8 FILE INPUT/OUTPUT

INTRODUCTION

Most instrumentation applications require storage of data in files for future use. Often, the data saved will be read by some application, which in turn will analyse the data, and present it in some form, such as graph, chart, etc. It may be combined with data from multiple runs for exhaustive analysis, etc. It is thus important to store the data in some standard format. This may be a format required by the application, or a more general format. LabVIEW allows data to be saved in three types of file formats, viz., ASCII text-format byte stream, binary-format byte stream and LabVIEW measurement files. These formats and various file input–output functions available in LabVIEW are discussed in this chapter.

8.1 FILE FORMATS

ASCII Text-Format Files

ASCII text-format files store data in the ASCII format. This is the simplest format and since most software packages can retrieve data from an ASCII text file, naturally it becomes the most commonly used format, especially in instrumentation applications. Needless to say, one has to convert all data to ASCII strings before saving data. The major disadvantage of this format is that the text files tend to be larger than the other formats. At one time this was a major handicap, but with the ever-increasing capacity of storage media, this is no longer a major handicap. Another disadvantage was speed, since the conversion of binary data (internal format) to ASCII, and the transfer of the larger volume of data to the disk were both slow. This is no longer so with modern systems.

Binary-Format Files

These files store data in binary form and look like an image of the data stored in the computer's memory. These files cannot be read by word processors. Also, it is impossible to retrieve data from these files without detailed knowledge of the file's format. The main advantage of these files is that they are much smaller than text files and reading and writing operations can be done with little data conversion. Binary file format is useful in applications requiring writing and reading of large chunks of data due to its size advantage. However, with the increasing availability of large storage devices at reasonable cost, this advantage is being progressively lost. Also, binary format is often system (or processor) dependent.

LabVIEW Measurement Files

LabVIEW measurement files are special files which are either text-based files (.lvm) or binary files (.tdm or .tdms). Generally, these files are written and read using Express VIs. Here data is stored as a sequence of records of a single arbitrary data type specified during creation of the file. The data is indexed in terms of these records. Each record in a measurement file must have the same data types associated with it. One interesting feature of these files is that timestamps are also included with each record. Also, LabVIEW allows random read access to the binary versions of these files.

What to Use and When?

Text and binary files are both byte stream files, i.e., they store data as a sequence of characters or bytes. Text files are the easiest format to use and to share. Almost any computer can read from or write to a text file. When one is interested in retrieving the data using a word processor or spreadsheet, text files are the best. However, text files take up more space than binary files if the data is not originally in text form, like in the case of graph or chart data. This is because ASCII representation of data usually is larger than the data itself. For example, a 32-bit integer occupies 4-bytes in binary, but may occupy more than double the space in ASCII. Furthermore, it is difficult to randomly access numeric data in text files.

Storing of binary data, such as an integer, takes a fixed number of bytes on disk. Hence binary files are useful in saving and accessing numeric data from a file, and also in random access of data from a file. Binary files are the most compact and fastest format for storing data. However, binary files are machine readable only. They are the most efficient since for numeric data the files contain a byte-by-byte image of the data stored in the memory.

Measurement files may be used only in applications where LabVIEW is used to access and manipulate data. Measurement files when used require little manipulation, thus making access faster during both reading and writing. Data retrieval is simplified since the original blocks of data can be read back as a record without having to read all records that precede it in the file. Random access is thus fast and easy with Measurement files. However, measurement file formats are peculiar to LabVIEW so files are not interchangeable with other applications, and converters have to be written to make them available.

Nowadays, text files (mostly Tab delimited in a spreadsheet format) are almost universally used. The binary format (regular binary or measurement) is used when there are transfer rate bottlenecks. Also, when working with Flash media as in cRIO the storage space is limited and binary files may be preferred. For massive files with indexing, measurement files (binary formats) may be the format of choice.

8.2 FILE I/O FUNCTIONS

The three basic file input and output (I/O) operations to store and retrieve data from files are

- Open an existing file or create a new file
- Write to or read from a file
- Close a file

LabVIEW provides several file I/O functions to write or read. Most applications require only writing or reading of one of the following data types:

- Strings to or from text files
- One-dimensional (1D) or two-dimensional (2D) arrays of single numbers. The most convenient format for these is the spreadsheet file format (tab delimited ASCII). This is the most popular format for data access and an extensive library of VIs is available to support it, both as files records and as strings.
- 1D or 2D arrays of numbers to binary files.

The file I/O sub-palette on the diagram is shown in Fig. 8.1. The file I/O functions comprise high-level and intermediate file function as well as path function and other special file VIs. The high-level VIs are placed at the top row of the palette, intermediate VIs at the second and third rows and the sub-palette for the advanced VIs on the rightmost end of the last row. Path and other special file functions are placed below the intermediate VIs.

The first two polymorphic functions VIs for file I/O can be used for writing and reading string, as well as 1D or 2D data of integers and double in spreadsheet formats. The other two high-level VIs can be used to write and read binary and measurement files. For most applications, the high-level VIs are all that are required. Occasionally one might like to use the Intermediate VIs. The commonly used VIs are briefly discussed below.

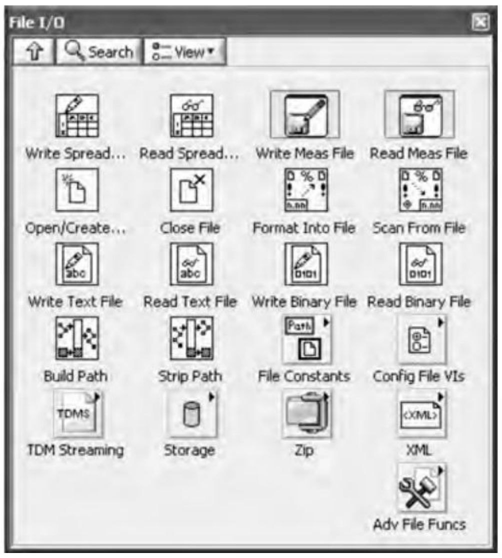


Fig. 8.1 *File I/O palette*

Write to Spreadsheet File

As most data is stored (and read) in a spreadsheet format (line-wise tab delimited ASCII) readymade VIs to handle these are available. The function *Write to Spreadsheet File* (Fig. 8.2) first converts a two-dimensional or a one-dimensional array of numbers (by default real numbers of format x.3f) to a text string separated by tab characters for elements in the same row. This is a polymorphic VI which can choose string, integer or double as inputs. The figure shows the VI for the double instance. The rows are separated by EOLs. The string is then written to the file. This may be a new file or appended to an existing file depending on the status of the Boolean *Append to File*. The facility to transpose data is also available. This is often required to display data collected from two or more channels using DAQ boards, since the graphs and charts require data row-wise whereas the data from multiple channels are stored in the array columnwise. However, it is more common to store data column wise since then the data can be more easily handled outside LabVIEW).

Using the *format* input one can specify how to convert the numbers to characters. The default is %.3f, meaning that the decimal numbers will have a precision of three digits after the decimal point. Various formats are possible using the *format specifier syntax* for strings. The output, *new file path* can be wired to all subsequent writes to the same file as a file handle. Once writing of characters is over, VI automatically closes the file.

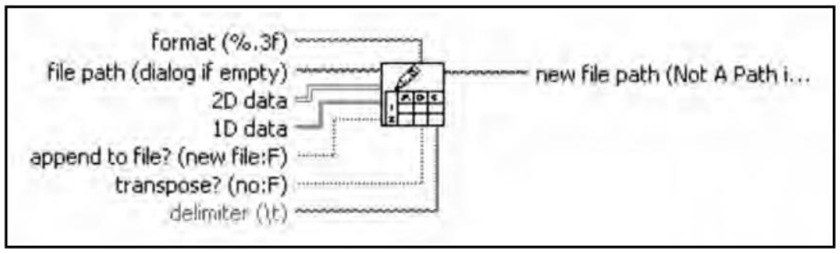

format (%.3f)
file path (dialog if empty)
2D data
1D data
append to file? (new file:F)
transpose? (no:F)
delimiter (\t)
new file path (Not A Path i...

Fig. 8.2 *Write to Spreadsheet File Function*

Read From Spreadsheet File

Data stored in a spreadsheet file can be read using the *Read from Spreadsheet File* VI (Fig. 8.3). When placed on the diagram the instance (double, integer or string) of this VI can be selected using a pull down menu. One can specify the number of rows to be read from the file. If this input is left open, all rows of the file are read. This VI also allows a delimiter to be specified, which may be the character or string of characters, such as tabs, commas, and so on, to use to delimit fields in the spreadsheet file. The default delimiter used is *Tab*. The *all rows* output stands for the data read from the file, while *first row* output is the first row of the *all rows* array. This output may be used when you want to read one row into a one-dimensional array.

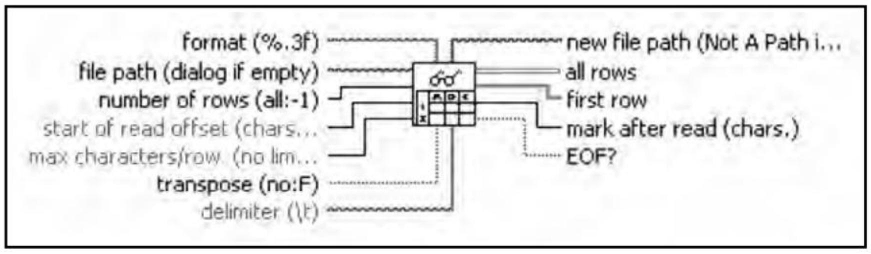

format (%.3f)
file path (dialog if empty)
number of rows (all:-1)
start of read offset (chars...
max characters/row (no lim...
transpose (no:F)
delimiter (\t)
new file path (Not A Path i...
all rows
first row
mark after read (chars.)
EOF?

Fig. 8.3 *Read from spreadsheet file function*

Write to Text File

The *Write to Text File* function intermediate VI (Fig. 8.4) writes a string of characters or an array of strings as lines to a file. If path to the file is not given the VI would open a file dialog. Appending to an existing file can be done by setting the file position to the end of the file by using the *Set File Position* function. Similarly, file position needs to be set appropriately for performing random access file reads or writes.

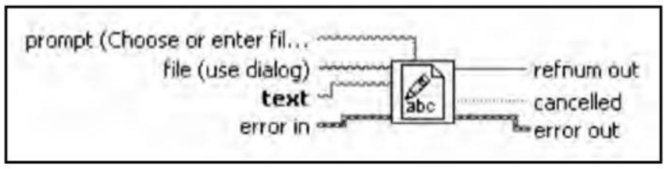


Fig. 8.4 *Write to Text File Function*

Read from Text File

The *Read from Text File* intermediate VI (Fig. 8.5) reads a specified number of characters or lines from a byte stream file. Count input is the maximum number of characters or lines the function reads. In the default mode, count specifies the number of characters to be read. By right-clicking on the function count could be changed to specify the number of lines. If count is < 0, the function reads the entire file.

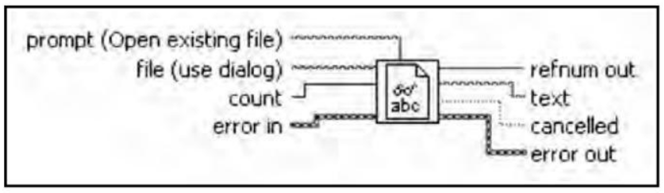


Fig. 8.5 *Read from text file function*

Write to Binary File

The *Write to Binary File* function (Fig. 8.6) writes binary data to a new file, appends data to an existing file, or replaces the contents of a file. The input data can be of any type. The byte order (big-endian, little-endian, etc).can also be specified for integers, default being big-endian.

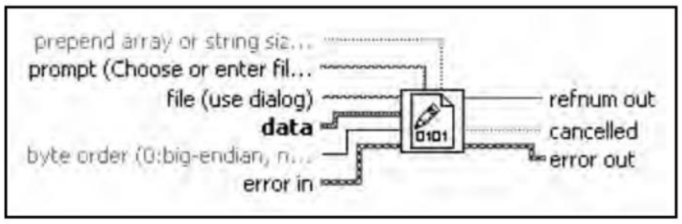

prepend array or string siz...
prompt (Choose or enter fil...
file (use dialog)
data
byte order (0:big-endian, n...
error in
refnum out
cancelled
error out

Fig. 8.6 *Write to binary file function*

Read from Binary File

The *Read from Binary File* function (Fig. 8.7) reads binary data from a file and returns it in data. How the data is read depends on the format of the specified file. The data type input sets the type of data the function uses to read from the binary file.

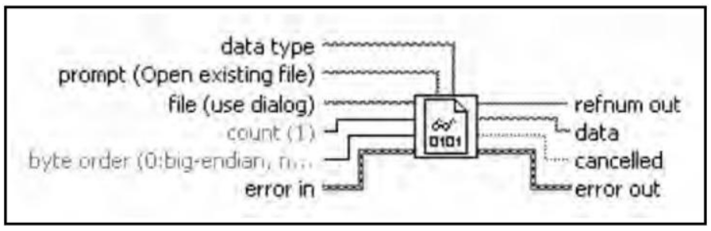


Fig. 8.7 *Read from to binary file function*

File I/O for Writing and Reading Waveforms

With the increasing use of waveform data type, a set of VIs for supporting this data type have now become available (Fig. 8.8). These are used in a fashion akin

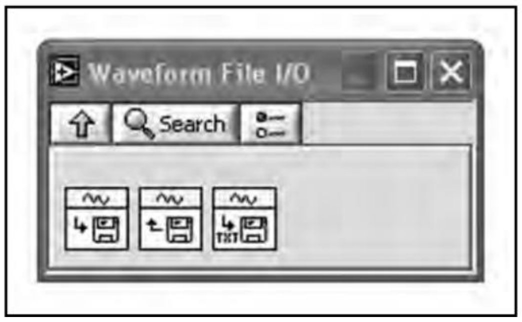


Fig. 8.8 *Waveform File VIs*

to that for spreadsheet access, and are self-explanatory. With distributed controls waveform I/O is very handy for data transfer across nodes. Waveform file I/O VIs can be accessed through the *Waveform File I/O* sub-palette on the *Waveform Palette*. The VIs available for the purpose are *Write Waveforms To File*, *Read Waveform From File*, and *Export Waveforms to Spreadsheet File*.

8.3 PATH FUNCTIONS

8.3 not in Syllabus

Path management is a critical part of practical file I/O operations. Even through paths are very similar to strings they are handled internally in a different manner, and VIs to interconvert paths and strings are available. The user wants to write (as well as access) his files in a specified directory. Sometimes this path may be an absolute path in which case the path management functions are not required. However, more often than not, the path is set relative to the place from which the VI is run. For example, data may be stored in a subdirectory data under the current path of the VI. For this path management functions are critical. Most of the work can be done with three functions, *(Get) Current VIs Path, Strip Path*, and *Build Path*. These are now discussed.

Current VIs Path

This VI is the starting point for all relative path operations. As is evident from Fig. 8.9, the VI simply provides the path of the current (calling) VI. LabVIEW has several functions for file path manipulations, such as *Build path, Strip Path*.

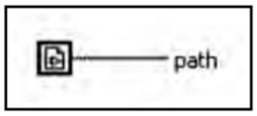


Fig. 8.9 *Current VIs path*

Strip Path

The Strip Path function (Fig. 8.10) returns the name of the last component of a path and the stripped path that leads to that component. This if called after *Current VIs Path* function gives the relative path to the current VI. This can be used as a starting point to define the relative path of the VI.

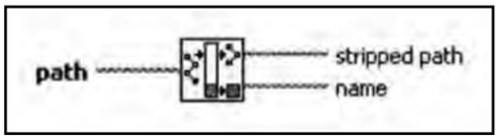
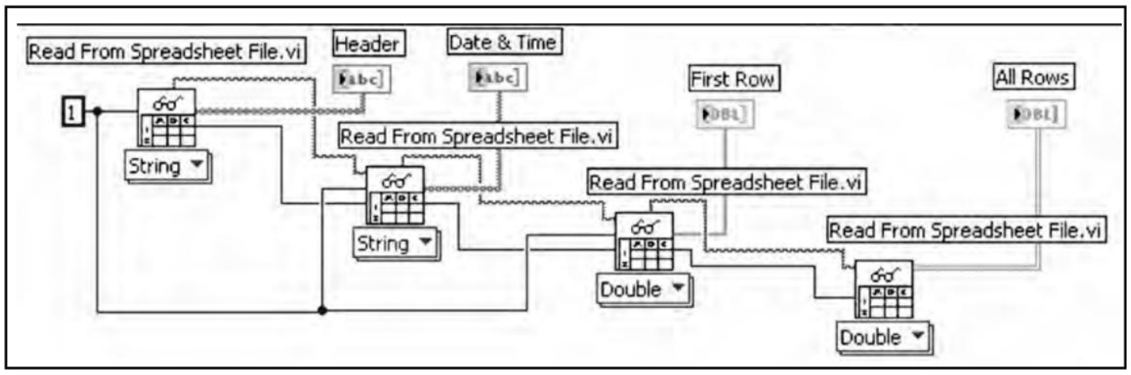


Fig. 8.10 *Strip path function*

of the next call to start the read operation from where the previous operation left. As in the earlier example, LabVIEW opens up the file dialog to obtain file path, which is used by the subsequent read functions. The parameters, stored in a single line, are read through the second call of *Read from Spreadsheet File*, while the last read operation gets the remaining (columnar) data.


Read From Spreadsheet File.vi
Header
Date & Time
First Row
All Rows
1
String
Read From Spreadsheet File.vi
String
Read From Spreadsheet File.vi
Double
Read From Spreadsheet File.vi
Double

From 8.5

Fig. 8.13 *Reading data from ASCII file*

8.5 GENERATING FILENAMES AUTOMATICALLY

Often there is a need to check whether a file exists. Figure 8.14 shows a useful function which may be used for this purpose, *Check if File or Folder Exists. vi*. As the name implies, this function checks whether a file or folder exists on disk at a specified path. The Boolean output *file or folder exits* could be used as a conditional input to create a folder/file.

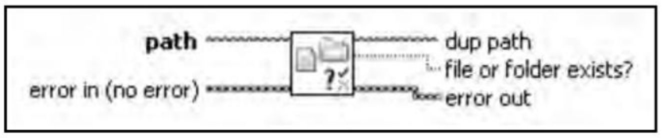


Fig. 8.14 *Check if file or folder exists function*

It is often required to generate filenames sequentially. The VI shown in Fig. 8.15 generates filenames automatically through the sub VI *AutoName* (Fig. 8.16) and outputs the name of the first blank file found. It loops until the

File/Directory Info gives a file size of zero for the named file. It can be easily modified through code for rejecting filenames which contain data.

AutoName generates file names in the format <prefix>mmddnn where nn starts from 00 and increments by 1 at each call.

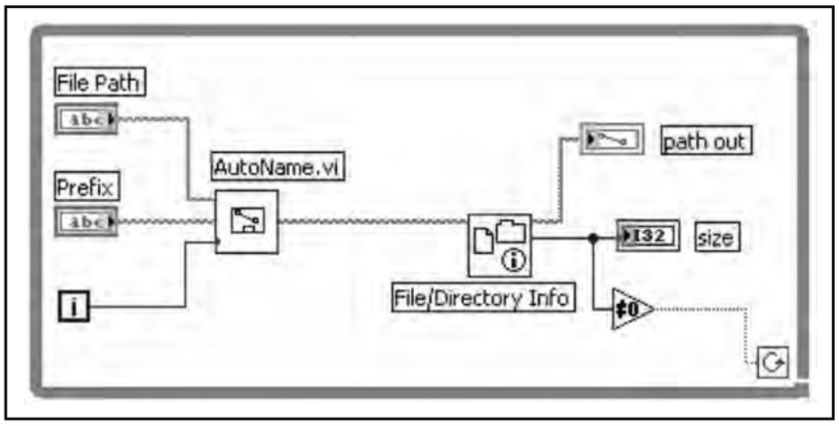


Fig. 8.15 *Code to generate sequential file names automatically*

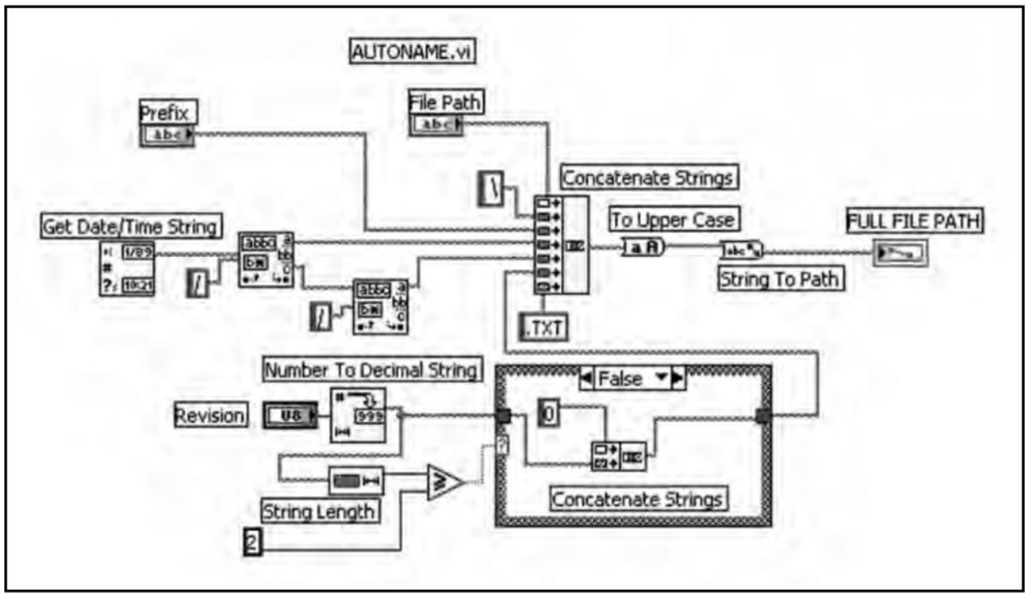


Fig. 8.16 *Autoname for naming files sequentially*