

# The BackDoor System

Duane C. Voy

## Introduction

BackDoor is a protocol and associated software system for exchanging data between computers. The name BackDoor suggests an alternate portal through which 'friends' gain access. BackDoor provides quick access to a front-end's data structures without having to attach them to ACNet and write custom console application programs. The BackDoor system is not intended to replace ACNet functionality, it merely provides easy to use tools to support the front-end development process.

BackDoor is a point-to-point protocol that can support peer-peer or client-server operation, and can be implemented with any *reasonable* programming language on any IP based operating system. The current implementation supports client-server operation only. To date there is a single BackDoor server implementation written in C++ running on VxWorks. There is also a client written in C++ running on VxWorks, and another written in LabVIEW's G programming language that will run on any platform supported by LabVIEW. This document describes client-server operation only.

Numerous examples are available to illustrate use of the BackDoor system including:

- an example project called 'backdoorexample' that demonstrates how to get the default BackDoor server operational,
- several useful BackDoor accessors in the BackDoorServer library and
- a test project called 'backdoortest' that demonstrates how to implement and use BackDoor accessors and events.

This draft of the documentation covers the second generation of the BackDoor software system. This latest version represents a complete rewrite of the C++ server and significant changes to the LabVIEW client with the goal of enhancing cross-platform operation. Additionally, a C++ client has been added to the system.

## BackDoor (Client-Server) System

BackDoor operation typically involves a server in a front-end node and a client on a workstation. BackDoor defines a session based symmetrical<sup>1</sup> request-reply transaction protocol. The client establishes a session with its target server, conducts one or more transactions and ultimately ends the session. Each transaction's request message elicits a reply message that may or may not contain data but always contains the status response to the original request. Sessions may contain a mixture of one-shot and repetitive transactions. One-shot transactions involve a single request with a single immediate reply while repetitive transactions involve a single request with multiple replies. Typical transactions involve reading or setting a remote object, but it is also possible to accomplish a form of 'remote method call' by doing a set-read transaction where the request contains an object setting value and the reply contains the resultant object reading value.

BackDoor makes no real distinction between data and parameters. Server side data structures are made available to the server, and thus the clients, via **BackDoor::Accessor** (accessor) objects. Accessors have five methods that allow clients to observe and/or modify the associated structure. Looking from the client's perspective the five methods are:

- **Set** - send a value to the setting of the object,
- **ReadSetting** - read the current value of the object's setting,
- **Read** - read the current value of the object's reading,
- **ReadFifo** - read the object in FIFO fashion, and
- **SetRead** - send a value to the setting of the object and then read the resulting value of the object's reading.

Objects with reading and setting capabilities are familiar to users of ACNET which supports numerous data properties; however, BackDoor supports setting and reading only. As the ReadSetting method implies the setting and reading may be associated with the same or different structures within the front-end. The SetRead method is unique in that it supports a simple remote procedure call mechanism -- set the object and return the result. Associated with each of the five methods is an **Index** value provided by the client and passed through the system to the targeted accessor method. The exact meaning of the index

---

<sup>1</sup> The symmetrical nature of the BackDoor protocol message structure means that expansion to full peer-peer operation is fully supported.

is defined by the individual accessor classes. An example use for the index value would be to identify a unique channel within an object.

In a BackDoor system time related events such as hardware triggers or changes of software state are made available to server objects as **BackDoorEvent** (event) objects. For example, the passage of time is indicated by a special periodic event called 'BackDoorTick' that occurs at a user specified rate. The server object uses events to trigger responses to periodic requests. Events can also be used by accessors to trigger data collection.

BackDoor represents aggregates of accessor objects with **BackDoorList** (list) objects. List objects are created by the client, forwarded to the server for processing, and persist within the server until the client deletes them. A list is an ordered list of accessor/method/index triplets specifying a sequence of accessor method invocations to be done each time the list is processed by the server. Lists may include any number of accessor/method/index triplets, in any order, with each identifying any of the five available accessor methods. Lists can be configured to return data periodically or only when explicitly read by the client.

Clients may also access individual accessors or lists with one-shot requests. To access accessors periodically clients must establish periodic list objects. Repetitive list data are returned on any specified multiple of any event known to the server.

### **BackDoorServer Class**

The BackDoor server uses standard IP networking technology to provide remote clients with access to the data and parameters of an embedded system. The server is intended to be lightweight in every respect including:

- small memory profile,
- prompt and quick processing of client requests (not a CPU hog),
- easy to implement (minimal user programming),
- operate entirely at user determined priorities and
- requires no hardware timing resources.<sup>2</sup>

---

<sup>2</sup>At instantiation the BackDoor server creates a default 15 Hz. "tick event task" that supports periodic activity in the BackDoor server. This task as well as all other BackDoor related tasks run at a user specified task priority. The default tick event can be replaced with a user provided event if desired.

In short the server works only with resources defined by the user, while consuming a minimal amount of memory and CPU cycles.

The server class has two friend classes: `BackDoorEvent` which provides synchronization with user timing, and `BackDoorAccessor` which provides access to user data structures. A system that uses `BackDoor` will have one or more server instances each having zero or more event instances and zero or more accessor instances. The server itself has no knowledge of the user's application, the user must create an event instance for each event and an accessor instance for each data structure to be made known to clients. Events may be periodic or sporadic and the data associated with an accessor may be of any size and complexity. Each event and accessor is assigned a numerical identifier by the server, called an **Ident**, that provides a shorthand way of addressing the object. Events and accessors will be described in more detail below.

Upon instantiation the server will create a default periodic 'tick' event called 'BackDoorTick' and a default time stamp accessor called 'BackDoorTime'. The default tick rate is 15 Hz and the default time stamp is the familiar 64 bit UN\*X time containing seconds and nanoseconds. If desired the default periodic event and time stamp accessors may be replaced with user provided versions in order to meet unique front-end requirements. Overriding the default tick and time stamp will be discussed in more detail below.

In addition to the default tick and time stamp objects the `BackDoor` server provides several built-in objects that provide utility services in the system. The names for these objects include:

- **DataIdent** - provides Ident lookup services for accessors,
- **DataName** - provides Name lookup services for accessors,
- **EventIdent** - provides Ident lookup services for events,
- **EventName** - provides Name lookup services for events,
- **TickRate** - provides the frequency in ticks per second of the `BackDoor` server's periodic tick event,
- **TimeOfDay** - UN\*X format consisting of two 32 bit words with the first containing seconds since the beginning of the epoch and the second containing the number of nanoseconds within those seconds,
- **Memory** - provides direct read/write access to system memory,
- **SymbolTable** - provides access to the front-end system symbol table and

- various BackDoor server diagnostic accessors.

See \$RFINST\_INC\_DIR/backdoorserver.h for the declaration of the BackDoorServer class and its methods.

### **BackDoorEvent Class**

Events provide a sense of time to the BackDoor server. Each event instance must have a unique name and is automatically attached to the most recently instantiated server when the event itself is instantiated. Events are created by the user and at the appropriate time are 'announced' by the user.

The class **EventWorkRequest** is used in conjunction with the **BackDoorEvent::AddWorkRequest()** method to instruct an event to perform a specified action whenever the event is announced. The defined work request types include:

- **ProcessList**,
- **AccessorAnnounceEvent** and
- **AccessorAcquire**.

The ProcessList request is used by the server and is not intended for use in front-end code. When the server is asked to process a list at some multiple of a specified event it uses the ProcessList EventWorkRequest to tell the specified event to send a list processing request to the event handler task each time that the event is announced. By using a task to handle list related events the potentially lengthy process of assembling lists of data is transferred from (potentially) interrupt level to task level.

The AccessorAnnounceEvent request is used by accessors to instruct an event to call the accessor's **AnnounceEvent()** method whenever the event is announced. Through this facility accessors can perform (hopefully quick) operations upon events. For example buffered accessors can use this mechanism to swap buffers upon some event.

The AccessorAcquire request is used by accessors to instruct an event to call the accessor's **Acquire()** method whenever the event is announced. The Acquire() method should be quick, simply copying a data point for the accessor.

Events can be announced at interrupt or task context. You must understand that any of the work request types described above may result from calling an event's announce() method, and that the total number of event work requests in a system is potentially unbounded. Further, any of the event work requests can potentially execute floating point operations so announce event calls at interrupt context must be surrounded by a floating point context save and restore.

To aid in understanding the impact of event work requests on the front-end there are several diagnostic objects built-in to the server's code:

- **TickTimeLine** provides a time line display of event processing activity for the BackDoorServer periodic tick event,
- **ServerTimeLine** provides a time line display of request message processing activity within the server,
- **TickPeriod** provides a histogram of the period of the periodic tick event,
- **AnnounceEventExecutionTime** provides a histogram of the execution time for all events announced in the system,
- **EventTaskExecutionTime** provides a histogram of the execution time of the event handler task,
- **ReplyTaskExecutionTime** provides a histogram of the execution time of the reply (network write) handler task and
- **EventFIFO** provides an accounting of the events that have been announced in the system.

Each of the diagnostic objects has an associated LabVIEW demo program for viewing the data.

See \$RFIINST\_INC\_DIR/backdoorserver.h for the declaration of the BackDoorEvent class and its methods.

### **BackDoorAccessor Class**

Accessors support remote observation and manipulation of user data and parameter structures. Each accessor instance must have a unique name and is automatically attached to the most recently instantiated server when the accessor itself is instantiated.

The `BackDoorAccessor` class declared in `backdoorserver.h` is abstract to support run-time polymorphism. This means that you cannot directly instantiate a `BackDoorAccessor`. All user accessors must be derived from the `BackDoorAccessor` class.

There are several standard accessors available to users. Standard accessor header files, located in `$RFIINST_INC_DIR`, with comment blocks providing some useful documentation. The standard accessors include:

- **InSituAccessor** - `insituaccessor.h` (template),
- **InSituReadAccessor** - `insituaccessor.h` (template),
- **InSituSetAccessor** - `insituaccessor.h` (template),
- **BufferAccessor** - `bufferaccessor.h` (template),
- **TwoDArrayAccessor** - `twodarrayaccessor.h` (template),
- **OneDArrayAccessor** - `onedarrayaccessor.h` (template),
- **TwoDimension** - `twodimension.h` (template),
- **OneDimension** - `onedimension.h` (template),
- **Histogram** - `histogram.h` (template),
- **SHistogram** - `shistogram.h` (template),
- **FIFOAccessor** - `fifoaccessor.h` (template),
- **PeriodHist** - `periodhist.h`, and
- **TimeLine** - `timeline.h`.
- **SystemReboot** - `systemreboot.h`.

`InSituAccessor` provides for reading and setting data as they are located in memory. That is to say the `InSituAccessor` class does asynchronous reading and setting of data in situ. The class is implemented as a template and can handle any data type.

`InSituReadAccessor` provides for reading data as they are located in memory. That is to say the `InSituReadAccessor` class does asynchronous reading of data in situ. The class is implemented as a template and can handle any data type.

`InSituSetAccessor` provides for setting and reading the setting of data as they are located in memory. That is to say the `InSituSetAccessor` class does asynchronous setting/reading of data in situ. The class is implemented as a template and can handle any data type.

`BufferAccessor` provides for reading data through a double buffering scheme. The `BufferAccessor` has an `Acquire()` method that can be called by the user or by an event

occurrence. The Acquire() method places a copy of the data associated with the accessor instance into a double buffer for later readout. The class is implemented as a template and can handle any data type.

TwoDArrayAccessor provides for accessing a LabVIEW dynamic length two dimensional array in situ. The class is implemented as a template and can handle any data type.

OneDArrayAccessor provides for accessing a LabVIEW dynamic length one dimensional array in situ. The class is implemented as a template and can handle any data type.

TwoDimension provides for accessing a fixed size two dimensional array in situ. The class converts a C array to a LabVIEW array, is implemented as a template and can handle any data type.

OneDimension provides for accessing a fixed size one dimensional array in situ. The class converts a C array to a LabVIEW array, is implemented as a template and can handle any data type.

Histogram provides for histogramming a quantity. The class is implemented as a template and can handle any fundamental C data type.

SHistogram provides for processing simple integer histograms of ADC data. The class is implemented as a template and can handle any integral data type.

FIFOAccessor provides for collecting data at rates higher than the periodic tick event by FIFO buffering the data for later readout.

PeriodHist provides for histogramming the execution time of code or the period of repetitive events.

TimeLine provides a logic analyzer like display of an arbitrary time line. This is intended to simulate the use of hardware "SSM LEDs" in displaying the performance of a thread of execution.

SystemReboot simply calls the VxWorks reboot() function whenever any 4 byte value is sent to the Set() method..

Each of the standard accessor types has a LabVIEW demo program that exercises the example. See \$RFIINST\_INC\_DIR/backdoorserver.h for the declaration of the BackDoorAccessor class and its methods.

### **Overriding the Default Tick Event and Time Stamp**

The default periodic tick event provided by the server is announced at 15 Hz by a task that loops announcing the tick event and then waiting for 1/15th of a second. The timing provided by this method is not precise but is adequate for most repetitive list activity. If this "sloppy" 15 Hz does not match front-end requirements you may instantiate a new tick event that will supersede the default. The example below shows how the user can create a periodic tick event. The periodic tick event is unique from all other events in that its constructor contains a tick frequency (in Hz) parameter in place of an instance name:

```
const int kTickRate = 15;      // BackDoor periodic tick rate
ptr = new BackDoorEvent( kTickRate, diagnosticControl );
ptr->Announce()
```

The default time stamp provided by the server follows the well-known UN\*X format of two 32 bit words with the first containing seconds since the beginning of the epoch and the second containing the number of nanoseconds beyond those seconds. If this form of time stamp does not match front-end requirements you may instantiate a new time stamp accessor that will supersede the default. The example below shows how the user can create a time stamp accessor. The time stamp accessor is unique from all other accessors in that its constructor does not contain an instance name:

```
new TimeStampAccessor( diagnosticControl );
```

### **The BackdoorExample Project**

The files in the backdoorexample project (/home/rfies/esd/examples/backdoorexample/\*) demonstrate how to get a BackDoor server running and customize it with a user provided tick event and time stamp accessor:

- **Makefile** - shows how to link your project to the BackDoor and other support libraries,
- **Targets** - shows the targets currently supported by BackDoor,

- **backdoorexample.cpp** - shows how to get a BackDoor server running in a system,
- **backdoorexamplestartup** - shows how to invoke the BackDoor instance and enable client access from within startup scripts,
- **BackDoorExample.vi** - shows how to manipulate the BackDoorAccessor instances used in the example and

The backdoorexample project provides three functions that together create a fully functioning BackDoor system:

- **BackDoorExampleNew** - creates an instance of the BackDoor server.
- **BackDoorAddPrivileges** - allows the specification of access privileges on a client by client basis.
- **BackDoorDisplay** - displays BackDoorServer instance information including server parameters, and lists of BackDoorAccessor and BackDoorEvent instances in the system.

The example creates an accessor instance named "TestData" that makes a floating point data value available for read access, and an accessor instance named "E-Max" that makes a floating point parameter value available for read and set access.

## **LibBackDoorDefault**

A single library containing all required BackDoor support files has been created to simplify use of the BackDoor server. The library is available in:

`${RFINST_LIB_DIR}/VW_${VXWORKS_VERSION}/${TARGET}/libbackdoorDefault.out.`

When loaded this library will create a default instance of the BackDoor server with the following characteristics:

- server TCP port – 2048
- maximum session count – 5
- server priority – 100
- maximum accessors – 100
- maximum events – 25
- additional work space – 2 MB
- additional stack space – 32 kB
- server name - "BackDoorDefault"
- diagnostic flags - kNoDiagnostic

The library provides three utility routines that may be called from the shell:

```
extern "C" int BackDoorAddPrivileges(  
    eClientAccessType const accessType,  
    unsigned long int const accessAddress,  
    unsigned long int const accessMask );  
  
extern "C" int BackDoorDisplay( void );  
  
extern "C" int BackDoorDiagnosticControl (  
    eDiagnosticType const diagnosticFlags );
```

End.