ACT API Developers Guide

For Act Enterprise v2.0.0.7+

Contents

Code Samples

Introduction

This ACT API Developers Guide is an introduction to developing third-party applications that integrate with ACT's access control software and hardware. The guide assumes you are an experienced software developer, preferably with C# .NET. Prior knowledge of WCF and/or using XML web services with ASP is an advantage, but not essential.

The guide assumes you are writing a .NET application. Although it might be possible to integrate using another platform, such as Java, that is beyond the scope of the guide.

We recommend the use of Visual Studio 2010 (or higher) and the related tools and mechanisms for adding web service references over WCF. It is beyond the scope of the guide to explain these tools in depth.

Note on API Versions

In ACT Enterprise v1 versions, there was a similar public API made available. That API is still available for use, but we recommend the newest one documented here. The older API used C# enums in the data contracts. Previously, if a new version of Enterprise was installed alongside a third-party tool which had been built against an earlier version, there was a problem handling new enum values. These have been changed to uints in the latest version, to improve backward compatibility. Steps have been taken to include descriptions of the uint values and assistance when interpreting them during development, which are documented below.

Related Documentation

The API and related classes have been documented in a Windows standard help format, in ACTPublicAPI.chm.

Some sample code is available in a small Windows solution called PublicAPISamples. This is a full working example that shows:

- reading user data
- getting and displaying user photographs
- reading doors
- issuing commands on doors
- registering for log events
- processing live log events

Who can use ACT's API

All third-party application developers are welcome to integrate with ACT's software and hardware. The types of application supported include HR systems, DVR systems or other access control systems. All developers need to have a licenced copy of ACT Enterprise v2 or higher before they can get access to the API.

Usage policy (licencing)

To develop against the API, you must first access the published web service. The API is only exposed on platforms where it has been specifically licenced. A licence for the API must also be purchased by each customer wishing to avail of your application after it has been developed. Please contact ACT through http://www.accesscontrol.ie to enquire about purchasing an API licence.

Technical description

The API is a set of XML web services, built using Microsoft's WCF, available over an Intranet using the TCP protocol. The API was developed as a subset of ACT's proprietary and internal web service API for developing Intranet-based client-server applications. ACT Enterprise is hosted inside a Windows Service running as Local System account on a server machine.

The API contains a set of web service methods enabling access to users, doors, log events, and other essential access control data. The methods allow reading, writing, and updating basic records (such as users) and access to live status and log event data.

Requirements and recommended tools

- .NET Framework 4.5.2 or higher
- Visual Studio 2010 or higher
- ACT Enterprise v2.0.0.6 or higher (released December 2016)
- SQL Compact (installed with Act Enterprise) or SQL Server
- Licence(s) purchased for ACT Enterprise to enable the API (available from ACT)

Understanding the API

This section covers important background topics that prepare you for using the API.

Session Based API

The ACT API is built as a session-based WCF service. This means that every client accessing the service must establish a proper session (see below). Sessions mean that inter-method call data can be stored and processed by the service. However, one potential disadvantage of a session is that if a fault occurs, then the WCF session can be put into a fault condition, necessitating a new session. Clients accessing the API will have to handle the creating and maintaining of valid WCF sessions themselves. Sessions can impose higher overheads on a service, but the ACT service usually handles low numbers of simultaneous sessions, so this is not a particular problem.

Limits on data transfer

When using the API, you need to bear in mind some important limitations and considerations.

The API was designed for optimal efficiency in an Intranet-type environment, so mandates the use of the TCP protocol, which supports encrypted, binary transfers of data. Each message is limited to 64k maximum, the default set by WCF. Enforcing a limit prevents accidental or deliberate misuse of the service, where enormous messages would degrade the user experience. Instead, the service API assumes that each data read is for a limited set of sequential records at a time. If more data is required than can be provided in one request, then the client must make more requests, using the appropriate arguments.

The practical upshot of this limit is that most data needs to be read in a loop where you cannot know beforehand the amount of records returned for a datatype.

An example of this limit in action is when reading the user data. A maximum of around 50[1] users can be transfered in each read. Therefore, you must provide a start index and maxCount for each read from the user table, in order to specify the subset of data you want. We will see a relevant code sample in the next section.

Another example is the following method. It allows for getting the log events for a specific user between the start and end datetimes:

```
List<LogValueExt> GetLogsOfUserID(DateTime startWhen, DateTime endWhen,
int user, int start, int maxCount, bool order);
```

Note the `start` and `maxCount` parameters, which are used to control the subset of log events returned. The number of log events returned is limited to the larger of `maxCount` and the built-in limit. The built-in limit is determined by the known safe amount of data to return for the return type, which can vary.

[1] This number could vary, so do not assume it will be 50!

For example, to use this method to read the most recent 500 log events for a user over the last ten days (where the variable "proxy" refers to the service proxy generated by Visual Studio), the method has to be called in a loop[2] such as the following:

```
    int start = int.MaxValue;
    int LOGS_WANTED = 500;
    bool MOST_RECENT = false;
    List<LogValueExt> results = new List<LogValueExt>();
    LogValueExt[] logs = null;
    do
    {
        logs = connection.GetLogsOfUserID(startWhen: DateTime.Now,
endWhen: DateTime.Now.AddDays(-10), user: user, start: start, maxCount:
LOGS_WANTED - results.Count, order: MOST_RECENT);
        if (logs.Length > 0)
        {
            start = logs[logs.Length - 1].EventID - 1;
            results.AddRange(logs);
        }
    }
    while (logs.Length > 0 && results.Count < LOGS_WANTED);
```

Code Sample 1: Reading recent log events for a user in a loop

Note that in the above code, no assumption can be made about the number of records returned on each iteration. It will depend on the datatype returned and on the WCF limits configured for the system. In general, the limits follow the defaults of WCF and are large enough to allow for efficient pagination.

Avoiding deadlock

When you are registered for callbacks, a WCF client-server application can produce deadlocks. This can happen, for example, when your application is in the middle of a call to the service and it is simultaneously calling you back to notify you about a log or status event. If your callback is handled on the same thread (typically the UI thread) as you are calling out on, then your application will be deadlocked.

One known solution to this common problem is explained in depth by Juval Lowy in his book *Programming WCF Services*, O'Reilly, pps 422-427. This is the recommended solution (and is shown below).

You need to mark your handling class as not using its current synchronisation context to choose the thread of execution:

[2] The most recent log events have the highest log event ids, so int.MaxValue is the valid starting point.

```
    [CallbackBehavior (UseSynchronizationContext=false)]
    public class ServerConnections : IActEnterprisePublicAPI_ExtCallback
    {
...
```

Code Sample 2: Marking a class as not using its own synchronisation context

At some suitable point in your code, you need to manually assign the synchronisation context.

And then you marshall callbacks to the correct context.

```
        private SynchronizationContext scontext = null;
        ....
        {
            // Set the context manually
            scontext = SynchronizationContext.Current;
        }
        ...

        public void OnLogEvents(LogValueExt[] loggedEvent)
        {
            SendOrPostCallback processLogs = delegate
            {
                for (int i = 0; i < loggedEvent.Length; i++)
                {
                  ...// handle log events here
                }
            };
            scontext.Post(processLogs, null);
        }
```

Code Sample 3: Marshalling callbacks to a synchronisation context

You will need to code this solution for the two callbacks in the API: `OnLogEvents` and `OnStatusChangeEvents`.

Lists returned as arrays

The API declares list types as List<T> internally. However, these types are returned as arrays to the caller by WCF. Therefore, where you see might this method in the documentation:

```
List<DoorValueExt> GetDoorsOnController(int controller, bool enabled);
```

It actually looks like this on the client side:

```
DoorValueExt[] GetDoorsOnController(int controller, bool enabled);
```

When using the Object Browser in Visual Studio, you will see the correct types (i.e. arrays) for the proxy class.

Using BaseValue types

All complex data types (e.g. for doors or log events or users) are returned as sub classes of `BaseValue` (e.g. `DoorValue`, `LogValue`, or `UserValue`). These are purely data-transfer classes and do not contain behaviour. `BaseValue` provides some important attributes that you need to understand.

The bool **`IsValid`** attribute marks a record as valid or not. Blank records default to `IsValid=false`. When updating or inserting new values, you must ensure that `IsValid` is set to true before the service will undertake the action. Similarly, if you request data from the service, you should check the `IsValid` property of any returned data in order to verify that it is valid. For example, asking for a non-existent user will return a `UserValue` record with `IsValid=false` to indicate that it does not exist.

The **`PrimaryKey`** property is an alternative way of accessing the record's unique id value. Some generic sort methods rely on this value. The `UserNumber` in `UserValue` is an example of a unique id value.

When reading user lists (and some other long sets of data) based on a complex query, the **`RowCount`** property of the first record returned can contain the number of records in the query. In general, unless explicitly documented, you cannot rely on the RowCount property to return a valid value. For most regular queries, this field is left as 0.

DBUser permissions

ACT Enterprise uses a proprietary set of user permissions to control access to data and services. These are managed in ACT Manage and stored as DBUsers. Customers are recommended to assign a unique DBUser record for each person authorised to access the system[3]. This helps with auditing and logging too.

When you establish a session with the API, you must provide a valid DBUser name and password. The rights assigned to that user determine the rights your client application then obtains. For example, you might establish a session using a DBUser who can read and write user data, but who is not allowed to issue commands on doors. Any attempt to issue a command via the relevant API method will be blocked by the service.

It is therefore essential to understand the permission system of Act Enterprise and to cater for users who might be denied access to some service methods. In general, service methods just return no data or fail to perfom an action when the client accessing them does not have the required permissions.

[3] A default user of "Administrator" with a blank password is available with new databases.

Retaining Forward Compatibility

The ACT Enterprise version you build and test your software against is the ideal version for your customer to use. If the customer intends upgrading their version of ACT Enterprise, then they will need to consider updating any third-party software that uses the API.

While ACT will endeavour to maintain compatibility with older third-party software at all times, there might be changes that require breaking it. In those cases where it is not possible to update the third-party software in a timely fashion, then the customer should postpone upgrading their ACT Enterprise until it is reworked.

Accessing Enum Values

In this newer version of the API, enums have been replaced in every data contract with uints. This means that if the service introduces a new enum value, it will not break compatibility with older third-party software which has been compiled against an earlier service reference.

Some steps have been taken to document these uints and to make their interpretation easier for third-party developers.

Firstly, where enums appear in a Value object passed back by the service, the uint value has been supplemented with a string desciptor. For every value X which is a uint but was an enum, it will have an XDescriptor field with the enum value's name. For example, in UserValueExt, the CardType field has a CardTypeDescriptor field too. If the CardType is 1, then the CardTypeDescriptor will be "LearnedCards", etc.

Secondly, a new dummy method `GetEnums()` has been added to the API to expose an `EnumDescriptorValue`. This includes all enums required in the data contracts. Consult these values to be able to interpret the returned uints.

Finally, the required enums are included in an appendix to this document.

Using the API

This section gives some practical guidance to undertaking common tasks with the API. It also tries to show how likely scenarios can be coded, such as updating user records.

Accessing metadata

The API's metadata is published by a running instance of the Act Enterprise Server which has been licenced to expose the API. You can use Visual Studio's Add Service Reference tool to get the metadata and to generate suitable proxy classes in your application.[4]

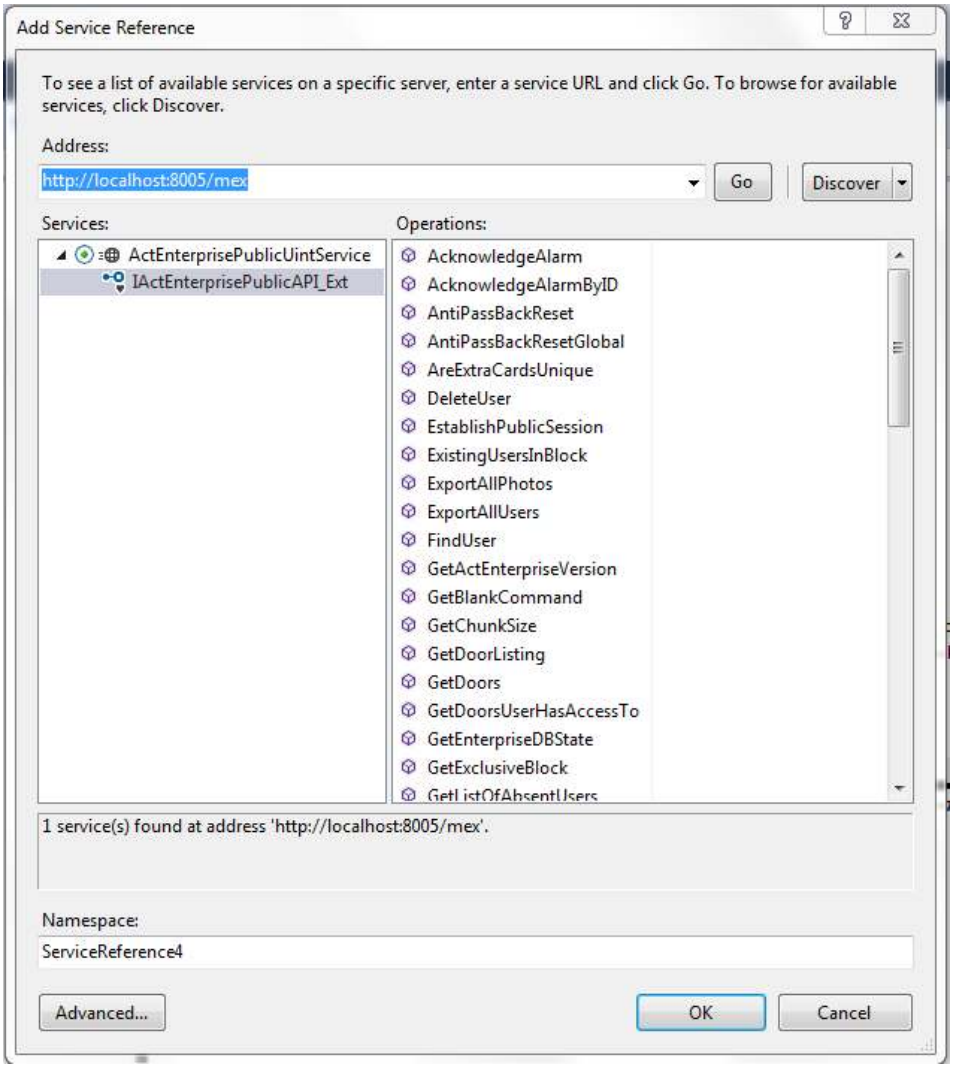

Figure 1: Adding a service reference from Visual Studio 2010

The service metadata (assuming you are running the service on your local machine) is exposed at the endpoint **http://localhost:8005/mex**[5]. Note that the service should be running with a properly configured database (which can be empty) and with a correctly licenced-for-the API system.

[4] There are other tools for generating proxy classes from metadata, but they are beyond the scope of this document.

Implementing the client class

After adding a Service Reference to your project, you should see the API through your object browser (where "ACTPublicAPI" can be changed to an arbitrary name).

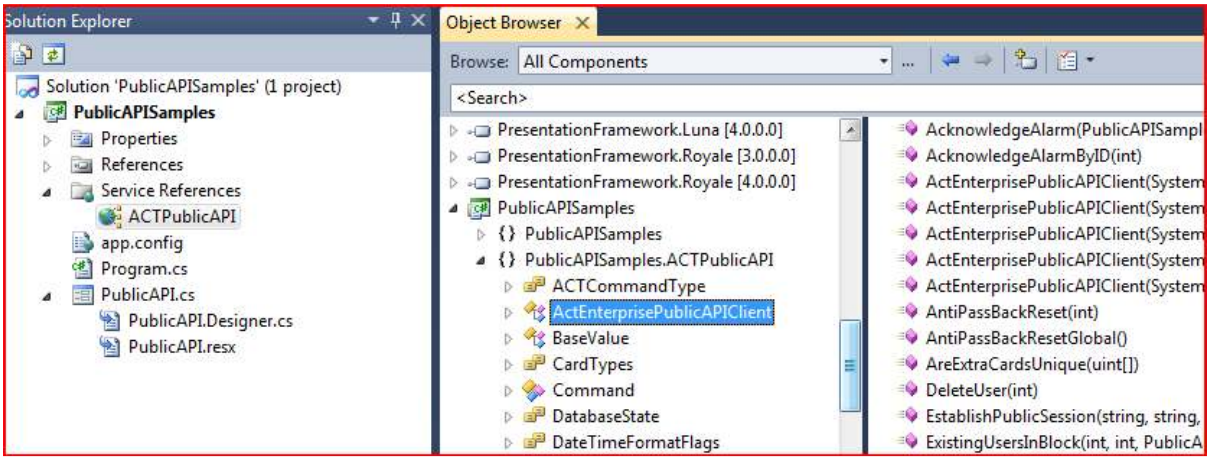

Figure 2: Browsing service reference

You need to create a proxy class on your client. The code sample shows a simple demonstration of such a class, which illustrates the main features you need to be aware of.

[5] The older version of this API is still available at port 8003. We do not recommend using that version and it is maintained purely for backward compatibility reasons.
Note that the port for the API itself is 8004.

```
    using System.ServiceModel; // [1]
...
    /// <summary>
    /// Creates a proxy class for handling communication with the service.
    /// Set CallbackBehavior to not use the default synchronisation
context
    /// Implement the IActEnterprisePublicAPI_ExtCallback callback
    /// Create the proxy object
    /// Establish a session
    /// Implement callback methods
    /// </summary>
    [CallbackBehavior(UseSynchronizationContext = false)] // [2]
    public class MyHandler : IActEnterprisePublicAPI_ExtCallback // [3]
    {
        ActEnterprisePublicAPI_ExtClient proxy = null; // [4]

        public bool CreateProxy()
        {
            try
            {
                proxy = new ActEnterprisePublicAPI_ExtClient(new
System.ServiceModel.InstanceContext(this)); // [5]
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
                return false;
            }
            uint ok = proxy.EstablishPublicSession("DBUser name",
"Password", System.Environment.UserName, System.Environment.MachineName,
"My application"); // [6]
            return ok == 1;
        }

        public void OnLogEvents(LogValueExt[] logEvent) { } // [7]
        public void OnStatusChangeEvents(StatusValueExt[] statusEvents) {
}

    }
```

Code Sample 4: Implement proxy class

The main features to be aware of are[6]:

1. You will be using the `System.ServiceModel` assembly, so it has to be added to your references.
2. The port for the service is 8004 (the metadata is published on port 8005).
3. To avoid deadlock (as explained above) you should set the automatic choosing of the current synchronisation context off.

[6] This example assumes you have added a service reference using Visual Studio

4. Declare a class as implementing `IActEnterprisePublicAPI_ExtCallback`.
5. Declare an object of type `ActEnterprisePublicAPI_ExtClient`.
6. Create an instance of the object, passing in an `InstanceContext`.
7. Call the ACT API method `EstablishPublicSession` with your ACT DBUser login and password.
8. Declare handlers for the callbacks, `OnLogEvents` and `OnStatusChangeEvents` to fulfill the `IActEnterprisePublicAPI_ExtCallback` interface requirements.

The following sample app.config file illustrates how to configure the WCF bindings and endpoints.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <bindings>
            <netTcpBinding>
                <binding name="NetTcpBinding_IActEnterprisePublicAPI_Ext"
/>
            </netTcpBinding>
        </bindings>
        <client>
            <endpoint
address="net.tcp://localhost:8004/ActEnterprisePublicUintAPI"
                binding="netTcpBinding"
bindingConfiguration="NetTcpBinding_IActEnterprisePublicAPI_Ext"
                contract="ACTPublicAPI.IActEnterprisePublicAPI_Ext"
name="NetTcpBinding_IActEnterprisePublicAPI_Ext">
                <identity>
                    <userPrincipalName value="userprinciple@local" />
                </identity>
            </endpoint>
        </client>
    </system.serviceModel>
</configuration>
```

5 Code Sample: Sample api.config file

Starting and ending a session

The ACT API is session-based. You *must* establish a session using the `EstablishPublicSession()` method, providing valid ACT login details, before being allowed to call any other method. If this method is not called, or if the call fails, then you will be blocked out of the service[7].

You must be in possession of a valid DBUser name and password, which are stored in the ACT database and configured using the ACT Manage client application. You must allow for this to be

[7] When the ACT Enterprise service boots up, it loads user and log data into an internal cache, which on large databases might take some time. Sessions cannot be started until this data is loaded.

provided by the customer when running your application - by providing for a login page that accepts a DBUser login name and password, for example[8].

For accurate auditing, you are recommended to provide the Windows identify of the user creating the session (logonUserName) and their PC. Finally, you should provide the application name (clientName) which should be the name of the client you are creating.

```
uint EstablishPublicSession(string DBName, string DBPassword, string
logonUserName, string logonPC, string clientName = "Unknown");
```

The result of a call is a uint which matches the values of the following **`EstablishSessionResult`** enum. It provides details of the login attempt and should be checked after the call:

```
public enum EstablishSessionResult : uint {
        SuccessfulLogin = 1,
        OnlyServerClientAllowedAccess = 2,
        DBNameIsBlank = 3,
        DBNameDoesNotExist = 4,
        PasswordInvalid = 5,
        ServicePreventingAccess = 6,
        AlreadyEstablishedSession = 7,
        UnknownError = 8,
        DeniedAccess = 9,
        ClientsLicencedExceeded = 10,
        TrialPeriodEnded = 11,
        FallbackLicenceDenied = 12
}
```

Code sample 6: The EstablishSessionResult enumeration

All sessions must be shut down using **`ShutDownSession().`** This automatically deregisters your application for event callbacks, audits the logoff, and frees up a licenced user. Not making this call might cause unexpected side-effects.

`ShutDownSession`() is marked in WCF as terminating the session. No other method can be called from this session after `ShutDownSession`() has been called.

Handling log events

The ACT Enterprise service allows third-parties to register for a live stream of log events. These log events arise from all controllers on the system and include some PC-generated ones too.

To register for log events, call the `RegisterForLogEvents` method. Log events are then sent to the registered client via the `OnLogEvents` callback.

[8] After creating a blank ACT database, there will be one default DBUser made available to kick-start the service, of "Administrator" and a blank password. It is recommended that this default user either be deleted, or for a suitable secure password to be set.

To handle log events (and avoid WCF deadlock problems), follow the suggestions in the relevant section above. Here is a sample `OnLogEvents` handler[9]:

```
public void OnLogEvents(LogValueExt[] loggedEvent)
{
    SendOrPostCallback processLogs = delegate
    {
        for (int i = 0; i < loggedEvent.Length; i++)
        {
            RecentLogEvents.Insert(0, new LogEvent(loggedEvent[i]));
            if (RecentLogEvents.Count >= 50)
            {
                RecentLogEvents.RemoveAt(RecentLogEvents.Count - 1);
            }
        }
    };
    scontext.Post(processLogs, null);
}
```

Code Sample 7: Handling the log event callback

The handler dispatches to a synchronisation context in order to avoid potential deadlocks. It is highly recommended that all callbacks be handled in this way, as described above.

Log events feature localisable log event descriptions (`LogValueExt.DisplayEventDescription`) which can be set on a per session basis. Use the `bool SetCulture(string culture)` method to specify the required culture. A list of available cultures can be obtained via: `List<string> GetRecognisedCultures()`. The default culture is EN.

Handling status events

When status changes, such as doors locking or unlocking, occur on the system, these are alerted to observers via the `OnStatusChangeEvents` method. You register for status events using:

```
void RegisterForStatusEvents(string ClientName);
```

Here is a sample `OnStatusChangeEvents` handler on a client:

[9] This handler simply stores the new log events into an ObservableCollection of recent log events, which are bound to a WPF control.

```
public void OnStatusChangeEvents(StatusValueExt[] status)
{
    SendOrPostCallback processStatus = delegate
    {
        foreach (StatusValueExt state in status)
        {
            // Handle the status change
        }
    };
    scontext.Post(processStatus, null);
}
```

Code Sample 8: Handling status events

Note again how marshalling to explicit synchronisation contexts is used to avoid deadlocks.

Status events are grouped by controller. Any status change on a controller causes a new `StatusValueExt` to be sent to the clients. An example of a status change would be a door going from open to closed, or vice versa.

Getting door information

Doors have two types of identifier: a global door number (unique for the whole system) and a local door number plus controller address combination. Status events use the second addressing scheme. You will need to be able to convert from global to local addresses and vice versa, depending on the context.

To get a list of all doors on the system (and hence all global to local address mappings), use the `GetDoors()` method in something like the following:

```
private void GetListOfAllDoors()
{
    int start = 0;
    int items = 0;
    List<DoorValueExt> doors = new List<DoorValueExt>();
    do
    {
        DoorValueExt[] results = connection.GetDoors(systemIndex: start,
max: int.MaxValue, next: true, enabled: false);
        items = results.Length;
        if (items > 0)
        {
            doors.AddRange(results);
            start = results[items - 1].GlobalDoorNumber + 1;
        }
    } while (items > 0);

    foreach (DoorValueExt item in doors)
    {
        textBox1.Text += string.Format("Door {0} {1} is local door {2} on
controller {3}", item.GlobalDoorNumber, item.Name, item.LocalDoorNumber,
item.ControllerAddress) + System.Environment.NewLine;
    }
}
```

Code Sample 9: Get a listing of all doors

Getting a specific user

To obtain a user's record when you know the user's unique ID, it is only necessary to make a call to `GetUser` and to pass the ID in:

```
UserValueExt GetUser(int index, bool specific, bool order, bool enabled);
```

```
    // Calls the service to get a user
    bool SPECIFIC = true;
    bool DESC = false;
    bool BOTH = false;
    UserValueExt user = proxy.GetUser(index:userNumber, specific:SPECIFIC,
order:DESC, enabled:BOTH);

    if (!user.IsValid)
    {
        Console.WriteLine("That user does not exist");
    }
```

Code Sample 10: Getting a user

The `specific` parameter just specifies the behaviour of the service when the requested user does not exist. Setting `specific=false` means it will skip ahead (or back, if order=false) to the next available user record. Setting it to true will mean it only attempts to get the exact record requested.

The order parameter only operates when specific=false. It can be set to true for ascending, or false for descending.

Finally, if you are only interested in enabled users, then set `enabled=true`. When `enabled=false`, all users are returned (i.e., not just disabled users).

Getting a list of users

When you do not know the user's ID, or where you are trying to list all available users, then you have to use the `GetUsers` method:

```
List<UserValueExt> GetUsers(Dictionary<string, string> matchers,
Dictionary<string, int> exactMatchers, int start, int finish, int
maxCount, bool order, bool enabled, uint enabledOption);
```

A simple query will use `null` for the `matchers` and `exactMatchers` arguments (which we will cover below). Then `GetUsers` returns all users as specified by the start, finish, and other arguments. For example, to retrieve the first 1000 enabled users on the system, use the following code.

The enabledOption argument can be 0 (meaning All users, both enabled and disabled), 1 (meaning only enabled users), or 2 (meaning only disabled users).

```
// Objective: Obtain the first 1000 enabled users in the database
List<UserValueExt> users = new List<UserValueExt>();
int start = 0;
int upperLimit = 10000; // highest user number to be returned
int total = 1000; // amount of users required
int returnedUsers = 0;

do
{
    returnedUsers = 0;
    UserValueExt[] results = proxy.GetUsers(null, null, start, upperLimit,
total - users.Count, true, true, 1);
    if (results != null && results.Length > 0)
    {
        returnedUsers = results.Length;
        users.AddRange(results);
        start = results[returnedUsers - 1].UserNumber + 1; // start at
next available user
    }
} while (returnedUsers > 0 && start < upperLimit && users.Count < total);

Console.WriteLine(string.Format("{0} users found.", users.Count));
foreach (UserValueExt user in users)
{
    Console.WriteLine(string.Format("User {0} is {1} {2}",
user.UserNumber, user.Forename, user.Surname));
}
```

Code Sample 11: Retrieve the first 1000 enabled users

The important points to the note about this code sample are:

- Because of WCF limits on data transfers, you have to obtain the results in a loop.
- The `start` argument must be reset after each iteration ends.
- You can use the `finish` argument to set an upper limit on the user number returned (in the sample it is 10,000). Set this argument to 0 if you want no upper limit.
- You use `maxCount` to determine the maximum number of records returned, but it is only relevant if it is less than the transfer limit.
- For the last argument, use 1 to only return enabled users. 0 returns all users, while 2 returns disabled users only.

A more complex requirement is when you want to filter the returned users. The typical requirement is to filter by `UserGroup` or by a search term in the name. The following code samples illustrate using the `matchers` and `exactMatchers` parameters for these kinds of search.

This first code sample shows how to obtain all users in a user group. You must specify the `UserGroup` as a key/value pair in the `exactMatchers` parameter.

```
    // Objective: Obtain the set of users in a specific user group
    List<UserValueExt> users = new List<UserValueExt>();
    int start = 0;
    int returnedUsers = 0;
    int userGroup = 52; // set to whatever is required...

    Dictionary<string, int> exactSearch = new Dictionary<string, int>();
    exactSearch.Add("Group", userGroup);
    do
    {
        returnedUsers = 0;
        UserValueExt[] results = proxy.GetUsers(null, exactSearch, start,
0, 0, true, true, 1);
        if (results != null && results.Length > 0)
        {
            returnedUsers = results.Length;
            users.AddRange(results);
            start = results[returnedUsers - 1].UserNumber + 1; // start at
next available user
        }
    } while (returnedUsers > 0);
```

Code Sample 12: Get all users in a particular user group

The `matchers` keys[10] are limited to the following:

"Forename"	Looks for matches to the user's forename
"Surname"	Looks for matches to the user's surname
"UserField1".."UserField10"	Looks for matches to the user fields 1 to 10

The `exactMatchers` keys are limited to:

"Group"	Will search on the UserGroup number
"CardNo"	Will find cards matching the key (in any card field)
"UserNumber"	An alternative way to match the user number

The matchers use logical AND to combine, so all of them must match before a result is returned.

Another popular search is to look for user's matching a search string. The following sample gets all users with an "S" (or "s") in their surnames and an "A" (or "a") in their forenames.

[10] Use the exact string specified in the list

```
    // Objective: Obtain the set of users matching a search term
    List<UserValueExt> users = new List<UserValueExt>();
    int start = 0;
    int returnedUsers = 0;

    Dictionary<string, string> nameSearch = new Dictionary<string,
string>();
    nameSearch.Add("Surname", "S");
    nameSearch.Add("Forename", "A");
    do
    {
        returnedUsers = 0;
        UserValueExt[] results = proxy.GetUsers(nameSearch, null, start,
0, 0, true, true, 1);
        if (results != null && results.Length > 0)
        {
            returnedUsers = results.Length;
            users.AddRange(results);
            start = results[returnedUsers - 1].UserNumber + 1; // start at
next available user
        }
    } while (returnedUsers > 0);
```

Code Sample 13: Get users that contain search terms in their names

Adding and updating users

To add a new user to the system, here is the recommended procedure.

You can obtain a blank `UserValueExt` record via the service (optional step). This has default values set for every property. In particular, it allocates the appropriate arrays of the right sizes expected (example, for `card[]`). It will return the next available user number from the database. Note that this number is provisional and is not guaranteed to be available when you go to save the user. If another client is simultaneously accessing the same method, then they'll receive the same provisional user number.

You can choose to ignore this value and try to assign your own value manually. Valid user numbers are normally between 1 and 60,000 only, but the upper limit may be lower on a specific system, depending on the controllers installed. Site-coded systems use the user number to assign a card, so you must explicitly assign the right user number to the right card holder.

You allow for the setting of `UserValueExt` properties[11]. Photographs are added separately (see below).

[11] Note that the ExternallyModified field should be left as ExternalModification.DoNothing. The service only requires this field to be set when a script bypasses the service entirely.

You must set `IsValid` to true. The `IsValid` property is set to false by default. Then call the `InsertUser` method:

```
int InsertUser(UserValueExt newValues, bool UseMyID);
```

The argument `UseMyID` indicates to the service whether the `UserNumber` in the newValues record should be used, or whether the service should automatically add the next available number. The return value is the new user number. A 0 return value indicates an error. All controllers are updated automatically, using a download on the fly.

```
// Optional - get blank UserValue from service
UserValueExt user = proxy.getBlankUserValue();

// Set values as required...
user.Forename = "Patricia";
user.Surname = "Smyth";
user.ActivateOP2 = false;
user.Enabled = true;
user.EndValid = new DateTime(2014, 09, 25);
user.Group = StandardUserGroup;
user.UserFields[0] = Department;
user.UserFields[1] = PhoneNumber;
// You can ask the service for a random and unique pin number for the user
int randomPin = proxy.GetRandomPINForUser();
user.Pin = randomPin;
// Necessary to set IsValid to true before attempting to insert
user.IsValid = true;
// Insert the user, asking for a user number to be generated by the
service
user.UserNumber = proxy.InsertUser(user, false);
if (user.UserNumber == 0)
{
    Console.WriteLine("The user could not be inserted");
}
```

Code Sample 14: Insert a new user

This code also shows how to obtain a random, unique PIN number for the user. You can call the service's `GetRandomPINForUser()` method with a guarantee that it will return a number not shared by any other user on the system[12].

Pin numbers are not displayed for users in the UI of ACT Manage. You will have to decide how to store the unique pin (not shown in code above) and inform users of their number.

[12] The uniqueness guarantee applies to existing users. Obviously, if the same Pin is assigned twice, then it is not unique.

Deleting users

To delete a user, call the method:

```
bool DeleteUser(int user)
```

providing the user number. A return value of true indicates that the user was successfully deleted. Note that all extra rights and door plans for the user are also deleted, but that log events for that user are not deleted. All controllers are updated dynamically.

Exporting users

There are some scenarios where it is useful to export all users, modify the data using Microsoft Excel, and then re-import the users[13].

The API provides for the export of all user data in one easy method call, `ExportAllUsers()`.

```
private void Export_Click(object sender, EventArgs e)
{
    FolderBrowserDialog folder = new FolderBrowserDialog();
    folder.RootFolder = Environment.SpecialFolder.MyComputer;
    bool ok = false;
    string filename = string.Empty;
    if (folder.ShowDialog() == DialogResult.OK)
    {
        filename = folder.SelectedPath +
System.IO.Path.DirectorySeparatorChar + "Users.csv";
        ok = connection.ExportAllUsers(filename);
    }
    if (ok)
    {
        MessageBox.Show("Users exported to: " + filename, "Export",
MessageBoxButtons.OK);
    }
}
```

Code Sample 15: Exporting users

The users are then saved as a CSV file, which can be opened in Microsoft Excel (or other spreadsheets) or as a text file.

Importing users

Another common requirement will be to update user data from a third-party system, typically a HR system. There are several ways to achieve this. As already explained above, you can write a custom

[13] User import and export are also available from the ACT Server tool.

piece of code for reading from the third-party system (presumably through its own API), converting the data to a `UserValue`, and storing it using the `InsertUser()` method.

Alternatively, you can write all user data out to a CSV file from the third-party software, and import that file back into ACT Enterprise. The file can be imported manually using the Server tool with ACT Enterprise. And the API contains an `ImportAllUsers()` method which will import an entire CSV file for you[14].

The `ImportAllUsers` method requires that the imported file be formatted quite precisely. The current format[15] can be obtained by exporting the user list (with at least one user). The columns exported match the required import columns.

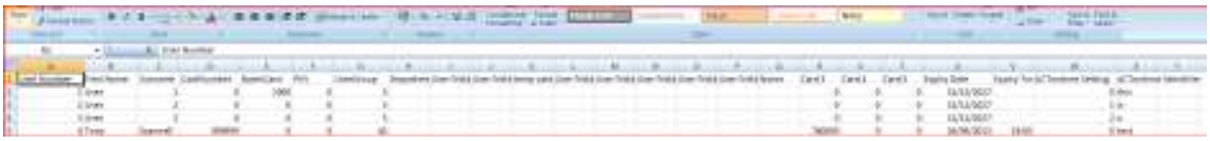

Figure 3: Viewing a user CSV file in Microsoft Excel

Note that the User Field 1…User Field 10 fields will have a column heading to match the customised heading set by the customer.

```
private void Import_Click(object sender, EventArgs e)
{
    bool ok = false;
    OpenFileDialog file = new OpenFileDialog();
    file.Filter = "CSV files | *.csv";
    file.InitialDirectory =
Environment.GetFolderPath(Environment.SpecialFolder.MyComputer);
    file.Title = "Import Users";
    if (file.ShowDialog() == DialogResult.OK)
    {
        ok = connection.ImportAllUsers(file.FileName, deleteExisting:
false);
    }
    if (ok)
        MessageBox.Show("Users are being imported from: " + file.FileName,
"Import", MessageBoxButtons.OK);
}
```

Code Sample 16: Importing users from a CSV file

To discard existing users, set the `deleteExisting` argument to true. To retain existing users, set `deleteExisting` to false. It is important to note: where imported users have the same User Number as existing users, the existing user is overwritten. There is no way to merge values.

[14] As long as the CSV file is in the correct format.
[15] Which is subject to change

Managing user photographs

User photographs (in JPEG format) can be stored into the database. These are handled separately to all other user values, to make the system more streamlined and efficient.

A `UserValueExt` record will indicate whether a user has a photograph assigned or not, in the `HasPhotograph` property.

Photographs are transmitted to and from the service in "chunks", which are blocks of byte data with an upper size limit. You can get the current chunk size defined by the service using:

```
int GetChunkSize();
```

To then read a user's photograph, it must be pulled back in chunks:

```
byte[] GetUserPhotoChunk(int user, ref int chunk, ref bool isLastChunk);
```

Here is sample code for getting a user's photograph and displaying it in a picture box control:

```
if (user.HasPhotograph)
{
    int chunk = 0;
    bool islast = false;
    int sizeOfChunk = proxy.GetChunkSize();
    List<byte> totalImage = new List<byte>(sizeOfChunk);
    do
    {
       byte[] image = proxy.GetUserPhotoChunk(user.UserNumber, ref chunk,
ref islast);
       totalImage.AddRange(image);
    } while (!islast);

    try
    {
        Image pic = Image.FromStream(new MemoryStream
(totalImage.ToArray()) );
        this.pictureBox1.Image = pic;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Could not display image: " + ex.Message);
        this.pictureBox1.Image = null;
    }
}
```

Code Sample 17: Getting a user's photograph

Inserting a photograph is similar, but in reverse. Each chunk of data must be transferred, until the last chunk is sent, when the lastChunkOK argument must be set to true[16]. This triggers the uploading of the photograph to the user's record.

```
if (pictureBox1.Image != null)
{
    Image img = pictureBox1.Image;
    MemoryStream ms = new MemoryStream();
    img.Save(ms, pictureBox1.Image.RawFormat);
    byte[] bytes = ms.ToArray();
    int chunksize = proxy.GetChunkSize();
    int len = bytes.Length;
    bool last = false;
    for (int i = 0; i <= len / chunksize; i++)
    {
        int numOfBytes = (int)Math.Min(chunksize, len - (i * chunksize));
        byte[] buffer = new byte[numOfBytes];
        Array.Copy(bytes, i * chunksize, buffer, 0, numOfBytes);
        last = ((i * chunksize) + numOfBytes >= len);
        proxy.InsertUserPhotoChunk(user.UserNumber, buffer, last);
    }
}
```

Code Sample 18: Setting a user's photograph

Generating Muster Reports

A muster[17] report will return a list of all users currently on-site. To obtain a muster report, you must make a call to `GetLogsOfUserTracking()` passing in a request for of 0 for a User Tracking Report Type of Muster. Generally speaking, you will want a valid muster report for today, so you specify a start time of 00:00 today and an end time of `DateTime.Now`.

For example, to get all users on site right now, you can issue this report:

[16] The service relies on chunks being transferred sequentially, for the same user and in the right order, and always within the same WCF session.

[17] In order for mustering to be valid for a site, you must have a properly defined perimeter door group, and users must swipe in and out when they enter or exit the building(s).

```
List<UserTrackValueExt> musterUsers = new List<UserTrackValueExt>();
int startUser = 0;
int items = 0;
do
{
    DateTime start = new DateTime(DateTime.Now.Year, DateTime.Now.Month,
DateTime.Now.Day, 0, 0, 0);
    UserTrackValueExt[] results = connection.GetLogsOfUserTracking // [1]
        (type: 0, startWhen: start, endWhen: DateTime.Now, doorGroup: 0,
userGroup: 0, // [2]
perimeter: false, antipassback: false, // [3]
monitored: false, enabledUser: true, startUser: startUser, maxCount:
int.MaxValue, order:true, OptionalUserField:1);
    items = results.Length;
    if (items > 0)
    {
        musterUsers.AddRange(results);
        startUser = results[items - 1].UserNumber + 1; // [4]
    }
} while (items > 0);

foreach (UserTrackValueExt user in musterUsers)
{
    textBox1.Text += string.Format("User {0} {1} ", user.UserNumber,
user.UserName) + System.Environment.NewLine;
}
```

Code Sample 19: Obtaining a muster report for all users on any door

Here are the important points about this code sample:

1. The method's arguments have been named in order to make them clearer.
2. The available types are Muster=0, LastEntry=1, LastEntryOrExit=2, and LastLocation=3.
3. The important thing to know when trying to get all users in a muster report is to not specify any door group or user group.
4. Set perimeter and antipassback to false when looking for users on all doors.
5. As for other complex queries which might return lots of data, you have to request the results in a loop, making sure to increment the startUser argument on each iteration.

Where you have a specific set of perimeter doors defined, then you need to specify perimeter=true. Only access granteds/exit granteds on the perimeter's doors will then count for the mustering.

```
List<UserTrackValueExt> musterUsers = new List<UserTrackValueExt>();
int startUser = 0;
int items = 0;
do
{
    DateTime start = new DateTime(DateTime.Now.Year, DateTime.Now.Month,
DateTime.Now.Day, 0, 0, 0);
    UserTrackValueExt[] results = connection.GetLogsOfUserTracking
        (type: 0, startWhen: start, endWhen: DateTime.Now, doorGroup:0,
perimeter:true, antipassback:false, userGroup:1, monitored:false,
enabledUser:true, startUser:startUser,         maxCount:int.MaxValue,
order:true, OptionalUserField:1);
    items = results.Length;
    if (items > 0)
    {
        musterUsers.AddRange(results);
        startUser = results[items - 1].UserNumber + 1;
    }
} while (items > 0);

foreach (UserTrackValueExt user in musterUsers)
{
    textBox1.Text += string.Format("User {0} {1} ", user.UserNumber,
user.UserName) + System.Environment.NewLine;
}
```

Code Sample 20: Getting a muster report for perimeter doors only

There are other UserTrackingReportTypes (passed as uints) for getting the last locations of users, or their last entry/exit. It is important that the doors you choose should reflect the way the system has been installed and configured at a hardware level.

Issuing commands on doors and controllers

You can issue commands on doors (and controllers) on the system. You need to create a Command object and then set its properties appropriately.

```
// Lock all doors on the system
int commandNumber = 1;
foreach (DoorValueExt door in doors)
{
    CommandExt lockDoor = connection.GetBlankCommand(); // [1]
    // Set controller address and local door numbers
    lockDoor.Controller = door.ControllerAddress;
    lockDoor.Door = (byte)door.LocalDoorNumber; // [2]
    // Set the type as Door command and the instruction
    lockDoor.Type = 2; // [3]
    lockDoor.DoorCommandInstruction = 2; // [4]
    // Optional ID number for the command
    lockDoor.CommandID = commandNumber++; // [5]
    // Issue command
    bool ok = connection.IssueCommand(lockDoor); // [6]
}
```

Code Sample 21: Lock all doors on the system

For example, in the code sample, we have a list of all doors on the system in doors. It is then relatively easy to lock them all.

1. A blank command can, optionally, be obtained via a call to `GetBlankCommand()`.
2. You must set the controller address and local door number.
3. For door commands, you have to set the command's Type to 2.
4. You then set the `DoorCommandInstruction` to the right command. In our case, 2 stands for Lock. See below for a full list of available commands. For controllers, you'd be setting `ControllerCommandInstruction`.
5. There is an optional `CommandID` which can help in auditing commands.
6. Finally, you issue the command using the `IssueCommand()` method. `IssueCommand` returns a boolean true if the command was successfully issued.

Appendix: Enums replaced by uints

```
public enum ACTCommandType
{
    None = 0,
    Device = 1,
    Door = 2,
    IOModule = 3
}

public enum ACTontimeUserSetting
{
    UserGroupDefault = 0,
    Enabled,
    Disabled
}

public enum CardTypes : byte
{
    Unknown = 0x00,
    LearnedCards = 0x01,
    OneToOneCards = 0x02,
    SiteCodedCards1 = 0x04,
    SiteCodedCards2 = 0x08,
    BatchCards = 0x10,
    ThisUserHasMultipleCards = 0x20
}

public enum DatabaseState
{
    NoValidConnectionString,
    DatabaseClosed,
    DatabaseOpenOK,
    CannotContactDatabase,
    DatabaseOpenOKButEmpty,
    DatabaseNotOpenedYet
}

public enum DeviceCommands : byte
{
    NormalizeDoors = 0,
    ActivateDoorRelays = 1,
    LockDoors = 2,
    UnlockDoors = 3,
    OutputsOn = 4,
    OutputsOff = 5,
    ActivateOutputs = 6,
    ClearAntipassback = 32,
    ClearMuster = 33,
    SetTracking = 34,
    CaptureCard = 48,
    GetSignature = 49,
    GetCardData = 50,
    StartFlashUpdate = 51,
    FlashSRecordData = 52,
    EndFlashUpdate = 53,
    FlashRawData = 54,
    FindDoorNumber = 61,
    ProgramDoorNumbers = 62,
    SetCurrentAreaCount = 63,
```

```
        GetCurrentAreaCount = 64,
        StartRemoteDiscovery = 65, // added this and the remaining commands for Act
4000
        GetDoorDiscovery = 66,
        GetIOMDiscovery = 67,
        StartCoughMode = 68,
        EndCoughMode = 69,
        CoughNow = 70,
        GetDoorDiscovery1732 = 71,
        DeviceOutputOff = 72,
        DeviceOutputOn = 73,
        SetControllerAddress = 80,
        SetFixedIPAddress = 81,
        DHCP = 82,
        StartLockDiscovery_DEPRECATED = 83,
        EndLockDiscovery_DEPRECATED = 84,
        StartRFMacTest = 85,
        ClearRFTestResults_DEPRECATED = 86,
        RemoveRFPersistentConnection = 90,
        PrepareRFTestResults_DEPRECATED = 91,
        EndRFMacTest = 92
    }

    public enum DeviceType
    {
        Unknown = 0,
        ACT1000 = 1,
        ACT2000 = 2,
        ACT3000 = 3,
        ACT4000 = 4,
        ACT1500 = 5,
        ACTeLock = 6,
        ACT1520e = 7, // Jan 2016
        DS100 = 16,
        DS200 = 17,
        IOModule = 18,
        Lock125KHz = 19,
        LockMifare = 20,
        ACTSmartController = 28,
        ACTSmartPinandProx = 29,
        ACTSmartPinOnly = 30,
        ACTSmartProxOnly = 31
    }

    public enum DoorCommands : byte
    {
        Normalize = 0,
        ActivateRelay = 1,
        LockDoor = 2,
        UnlockDoor = 3,
        OutputOn = 4,
        OutputOff = 5,
        ActivateOutput = 6,
        TimedActivateRelay = 7,
        RandomChallengeEntry = 8,
        RandomChallengeExit = 9,
        RandomChallengeOff = 10,
        RequestStartTests = 11,
        AVISGrant = 12,
        CaptureCard = 48,
```

```
    GetSignature = 49,
    GetCardData = 50,
    StartFlashUpdate = 51,
    EndFlashUpdate = 52,
    FlashSRecordData = 53,
    FlashRawData = 54,
    FindDoorNumber = 61
}

public enum IOModuleCommands : byte
{
    OutputRelayOff = 0,
    OutputRelayOn = 1,
    InputEnable = 2,
    InputDisable = 3
}

public enum EnabledField
{
    All,
    EnabledOnly,
    DisabledOnly
}


public enum EstablishSessionResult
{
    SuccessfulLogin = 1,
    OnlyServerClientAllowedAccess,
    DBNameIsBlank,
    DBNameDoesNotExist,
    PasswordInvalid,
    ServicePreventingAccess,
    AlreadyEstablishedSession,
    UnknownError,
    DeniedAccess,
    ClientsLicencedExceeded,
    TrialPeriodEnded,
    FallbackLicenceDenied
}

public enum EventLocation
{
    NoLocation,
    Door,
    IOModule,
    Controller,
    PC
}

public enum ExternalModification
{
    DoNothing = 0,
    Modified = 1,
    New = 2,
    Delete = 3,
    Modified_BUSY = 100,
    New_BUSY = 200,
    Delete_BUSY = 300
}
```

```
public enum ExtraCardType : byte
{
    None = 0x00,
    Card3 = 0x01,
    Card4 = 0x02,
    Card5 = 0x04,
    OneToOne = 0x08,
    AllTypes = 0x0F
}

public enum LockFault
{
    NoFault = 0,
    MBDecryptFail = 1,
    MBCommFail = 2,
    PivotStuckUp = 3,
    PivotStuckDown = 4
}

public enum LogEventCategory : uint
{
    None = 0x0000,
    AccessGranted = 0x0001,
    AccessDenied = 0x0002,
    Alarm = 0x0004,
    Door = 0x0008,
    Operator = 0x0010,
    System = 0x0020,
    Entry = 0x0100,
    Exit = 0x0200,
    Manual = 0x0400,
    PC = 0x8000,
    User = 0x0003,
    Normal = 0x000F,
    Other = 0x78F0,
    Undefined = 0x78C0,
    All = 0xFFFF
}


public enum LogEventType : byte
{
    EVT_NOACTION = 0,
    EVT_READ2 = 1,
    EVT_READ1 = 2,
    EVT_READERR = 3,
    EVT_RANGE = 4,
    EVT_PININ = 5,
    EVT_PINOUT = 6,
    EVT_TAMPER = 7,
    EVT_RESET = 8,
    EVT_POWERUP = 9,
    EVT_MAINS = 10,
    EVT_BUTTON = 11,
    EVT_OPEN = 12,
    EVT_CLOSED = 13,
    EVT_AJAR = 14,
    EVT_FORCED = 15,
```

```
EVT_READ = 16,
EVT_INFO = 17,
EVT_UNKNOWN = 18,
EVT_BADISSUE = 19,
EVT_NOTAMPER = 20,
EVT_READPWR = 21,
EVT_MAINS_RESTORE = 22,
EVT_AUX_CLOSED = 23,
EVT_AUX_OPEN = 24,
EVT_FUSEOK = 25,
EVT_FUSEBLOWN = 26,
EVT_PINTMO = 27,
EVT_HISTORIC = 28,
EVT_NOTPROG = 29,
EVT_SPARE30 = 30,
EVT_INTERLOCK = 31,
EVT_IOM_OP_ON = 32,
EVT_IOM_OP_OFF = 33,
EVT_IOM_IP_ACTIVE = 34,
EVT_IOM_IP_NORM = 35,
EVT_IOM_IP_SHORT = 36,
EVT_IOM_IP_DISCON = 37,
EVT_IOM_IP_ENABLE = 38,
EVT_IOM_IP_DISABLE = 39,
EVT_DONOTUSE40 = 40,
EVT_DONOTUSE41 = 41,
EVT_IOM_IP_ALARM = 42,
EVT_BATTLOW = 43,
EVT_BATTOK = 44,
EVT_READERFUSEBLOWN = 45,
EVT_READERFUSEOK = 46,
EVT_SPARE47,
EVT_MENU = 48,
EVT_SPARE49 = 49,
EVT_GRANTED1 = 50,
EVT_DENIED1 = 51,
EVT_GRANTED2 = 52,
EVT_DENIED2 = 53,
EVT_BADSITE = 54,
EVT_UNRECOG = 55,
EVT_BADTIME = 56,
EVT_BADPIN = 57,
EVT_DURESS = 58,
EVT_TIMESET = 59,
EVT_OFFLINE = 60,
EVT_ONLINE = 61,
EVT_DEFAULTED = 62,
EVT_LOGRESET = 63,
EVT_NORMAL = 64,
EVT_LOCKED = 65,
EVT_UNLOCKED = 66,
EVT_NOTSPARE67 = 67,
EVT_FIREDOOR = 68,
EVT_ANTIPASS = 69,
EVT_LOGIN = 70,
EVT_LOGOUT = 71,
EVT_MRESET = 72,
EVT_APRESET = 73,
EVT_SPARE74 = 74,
EVT_SPARE75 = 75,
```

```
EVT_USERLIMIT = 76,
EVT_NOTVALID = 77,
EVT_PASS = 78,
EVT_ANTIPASS_IN = 79,
EVT_ANTIPASS_OUT = 80,
EVT_LOGOVERFLOW = 81,
EVT_DOOR_DISARM = 82,
EVT_DOOR_ARM = 83,
EVT_IO_OFFLINE = 84,
EVT_IO_ONLINE = 85,
EVT_RANDOMDENIED1 = 86,
EVT_RANDOMDENIED2 = 87,
EVT_RANDOMGRANTED = 88,
EVT_TWINDENIED1 = 89,
EVT_TWINDENIED2 = 90,
EVT_CLOCKIN = 91,
EVT_CLOCKOUT = 92,
EVT_DENIED_AREA = 93,
EVT_DENIED_AREAGRP = 94,
EVT_COUNTAREA_RESET = 95,
EVT_LREAD = 96,
EVT_INPUT = 97,
EVT_BUTTONUP = 98,
EVT_COMMAND = 99,
EVT_KEYPAD = 100,
EVT_MINUTE = 102,
EVT_KEYPAD_IN = 103,
EVT_KEYPAD_OUT = 104,
EVT_IOM_COMMAND = 105,
EVT_CEN_SAT_ESTAB = 106,
EVT_CEN_SAT_LOST = 107,
EVT_DENIED1_INTERNAL = 108,
EVT_DENIED2_INTERNAL = 109,
EVT_EXT_FLASH_READY = 110,
EVT_BREAKGLASS = 111,
EVT_V_SUPPLY_OK = 112,
EVT_V_SUPPLY_WARN = 113,
EVT_DR_SUPPLY_V = 114,
EVT_IOM_SUPPLY_V = 115,
EVT_MEM_FAIL = 116,
RFU8 = 117,
RFU9 = 118,
RFU10 = 119,
RFU11 = 120,
RFU12 = 121,
RFU13 = 122,
RFU14 = 123,
RFU15 = 124,
RFU16 = 125,
RFU17 = 126,
RFU18 = 127,
PC_EVT_START = 128, /* Program Started */
PC_EVT_EXIT = 129,         /* Program Exited */
PC_EVT_OFFLINE = 130,              /* Device Offline */
PC_EVT_ONLINE = 131,               /* Device Online */
PC_EVT_NWFAIL = 132,               /* Network Fail */
PC_EVT_MRESET = 133,               /* Muster Reset */
PC_EVT_APRESET = 134,              /* Antipassback Reset */
PC_EVT_DOWNLOADED = 135,   /* Downloaded OK */
PC_EVT_DLDFAIL = 136,              /* Download failed */
```

```
        PC_EVT_ACKNOWLEDGE = 137,  /* Alarm event ackn. */
        PC_EVT_NEWDB_CREATED = 138, // New database created
        PC_EVT_FW_UPDT_SUCCD = 139, // Firmware update succeeded
        PC_EVT_FW_UPDT_FAILD = 140, // Firmware update failed
        PC_EVT_UPLOADFAIL = 141,   // Upload failed
        PC_EVT_UPLOADED = 142,           //Upload successful
        PC_EVT_COMMS_TIMEOUT = 143,//Comms Timeout in SiteServer
        LAST_EVENT_NUMBER = 144, // SHOULD NOT HAPPEN
        EVT_BATTERY_INSERT = 145, // Battery inserted into eLock
        EVT_RFCOMMS_FAIL = 146, // Communications failed between eLock and controller
        EVT_LOCK_FAULT = 147, //  Fault detected by eLock
        EVT_LOCK_FAULT_CLEAR = 148, // Fault cleared on the eLock
        EVT_HANDLE_HELD = 149, // eLock handle held down
        EVT_CLONED_CARD = 150,
        EVT_USB_CONNECT = 151,
        EVT_USB_DISCONNECT = 152,
        EVT_GRANT_WHITE = 153,
        EVT_GRANT_UNKN_WHITE = 154,
        EVT_GRANT_UNKN_CARD = 155,
        EVT_RFCOMMS_RESTORE = 156,
        EVT_BATTERY_CRITICAL = 157,
        EVT_BATTERY_FAIL = 158,
        PC_EVT_ACTONTIME_SERVICE_DOWN = 159,
        PC_EVT_ACTONTIME_SERVICE_UP = 160,
        PC_EVT_DBUSER_LOCK = 161, // DBUser lock command issued
        PC_EVT_DBUSER_UNLOCK = 162, // DBUser unlock command issued
        PC_EVT_DBUSER_PASS = 163, // DBUser pass command issued
        PC_EVT_DBUSER_NORMALISE = 164, // DBUser normalise command issued
        EVT_UNLOCK_ON_EXIT = 165, // Door unlocked on exit
        EVT_UNLOCK_ON_EXIT_NORMALIZED = 166, // Normalize door in unlocked on exit
state may contain the user number of card presented
        EVT_ELOCK_EGRESS = 167, // Handle used from inside
        PC_EVT_ACTONTIME_PUNCH_SKIPPED = 168,
        EVT_VISITOR_GRANTED = 169,
        EVT_VISITOR_EXIT_GRANTED = 170,
        EVT_PIN_ENTRY = 171, // internal event that is not logged
        EVT_RENTAL_EXPIRED = 172, // the rental agreement expired before it was used
        EVT_RENTAL_CANCELLED = 173, // the rental agreement was cancelled by an agent
        EVT_RENTAL_USED = 174, // the car was taken out of the car park after a user
entered a valid pin
        EVT_ANPR_IN = 175, // a car/vehicle approaches a barrier, direction IN
        EVT_ANPR_OUT = 176,  // a car/vehicle approaches a barrier, direction OUT
        EVT_RENTAL_INVALIDATED = 177, // a number of invalid pins have been entered at
a barrier so the rental agreement has been invalidated
        EVT_RENTAL_ISSUED = 178,
        PC_EVT_ACTONTIME_ERROR = 179,
        PC_EVT_ACTONTIME_INFO = 180
    }

    public enum MemoryFailure
    {
        NoError = 0,
        NANDFlashFatalError = 1,
        NANDFlashCorrupted = 2,
        MemoryCardFatalError = 3,
        InvalidMemoryCard = 4,
        MemoryCardCorrupted = 5
    }

    public enum ReplyCode : byte
```

```
{
    OK,
    Ack,
    Pause,
    Retransmit,
    Busy,
    Abandon,
    Cough,
    NoReplyReceived,
    InvalidChecksum,
    Duplicate
}


public enum SignalQuality
{
    Unknown = 0,
    Good = 1,
    OK = 2,
    Bad = 3
}

public enum UserTrackingReportType
{
    Muster,
    LastEntry,
    LastEntryOrExit,
    LastLocation
}
```