up, the ADS7142 goes into manual mode. In this mode, the device uses the high frequency oscillator for conversions. This mode allows the external host processor to directly request and control when the data is sampled. The resolution is set to 12 bits and samples are clocked out by providing continuous SCL when the desired channels are selected.

If the device scans both channels in AUTO sequence, the first data (sample A) is from CH0 and second data (sample A+1) is from CH1. The main routine for manual mode operation with AutoSequencing of both channels is as follows:

```
#include "ADS7142RegisterMap.h"

/*
ADS7142_ManualMode_AutoSequencing_CH0_CH1_Scan.c
*/

int main(void)
{

//Initialize the master MCU (0 = 100 kHz SCL, 1 = 400 kHz SCL)
TM4C1294Init(0);

//Calibrate out the offset of the ADS7142
ADS7142Calibrate();

//Let's put the ADS7142 into Manual Mode with both Channels enabled in Single-
Ended Configuration for AUTO Sequencing

//Select the channel input configuration
ADS7142SingleRegisterWrite(ADS7142_REG_CHANNEL_INPUT_CFG,
ADS7142_VAL_CHANNEL_INPUT_CFG_2_CHANNEL_SINGLE_ENDED);

//Confirm the input channel configuration
uint32_t channelconfig;
ADS7142SingleRegisterRead(ADS7142_REG_CHANNEL_INPUT_CFG, &channelconfig);

//Select the operation mode of the device
ADS7142SingleRegisterWrite(ADS7142_REG_OPMODE_SEL,
ADS7142_VAL_OPMODE_SEL_I2C_CMD_MODE_W_AUTO_SEQ_EN);

//Confirm the operation mode selection
uint32_t opmodeselconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_SEL, &opmodeselconfig);

//Set the I2C Mode to High Speed (optional)
//ADS7142HighSpeedEnable(ADS7142_VAL_OPMODE_I2CMODE_HS_1);

//Check the I2C Mode Status
uint32_t opmodei2cconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_I2CMODE_STATUS, &opmodei2cconfig);

//Configure Auto Sequencing for both channels
ADS7142SingleRegisterWrite(ADS7142_REG_AUTO_SEQ_CHEN, ADS7142_VAL_AUTO_SEQ_CHENAUTO_SEQ_CH0_CH1);

//Confirm Auto Sequencing Register Configuration
uint32_t autoseqchenconfig;
ADS7142SingleRegisterRead(ADS7142_REG_AUTO_SEQ_CHEN, &autoseqchenconfig);

//Set SEQ_START Bit to start the sampling Sequence
ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);

//Begin manual mode operation with AUTO Sequencing

//Start Sampling Ch0 and Ch1
while (ADS7142DataRead_continuous() < 0 );
```

```
//Return no errors
return(0);

}
```

The I^2C address of the ADS7142 is a 7-bit address followed by a read/write bit indicating the direction of data transfer. The I^2C address of the ADS7142 is set to 0x18 (001 1000 R/W) by the hardware configuration of the ADS7142BoosterPack™. The analyzer sees an 8-bit address with the R/W bit as part of the address, and therefore reports the I^2C address as 0x30 (0011 0000) for a write operation and 0x31 (0011 0001) for a read operation. Figure 14 illustrates the test data from the salae logic analyzer.

Masks that contain the letters REG refer to ADS7142 register masks, and masks that contain the letters VAL refer to ADS7142 value masks.

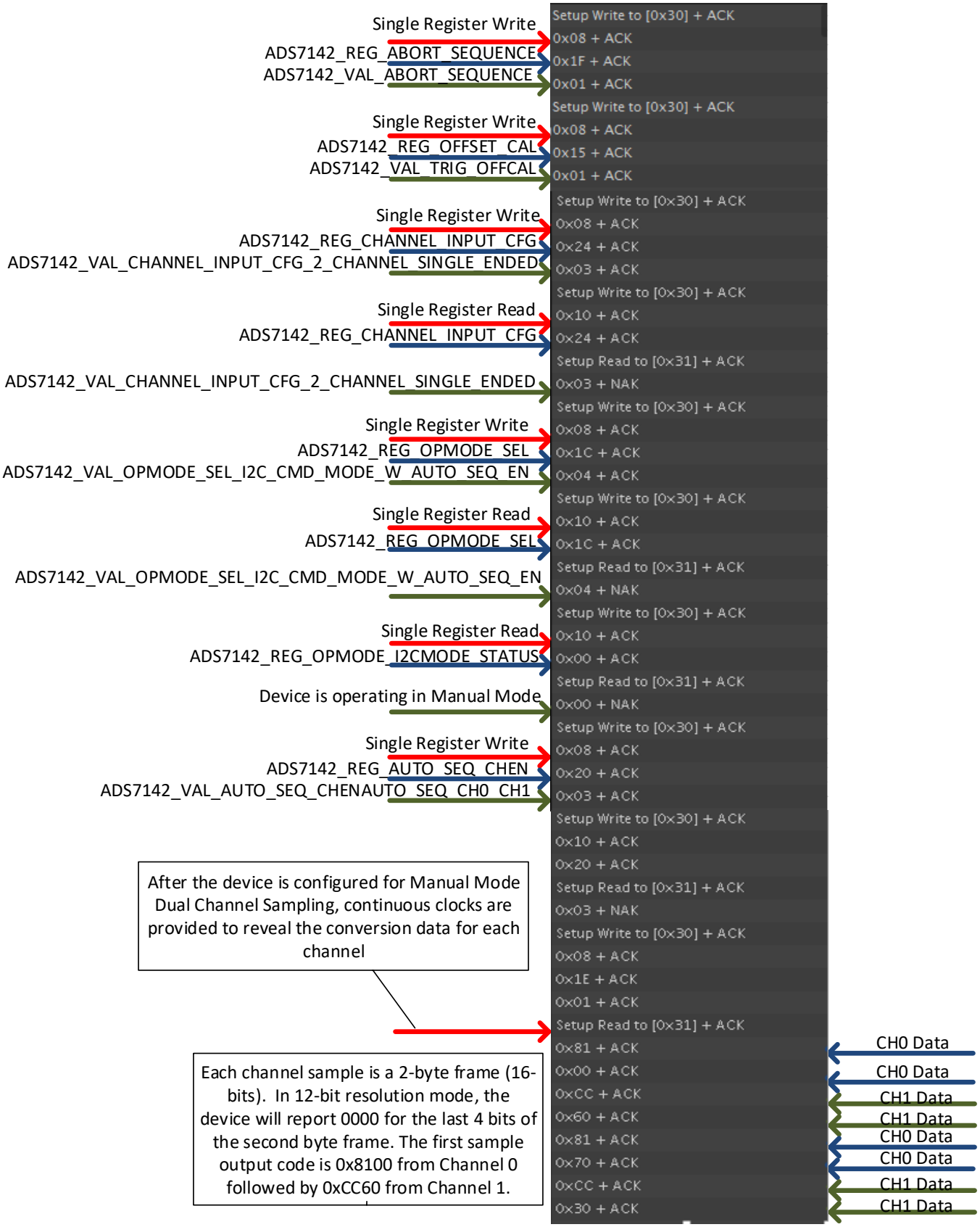


Figure 14. Manual Mode Dual Channel Sampling Test Data

5.2 Autonomous Mode With Pre-Alert

In autonomous mode with pre alert data, the ADS7142 automatically scans the input voltage on the input channels and generates a signal when the programmable high or low threshold values are crossed. The device stores the 16 conversion results prior to the activation of the alert. When the alert is activated, conversion stops and the data buffer is no longer filled. Figure 15 shows the filling of the data buffer in autonomous mode with pre alert data.

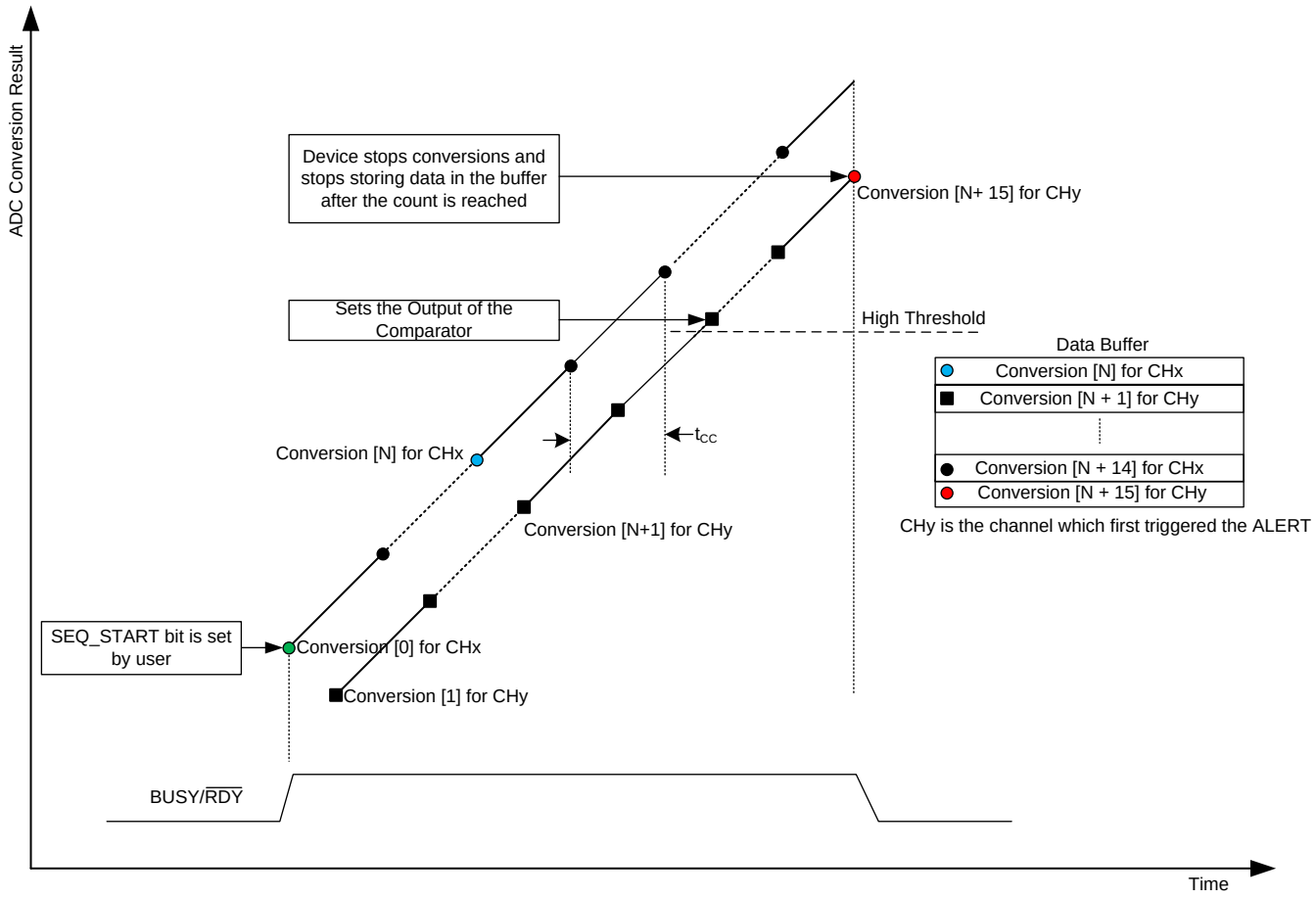


Figure 15. Pre Alert Data for Dual-Channel Configuration

The example main routine for this mode is the following:

```
#include "ADS7142RegisterMap.h"

/*
ADS7142_AutonomousMode_PreAlert_AutoSequencing_CH0_CH1_Scan.c
*/

int main(void)
{

//Initialize the master MCU (0 = 100 kHz SCL, 1 = 400 kHz SCL)
TM4C1294Init(0);

//Calibrate out the offset from the ADS7142
ADS7142Calibrate();

//Let's put the ADS7142 into Autonomous Mode with both Channels enabled in Single-
Ended Configuration

//Select the channel input configuration
```

```
ADS7142SingleRegisterWrite(ADS7142_REG_CHANNEL_INPUT_CFG,
ADS7142_VAL_CHANNEL_INPUT_CFG_2_CHANNEL_SINGLE_ENDED);

//Confirm the input channel configuration
uint32_t channelconfig;
ADS7142SingleRegisterRead(ADS7142_REG_CHANNEL_INPUT_CFG, &channelconfig);

//Select the operation mode of the device
ADS7142SingleRegisterWrite(ADS7142_REG_OPMODE_SEL,
ADS7142_VAL_OPMODE_SEL_AUTONOMOUS_MONITORING_MODE);

//Confirm the operation mode selection
uint32_t opmodeselconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_SEL, &opmodeselconfig);

//Set the I2C Mode to High Speed (optional)
//ADS7142HighSpeedEnable(ADS7142_VAL_OPMODE_I2CMODE_HS_1);

//Check the I2C Mode Status
uint32_t opmodei2cconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_I2CMODE_STATUS, &opmodei2cconfig);

//Select both channels for AUTO Sequencing
ADS7142SingleRegisterWrite(ADS7142_REG_AUTO_SEQ_CHEN, ADS7142_VAL_AUTO_SEQ_CHENAUTO_SEQ_CH0_CH1);

//Confirm Auto Sequencing is enabled
uint32_t autoseqchenconfig;
ADS7142SingleRegisterRead(ADS7142_REG_AUTO_SEQ_CHEN, &autoseqchenconfig);

//Select the Low Power or High Speed Oscillator
ADS7142SingleRegisterWrite(ADS7142_REG_OSC_SEL, ADS7142_VAL_OSC_SEL_HSZ_HSO);

//Confirm the oscillator selection
uint32_t oscconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OSC_SEL, &oscconfig);

 //Set the minimum nCLK value for one conversion to maximize sampling speed
ADS7142SingleRegisterWrite(ADS7142_REG_nCLK_SEL, 21);

//Confirm the nCLK selection
uint32_t nCLKconfig;
ADS7142SingleRegisterRead(ADS7142_REG_nCLK_SEL, &nCLKconfig);

//Select the Data Buffer output data Configuration
ADS7142SingleRegisterWrite(ADS7142_REG_DOUT_FORMAT_CFG, ADS7142_VAL_DOUT_FORMAT_CFG_DOUT_FORMAT2);

//Select the Data Buffer opmode for Pre-Alert mode
ADS7142SingleRegisterWrite(ADS7142_REG_DATA_BUFFER_OPMODE,
ADS7142_VAL_DATA_BUFFER_STARTSTOP_CNTRL_PREALERT);

//Configure CH0 High Threshold MSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HTH_CH0_MSB, 0x90);

//Configure CH0 High Threshold LSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HTH_CH0_LSB, 0x00);

//Configure CH0 Low Threshold MSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_LTH_CH0_MSB, 0x00);

//Configure CH0 Low Threshold LSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_LTH_CH0_LSB, 0x00);

//Set the Hysteresis for CH0
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HYS_CH0, 0x00);

//Configure CH1 High Threshold MSB
```

```
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HTH_CH1_MSB, 0xE0);

//Configure CH1 High Threshold LSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HTH_CH1_LSB, 0x00);

//Configure CH1 Low Threshold MSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_LTH_CH1_MSB, 0x00);

 //Configure CH1 Low Threshold LSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_LTH_CH1_LSB, 0x00);

//Set the Hysteresis for CH1
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HYS_CH1, 0x00);

//Set the Pre-Alert Event Counter
ADS7142SingleRegisterWrite(ADS7142_REG_PRE_ALERT_EVENT_COUNT, ADS7142_VAL_PRE_ALERT_EVENT_COUNT4);

//Confirm the Pre-Alert Event Counter setting
uint32_t eventcount;
ADS7142SingleRegisterRead(ADS7142_REG_PRE_ALERT_EVENT_COUNT, &eventcount);

//Enable Alerts
ADS7142SingleRegisterWrite(ADS7142_REG_ALERT_CHEN, ADS7142_VAL_ALERT_EN_CH0_CH1);

//Enable the digital window comparator block
ADS7142SingleRegisterWrite(ADS7142_REG_ALERT_DWC_EN, ADS7142_VAL_ALERT_DWC_BLOCK_ENABLE);

//Set the SEQ_START Bit to begin the sampling sequence
ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);

//Begin Autonomous Mode Pre-Alert operation and Scan both channels 0 and 1

while (1)

{

//Start Scanning CH0 and CH1
//If Alert is set, device stops conversions and filling the data buffer
//Read the latched flags of the Digital Window Comparator

while (ADS7142DataRead_autonomous() < 0);

//Reset the alert flags
ADS7142SingleRegisterWrite(ADS7142_REG_ALERT_HIGH_FLAGS, 0x03);
ADS7142SingleRegisterWrite(ADS7142_REG_ALERT_LOW_FLAGS, 0x03);

//Read the Data Buffer Status
uint32_t databufferstatus;
ADS7142SingleRegisterRead(ADS7142_REG_DATA_BUFFER_STATUS, &databufferstatus);

//Read the Data Buffer
while (ADS7142DataRead_count(16) < 0);

//Restart the sequence
ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);
 }
//Return no errors
return 0;

}
```

The data output seen from the logic analyzer corresponds to what is written in main. Figure 16 shows the ADS7142 configuration.

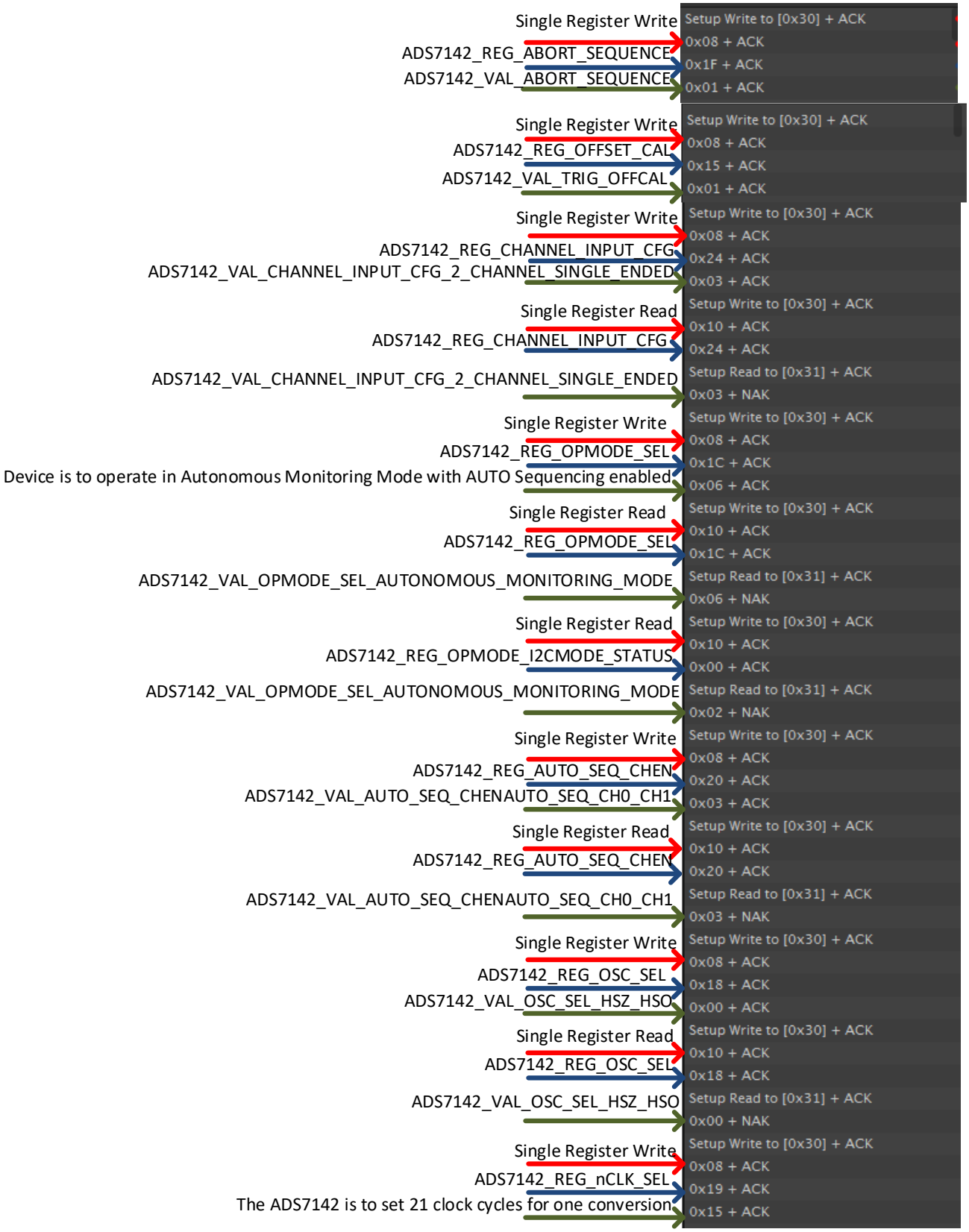


Figure 16. Autonomous Mode Pre-Alert Dual Channel Sampling Test Data 1

The data in Figure 17 show further configuration and settings of the thresholds.

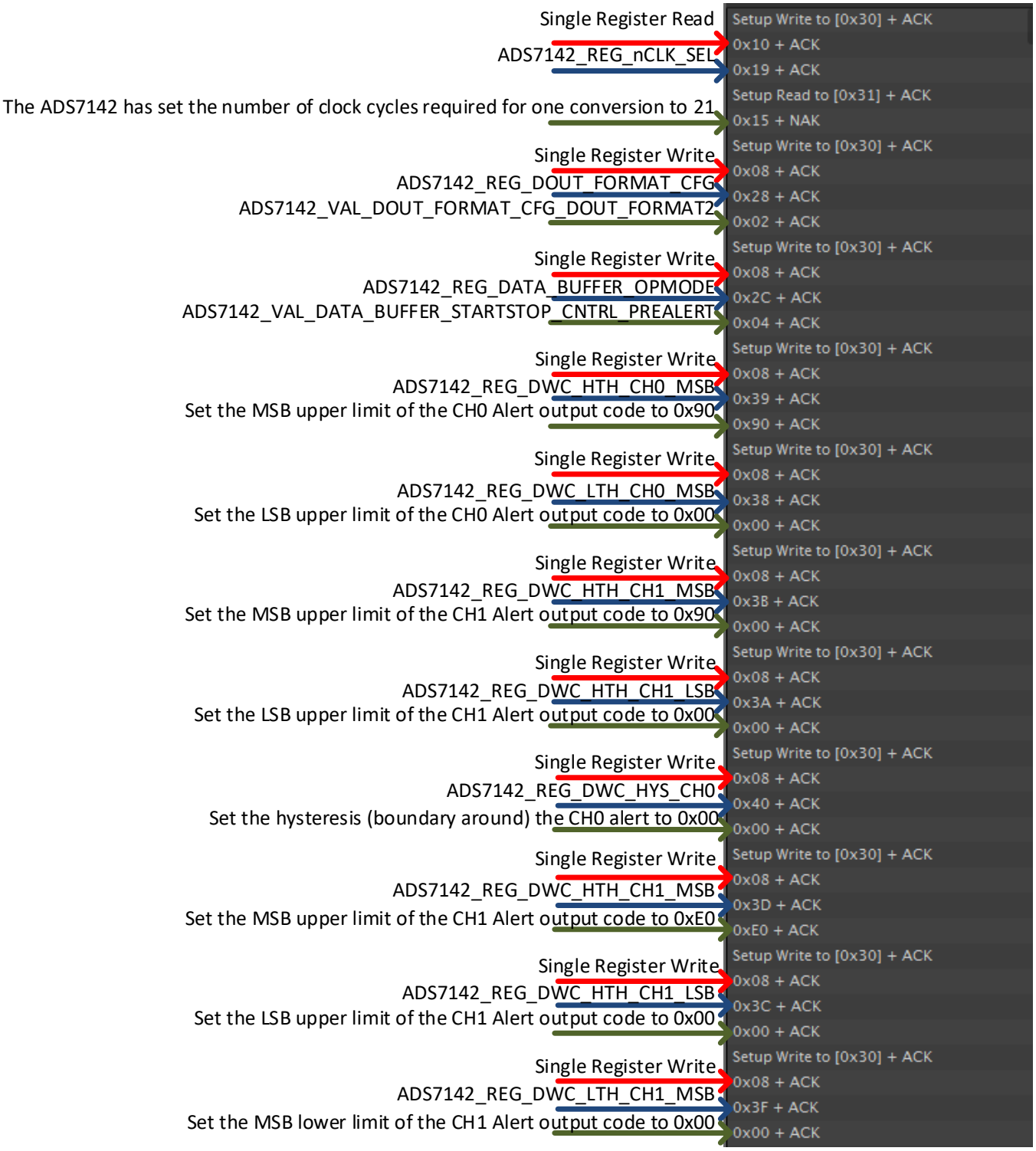


Figure 17. Autonomous Mode Pre-Alert Dual Channel Sampling Test Data 2

The data in Figure 18 shows how the alerts are enabled and the start of the conversion sequence.

Figure 18. Autonomous Mode Pre-Alert Dual Channel Sampling Test Data 3

Figure 19 shows the contents of the data buffer after triggering an alert.

Figure 19. Autonomous Mode Pre-Alert Dual Channel Sampling Test Data 4

5.3 Autonomous Mode With Post-Alert

In autonomous mode with post alert data, the ADS7142 captures the 16 conversion results after the alert is activated. Figure 20 shows the filling of the data buffer in autonomous mode with post alert data.

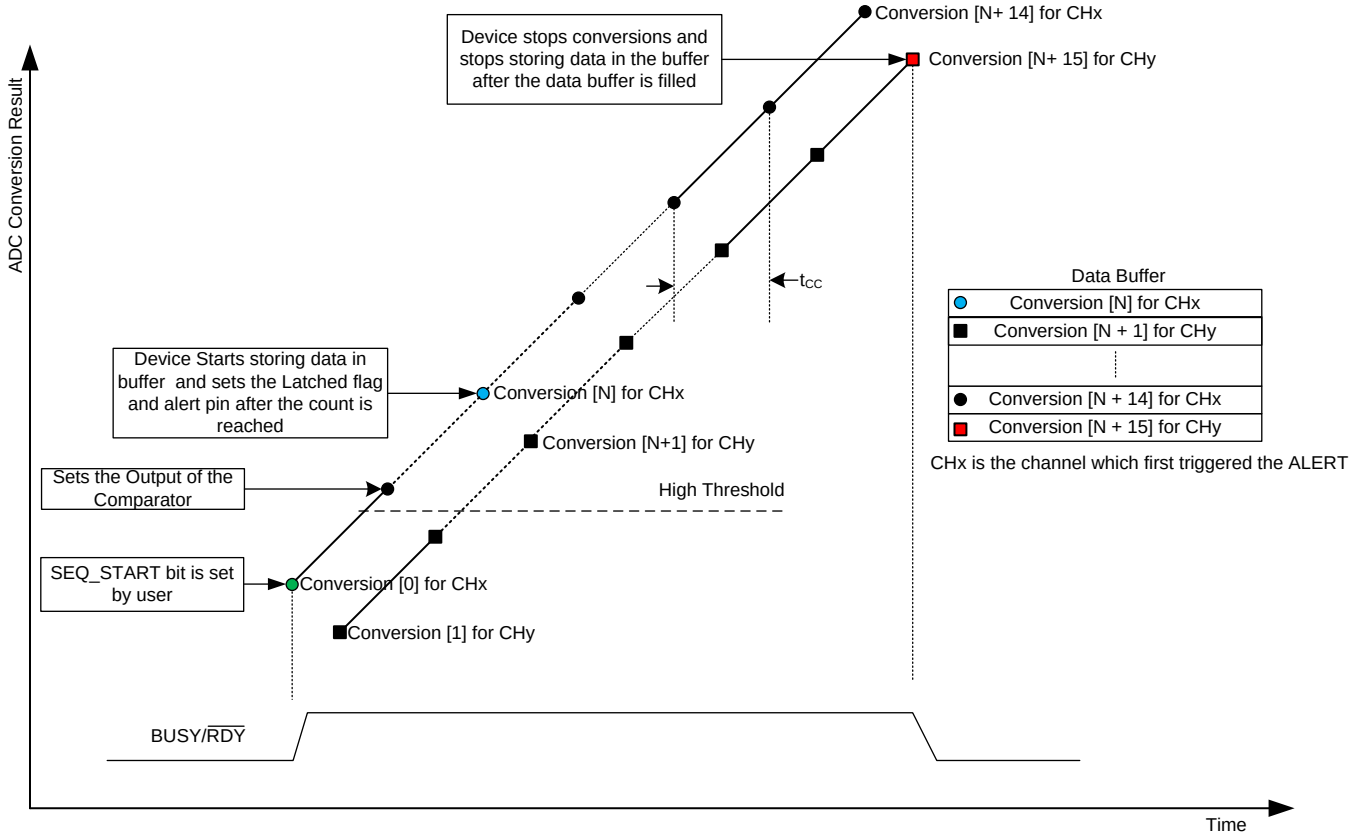


Figure 20. Post Alert Data for Dual-Channel Configuration

The main routine for this functional mode is the following:

```
#include "ADS7142RegisterMap.h"

/*
ADS7142_AutonomousMode_PostAlert_AutoSequencing_CH0_CH1_Scan.c
*/

int main(void)
{

//Initialize the master MCU (0 = 100 kHz SCL, 1 = 400 kHz SCL)
TM4C1294Init(0);

//Calibrate out the offset from the ADS7142
ADS7142Calibrate();

//Let's put the ADS7142 into Autonomous Mode with both Channels enabled in Single-
Ended Configuration

//Select the channel input configuration
ADS7142SingleRegisterWrite(ADS7142_REG_CHANNEL_INPUT_CFG,
ADS7142_VAL_CHANNEL_INPUT_CFG_2_CHANNEL_SINGLE_ENDED);

//Confirm the input channel configuration
uint32_t channelconfig;
ADS7142SingleRegisterRead(ADS7142_REG_CHANNEL_INPUT_CFG, &channelconfig);
```

```
//Select the operation mode of the device
ADS7142SingleRegisterWrite(ADS7142_REG_OPMODE_SEL,
ADS7142_VAL_OPMODE_SEL_AUTONOMOUS_MONITORING_MODE);

//Confirm the operation mode selection
uint32_t opmodeselconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_SEL, &opmodeselconfig);

//Set the I2C Mode to High Speed (optional)
//ADS7142HighSpeedEnable(ADS7142_VAL_OPMODE_I2CMODE_HS_1);

//Check the I2C Mode Status
uint32_t opmodei2cconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_I2CMODE_STATUS, &opmodei2cconfig);

//Select both channels for AUTO Sequencing
ADS7142SingleRegisterWrite(ADS7142_REG_AUTO_SEQ_CHEN, ADS7142_VAL_AUTO_SEQ_CHENAUTO_SEQ_CH0_CH1);

//Confirm Auto Sequencing is enabled
uint32_t autoseqchenconfig;
ADS7142SingleRegisterRead(ADS7142_REG_AUTO_SEQ_CHEN, &autoseqchenconfig);

//Select the Low Power or High Speed Oscillator
ADS7142SingleRegisterWrite(ADS7142_REG_OSC_SEL, ADS7142_VAL_OSC_SEL_HSZ_HSO);

//Confirm the oscillator selection
uint32_t oscconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OSC_SEL, &oscconfig);

//Set the minimum nCLK value for one conversion to maximize sampling speed
ADS7142SingleRegisterWrite(ADS7142_REG_nCLK_SEL, 21);

//Confirm the nCLK selection
 uint32_t nCLKconfig;
ADS7142SingleRegisterRead(ADS7142_REG_nCLK_SEL, &nCLKconfig);

//Select the Data Buffer output data Configuration
ADS7142SingleRegisterWrite(ADS7142_REG_DOUT_FORMAT_CFG, ADS7142_VAL_DOUT_FORMAT_CFG_DOUT_FORMAT2);

//Select the Data Buffer opmode for Post-Alert mode
ADS7142SingleRegisterWrite(ADS7142_REG_DATA_BUFFER_OPMODE,
ADS7142_VAL_DATA_BUFFER_STARTSTOP_CNTRL_POSTALERT);

//Configure CH0 High Threshold MSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HTH_CH0_MSB, 0x90);

//Configure CH0 High Threshold LSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HTH_CH0_LSB, 0x00);

//Configure CH0 Low Threshold MSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_LTH_CH0_MSB, 0x00);

//Configure CH0 Low Threshold LSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_LTH_CH0_LSB, 0x00);

//Set the Hysteresis for CH0
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HYS_CH0, 0x00);

//Configure CH1 High Threshold MSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HTH_CH1_MSB, 0xE0);

//Configure CH1 High Threshold LSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HTH_CH1_LSB, 0x00);

//Configure CH1 Low Threshold MSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_LTH_CH1_MSB, 0x00);
```

```
//Configure CH1 Low Threshold LSB
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_LTH_CH1_LSB, 0x00);

//Set the Hysteresis for CH1
ADS7142SingleRegisterWrite(ADS7142_REG_DWC_HYS_CH1, 0x00);

//Enable Alerts
ADS7142SingleRegisterWrite(ADS7142_REG_ALERT_CHEN, ADS7142_VAL_ALERT_EN_CH0_CH1);

//Enable the digital window comparator block
ADS7142SingleRegisterWrite(ADS7142_REG_ALERT_DWC_EN, ADS7142_VAL_ALERT_DWC_BLOCK_ENABLE);

//Set the SEQ_START Bit to begin the sampling sequence
ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);

//Begin Autonomous Mode Post-Alert operation and Scan both channels 0 and 1

while (1)

{

//Start Scanning CH0 and CH1
//If Alert is set, device stops conversions and filling the data buffer
//Read the latched flags of the Digital Window Comparator
while (ADS7142DataRead_autonomous() < 0);

//Reset the alert flags
ADS7142SingleRegisterWrite(ADS7142_REG_ALERT_HIGH_FLAGS, 0x03);
ADS7142SingleRegisterWrite(ADS7142_REG_ALERT_LOW_FLAGS, 0x03);

//Read the Data Buffer Status uint32_t databufferstatus;
ADS7142SingleRegisterRead(ADS7142_REG_DATA_BUFFER_STATUS, &databufferstatus);

//Read the Data Buffer
while (ADS7142DataRead_count(16) < 0);

//Restart the sequence
ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);

}

//Return no errors
return 0;

}
```

Figure 21 shows the channel configuration and selection of the opmode for this ADS7142 functional mode.

Figure 21. Autonomous Mode Post Alert Dual-Channel Sampling Test Data 1

Figure 22 shows the conversion clock, data buffer configuration, output data format configuration, setting of thresholds for channel 0, and setting of hysteresis for channel 0.

Figure 22. Autonomous Mode Post Alert Dual-Channel Sampling Test Data 2

Figure 23 shows the setting of thresholds and hysteresis for channel 1, the enabling of alerts, and the start of the conversion sequence.

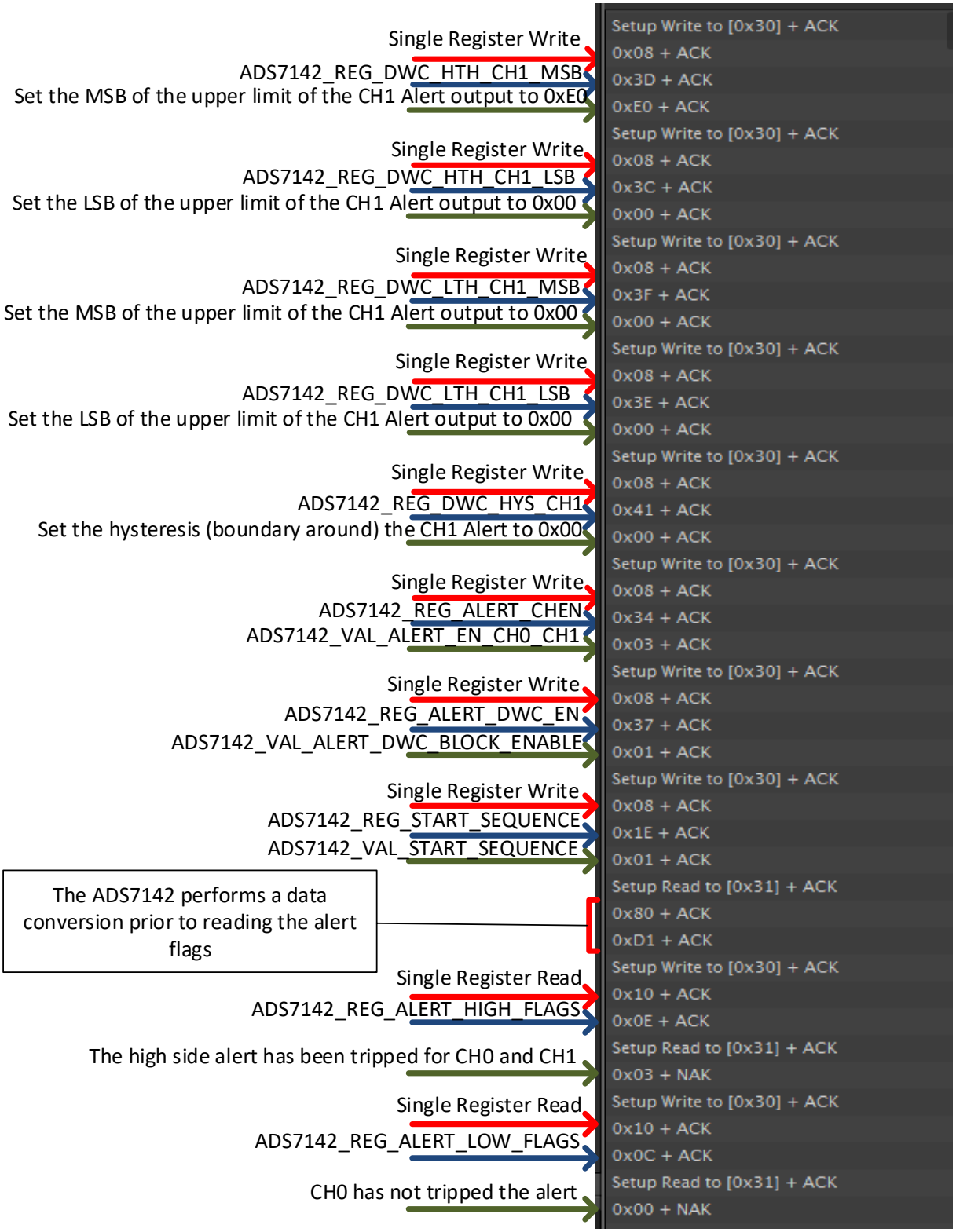


Figure 23. Autonomous Mode Post Alert Dual Channel Sampling Test Data 3

Figure 24 shows the data results for the alert flag resets, reading of the data buffer status, and clocking out the data inside the data buffer after the alert is triggered.

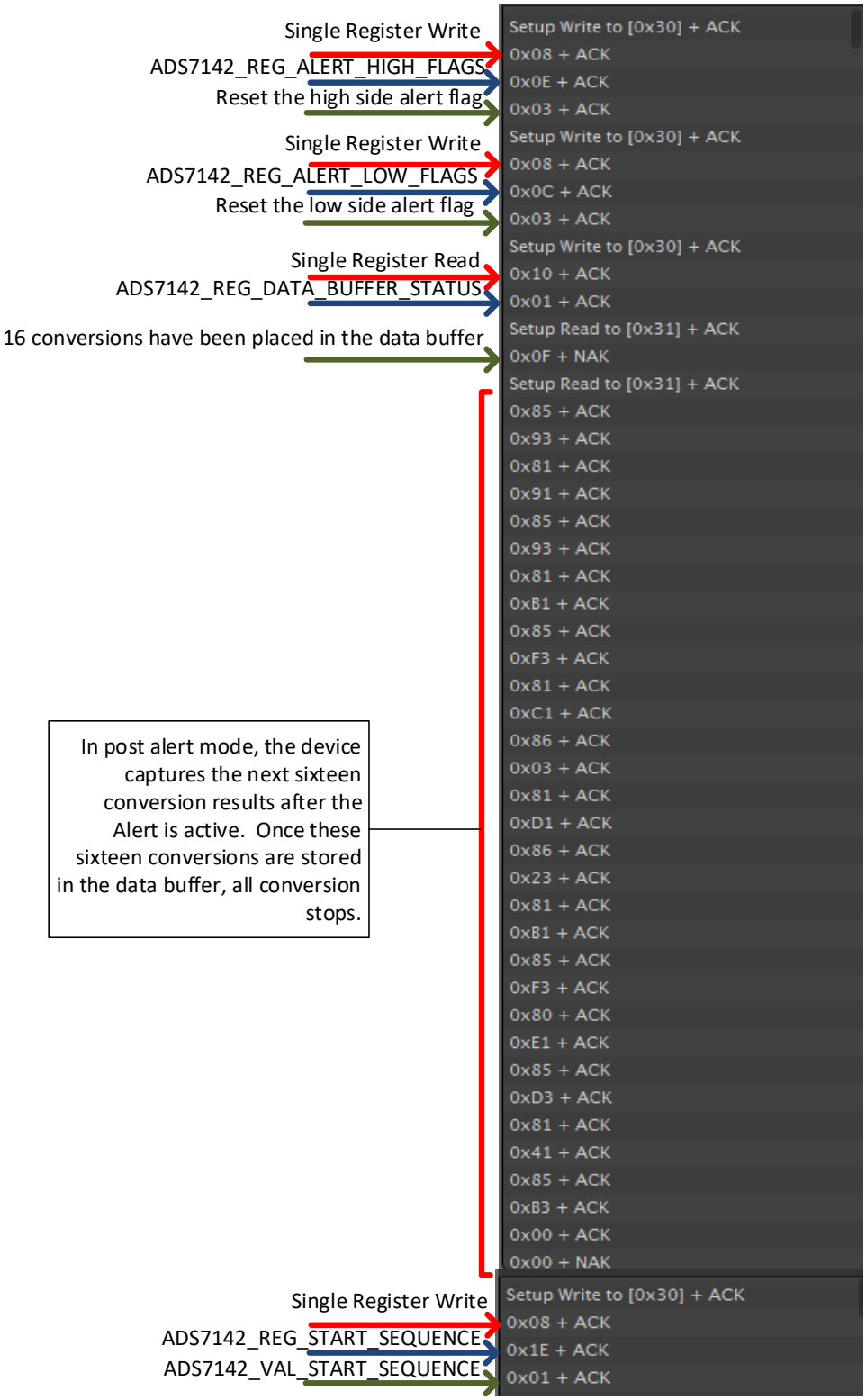


Figure 24. Autonomous Mode Post Alert Dual-Channel Sampling Test Data 4

5.4 Autonomous Mode With Start Burst Data

In autonomous mode with start burst data functional mode, the ADS7142 can be configured to start filling the data buffer when starting the conversion sequence. After the data buffer is filled, conversion stops. Figure 25 shows how this functional mode handles conversions.

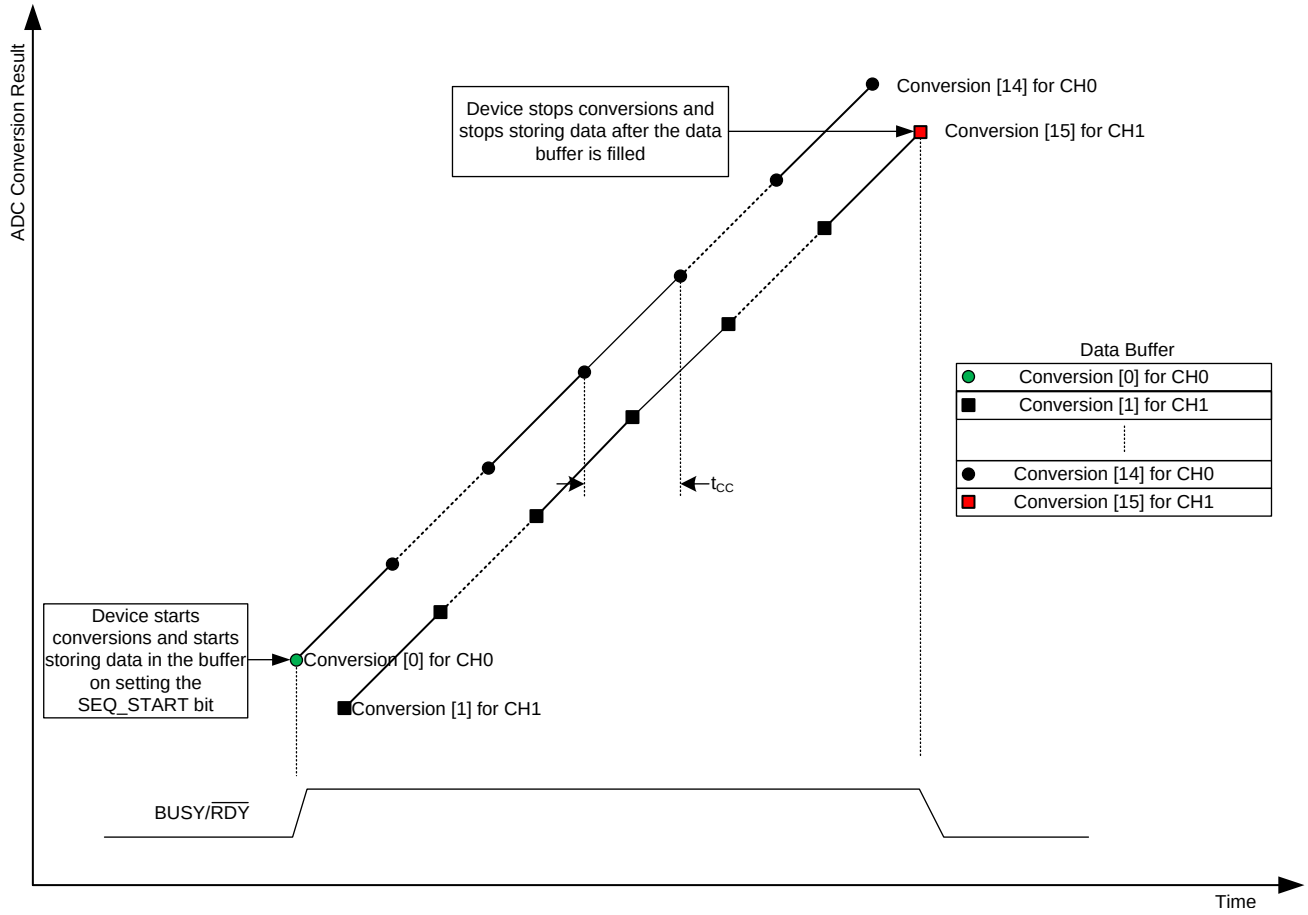


Figure 25. Start Burst With Dual-Channel Configuration

The main routine is the following:

```
#include "ADS7142RegisterMap.h"
/* *ADS7142_AutonomousMode_StartBurst_AutoSequencing_CH0_CH1_Scan.c */

int main(void)
{

//Initialize the master MCU (0 = 100 kHz SCL, 1 = 400 kHz SCL)
TM4C1294Init(0);

//Calibrate out the offset from the ADS7142
ADS7142Calibrate();

//Let's put the ADS7142 into Autonomous Mode with both Channels enabled in Single-
Ended Configuration

//Select the channel input configuration
ADS7142SingleRegisterWrite(ADS7142_REG_CHANNEL_INPUT_CFG,
ADS7142_VAL_CHANNEL_INPUT_CFG_2_CHANNEL_SINGLE_ENDED);

//Confirm the input channel configuration
uint32_t channelconfig;
ADS7142SingleRegisterRead(ADS7142_REG_CHANNEL_INPUT_CFG, &channelconfig);
```

```
//Select the operation mode of the device
ADS7142SingleRegisterWrite(ADS7142_REG_OPMODE_SEL,
ADS7142_VAL_OPMODE_SEL_AUTONOMOUS_MONITORING_MODE);

//Confirm the operation mode selection
uint32_t opmodeselconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_SEL, &opmodeselconfig);

//Set the I2C Mode to High Speed (optional)
//ADS7142HighSpeedEnable(ADS7142_VAL_OPMODE_I2CMODE_HS_1);

//Check the I2C Mode Status
uint32_t opmodei2cconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_I2CMODE_STATUS, &opmodei2cconfig);

//Select both channels for AUTO Sequencing
ADS7142SingleRegisterWrite(ADS7142_REG_AUTO_SEQ_CHEN, ADS7142_VAL_AUTO_SEQ_CHENAUTO_SEQ_CH0_CH1);

//Confirm Auto Sequencing is enabled
uint32_t autoseqchenconfig;
ADS7142SingleRegisterRead(ADS7142_REG_AUTO_SEQ_CHEN, &autoseqchenconfig);

//Select the Low Power or High Speed Oscillator
ADS7142SingleRegisterWrite(ADS7142_REG_OSC_SEL, ADS7142_VAL_OSC_SEL_HSZ_HSO);

//Confirm the oscillator selection
uint32_t oscconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OSC_SEL, &oscconfig);

//Set the minimum nCLK value for one conversion to maximize sampling speed
ADS7142SingleRegisterWrite(ADS7142_REG_nCLK_SEL, 21);

//Confirm the nCLK selection
uint32_t nCLKconfig;
ADS7142SingleRegisterRead(ADS7142_REG_nCLK_SEL, &nCLKconfig);

//Select the Data Buffer output data Configuration
ADS7142SingleRegisterWrite(ADS7142_REG_DOUT_FORMAT_CFG, ADS7142_VAL_DOUT_FORMAT_CFG_DOUT_FORMAT2);

//Select the Data Buffer opmode for Start Burst mode
ADS7142SingleRegisterWrite(ADS7142_REG_DATA_BUFFER_OPMODE,
ADS7142_VAL_DATA_BUFFER_STARTSTOP_CNTRL_STARTBURST);

//Set the SEQ_START Bit to begin the sampling sequence
ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);

//Begin Autonomous Mode StartBurst operation and scan both channels 0 and 1

while (1)

{

uint32_t databufferstatus;
uint32_t sequencestatus;

//Fill and read the data buffer

do

{

while (ADS7142DataRead_count(16) < 0);

}
```

```
//Ensure that the data buffer is not filled and the sequence is not aborted

while ((ADS7142SingleRegisterRead(ADS7142_REG_DATA_BUFFER_STATUS, &databufferstatus) < 0x10) \
&& (ADS7142SingleRegisterRead(ADS7142_REG_SEQUENCE_STATUS, &sequencestatus) == 0x02));

//Restart the sequence
ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);

}

//Return no errors
return 0;

}
```

Figure 26 shows the channel configuration and selection of the opmode for this autonomous mode.

Figure 26. Autonomous Mode Start Burst Dual-Channel Sampling Test Data 1

Figure 27 shows the contents of the data buffer when filled.

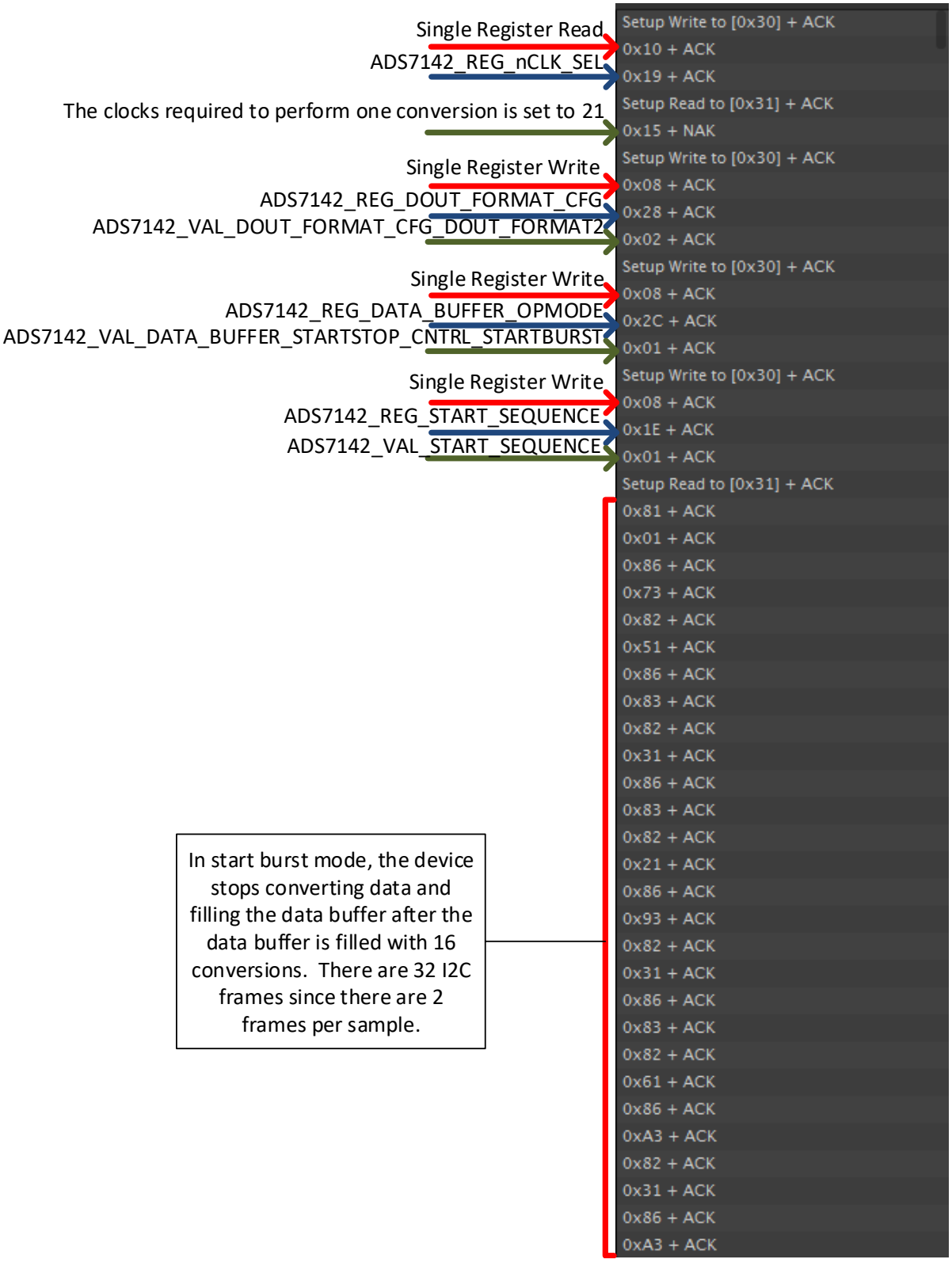


Figure 27. Autonomous Mode Start Burst Dual-Channel Sampling Test Data 2

After the data buffer contents are clocked out, there are no remaining conversions within the buffer. The conversion sequence must be restarted for the buffer to be filled again. Figure 28 shows this process.

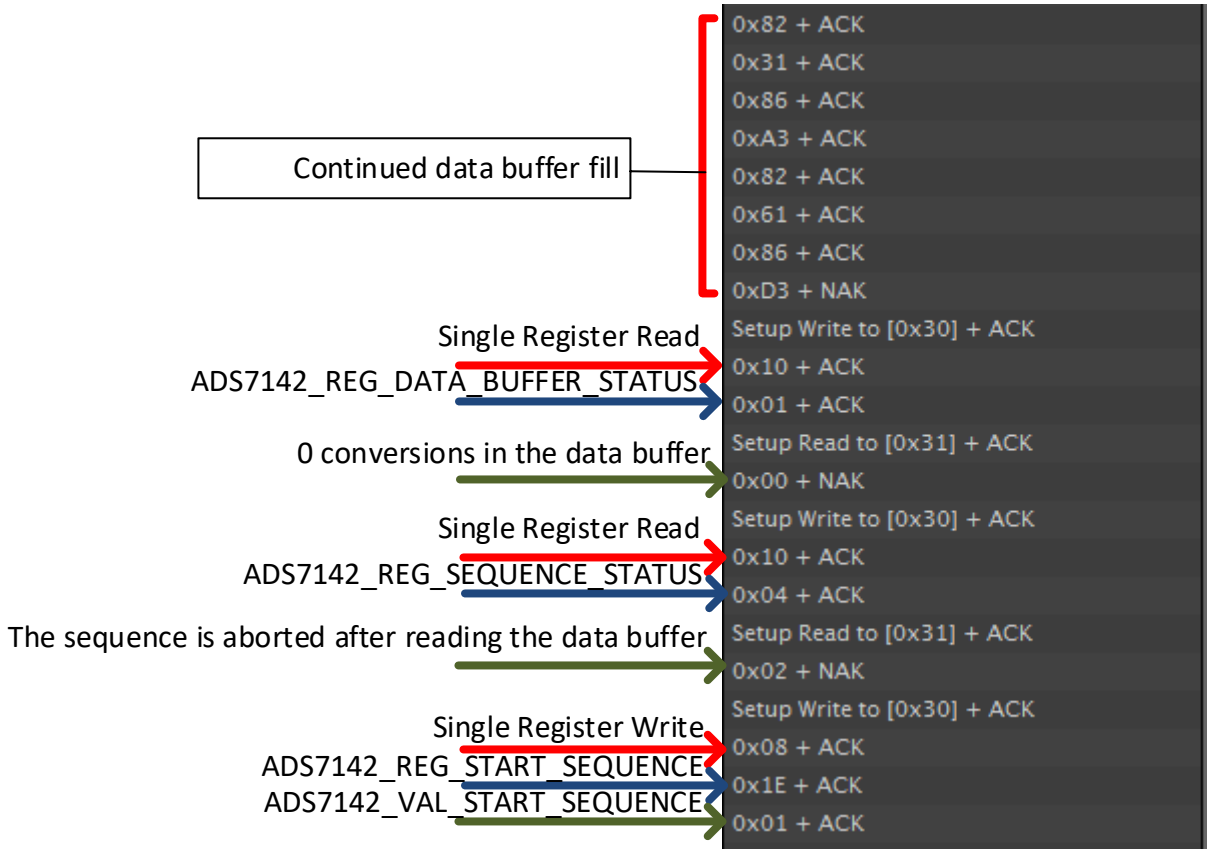


Figure 28. Autonomous Mode Start Burst Dual-Channel Sampling Test Data 3

5.5 Autonomous Mode With Stop Burst Data

In autonomous mode with stop burst, the user is able to configure the ADS7142 to stop filling the data buffer with conversion results by aborting the conversion sequence. If more than 16 conversions occur between the start of the sequence and when the sequence is aborted, the entries first written in to the data buffer are overwritten. Figure 29 shows the filling of the data buffer in autonomous mode with stop burst data.

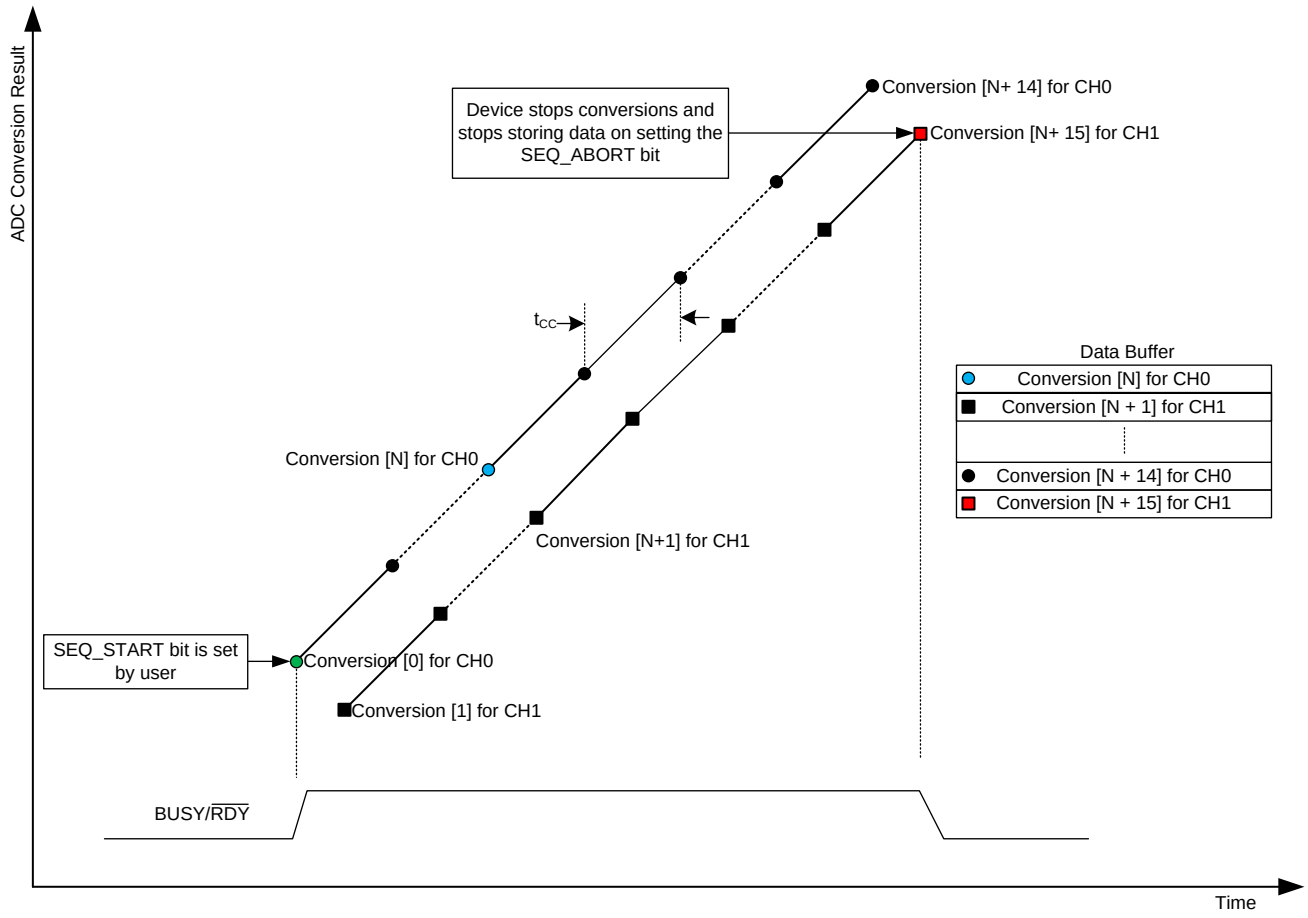


Figure 29. Stop Burst With Dual-Channel Configuration

The main routine is the following:

```
#include "ADS7142RegisterMap.h"

/*
ADS7142_AutonomousMode_StopBurst_AutoSequencing_CH0_CH1_Scan.c
*/

int main(void)
{

//Initialize the master MCU (0 = 100 kHz SCL, 1 = 400 kHz SCL)
TM4C1294Init(0);

//Calibrate out the offset from the ADS7142
ADS7142Calibrate();

//Let's put the ADS7142 into Autonomous Mode with both Channels enabled in Single-
Ended Configuration
//Select the channel input configuration
ADS7142SingleRegisterWrite(ADS7142_REG_CHANNEL_INPUT_CFG,
```

```
ADS7142_VAL_CHANNEL_INPUT_CFG_2_CHANNEL_SINGLE_ENDED);

//Confirm the input channel configuration
uint32_t channelconfig;
ADS7142SingleRegisterRead(ADS7142_REG_CHANNEL_INPUT_CFG, &channelconfig);

//Select the operation mode of the device
ADS7142SingleRegisterWrite(ADS7142_REG_OPMODE_SEL,
ADS7142_VAL_OPMODE_SEL_AUTONOMOUS_MONITORING_MODE);

//Confirm the operation mode selection
uint32_t opmodeselconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_SEL, &opmodeselconfig);

//Set the I2C Mode to High Speed (optional)
//ADS7142HighSpeedEnable(ADS7142_VAL_OPMODE_I2CMODE_HS_1);

//Check the I2C Mode Status
uint32_t opmodei2cconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_I2CMODE_STATUS, &opmodei2cconfig);

//Select both channels for AUTO Sequencing
ADS7142SingleRegisterWrite(ADS7142_REG_AUTO_SEQ_CHEN, ADS7142_VAL_AUTO_SEQ_CHENAUTO_SEQ_CH0_CH1);

//Confirm Auto Sequencing is enabled
uint32_t autoseqchenconfig;
ADS7142SingleRegisterRead(ADS7142_REG_AUTO_SEQ_CHEN, &autoseqchenconfig);

//Select the Low Power or High Speed Oscillator
ADS7142SingleRegisterWrite(ADS7142_REG_OSC_SEL, ADS7142_VAL_OSC_SEL_HSZ_HSO);

//Confirm the oscillator selection
uint32_t oscconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OSC_SEL, &oscconfig);

//Set the minimum nCLK value for one conversion to maximize sampling speed
ADS7142SingleRegisterWrite(ADS7142_REG_nCLK_SEL, 21);

//Confirm the nCLK selection
uint32_t nCLKconfig;
ADS7142SingleRegisterRead(ADS7142_REG_nCLK_SEL, &nCLKconfig);

//Select the Data Buffer output data Configuration
ADS7142SingleRegisterWrite(ADS7142_REG_DOUT_FORMAT_CFG, ADS7142_VAL_DOUT_FORMAT_CFG_DOUT_FORMAT2);

//Select the Data Buffer opmode for Stop Burst mode
ADS7142SingleRegisterWrite(ADS7142_REG_DATA_BUFFER_OPMODE,
ADS7142_VAL_DATA_BUFFER_STARTSTOP_CNTRL_STOPBURST);

//Set the SEQ_START Bit to begin the sampling sequence
ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);

//Begin Autonomous Mode StopBurst operation and scan channels 0 and 1

while (1)
{

//Abort the sequence
ADS7142SingleRegisterWrite(ADS7142_REG_ABORT_SEQUENCE, ADS7142_VAL_ABORT_SEQUENCE);

//Read the databuffer status
uint32_t databufferstatus;
ADS7142SingleRegisterRead(ADS7142_REG_DATA_BUFFER_STATUS, &databufferstatus);

//Read the databuffer
while(ADS7142DataRead_count(16) < 0);
```

```
//Restart the sequence
ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);

}

//Return no errors
return 0;

}
```

Figure 30 shows the channel configuration, selection of the opmode, and selection of the oscillator for this ADS7142 functional mode.

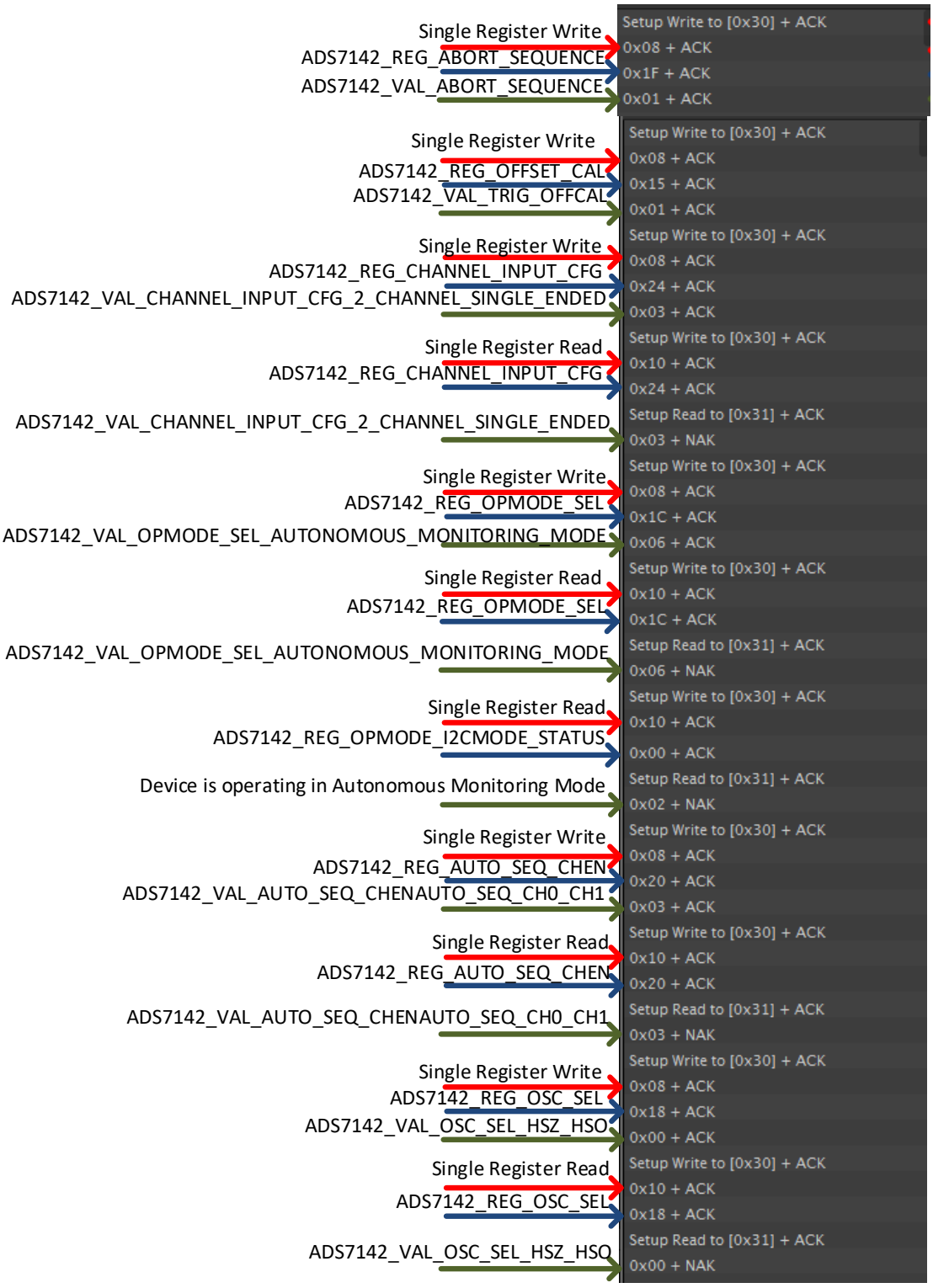


Figure 30. Autonomous Mode Stop Burst Dual-Channel Sampling Test Data 1

Figure 31 shows the setting of the number of clock cycles required to perform a conversion, the data output format, and the contents of the data buffer after the conversion sequence is aborted.

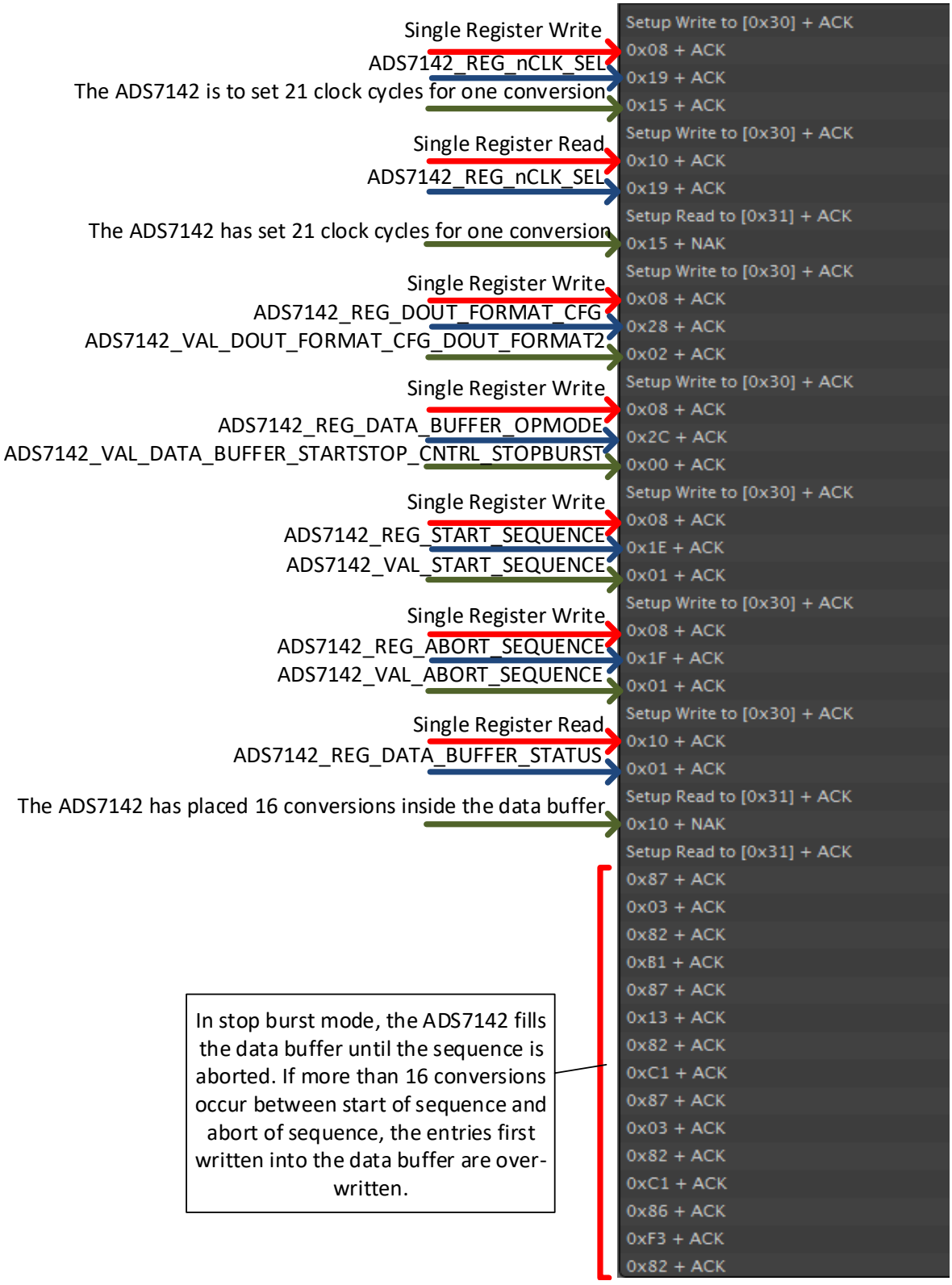


Figure 31. Autonomous Mode Stop Burst Dual-Channel Sampling Test Data 2

Figure 32 shows the continued data buffer results and the conversion sequence restart.

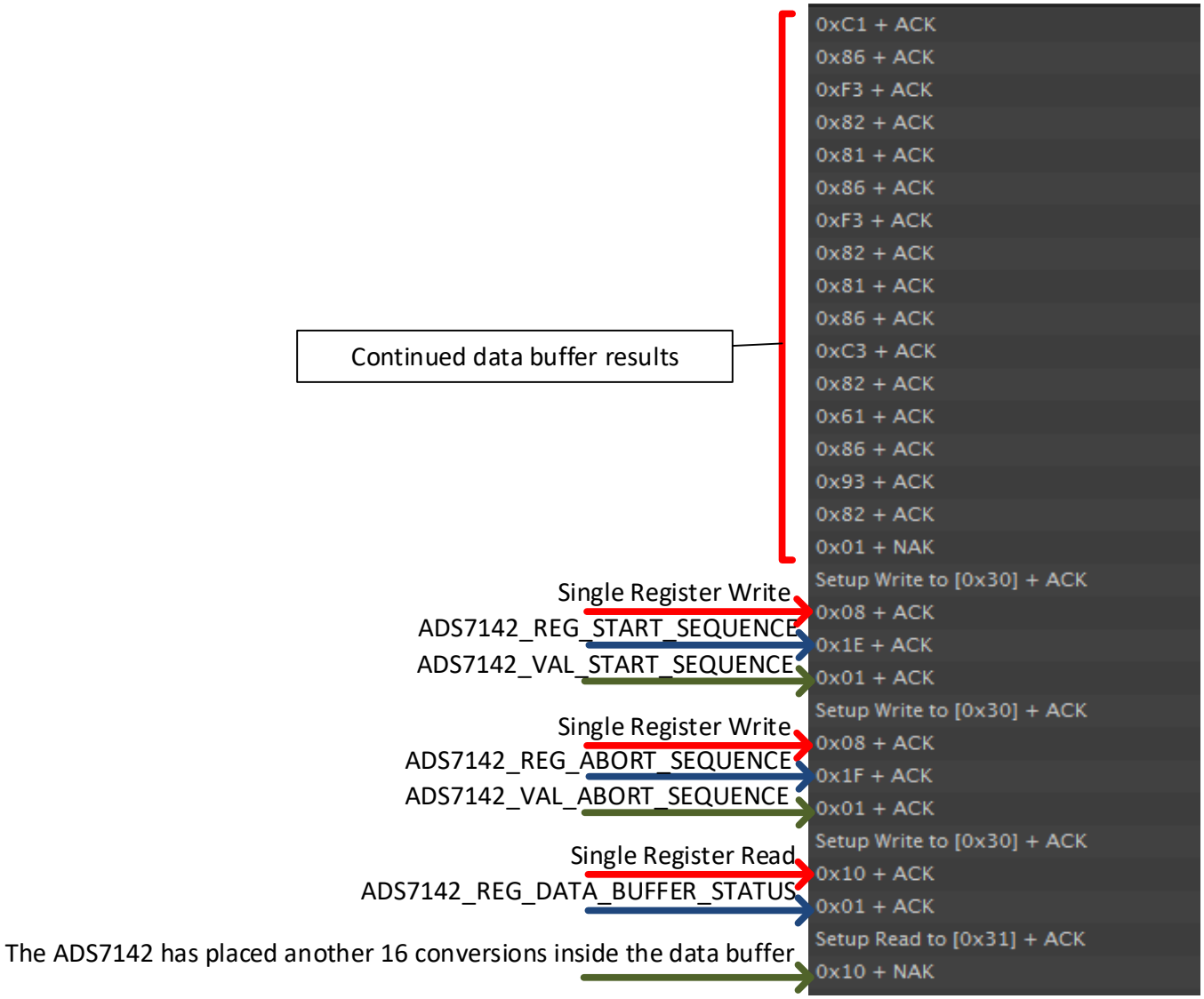


Figure 32. Autonomous Mode Stop Burst Dual-Channel Sampling Test Data 3

5.6 High Precision Mode

High precision mode increases the accuracy of the data measurement to 16-bit accuracy. In this mode, 16 12-bit conversions are placed into an accumulator that sums up the conversions in the accumulator. Equation 2 calculates the high precision data for each channel.

$$\text{High Precision Data for CHx} = \sum_{k=1}^{16} \text{Conversion Result[k] for CHx} \tag{2}$$

Each channel has an accumulator. Figure 33 shows the high precision mode conversion dynamics.

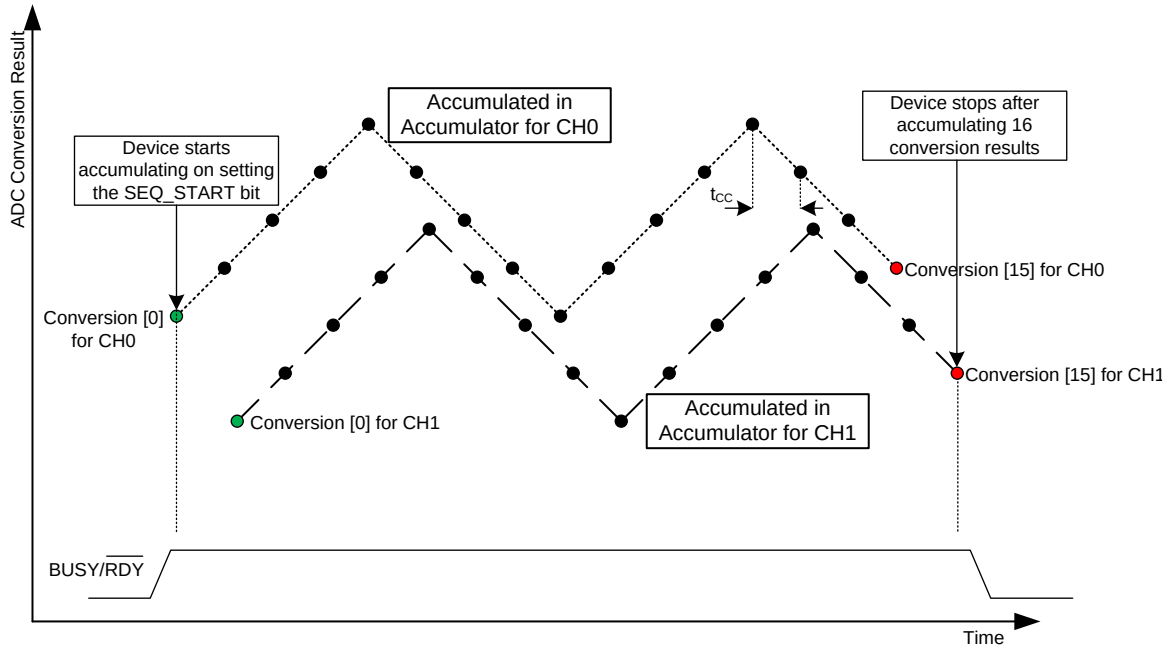


Figure 33. High Precision Mode With Dual-Channel Configurations

The main routine is the following:

```
#include "ADS7142RegisterMap.h"

/*
ADS7142_HighPrecisionMode_AutoSequencing_CH0_CH1_Scan.c
*/

int
main(void)

{

//Initialize the master MCU (0 = 100 kHz SCL, 1 = 400 kHz SCL)
TM4C1294Init(0);

//Calibrate the offset out of ADS7142
ADS7142Calibrate();

//Let's put the ADS7142 into High Precision Mode with both channels enabled in Single-
Ended Configuration
//Select the channel input configuration
ADS7142SingleRegisterWrite(ADS7142_REG_CHANNEL_INPUT_CFG,
ADS7142_VAL_CHANNEL_INPUT_CFG_2_CHANNEL_SINGLE_ENDED);

//Confirm the input channel configuration
uint32_t channelconfig;
ADS7142SingleRegisterRead(ADS7142_REG_CHANNEL_INPUT_CFG, &channelconfig);
```

```
//Select the operation mode of the device
ADS7142SingleRegisterWrite(ADS7142_REG_OPMODE_SEL, ADS7142_VAL_OPMODE_SEL_HIGH_PRECISION_MODE);

//Confirm the operation mode selection
uint32_t opmodeselconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_SEL, &opmodeselconfig);

//Set the I2C Mode to High Speed (optional)
//ADS7142HighSpeedEnable(ADS7142_VAL_OPMODE_I2CMODE_HS_1);

//Check the I2C Mode Status
uint32_t opmodei2cconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OPMODE_I2CMODE_STATUS, &opmodei2cconfig);

//Auto Sequence both channels 0 and 1
ADS7142SingleRegisterWrite(ADS7142_REG_AUTO_SEQ_CHEN, ADS7142_VAL_AUTO_SEQ_CHENAUTO_SEQ_CH0_CH1);

//Confirm Auto Sequencing is enabled
uint32_t autoseqchenconfig;
ADS7142SingleRegisterRead(ADS7142_REG_AUTO_SEQ_CHEN, &autoseqchenconfig);

//Select the Low Power Oscillator or high speed oscillator
ADS7142SingleRegisterWrite(ADS7142_REG_OSC_SEL, ADS7142_VAL_OSC_SEL_HSZ_HSO);

//Confirm the oscillator selection
uint32_t oscselconfig;
ADS7142SingleRegisterRead(ADS7142_REG_OSC_SEL, &oscselconfig);

//Set the minimum nCLK value for one conversion to maximize sampling speed
ADS7142SingleRegisterWrite(ADS7142_REG_nCLK_SEL, 21);

//Confirm the nCLK selection
uint32_t nCLKselconfig;
ADS7142SingleRegisterRead(ADS7142_REG_nCLK_SEL, &nCLKselconfig);

//Enable the accumulator
ADS7142SingleRegisterWrite(ADS7142_REG_ACC_EN, ADS7142_VAL_ACC_EN);

//Set SEQ_START Bit to start the sampling sequence
ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);

//Begin High Precision Mode Scanning Ch0 and Ch1 continuously

while(1)
{

//Sample 16 conversions from each channel
while (ADS7142DataRead_count(32) < 0);

//Check the Accumulator Status to count the number of conversions complete
uint32_t accstatus;
ADS7142SingleRegisterRead(ADS7142_REG_ACCUMULATOR_STATUS, &accstatus);

//Read the MSB of Ch0 Accumulated Data after 16 accumulations are complete
uint32_t accch0MSB;
ADS7142SingleRegisterRead(ADS7142_REG_ACC_CH0_MSB, &accch0MSB);

//Read the LSB of Ch0 Accumulated Data after 16 accumulations are complete
uint32_t accch0LSB;
ADS7142SingleRegisterRead(ADS7142_REG_ACC_CH0_LSB, &accch0LSB);

//Read the MSB of Ch1 Accumulated Data after 16 accumulations are complete
uint32_t accch1MSB;
ADS7142SingleRegisterRead(ADS7142_REG_ACC_CH1_MSB, &accch1MSB);

//Read the LSB of Ch1 Accumulated Data after 16 accumulations are complete
```

```
    uint32_t accch1LSB;
    ADS7142SingleRegisterRead(ADS7142_REG_ACC_CH1_LSB, &accch1LSB);

    //Set the SEQ_START Bit again
    ADS7142SingleRegisterWrite(ADS7142_REG_START_SEQUENCE, ADS7142_VAL_START_SEQUENCE);

    }

    //Return no errors
    return 0;

    }
```

Figure 34 shows the channel configuration and the selection of the opmode and oscillator for this ADS7142 functional mode.

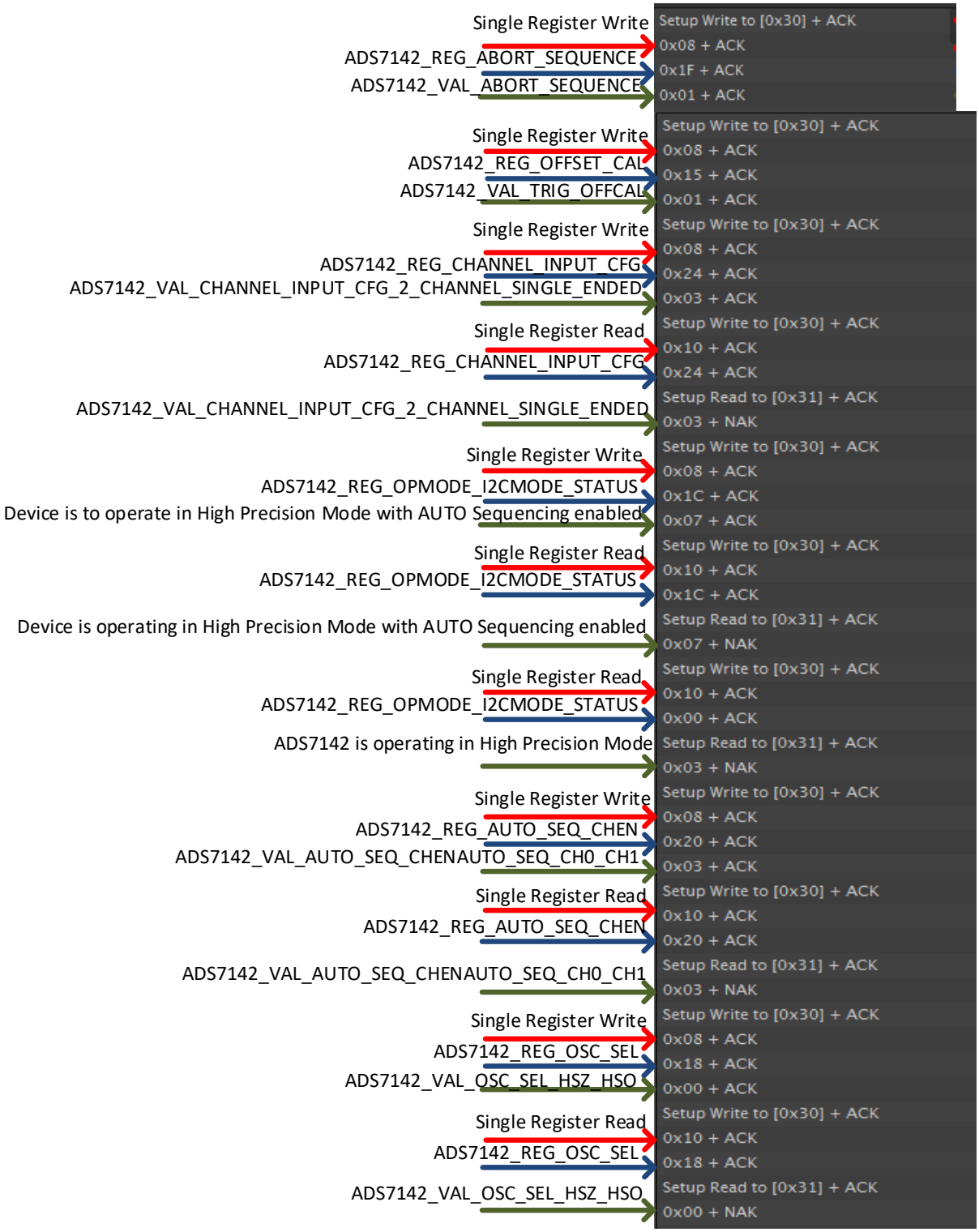


Figure 34. High Precision Mode Dual-Channel Sampling Test Data 1

Figure 35 shows the setting of clock cycles required for a conversion, the enabling of accumulators, and the start of the conversion sequence.

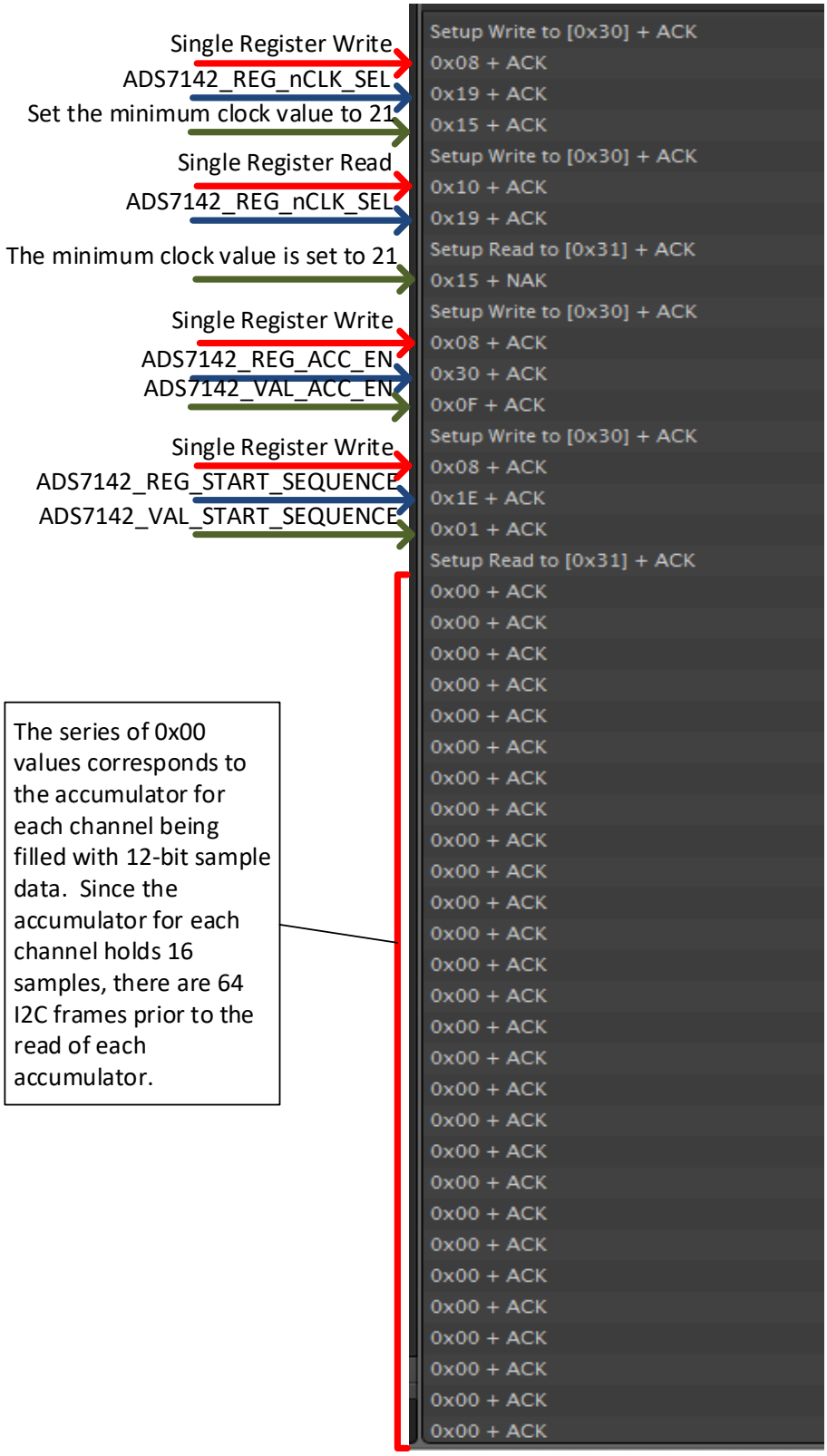


Figure 35. High Precision Mode Dual-Channel Sampling Test Data 2

Figure 36 shows the number of conversions that fill the accumulator and the results of the high precision accumulation.

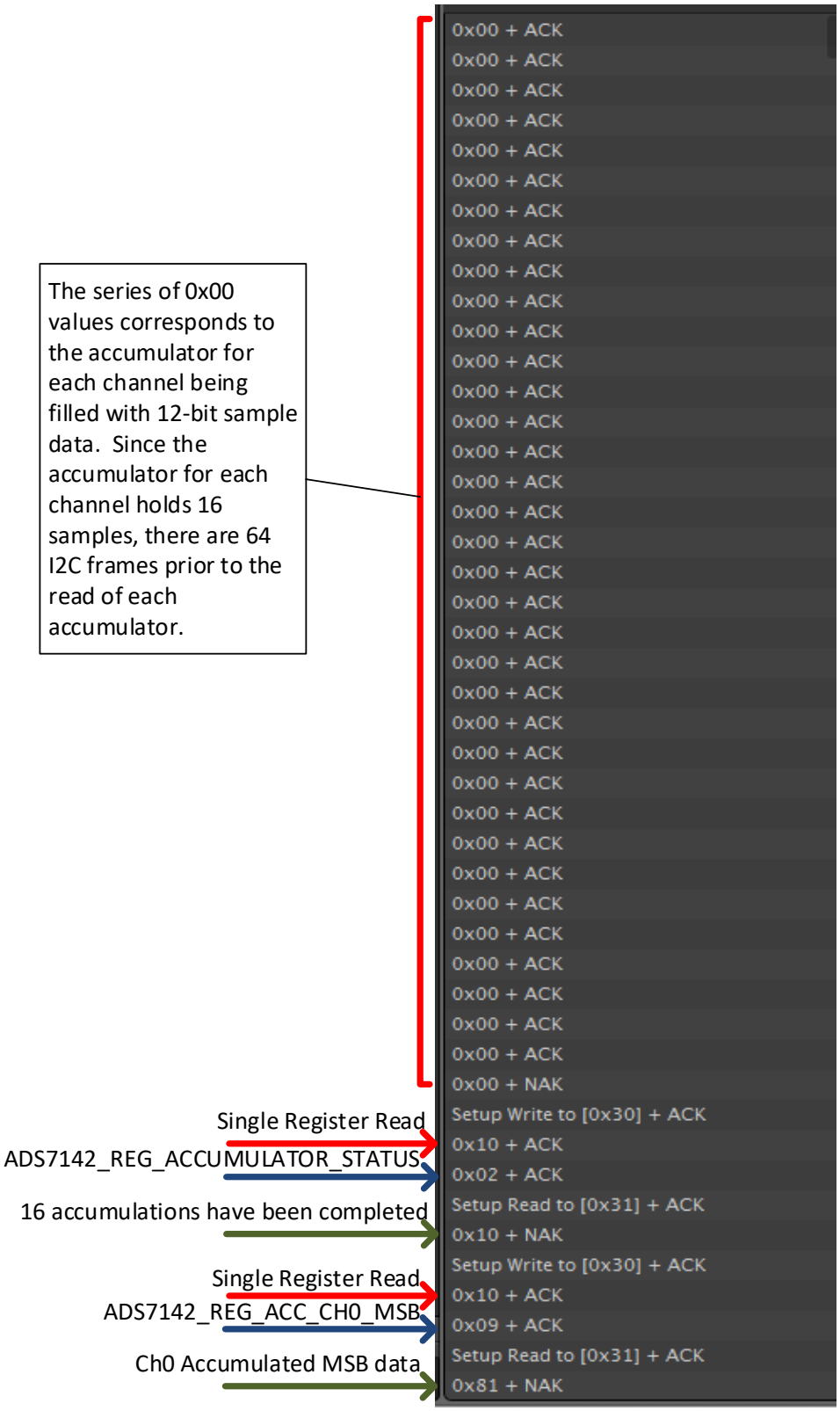


Figure 36. High Precision Mode Dual-Channel Sampling Test Data 3

Figure 37 shows the accumulated results and restart of the conversion sequence.

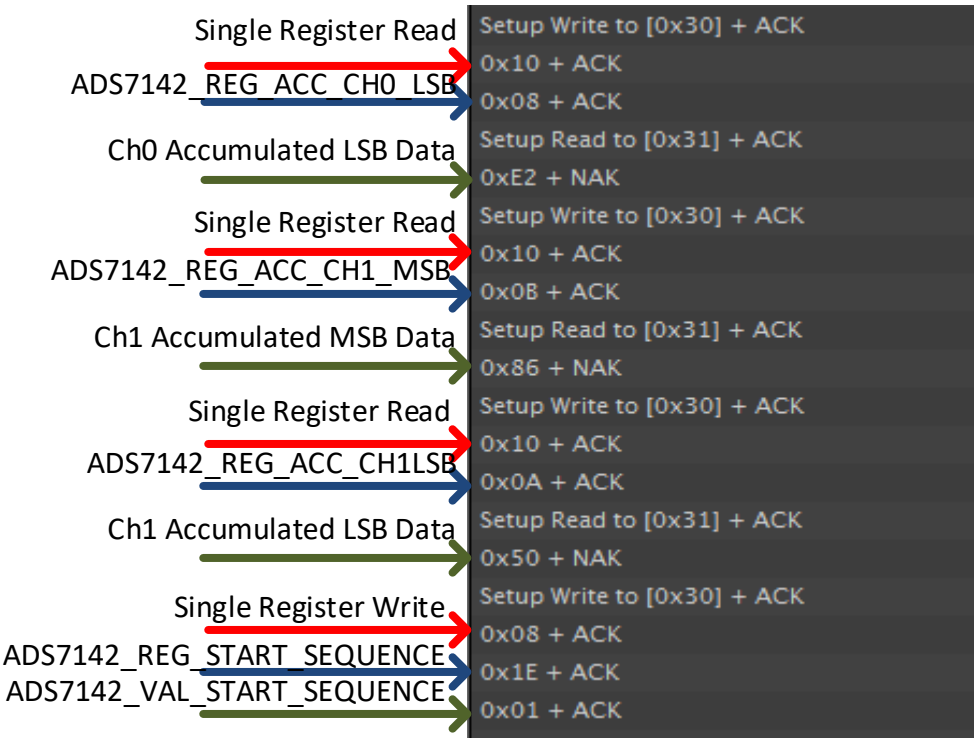


Figure 37. High Precision Mode Dual-Channel Sampling Test Data 4

6 References

1. *ADS7142 Nanopower, Dual-Channel, Programmable Sensor Monitor Data Sheet*
2. *ADS7142BoosterPack User's Guide*
3. *Tiva TM4C1294NCPDT Microcontroller Data Sheet*
4. *Tiva C Series TM4C1294 Connected LaunchPad Evaluation Kit User's Guide*
5. *Tiva C Series TM4C129x Microcontrollers Silicon Revisions 1, 2, and 3* Silicon Errata
6. TI I^2C Tips
7. *TivaWare Peripheral Driver Library* , SW-TM4C-DRL-UG-2.1.3.156
8. *Building a gateway to the Internet of Things*
9. TI Precision Labs
10. Salae Logic Analyzer Landing Page)

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Original (July 2018) to A Revision	Page
• Added author names to document	1

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265