Data Format Design for

Heterogeneous Environments

by M.D.Geib Vista Control Systems, Inc.

Introduction

This paper discusses some of the issues involved in designing data formats for use in multi-platform computing environments. Issues include how to design file formats that can be read on any platform and how to transmit data across a network to support communication between different platforms. Additional discussion is presented on choosing or defining data formats to represent non-atomic data types that hold information such as time.

Designing Platform independent files

Many systems make use of data external to the system, such as disk files. It may be advantageous to use these files on multiple platforms. For systems that make use of data files, one aspect of achieving platform independence is the support for these data files.

Many systems use read or read-and-write files that must be portable. For example, a utility on one platform is used to generate a file that other related utilities read and these other utilities may be running on various platforms. In this case the utilities must be able to read the file independently of which platform produced it.

File headers

One method of making data files portable is by prefixing them with a standard file header which includes information indicating how the file was written. The header should include enough information to allow an application to determine the byte-ordering of data in the file, the format of the floating-point data, the character collating sequence and possibly the format of any time data included in the file. Once this file header is developed, any utility can be developed to support reading and writing of portable binary data. This method has the advantage that for files written and read on similar systems in a homogeneous environment, no type conversion is required, resulting in better performance.

A file header like the following has been developed at Vista to prefix any binary data files that must be portable.

- A two-byte integer with value FFFE
- A four-byte integer with hex value FFFEFDFC
- A four-byte integer containing the major version number
- A four-byte integer containing the minor version number

At this point the byte ordering can be determined. With the byte order established the file version information can be read.

The remaining data in the header includes a four-byte integer that indicates the data type and size of the data that follows. That is, each of the remaining fields is made up of a four-byte prefix, followed by the data. The remainder of the header follows:

- A four-byte integer that indicates the single precision floating point format
- A four-byte integer that indicates the double precision floating point format
- A four-byte integer that indicates the character collating sequence, ASCII or EBCDIC
- An eight-byte integer dependent on the version for validating the format for 64-bit integer values.

- A single-precision floating-point value to validate the floating-point format for single-precision values
- A double-precision floating-point value to validate the double-precision floating-point format
- A four-byte integer indicating the platform where the file was written
- And finally a four-byte integer which indicates the end of the header

Because only the initial portion of the header is fixed, the header can easily be modified in future versions of applications that read or write the file. Prefixing each data item allows for some flexibility in the format of the header and makes it easy for applications to support the reading of data files produced with different versions of the system.

The remainder of the data file is written in a similar fashion. All data items are prefixed with a four-byte integer to indicate the data type of the data that follows. Using this prefix allows for greater flexibility in interversion support. When possible, older versions of the system can ignore data that they do not recognize in new files, and newer version applications can supply defaults for values not supplied when reading older version data files.

The above method relates specifically to binary data files, but a similar approach can be taken with text files. A file header of a standard known character sequence can be written to provide enough information for an application to determine the collating sequence. With this information, the application can make the necessary translation from the data file to the native character values.

Using standard data types

Another approach to producing portable data files is to adopt a standard data format for all required data types. Since a single format is used for all data applications one, simply converts to that type for writing and converts from that type for reading. For systems that have native data types that match the standard format chosen, no conversion is ever required. However, the use of standard data types prevents any performance gain when the platform where the data is written is similar to the platform where the data is read, if both have native types different from the chosen standard file format.

Data communication

As was the case for portable data files, many systems must pass data between different machines while they are running. This requires that all the machines read and write data that other machines write and read. Like the data file, to handle this problem for data communication, a number of approaches are available. Note that since different platforms use different character collating sequences, the numeric value for a given character cannot be assumed to be fixed.

RPC

A number of Remote Procedure Call packages are available that handle many of the problems associated with passing data between different machines. These packages automatically handle the conversion of data between the host machine and the client machine.

Typically, a tool is provided to define all the arguments that are passed in the RPC calls. A utility then reads the data definitions and produces code to handle the conversion of the data to and from the network representation.

Some RPC packages detect when the two communicating systems are of the same type and then use the native data types of the platform rather than converting to and from the standard network representation.

Low level data

Using an RPC package for network communication may not always be appropriate, or available. When lower level data communication is required between machines, code must be written to support the conversion of data to and from different platform types.

One approach is to prefix the data with information similar to the binary file header so that the receiver of the data can convert the data to the native types if required. As with the file header approach, this technique has the advantage that for two similar platforms, no conversion is necessary.

Another approach is to convert the data to a standard type for transmission to the other machine. All parties involved must then convert to and from the standard type in order to communicate with the other machines. This is the approach normally taken by the RPC packages. In fact, many of the platforms that support RPC packages provide access to the routines used by the RPC for data conversion, so that users can use the same routines for converting data to and from a standard network representation for their own use.

User defined data types and constants

When designing data for a system which supports many different platforms, one area to be aware of early on is the support of OS specific information. There are a number of data types that are specific to a platform or even proprietary in nature.

Time

Time has a number of different formats depending on the platform. There are a number of supported formats that define a time with a multi-field structure. This is a very flexible implementation, but it tends to consume more memory than required. At Vista it was decided to adopt the same format for time used by OpenVMS, a signed 64 bit integer. This time format supports both delta and absolute times, and has sufficient resolution for real time systems. There are routines supplied with Vsystem to manipulate this time and to convert it to and from a string representation and a representation similar to the popular time structures.

Status values

Status values returned from library functions are normally platform-specific. During a remote call, trying to return the status value from the remote system to the local one for display is useless in many cases. Vista developed a platform-independent facility for generating status values. These status values can be passed between different platforms, with the values always having the same meaning. To support OS specific errors, these are mapped into the Vsystem facility in a platform independent manner so that a similar type error on different platforms generates the same Vsystem error value and maps to a consistent text message.

Miscellaneous data

A number of other types of data may or may not have to be supported in a heterogeneous manner. In networked systems, like those supported by Vsystem, there is some platform data, like process and user identification and callback routine addresses, that are passed between different machines at runtime. Even though this data is never used on the remote platforms in such a system, the data may be stored on the remote platform and then retrieved by the local platform at a later time. In this case, it is important that the data is never disturbed by the remote communication. Sufficient space must be allocated for the storage of this data generated on all the supported platforms. For example, callback routine addresses on a Digital Alpha platform require 64 bits of storage, while most other platforms currently only require 32 bits to store an address. In order to store addresses in such a system, all address storage must be at least 64 bits.

System Constants

An additional problem created when supporting heterogeneous platforms is the value of system constants. One example is the use of the C constants normally defined in the header files limits.h and floats.h. When a data item is set to the value of FLT_MAX on one platform and then passed to a remote platform which uses a different floating point format, the resulting value may be illegal on the remote platform. An easy solution to this problem is to define within the system, constants that are valid on all the supported platforms. If a new platform is supported in the future, it is easy to change constant definitions if required. The code that does the type conversions for a system should detect this problem. The user code could also take the option of substituting the local platform’s equivalent value if an illegal value is detected.

Recommendations and Summary

The following is a list of some techniques successfully used at Vista to support the exchange of data between heterogeneous platforms:

- Define new data types for all external data and any internal data dependent on the external type to minimize the impact of incompatible implementation of data types by compilers.
- Define new data types for platform-specific types which will support the requirements of the system on all the platforms. These new data types include status values and error codes.
- Define new constants to replace those supplied by the local environments.
- Be aware of the supported range and valid values for all data types on all platforms.

Designing or choosing data formats for use with heterogeneous platforms requires knowledge of the requirements of the system being developed and knowledge of the data types on all the platforms to be supported. In addition, a well-developed plan must specify how new data types will be handled in the future.