# Storing Measurement Data

File I/O records or reads data in a file. A typical file I/O operation involves the following process.

1. Create or open a file. Indicate where an existing file resides or where you want to create a new file by specifying a path or responding to a dialog box to direct LabVIEW to the file location. After the file opens, a *refnum* represents the file.
2. Read from or write to the file.
3. Close the file.
4. Check for Errors.

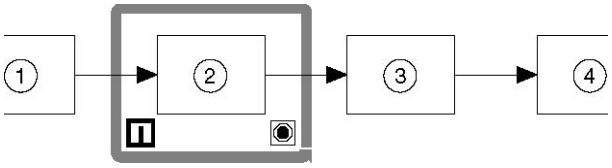



Figure 1: Steps in a Typical File I/O Operation

## File Formats

At their lowest level, all files written to your computer's hard drive are a series of binary bits. However, many formats for organizing and representing data in a file are available. In LabVIEW, three of the most common techniques for storing data are **direct binary storage**, **text file format**, and the **TDM file format**. Each of these formats has advantages and some formats work better for storing certain data types than others. We will discuss the advantages in chapter "File Format Selection" on page 3.

- **Binary**—Binary files are the underlying file format of all other file formats.
- **ASCII**—An ASCII file is a specific type of binary file that is a standard used by most programs. It consists of a series of ASCII codes. ASCII files are also called text files. The LabVIEW measurement data file (**.lvm**) is a tab-delimited text file you can open with a spreadsheet application or a text-editing application. The .lvm file includes information about the data, such as the date and time the data was generated. This file format is a specific type of ASCII file created for LabVIEW.
- **TDM**—This file format is a specific type of binary file created for National Instruments products. It actually consists of two separate files: an XML section contains the data attributes, and a binary file for the waveform.

## Understanding High-Level File I/O

Some File I/O VIs perform all three steps of a file I/O process: open, read/write, and close. If a VI performs all three steps, it is referred to as a **high-level VI**. However, these VIs may not be as efficient as the low-level VIs and functions designed for individual parts of the process. If you are writing to a file in a loop, use **low-level file I/O VIs**. If you are writing to a file in a single operation, you can use the high-level file I/O VIs instead.

High-level file I/O VIs include the following:

- **Write to Spreadsheet File**—Converts a 2D or 1D array of single-precision numbers to a text string and writes the string to a new ASCII file or appends the string to an

existing file. You also can transpose the data. The VI opens or creates the file before writing to it and closes it afterwards. You can use this VI to create a text file readable by most spreadsheet applications.

- **Read From Spreadsheet File**—Reads a specified number of lines or rows from a numeric text file beginning at a specified character offset and converts the data to a 2D single-precision array of numbers. The VI opens the file before reading from it and closes it afterwards. You can use this VI to read a spreadsheet file saved in text format.
- **Write to Measurement File**—An Express VI that writes data to a text-based measurement file (.lvm) or a binary measurement file (.tdm) format. You can specify the save method, file format (.lvm or .tdm), header type, and delimiter.
- **Read from Measurement File**—An Express VI that reads data from a text-based measurement file (.lvm) or a binary measurement file (.tdm) format. You can specify the file name, file format and segment size.

**Tip:** Avoid placing the high-level VIs in loops, because the VIs perform open and close operations each time they run.

## Low-Level File I/O

Low-level file I/O VIs and functions each perform only one piece of the file I/O process. For example, there is one function to open an ASCII file, one function to read an ASCII file, and one function to close an ASCII file. Use low-level functions when file I/O is occurring within a loop, especially with high loop rates.

### Disk Streaming with Low-Level Functions

Disk streaming is a technique for keeping files open while you perform multiple write operations, for example, within a loop. Wiring a *path* control or a constant to the Write to Text File function, the Write to Binary File function, or the Write to Spreadsheet File VI adds the overhead of opening and closing the file each time the function or VI executes.

VIs can be more efficient if you avoid opening and closing the same files frequently. **To avoid opening and closing the same file, you need to pass a *refnum* to the file into the loop.** When you open a file, device, or network connection, LabVIEW creates a *refnum* associated with that file, device, or network connection. All operations you perform on open files, devices, or network connections use the *refnums* to identify each object.

The examples in figures 2 and 3 show the advantages of using disk streaming. In the figure 2, the VI must open and close the file during each iteration of the loop. The example in figure 3 uses disk streaming to reduce the number of times the VI must interact with the operating system to open and close the file. By opening the file once before the loop begins and closing it after the loop completes, you save two file operations on each iteration of the loop.

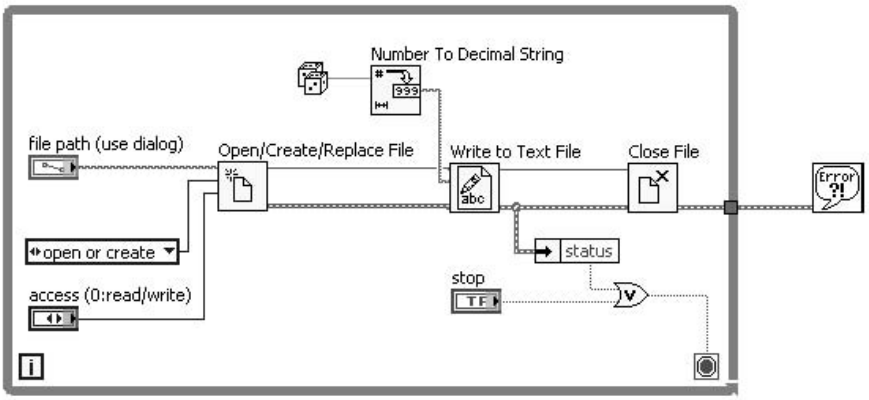



Figure 2: Non-Disk Streaming Example

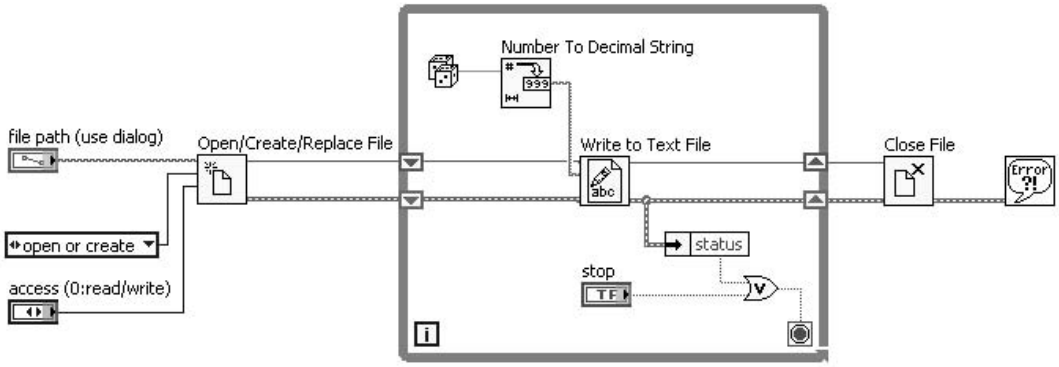



Figure 3: Disk Streaming Example

# File Format Selection

In this section some of the criteria for selecting the best file format are discussed.

## Text (ASCII) Files

Text files are the easiest format to use and to share. Almost any computer can read from or write to a text file. Use text format files for your data to make it available to other users or applications if disk space and file I/O speed are not crucial, if you do not need to perform random access reads or writes, and if numeric precision is not important.

Store data in text files when you want to **access it from another application**, such as a word processing or spreadsheet application. To store data in text format, use the **String** functions to convert all data to text strings. Text files can contain information of different data types.

Text files typically **take up more memory** than binary files if the data is not originally in text form, because the ASCII representation of data usually is larger than the data itself. For example, you can store the number -123.4567 in 4 bytes as a single-precision, floating-point number. However, its ASCII representation takes 9 bytes, one for each character.

In addition, it is **difficult to randomly access** numeric data in text files. Although each character in a string takes up exactly 1 byte of space, the space required to express a number as text typically is not fixed. To find the ninth number in a text file, LabVIEW must first read and convert the preceding eight numbers.

You might **lose precision** if you store numeric data in text files. Computers store numeric data as binary data, and typically you write numeric data to a text file in decimal notation. Loss of precision is not an issue with binary files.

## Binary Files

Storing binary data, such as an integer, uses a fixed number of bytes on disk. For example, storing any number from 0 to 4 billion in binary format, such as 1, 1000, or 1 000 000, takes up 4 bytes for each number.

Use binary files to save numeric data and to access specific numbers from a file or randomly access numbers from a file. Binary files are machine readable only, unlike text files, which are human readable. Binary files are the most compact and fastest format for storing data. You can use multiple data types in binary files, but it is uncommon.

Binary files are more efficient because they use less disk space and because you do not need to convert data to and from a text representation when you store and retrieve data. A binary file can represent 256 values in 1 byte of disk space. Often, binary files contain a byte-for-byte image of the data as it was stored in memory, except for cases like extended and complex numeric values. When the file contains a byte-for-byte image of the data as it was stored in memory, reading the file is faster because conversion is not necessary.

## Datalog Files

A specific type of binary file, known as a datalog file, is the easiest method for logging cluster data to file. Datalog files store arrays of clusters in a binary representation. Datalog files provide efficient storage and random access, however, the storage format for datalog files is complex, and therefore they are difficult to access in any environment except LabVIEW. Furthermore, in order to access the contents of a datalog file, you must know the contents of the cluster type stored in the file. If you lose the definition of the cluster, the file becomes very difficult to decode. For this reason, datalog files are not recommended for sharing data with others or for storing data in large organizations where you could lose or misplace the cluster definition.

## TDM Files

Test Data Exchange Format (TDM) is a hybrid file format that combines binary storage and XML formatted ASCII data. In a TDM file, the raw numerical data is stored in binary format. This provides the advantages of the binary file format, such as efficient space usage and fast write times. In addition to this data, an XML format stores the structure of the data and information about the data. This allows the information in the file to be easily accessible and searchable. Typically, the binary data and XML data are separated into two files, a `.tdm` file for the XML data and a `.tdx` file for the binary data.

TDM files are designed for **storing test or measurement data**, especially when the data consists of one or more arrays. TDM files are most useful when storing arrays of simple data types such as numbers, strings, or Boolean data.

TDM Files allow you to **create a structure for your data**. Data within a file is organized into channels. You also can organize channels into channel groups. A file can contain multiple channel groups. Well-grouped data simplifies viewing and analysis and can reduce the time required to search for a particular piece of data.

Use TDM files when you want to **store additional information about your data**. For example, you might want to record the following information:

- type of tests or measurements
- operator or tester name
- time of the test

TDM files each contain a File object and can contain as many Channel Group and Channel objects as you want. Each of the objects in a file has properties associated with it, which creates three levels of properties that you can use to store data. For example, test conditions are stored at the file level. UUT information is stored at the channel or channel group level. Storing plenty of information about your tests or measurements can make analysis easier, and also allows you to search for specific data sets. Searching for specific files or data sets based upon stored criteria is important when you gather large amounts of data—especially when you may need to share the data with others.

Searching for data in a file based upon one or more conditions is a feature of the TDM data storage format. With most file formats, you must read the entire file into a program and then programmatically search for certain fields in the file in order to locate a specific set of data. With a TDM file you can specify a condition when you read data, and the read returns only data that matches that condition. By using multiple reads and merging their results together you can construct complex queries for your data. You can use any property of the TDM File, Channel Group, or Channel as a query condition. Therefore, enter as many properties as possible when logging TDM files. These properties simplify locating data.

Like many binary file formats, only programs specifically designed to recognize and decode them can read TDM files. LabVIEW, LabWindows™/CVI™, DIAdem, and some other NI software can read TDM files. Furthermore, from ni.com you can download a free plug-in to MS Excel for importing TDM files.

To access your data using other software you should use a more universal format such as ASCII.

# Programming considerations

## Binary Files

Although all file I/O methods eventually create binary files, you can directly interact with a binary file by using the Binary File functions. The following list describes the common functions that interact with binary files.

Figure 4 shows an example that writes an array of doubles to a binary file.

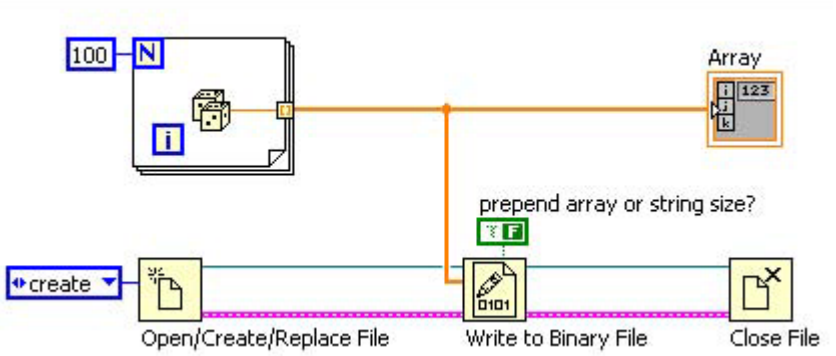



Figure 4: Writing a Binary File

**Open/Create/Replace File**—Opens a reference to a new or existing file for binary files as it does for ASCII Files.

**Write to Binary File**—Writes binary data to a file. The function works much like the Write to Text File function, but can accept most data types.

**Close File**—Closes an open reference to a file.

## Random Access

To randomly access a binary file, use the Set File Position function to set the read offset to the point in the file you want to begin reading. Notice that the offset is in bytes. Therefore, you must calculate the offset based upon the layout of the file. In Figure 5, the VI returns the array item with the index specified, assuming that the file was written as a binary array of double-precision numerics with no header, like the one written by the example in Figure 4.

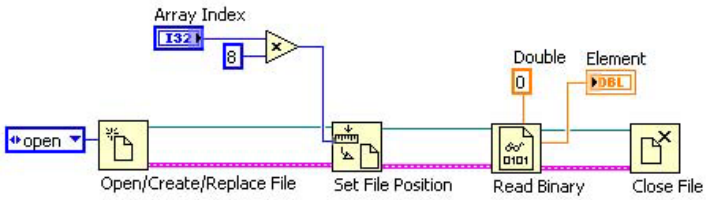



Figure 5: Randomly Accessing a Binary File

**Get/Set File Position**—These functions get and set the location in the file where reads and writes occur. Use these functions for Random File Access.

## TDM Files

In LabVIEW, you can create TDM Files in two ways. Use the Write to Measurement File Express VI and Read from Measurement File Express VI or the Data Storage API VIs.

The Express VIs allow you to quickly save and retrieve data from the TDM format. Figure 5-11 is the configuration dialog box for the Write to Measurement File Express VI. Notice that you can choose to create a LVM or a TDM file type. However, these Express VIs give you little control over your data grouping and properties and do not allow you to use some of the features that make TDM files useful, such as searching for data based on conditions.

To gain access to the full capabilities of TDM files, use the Data Storage API. The Data Storage API is a set of VIs that can write multiple file formats, however, they write TDM files by default. Figure 6 shows an example of a simple program that logs channel properties and numeric data to a TDM File using the Data Storage API.

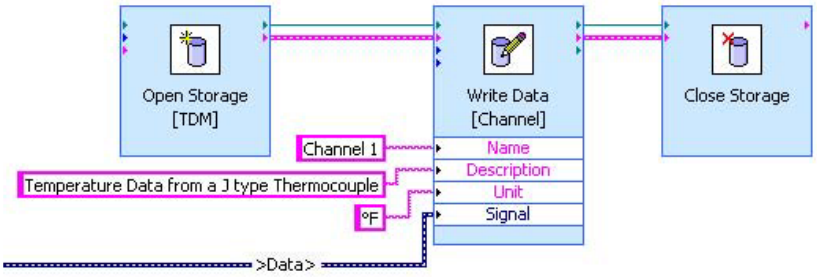


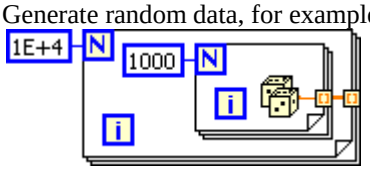

Figure 6: Using the Data Storage API to Write a Simple TDM File

To learn more about programming TDM files, refer to the article "*Writing TDM and TDMS Files in LabVIEW*" at http://zone.ni.com.

## Exercise – File I/O Performance Comparison

Create a VI that measures the time to file writes with three methods, ASCII, Binary and TDMS. To keep the results comparable use low-level file I/O function calls for each method.

1. Generate random data, for example as indicated in the image below.




   This code creates a 10 000 x 1000 matrix of double precision floating point data.

2. Create the timing framework. Since the times measured will be fairly short you should use the Tick Count (ms) function, call it once after you've acquired a file handle but before the repeating file write operations, another time after the file writes are completed but before releasing the file handle, then subtract the values. Use a sequence structure to force the execution order.
3. Each of the File I/O method requires three function calls,
   a. open or create a file and to receive a file handle
   b. write to the file
   c. close the file and release the handle.
4. Place the write to file function inside a For Loop to write the data in buffers (of 1000 samples each, if you used the data matrix indicated above).
5. Remember to wire the *error* and *file handle* parameters from one function to the other, and add an error handler to the end of the function call chain.
6. For easier testing, you may want to include a File Dialog VI to enter the file pathname.

Using the 10 000 x 1000 matrix, what were the file write times for:

ASCII _____

Binary _____

TDMS _____

Change the generated array dimensions to 100 000 x 1, run the same tests. What values did you receive now?

ASCII _____

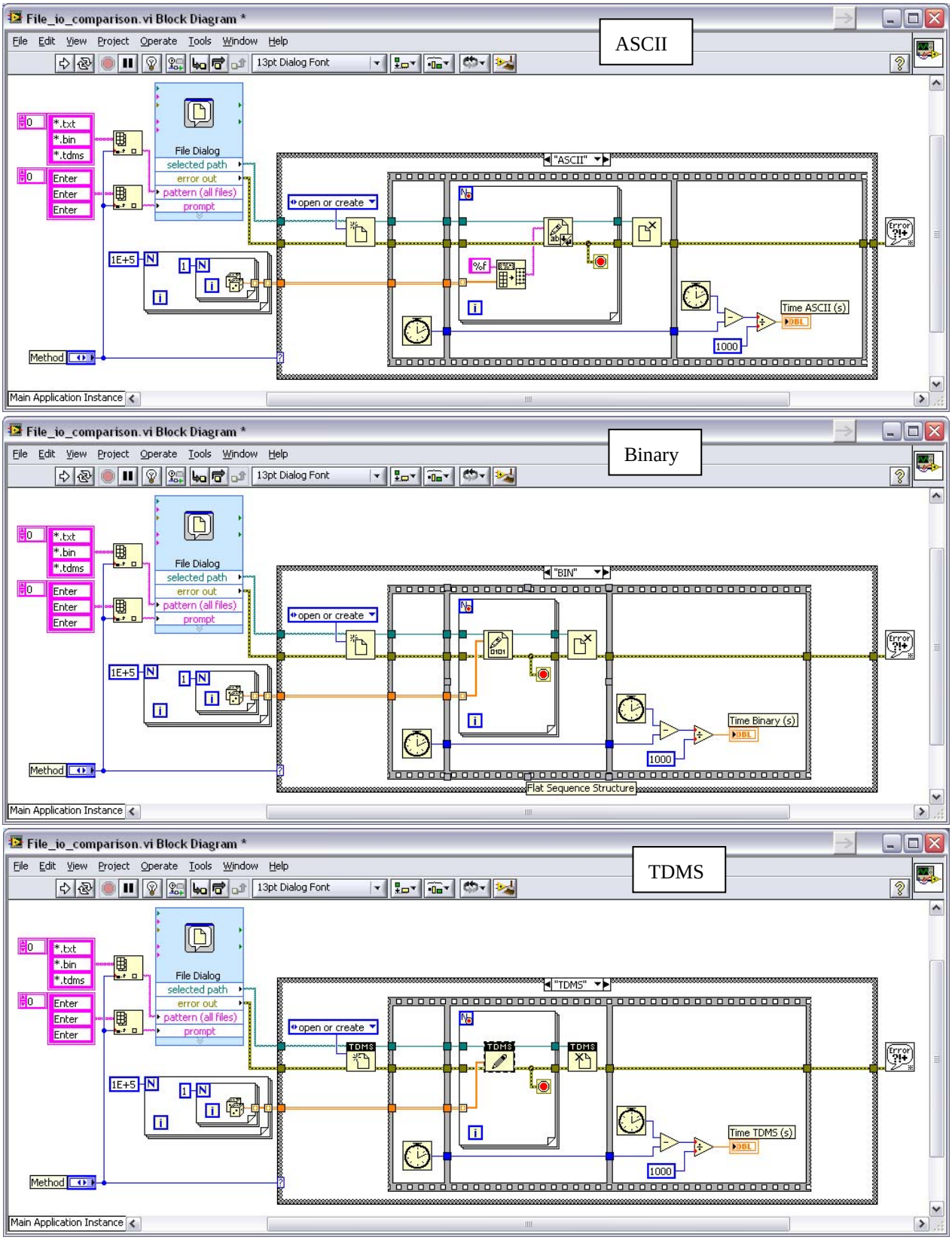
Binary _____

TDMS _____

Do you have an explanation for this (hint: look at the TDMS Write function parameters and the function help )?

# Solution




Explanation to why the 100 000 x 1 matrix takes longer time to write with TDMS: the file write function will need to update the channel parameters in every iteration. With a matrix of these dimensions you will enter one point of channel data at a time, forcing also the additional parameters to be updated for each data point.