16AI32SSC

16-bit, 32 channel, 200K S/S/Ch A/D Input

PMC66-16AI32SSC

snapshot Test Results

Manual Revision: July 1, 2024

General Standards Corporation
8302A Whitesburg Drive
Huntsville, AL 35802
Phone: (256) 880-8787
Fax: (256) 880-8788
URL: http://www.generalstandards.com
E-mail: sales@generalstandards.com
E-mail: support@generalstandards.com

Preface

Additional copies of this manual or other literature may be obtained from:

General Standards Corporation
8302A Whitesburg Dr.
Huntsville, Alabama 35802
Phone: (256) 880-8787
FAX: (256) 880-8788
URL: http://www.generalstandards.com
E-mail: sales@generalstandards.com

Table of Contents

Table of Figures

1. Introduction

1.1. Purpose

The purpose of the `snapshot` sample application is to demonstrate, for comparison purposes, the relative achievable performance for applications required to take snapshots of the analog inputs at a specified periodic rate. The application implements several methods, selectable from the command line, for detecting and retrieving ADC data. Command line options also permit selection of the periodic rate, the number of active channels and use of Data Packing (which is not usable with the Low Latency feature).

1.2. Host

The host on which the tests were performed was a relatively high-end desktop PC. The host included a single Intel Core i7-6900K CPU running at 3.2GHz with Hyperthreading enabled and with 32GB of RAM.

1.3. Operating System

The operating system was a virgin installation of Red Hat Fedora Core 38, 64-bit.

1.4. Device Driver

The driver installed was the 16AI32SSC Linux driver, version 2.8.111.50.0.

1.5. Test Hardware

All testing was performed using a PMC66-16AI32SSC-32-50-LL.

1.6. Test Software

The test software, the `snapshot` test application, is primarily designed to obtain the very latest analog input data scan at a fixed input sample rate. The secondary objective of the application is to measure the sustainable input sample rate under varying conditions. These conditions notably include the number of active channels, the desired sample rate, the use of Data Packing as well as the method used to detect the availability of data and of data retrieval. The detection options include both polling methods and interrupt driven methods. The retrieval options include both the API's read service as well as the Low Latency Read service. For each specified operating configuration, the application either tests a specified sample rate or it sweeps the sample rate over a range, as dictated by command line arguments. At each step, the requested and achieved sample rates are recorded for later analysis. In this instance, the range tested was from 1 S/S through 50,000 S/S, in increments of 100 S/S. The application is included as part of the 16AI32SSC Linux driver archive.

> **NOTE**: For all tests, the Read I/O Timeout is set to zero seconds. This prevents the read service from blocking in order to wait for additional data to be captured into the Input Buffer.

> **NOTE**: The API's read service retrieves data using PIO due to the small number of samples involved. With the small number of samples involved, PIO is more efficient than either DMA option.

2. Data Detection and Retrieval

2.1. Data Detection and Retrieval Methods

The following are the terms used to identify the means of detecting when data is present in the Analog Input Buffer and how the data is retrieved. Scan data is essentially injected into the Analog Input Buffer (a FIFO) all at once. Even if software's query for the FIFO status came when only a portion of a scan were in the FIFO, the entire scan would be in the FIFO well before application of driver software could turn around and retrieve it.

2.1.1. Poll-FIFO

This method polls for scan data availability by examining the Analog Input Buffer status, then retrieves the scan data by calling the API's read service (`ai32ssc_read()`). This polling option offers very finely grained detection of scan data, time wise, because checking the FIFO status should take only a very few microseconds. However, because it is polling based it puts a high load on the CPU.

2.1.2. Poll-FIFO w/ DP

This method is identical to the above option except that Data Packing is enabled. This method offers some performance improvement, but it still puts a high load on the CPU because it is polling based.

2.1.3. Poll-Read

This method polls for and retrieves scan data by calling the API's read service (`ai32ssc_read()`). With the I/O Timeout being zero, the service never sleeps to wait for data to appear in the FIFO. Instead, the service returns immediately when no data is present. But, as soon as data is available, the read service retrieves it from the FIFO and returns it to the caller. This method is inefficient due it being polling based. Also, this method is less efficient than the Poll-FIFO method because the read service takes longer to execute than just checking the FIFO status.

2.1.4. Poll-Read w/ DP

This method is identical to the above option except that Data Packing is enabled. Unfortunately, this method does not appear to offer increased performance. This is mainly because of its relatively high data detection overhead.

2.1.5. Wait-Read

This method uses the Scan Complete interrupt to detect availability of scan data, then retrieves that data by calling the API's read service (`ai32ssc_read()`). This detection method puts the least possible load on the CPU. Because of the ISR however, there is a 5 µs to 10 µs delay before the API's read service is called.

2.1.6. Wait-Read w/ DP

This method is identical to the above option except that Data Packing is selected. This method offers some performance improvement with slightly reduced CPU loading.

2.1.7. Wait-LL

This method uses the Scan Complete interrupt to detect availability of scan data, but uses the Low Latency feature to retrieve that data. This method also puts the least possible load on the CPU for data detection. The CPU load for its data retrieval is slightly reduced compared to the API's read service. Again however, because of the ISR, there is a 5 µs to 10 µs delay before the Low Latency service can be exercised to retrieve the data.

2.2. Method Comparison

The figure below, Figure 1, illustrates how the different detection and retrieval methods offer differing levels of performance capabilities. Each method offers stable performance matching the requested sample rate up to a point. Beyond that point, each method is unable to maintain the requested sample rate. (The graphs shown employ data smoothing for clarify. Without smoothing, the graphs mostly tend to produce very erratic results once they depart from the requested sample rate.)

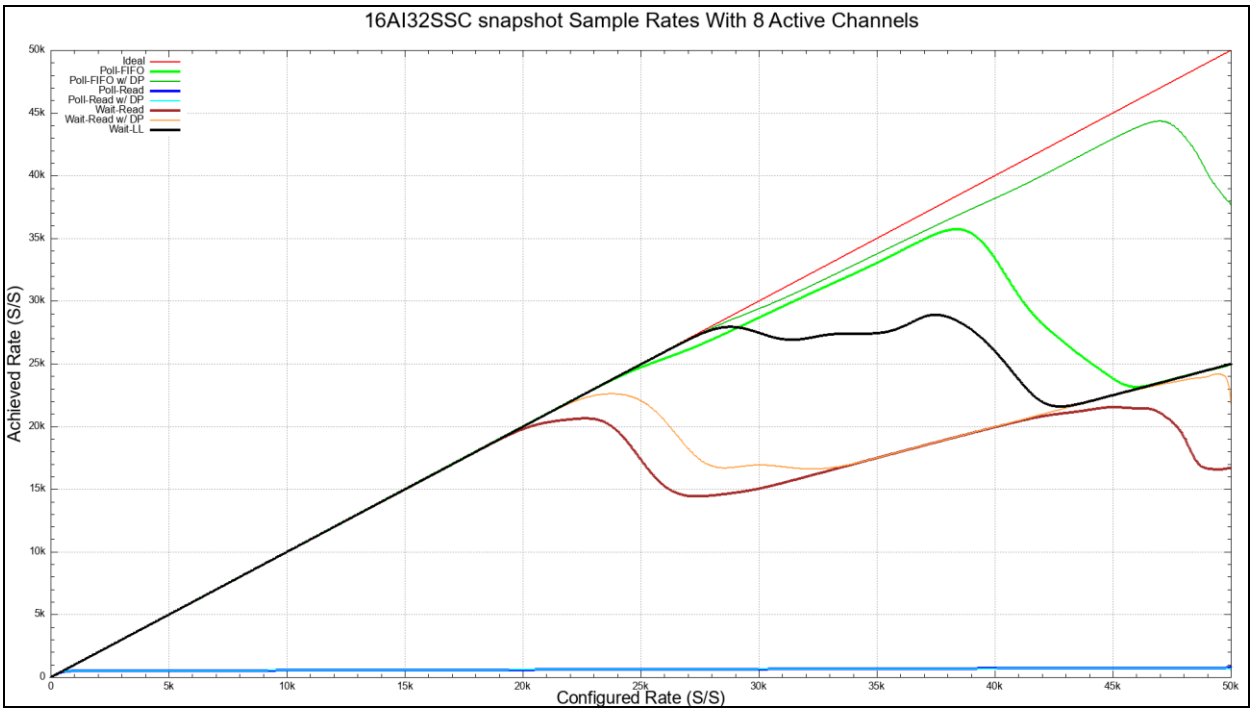

Figure 1 Data detection and retrieval method comparison with eight active channels.

> **NOTE**: The graphs represent the achieved performance when detecting and reading data with the respective methods. Once retrieved though, the data is discarded. Real life applications should expect different performance levels because they must do something with the retrieved data and would be using an entirely different host.

2.3. Active Channel Comparison

The figure below, Figure 2, illustrates the impact that the number of active channels has on performance achieved. As shown, as the number of active channels increases, the achievable performance rate decreases. (The graphs shown employ data smoothing for clarify. Without smoothing, the graphs mostly tend to produce very erratic results once they depart from the requested sample rate.)

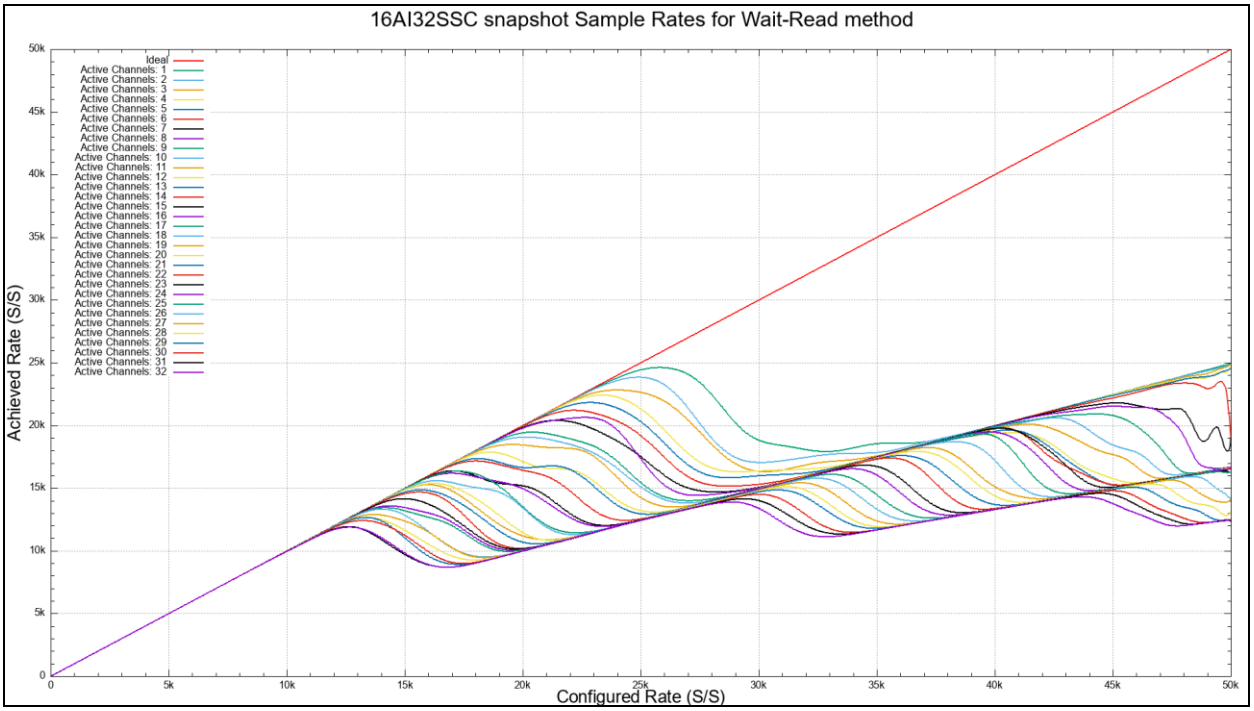

Figure 2 Active channel count comparison.

Document History

Revision	Description
July 1, 2024	Initial release.