NI-ATE Core User Manual

2025-03-26

Contents

NI-ATE Core Overview

NI-ATE Core is driver software that supports communication with an ATE Core Configuration Generation 2 (ATECCGEN2) rack. The NI-ATE Core User Manual User Manual provides detailed descriptions of product functionality and examples for use.

NI-ATE Core provides an API for controlling rack components and monitoring rack health and power. Use the NI-ATE Core C# library to create a Microsoft C# application that connects to a rack over a network to monitor and control rack components or sensors. The NI-ATE Core C# library also provides functions to perform rack operations such as updating firmware, changing rack power state, or rebooting the rack.

Looking for something else?

For information not found in the user manual for your product, like API reference or hardware documentation, browse Related Information.

Related information:

- NI-ATE Core API Reference Manual
- ATE Core Configurations Generation 2 System Components (in ATE Core Configurations Generation 2 User Manual)

NI-ATE Core System Terminology

Before you begin building your application, review the following terms related to managing an ATECCGEN2 rack with NI-ATE Core software.

Table 1. Terms Related to ATECCGEN2 Rack Management

Term	Description
ATE Core Configuration GEN2 (ATECCGEN2) Rack	Refers to an actual rack that can be accessed by a host over a network. A functional rack includes multiple rack components. The subsequent table provides component descriptions. Users can insert payloads, such as a NI PXIe chassis, into the slots of the rack.

Host	A controller or a PC that uses NI-ATE Core software to establish a session to monitor or control the rack over the network.
Payload	Items, such as a NI PXIe chassis, that users insert into the slots of the rack.
RCU Service	A service that runs automatically in the RCU embedded operating system when the rack is turned on. The RCU service initializes all rack components during rack initialization. In normal operation, it monitors and controls component health states. It also monitors and controls power supplies to components and payloads. Some advanced functions, like firmware updates, are also handled by RCU service.

Related information:

- ATE Core Configurations Generation 2 System Components (in ATE Core Configurations Generation 2 User Manual)

NI-ATE Core System Components

Before you begin building your application, review the following summary of rack components and their function. For more detailed documentation, refer to the hardware user manual.

The NI-ATE Core API allows you to interact with each rack component and any component sensors. Component sensors differ depending on the component type.

Table 2. ATECCGEN2 Rack Component Summary

Component	Description
Rack Control Unit (RCU)	The embedded controller in the rack. The RCU communicates with hosts over the network and controls rack components using internal electrical buses.
Fan Panel	Exhausts hot air out of the rack. Each panel has 3 fans and temperature sensors.
Power Entry Panel (PEP)	Rack entry point for facility power. The PEP also contains the safety disconnect relays which open when the Emergency Power Off (EMO) button is pressed.

AC Power Distribution Unit (AC PDU)	Receives single-phase power from the PEP and distributes it to equipment in the rack, including DC PDUs.
DC Power Distribution Unit (DC PDU)	Supplies DC power to devices or payloads in the rack.
Rack Temperature Sensor Unit	Temperature sensor units installed throughout a rack to monitor the temperature of the rack.

Related information:

- ATE Core Configurations Generation 2 System Components (in ATE Core Configurations Generation 2 User Manual)

Installing NI-ATE Core

Visit https://www.ni.com/download and search for NI-ATE Core to download and install NI-ATE Core. You can also use Package Manager to install NI-ATE Core.

NI-ATE Core New Features and Changes

Learn about updates, including new features and behavior changes, introduced in each version of NI-ATE Core.

Discover what is new in the latest releases of NI-ATE Core.

> **Note** If you cannot find new features and changes for your version, it might not include user-facing updates. However, your version might include non-visible changes such as bug fixes and compatibility updates. For information about non-visible changes, refer to your product ***Release Notes***.

NI-ATE Core 2025 Q1 Changes

- Added support for Hardware Configuration Utility (HWCU).

NI-ATE Core 2024 Q4 Changes

- Added API support for uninterruptable power supply (UPS).
- Added APIs to monitor interlock switch states.

Related reference:

- Getting Safety Interlock Status

NI-ATE Core 2024 Q2 Changes

- Added API support for tower light color control.
- Added password saving functionality to `PasswordManager` class, which allows you to save the password you use to access a target rack from a host.

Related reference:

- Credential Management

NI-ATE Core 2024 Q1 Changes

- Initial release.

Namespace, Minimum .NET Framework Version, and Dependencies

Use the NI-ATE Core C# library to create a Microsoft C# application that connects to the ATECCGEN2 rack over a network to monitor and control rack components or sensors.

The NI-ATE Core C# library is built with .NET Framework 4.5. Your user application must support .NET Framework 4.5 or above to use the NI-ATE Core C# library. The NI-ATE Core C# API is defined within the `NationalInstruments.AteCore` namespace. You must include this namespace to access and use the API.

Including the NI-ATE Core Namespace

```
using NationalInstruments.AteCore;
```

Dependencies

The NI-ATE Core C# library depends on third-party libraries, which are listed below and which are installed along with NI-ATE Core. You must include these dependencies in your application.

Table 3. NI-ATE Core C# Library Third-Party Dependencies

Third-Party Library	Version	Installation Path
Newtonsoft (Newtonsoft.Json.dll)	12.0.3	\Program Files (x86)\National Instruments\NI-ATE Core\Bin

Configuring a Basic C# Application in Visual Studio

This topic outlines how to create a C# application with NI-ATE Core using Microsoft Visual Studio.

1. Create a new .NET Framework 4.5.2 project.
2. In the Solution Explorer, right-click your project and select **Unload Project**.
3. Add the NI-ATE Core reference to the `CSPROJ` file and save the file.

```
<Project …> …
        <ItemGroup>
                <Reference Include="NationalInstruments.AteCore.Fx45" />
                        …
        </ItemGroup>
        …
</Project>
```

4. In the Solution Explorer, right-click your project and select **Reload Project with Dependencies**.
5. View the **Reference Properties** to see the reference path. The NI-ATE Core DLL loads automatically from the global assembly cache (GAC).
6. Include the namespace in your `cs` file: `using NationalInstruments.AteCore;`
7. Add a reference to the Newtonsoft driver dependency:
   a. In the Solution Explorer, expand your project to ensure the project contents are visible.
   b. Within your project, right-click on **References** and select **Add Reference**.
   c. Select **Browse** and click the **Browse** button.
   d. Navigate to `C:\Program Files (x86)\National Instruments\NI-ATE Core\Bin`.
   e. Select `Newtonsoft.Json.dll`.
   f. Click **Add**.
   g. Click **OK**.

You can now write your remaining logic and build your program.

Creating and Disposing of Sessions

To interact with the ATECCGEN2 rack, you must create an `IAteCoreSession` object using the static function `CreateAteCoreSessionAsync` from `AteCoreSession`. This static function takes in the following arguments to connect to a rack from a host:

- Hostname/IP address
- Password
- Optional `rememberPassword` parameter

When the session object is returned from `CreateAteCoreSessionAsync`, the network connection is successfully established and you can use the session to call the properties and functions of different classes and interfaces to perform the desired operation. Call `Dispose` to close the network connection.

Session Creation and Session Disposal

```
IAteCoreSession ateCoreSession = null;
try
{
    // To create an object of IAteCoreSession.
    ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password, true);

    // Call another function to perform an operation on the ATECCGEN2 rack.
}
catch(Exception ex)
{
    Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
    if (ateCoreSession != null)
    {
        // To close the session.
        ateCoreSession.Dispose();
    }
}
```

Related reference:

- Credential Management

Managing Errors and Exceptions

All functions and properties should be called within a try-catch block. When an error occurs in a function, an exception will be thrown and caught by the catch block. You can retrieve the `Data["ErrorCode"]` and `Data["ErrorMessage"]` in the exception to determine the error. The NI-ATE Core C# library has a defined list of error codes in the class `AteCoreStatusCode` which you can compare to the value of the error code in the exception.

Catching Exceptions during Session Creation

```
using NationalInstruments.AteCore;
using NationalInstruments.AteCore.StatusCode;

IAteCoreSession ateCoreSession = null;
try
{
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);
}
catch (Exception ex)
{
        if
(ex.Data["ErrorCode"].Equals(AteCoreStatusCode.NIATECORE_ERR_INCORRECT_PASSWORD))
        {
                Console.WriteLine("Exception: {0}", ex.Data["ErrorMessage"]);
        }
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Components and Sensors

The ATECCGEN2 rack contains several types of components with distinct component classes. Component classes are inherited from `IComponent`.

The NI-ATE Core API allows you to interact with each rack component and to access the sensors inside the component. Component sensors differ depending on the component type. To get the list of detected rack components, create a session and call the corresponding property of `IAteCoreSession`.

Table 4. NI-ATE Core C# Component Interfaces

Component	C# Interface
Rack Control Unit (RCU)	`IRackControlUnit`
Fan Panel	`IFanPanel` (Accessed via the `FanPanels` property of `IFanDomain`)
Power Entry Panel (PEP)	`IPowerEntryPanel`
AC Power Distribution Unit (AC PDU)	`IACPowerDistributionUnit`
DC Power Distribution Unit (DC PDU)	`IDCPowerDistributionUnit`
Uninterruptable Power Supply	`IUninterruptablePowerSupply`
Rack Temperature Sensor Unit	`IRackTemperatureSensorUnit`

AC Power Distribution Unit (AC PDU)

The AC power distribution unit (AC PDU) supplies AC power to payloads connected to AC PDU power outlets. You can read the current sensor on the AC power outlet to monitor the AC power supply on each outlet. Outlets are grouped as power banks. Each power bank can be turned off (`TurnOffPowerBankAsync()`) and on (`TurnOnPowerBankAsync()`) individually. To get the current state of a power bank use `GetPowerBankStateAsync()`. The electric current measured at each power outlet can be read independently by accessing the `IOutletSensor` objects stored in each power bank (`IPowerBank`).

Note AC PDU control and sensor readings are only accessible when the ATECCGEN2 rack is in the `Running` power state. When the rack is not in the `Running` power state, attempting to read a sensor returns `Double.NaN` and attempting to enable or disable a power bank triggers an exception.

Note A payload that discharges slowly can cause the AC PDU bank state detection circuitry to report that the bank is on. Add a delay in cases where you call `GetPowerBankStateAsync()` after `TurnOffPowerBankAsync()` while connected to a payload that discharges slowly.

Reading All Outlet Sensors on All AC Power Distribution Unit Banks

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);
        // Get the objects of ACPDU component
        IACPowerDistributionUnit[] acPowerDistributionUnits =
ateCoreSession.ACPowerDistributionUnits;

        foreach (var acPowerDistributionUnit in acPowerDistributionUnits)
        {
                // Print some of the properties of the component
                Console.WriteLine("AC power distribution unit name: {0}",
acPowerDistributionUnit.Name);
                Console.WriteLine("AC power distribution unit location: {0}",
acPowerDistributionUnit.Location);
                Console.WriteLine("AC power distribution unit vendor name: {0}",
acPowerDistributionUnit.VendorName);
                Console.WriteLine("AC power distribution unit model name: {0}",
acPowerDistributionUnit.ModelName);
                Console.WriteLine("AC power distribution unit serial number: {0}",
acPowerDistributionUnit.SerialNumber);
                Console.WriteLine("AC power distribution unit firmware version:
{0}", acPowerDistributionUnit.FirmwareVersion);

                foreach (var powerBank in acPowerDistributionUnit.PowerBanks)
                {
                        if (await powerBank.GetPowerBankState() == PowerState.On)
```

```
                        {
                                Console.WriteLine(powerBank.BankNumber);

                                foreach (var outletCurrentSensor in
powerBank.OutletCurrentSensors)
                                {
                                        Console.WriteLine("Outlet number: {0}",
outletCurrentSensor.OutletNumber);
                                        Console.WriteLine("Current sensor reading:
{0} A", await outletCurrentSensor.ReadSensorAsync());
                                }
                        }
                }
        }
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

DC Power Distribution Unit (DC PDU)

The DC power distribution unit (DC PDU) supplies DC power to payloads connected to DC power outlets. You can read the current sensor and voltage sensor on the DC power outlet to monitor the DC power supply. Each outlet can be turned off and on individually.

In certain circumstances, a rack with DC PDU installed may return an empty `DCPowerDistributionUnits` array from `IAteCoreSession`. A DC PDU is only detected when the `RackPowerState` has been set to the `Running` state at least once. To set the rack power state to `Running` state, press the power button on the rack or call `SetRackPowerStateToRunningAsync`. Then, trigger the system to update the DC PDU list by calling `RefreshDCPowerDistributionUnitsAsync`.

Note DC PDU control and sensor readings are only accessible when the ATECCGEN2 rack is in the `Running` power state. When the rack is not in the `Running` power state, attempting to read a sensor returns `Double.NaN` and attempting to enable or disable an outlet triggers an exception.

Checking Rack Power State, Setting Power State to Running, Calling RefreshDCPowerDistributionUnitsAsync, and Reading Current and Voltage Sensors for all DC Power Outlets

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);
        // Get the objects of ACPDU component
        IACPowerDistributionUnit[] acPowerDistributionUnits =
ateCoreSession.ACPowerDistributionUnits;

        foreach (var acPowerDistributionUnit in acPowerDistributionUnits)
        {
                // Print some of the properties of the component
                Console.WriteLine("AC power distribution unit name: {0}",
acPowerDistributionUnit.Name);
                Console.WriteLine("AC power distribution unit location: {0}",
acPowerDistributionUnit.Location);
                Console.WriteLine("AC power distribution unit vendor name: {0}",
acPowerDistributionUnit.VendorName);
                Console.WriteLine("AC power distribution unit model name: {0}",
acPowerDistributionUnit.ModelName);
                Console.WriteLine("AC power distribution unit serial number: {0}",
acPowerDistributionUnit.SerialNumber);
                Console.WriteLine("AC power distribution unit firmware version:
{0}", acPowerDistributionUnit.FirmwareVersion);

                foreach (var powerBank in acPowerDistributionUnit.PowerBanks)
                {
                        if (await powerBank.GetPowerBankState() == PowerState.On)
                        {
                                Console.WriteLine(powerBank.BankNumber);

                                foreach (var outletCurrentSensor in
powerBank.OutletCurrentSensors)
```

```
                                {
                                        Console.WriteLine("Outlet number: {0}",
outletCurrentSensor.OutletNumber);
                                        Console.WriteLine("Current sensor reading:
{0} A", await outletCurrentSensor.ReadSensorAsync());
                                }
                        }
                }
        }
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Power Entry Panel (PEP)

The power entry panel is the main AC power input to the ATECCGEN2 rack. It consists of a three-phase AC power supply or a single-phase AC power supply depending on the rack variant. You can take sensor readings to monitor the status of AC power supply for a particular power phase.

Reading the Sensors on the Single-Phase Power Entry Panel

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);
        // Get the object of Power Entry Panel component
        IPowerPanelEntry powerPanelEntry = ateCoreSession.PowerEntryPanel;

        Console.WriteLine("Power panel entry name: {0}", powerPanelEntry.Name);
        Console.WriteLine("Power panel entry location: {0}",
```

```
powerPanelEntry.Location);
        Console.WriteLine("Power panel entry vendor name: {0}",
powerPanelEntry.VendorName);
        Console.WriteLine("Power panel entry model name: {0}",
powerPanelEntry.ModelName);

        foreach (var temperatureSensor in powerPanelEntry.TemperatureSensors)
        {
                Console.WriteLine("Temperature sensor name: {0}", await
temperatureSensor.Name);
                Console.WriteLine("Temperature sensor reading: {0} degree C", await
temperatureSensor.ReadSensorAsync());
        }

        if (powerPanelEntry.PowerSupplyPhaseType ==
PowerSupplyPhaseType.SinglePhase)
        {
                // Only one powerPhase object (phase A) in the PowerPhases array
                // in powerPanelEntry for single-phase
                IPowerPhase powerPhase = powerPanelEntry.PowerPhases[0];

                if (PowerState.On == await powerPhase.GetPowerPhaseStateAsync())
                {
                        int numberOfSensors = powerPhase.VoltageSensors.Length;
                        Console.WriteLine("Power phase type: {0}",
powerPhase.PhaseType.ToString());

                        // The number of sensors are same across all sensor type in
power phase object
                        for (int i=0; i<numberOfSensors; i++)
                        {
                                Console.WriteLine("Voltage sensor reading: {0} V",
await powerPhase.VoltageSensors[i].ReadSensorAsync());
                                Console.WriteLine("Current sensor reading: {0} A",
await powerPhase.CurrentSensors[i].ReadSensorAsync());
                                Console.WriteLine("Temperature sensor reading: {0}
degree C", await powerPhase.LineFrequencySensors[i].ReadSensorAsync());
                        }
                }
        }

}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
```

```
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Reading the Sensors on the Three-Phase Power Entry Panel

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);
        // Get the object of Power Entry Panel component
        IPowerPanelEntry powerPanelEntry = ateCoreSession.PowerEntryPanel;

        Console.WriteLine("Power panel entry name: {0}", powerPanelEntry.Name);
        Console.WriteLine("Power panel entry location: {0}",
powerPanelEntry.Location);
        Console.WriteLine("Power panel entry vendor name: {0}",
powerPanelEntry.VendorName);
        Console.WriteLine("Power panel entry model name: {0}",
powerPanelEntry.ModelName);

        foreach (var temperatureSensor in powerPanelEntry.TemperatureSensors)
        {
                Console.WriteLine("Temperature sensor name: {0}", await
temperatureSensor.Name);
                Console.WriteLine("Temperature sensor reading: {0} degree C", await
temperatureSensor.ReadSensorAsync());
        }

        if (powerPanelEntry.PowerSupplyPhaseType ==
PowerSupplyPhaseType.SinglePhase)
        {
                // Only one powerPhase object (phase A) in the PowerPhases array
                // in powerPanelEntry for single-phase
                IPowerPhase powerPhase = powerPanelEntry.PowerPhases[0];

                if (PowerState.On == await powerPhase.GetPowerPhaseStateAsync())
                {
```

```
                        int numberOfSensors = powerPhase.VoltageSensors.Length;
                        Console.WriteLine("Power phase type: {0}",
powerPhase.PhaseType.ToString());

                        // The number of sensors are same across all sensor type in
power phase object
                        for (int i=0; i<numberOfSensors; i++)
                        {
                                Console.WriteLine("Voltage sensor reading: {0} V",
await powerPhase.VoltageSensors[i].ReadSensorAsync());
                                Console.WriteLine("Current sensor reading: {0} A",
await powerPhase.CurrentSensors[i].ReadSensorAsync());
                                Console.WriteLine("Temperature sensor reading: {0}
degree C", await powerPhase.LineFrequencySensors[i].ReadSensorAsync());
                        }
                }
        }

}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Rack Control Unit

The rack control unit (RCU) communicates with components and sensors using internal electrical buses and communicates with hosts using a network connection. The RCU has various onboard sensors including current sensors, voltage sensors, and temperature sensors.

Reading the Properties and Sensors from the Rack Control Unit

```
IAteCoreSession ateCoreSession;
try
```

```
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);
        // Get the object of RCU component
        IRackControlUnit rackControlUnit = ateCoreSession.RackControlUnit;
        Console.WriteLine("Rack control unit name: {0}", rackControlUnit.Name);
        Console.WriteLine("Rack control unit location: {0}",
rackControlUnit.Location);
        Console.WriteLine("Rack control unit vendor name: {0}",
rackControlUnit.VendorName);
        Console.WriteLine("Rack control unit model name: {0}",
rackControlUnit.ModelName);
        Console.WriteLine("Rack control unit serial number: {0}",
rackControlUnit.SerialNumber);
        Console.WriteLine("Rack control unit firmware version: {0}",
rackControlUnit.FirmwareVersion);

        foreach (var currentSensor in rackControlUnit.CurrentSensors)
        {
                Console.WriteLine("Current sensor name: {0}", currentSensor.Name);
                Console.WriteLine("Current sensor reading: {0} A", await
currentSensor.ReadSensorAsync());
        }

        foreach (var voltageSensor in rackControlUnit.VoltageSensors)
        {
                Console.WriteLine("Voltage sensor name: {0}", voltageSensor.Name);
                Console.WriteLine("Voltage sensor reading: {0} V", await
voltageSensor.ReadSensorAsync());
        }

        foreach (var temperatureSensor in rackControlUnit.TemperatureSensors)
        {
                Console.WriteLine("Temperature sensor name: {0}",
temperatureSensor.Name);
                Console.WriteLine("Temperature sensor reading: {0} degree C", await
temperatureSensor.ReadSensorAsync());
        }
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
```

```
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Fan Domain and Fan Panel

A fan domain is an area where fan panels are installed. Each fan domain contains a list of fan panels. Each fan panel contains a list of fans and temperature sensors. To control the fan speed, call `SetFanDomainModeAsync` and pass the expected mode from `FanDomainMode` as an input parameter to change the fan speed.

- `FanDomainMode.High`: the fan speed is set to `FanDomainMaxRpm` for the fans in the fan panels of the target fan domain.
- `FanDomainMode.Manual`: the fan speed is set to the user-defined fan RPM. To set a user-defined fan speed, call `SetFanDomainUserDefinedRpmAsync`.

> **Note** If the fan domain mode is set to `FanDomainMode.Manual` when a user-defined RPM is not set, the fan RPM will be set to `FanDomainMinRpm`. The user-defined RPM cannot be set higher than `FanDomainMaxRpm` or lower than `FanDomainMinRpm`.

> **Note** When the RCU is in standby mode, querying the fan RPM using `ReadRpmAsync` may return an incorrect value of `4294967295`. This value is generated by the default hardware behavior and can be disregarded.

Reading Temperature Sensors and Fan Speeds from All Fan Domains

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);
        // Get the objects of Fan Domain.
```

```
        IFanDomain[] fanDomains = ateCoreSession.FanDomains;

        foreach (var domain in fanDomains)
        {
                Console.WriteLine("Fan domain type: {0}",
domain.FanDomainType.ToString());

                foreach(var fanPanel in domain.FanPanels)
                {
                        Console.WriteLine("Fan panel name: {0}", fanPanel.Name);
                        Console.WriteLine("Fan panel location: {0}",
fanPanel.Location);
                        Console.WriteLine("Fan panel vendor name: {0}",
fanPanel.VendorName);
                        Console.WriteLine("Fan panel model name: {0}",
fanPanel.ModelName);
                        Console.WriteLine("Fan panel serial number: {0}",
fanPanel.SerialNumber);

                        foreach(var temperatureSensor in
fanPanel.TemperatureSensors)
                        {
                                Console.WriteLine("Temperature sensor name: {0}",
temperatureSensor.Name);
                                Console.WriteLine("Temperature sensor reading: {0}
degree C", await temperatureSensor.ReadSensorAsync());
                        }

                        foreach(var fan in fanPanel.Fans)
                        {
                                Console.WriteLine("Fan name: {0}", fan.Name);
                                Console.WriteLine("Fan speed reading: {0} RPM",
await fan.ReadRpmAsync());
                        }
                }
        }
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
```

```
                ateCoreSession.Dispose();
        }
}
```

Uninterruptable Power Supply

The uninterruptable power supply (UPS) provides backup power to the rack in the event of power outages or fluctuations.

Once you install a UPS in your ATE Core Configurations Generation 2 system, use the `RegisterUninterruptablePowerSupplyAsync` function to register the UPS to the system. After registering the UPS, use the `SetMinimumInputVoltageAsync` and `SetMaximumInputVoltageAsync` functions to set the rated minimum and maximum input voltages for the UPS.

You can use the `UnregisterUninterruptiblePowerSupplyAsync` function to unregister an installed UPS before uninstalling it.

Note After you register a UPS, UPS-related monitoring features do not start until the rack is rebooted.

Note You must unregister an installed UPS using `UnregisterUninterruptiblePowerSupplyAsync` before powering down the UPS and physically removing it from the rack.

Registering a UPS and Setting the Maximum and Minimum Input Voltages

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);

        // Register a UPS installed on Phase A, outlet 1 to ATECCGEN2 rack
        await
ateCoreSession.RegisterUninterruptiblePowerSupplyAsync(PowerPhaseType.PhaseA, 1);
        // Set the maximum and minimum input voltages of UPS
        foreach (var uninterruptiblePowerSupply in
```

```
ateCoreSession.UninterruptiblePowerSupplies)
        {
                if (uninterruptiblePowerSupply.PoweredPhase ==
PowerPhaseType.PhaseA)
                {
                        await
uninterruptiblePowerSupply.SetMaximumInputVoltageAsync(250);
                        await
uninterruptiblePowerSupply.SetMinimumInputVoltageAsync(210);
                }
        }
        Console.WriteLine("Successfully registered UPS to system.");
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Unregistering a UPS

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);

        // Unregister a UPS installed on Phase A, outlet 1 from ATECCGEN2 rack
        await
ateCoreSession.UnregisterUninterruptiblePowerSupplyAsync(PowerPhaseType.PhaseA, 1);
        Console.WriteLine("Successfully unregistered UPS from system.");
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
```

```
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Checking if the ATECC Gen 2 System is Running on UPS Power

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);

        // Check if ATECCGEN2 is on UPS power or facility power
        bool isOnUpsPower = await ateCoreSession.IsRackOnUpsPowerAsync();
        Console.WriteLine("The rack is powered by UPS: {0}", isOnUpsPower);
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Rack Temperature Sensor Unit

Rack temperature sensor units are installed at different locations to monitor temperatures throughout the ATECCGEN2 rack. Rack temperature sensor units are accessible via the `IRackTemperatureSensorUnit` interface.

Getting Temperature Sensor Readings from All Rack Temperature Sensor Units

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);
        // Get the objects of rack temperature sensor component
        IRackTemperatureSensorUnit[] rackTemperatureSensorUnits =
ateCoreSession.RackTemperatureSensorUnits;

        foreach (var rackTemperatureSensorUnit in rackTemperatureSensorUnits)
        {
                Console.WriteLine("Rack temperature sensor unit name: {0}",
rackTemperatureSensorUnit.Name);
                Console.WriteLine("Rack temperature sensor unit location: {0}",
rackTemperatureSensorUnit.Location);
                Console.WriteLine("Rack temperature sensor unit vendor name: {0}",
rackTemperatureSensorUnit.VendorName);
                Console.WriteLine("Rack temperature sensor unit model name: {0}",
rackTemperatureSensorUnit.ModelName);

                var tempSensor = rackTemperatureSensorUnit.TemperatureSensor;
                Console.WriteLine("Temperature sensor reading: {0} degree C", await
tempSensor.ReadSensorAsync());
        }
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Getting Safety Interlock Status

Call `IsInterlockClosedAsync` to read the signal state for safety interlocks monitored by the RCU. Call `IsInterlockDipSwitchClosedAsync` to read the DIP switch state for the safety interlocks monitored by the RCU.

Get the Interlock Connection Status

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);

        // Loop through the enum values of Interlock
        foreach (Interlock interlock in Enum.GetValues(typeof(Interlock)))
        {
                bool isClosed = await
ateCoreSession.IsInterlockClosedAsync(interlock);
                Console.WriteLine("The {0} is closed: {1}", interlock, isClosed);
        }
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Get the Interlock DIP Switch States

```
IAteCoreSession ateCoreSession;
try
{
```

```
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);

        // Loop through the enum values of InterlockDipSwitch
        foreach (InterlockDipSwitch dipSwitch in
Enum.GetValues(typeof(InterlockDipSwitch)))
        {
                bool isClosed = await
ateCoreSession.IsInterlockDipSwitchClosedAsync(dipSwitch);
                Console.WriteLine("The {0} is closed: {1}", dipSwitch, isClosed);
        }
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Related information:

- RMX-10101/10102 Safety Interlocks in the ATE Core Configurations Generation 2 User Manual

Get or Set System State or Power State (Run Remotely)

The `GetRackSystemStateAsync` function returns the current rack system state as a `RackSystemState` enum. The `GetRackPowerStateAsync` function returns the rack power state as a `RackPowerState` enum. You can can use this function, for example, to remotely change the ATECCGEN2 rack power state from `Standby` to `Running`.

When the rack is turned on and initialized, the RCU service cycles through the system states of `Inactive`, `Init1` and `Init2` before reaching the `Operational` system state. During rack initialization, the rack power state is set to `RackControlUnitBooting`. When the rack system state reaches the `Operational` state, the rack power state is set to `Standby`.

In the `Standby` power state, the rack does not supply power to the payloads. Pressing the power button on the rack or programmatically calling the `SetRackPowerStateToRunningAsync` function from a host changes the rack power state to `Running` and the system supplies power to the payloads.

To stop the system from supplying power to any payloads and return to the `Standby` power state from the `Running` power state, press the power button or programmatically call the `SetRackPowerStateToStandbyAsync` function.

Performing a Full Restart and Getting the System State

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);

        // Perform a full restart
```

```
        await
ateCoreSession.InitiateRackRestartAsync(RackRestartLevel.FullRestart); // Dispose
ateCoreSession will be handled in InitiateRackRestartAsync
        Stopwatch stopWatch = new Stopwatch();
        stopWatch.Start();
        while (true)
        {
                try
                {
                        ateCoreSession = await
AteCoreSession.CreateAteCoreSessionAsync(hostname, password);
                        break;
                }
                catch
                {
                        // Give some time (30 seconds) for a full restart
                        if (stopWatch.ElapsedMilliseconds > 30000)
                        {
                                throw new TimeoutException();
                        }
                }
        }
        stopWatch.Stop();
        stopWatch.Restart();
        while (await ateCoreSession.GetRackSystemStateAsync() !=
RackSystemState.Operational)
        {
                // throw if the ATECCGEN2 hasn't moved to Operational system state
over 5 seconds
                if (stopWatch.ElapsedMilliseconds > 5000)
                {
                        throw new TimeoutException();
                }
        }
        stopWatch.Stop();
        // Proceed with other operations after ATECCGEN2 is in an operational
system state.
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
```

```
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Get and Set the Rack Power State

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);

        // Set the ATECCGEN2 Power State to Running when it is in Standby
        if (await ateCoreSession.GetRackPowerStateAsync() ==
RackPowerState.Standby)
        {
                await ateCoreSession.SetRackPowerStateToRunningAsync();
                // Call other functions to perform operations that require Power
State Running on ATECCGEN2.
        }
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Related reference:

- Rack System State and Rack Power State Descriptions

Triggering a Full Restart or Service-Level Restart

Call `InitiateRackRestartAsync` with the input parameter `RackRestartLevel.FullRestart` to trigger a full restart, or `RackRestartLevel.ServiceLevelRestart` to trigger a service-level restart. The `InitiateRackRestartAsync` function internally calls `IAteCoreSession.Dispose` so you do not need to explicitly call the `Dispose` function.

Restarting the RCU Service

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);

        // Perform a full restart
        await ateCoreSession.InitiateRackRestartAsync(RackRestartLevel.
ServiceLevelRestart); // Dispose ateCoreSession will be handled in
InitiateRackRestartAsync
        Stopwatch stopWatch = new Stopwatch();
        stopWatch.Start();
        while (true)
        {
                try
                {
                        ateCoreSession = await
AteCoreSession.CreateAteCoreSessionAsync(hostname, password);
                        break;
                }
                catch
                {
                        // Timeout after 20 seconds.
                        if (stopWatch.ElapsedMilliseconds > 20* 1000)
                        {
                                throw new TimeoutException();
                        }
```

```
                }
        }
        stopWatch.Stop();
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Related reference:

- Full Restart and RCU Service-Level Restart Descriptions

Control the Rack Tower Light

An optional tower light kit is available for the ATECCGEN2 rack. The tower light can be controlled using the `SetTowerLightAsync()` method. Each tower light segment can be independently controlled using the TowerLight enum. `TowerLightRed`, `TowerLightAmber`, `TowerLightGreen`, `TowerLightBlue`, and `TowerLightWhite` control the LED segments and `TowerLightBuzz` controls the tower light buzzer. `TowerLightAll` provides the option to switch off all segments.

Controlling Tower Lights

```
IAteCoreSession ateCoreSession;
try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);

        // Reset Tower Light before turn on any Tower Light
        await ateCoreSession.SetTowerLightAsync(TowerLight.TowerLightAll, false);

        // Examples of setting Tower Light based on conditions
        if (failure conditions such as an emergency stop or machine fault)
        {
                // Turned On Tower Light Red
                await ateCoreSession.SetTowerLightAsync(TowerLight.TowerLightRed,
true);
        }
        else if (warnings such as over-temperature or over-pressure conditions)
        {
                // Turned On Tower Light Amber
                await ateCoreSession.SetTowerLightAsync(TowerLight.TowerLightAmber,
true);
        }
        else if (normal machine or process operation)
        {
                // Turned On Tower Light Green
                await ateCoreSession.SetTowerLightAsync(TowerLight.TowerLightGreen,
true);
        }
        else if (external help request, scheduling or maintenance personnel
```

```
assistance)
        {
                // Turned On Tower Light Blue
                await ateCoreSession.SetTowerLightAsync(TowerLight.TowerLightBlue,
true);
        }
        else if (user-defined conditions, machine is running on a custom user-
defined command set)
        {
                // Turned On Tower Light White
                await ateCoreSession.SetTowerLightAsync(TowerLight.TowerLightWhite,
true);
        }
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Updating Firmware

To apply a firmware update to the rack control unit, call the `UpdateFirmwareAsync` function and pass in the full firmware file path as an input parameter.

Note Download the firmware update file from `ni.com/downloads`.

If the `UpdateFirmwareAsync` function succeeds, perform a full restart by calling the `InitiateRackRestartAsync` function with `RackRestartLevel.FullRestart` as the input parameter.

After the full restart, call `GetFirmwareUpdateStatusAsync` to check the firmware update status. If the return value evaluates to `FirmwareUpdateStatus.Succeeded`, the firmware update succeeded and the new firmware is running. If the returned value evaluates to `FirmwareUpdateStatus.Failed`, the firmware update failed and the system has rolled back to the previous firmware.

Note The firmware file must have a `.bin` file extension to prevent `UpdateFirmwareAsync` from throwing an exception.

Updating Rack Firmware

```
IAteCoreSession ateCoreSession;
// New firmware file to update to
string firmwareFilePath = " C:\\new_rack_firmware_version_1.bin ";

try
{
        // To create an object of IAteCoreSession.
        ateCoreSession = await AteCoreSession.CreateAteCoreSessionAsync(hostname,
password);
        await ateCoreSession.UpdateFirmwareAsync(firmwareFilePath);

        // Perform a full restart
        await
```

```
ateCoreSession.InitiateRackRestartAsync(RackRestartLevel.FullRestart); // Dispose
ateCoreSession will be handled in InitiateRackRestartAsync
        Stopwatch stopWatch = new Stopwatch();
        stopWatch.Start();
        while (true)
        {
                try
                {
                        ateCoreSession = await
AteCoreSession.CreateAteCoreSessionAsync(hostname, password);
                        break;
                }
                catch
                {
                        // For full restart after firmware update, it takes longer
time to restart, timeout after 2 mins.
                        if (stopWatch.Elapsed.TotalSeconds > 2 * 60)
                        {
                                throw new TimeoutException();
                        }
                }
        }
        stopWatch.Stop();
        stopWatch.Restart();
        while (await ateCoreSession.GetRackSystemStateAsync() !=
RackSystemState.Operational)
        {
                // throw if the ATECCGEN2 hasn't moved to Operational system state
over 5 seconds
                if (stopWatch.ElapsedMilliseconds > 5000)
                {
                        throw new TimeoutException();
                }
        }
        stopWatch.Stop();

        FirmwareUpdateStatus firmwareUpdateStatus = await
ateCoreSession.GetFirmwareUpdateStatusAsync();
        if (FirmwareUpdateStatus.Succeeded != firmwareUpdateStatus)
        {
                throw new Exception("Firmware update failed.");
        }

        // Get the object of RCU component
        IRackControlUnit rackControlUnit = ateCoreSession.RackControlUnit;
        // Print new firmware version
```

```
        Console.WriteLine("Rack control unit firmware version: {0}",
rackControlUnit.FirmwareVersion);

}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (ateCoreSession != null)
        {
                // To close the session.
                ateCoreSession.Dispose();
        }
}
```

Related information:

- RMX-1010x Firmware Download

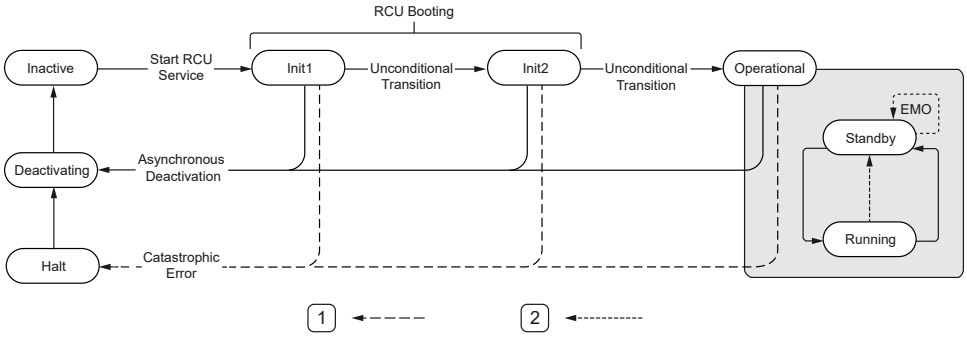
Rack System State and Rack Power State Descriptions

When a user powers on the ATECCGEN2 rack, the Rack Control Unit (RCU) loads firmware and runs the RCU service. The RCU service begins initializing components in the rack, enabling gRPC server communication, enabling RCU service status logging, and enabling system monitoring and control. The RCU service reports four system states during the initialization process: `Inactive`, `Init1`, `Init2`, and `Operational`. When the RCU service is terminated, it transitions from the `Operational` system state to the `Deactivating` system state and then the `Inactive` system state. In the event of an unrecoverable error in the system, the RCU service enters the `Halt` system state.

System and Power State Transitions

The following figure illustrates rack system state and simplified power states transitions.

Figure 1. System and Power States



1. Exception-triggered transition
2. Emergency off (EMO) transition

- **Inactive**—The RCU service is not running. The gRPC server communication and RCU service status logging are not available. The system transitions to the `Init1` state after the RCU service starts.
- **Init1**—The RCU service starts and initializes basic devices on the system, including the status LED. The gRPC server communication is started and RCU service status

logging is enabled. The system automatically transitions to the `Init2` state after the initialization is completed

- **Init2**—The RCU service completes the remaining devices initialization and starts the rack health monitoring. The system automatically transitions to `Operational` state after the initialization is completed.
- **Operational**—The system is fully operational. The power state is either `Running` or `Standby`.
  1. The rack power state starts with the `Standby` state where the rack does not supply power to the payloads.
  2. When the rack power state is in `Standby` state, pressing the power button on the rack or programmatically calling the `SetRackPowerStateToRunningAsync` function from a host changes the rack power state to `Running` and the system supplies power to payloads.
  3. When the rack power state is in `Running` state, pressing the power button or programmatically calling the `SetRackPowerStateToStandbyAsync` function from a host causes the system to turn off power supplies to the payloads and changes the rack power state to `Standby`.
  4. If a catastrophic event happens when the rack power state is in `Running` state, pressing the Emergency Off Button (EMO) immediately causes the system to turn off all power supplies to payloads and the system unconditionally transitions to the `Standby` power state. The system remains in `Standby` state until the RCU service is terminated or encounters a catastrophic error.
- **Deactivating**—When the RCU service is terminated asynchronously by calling the `InitiateRackRestartAsync` function from a host, the RCU service enters the `Deactivating` state before going into the `Inactive` state. All running tasks are terminated and resources are freed before the RCU service exits.
- **Halt**—the RCU service enters the `Halt` state in the event of an unrecoverable system error. The `Halt` state persists indefinitely until the system is deactivated asynchronously.

  In the `Halt` state the system terminates most of the services except for the gRPC server to allow host communication and RCU service status logging. The RCU service may also transition to this state in the event of critical errors in the `Init1`, `Init2` or `Operational` system states.

Related reference:

- Get or Set System State or Power State (Run Remotely)

Related information:

- Power Entry Panel (in ATE Core Configurations Generation 2 User Manual)

Full Restart and RCU Service-Level Restart Descriptions

Call `InitiateRackRestartAsync` to reboot the system or restart the RCU service as necessary (for example, to complete a firmware update).

Service-Level Restart Actions

During a service-level restart, only the RCU service is restarted. The restart sequence and effects are as follows:

1. If the rack is in the `Running` power state, the rack power state transitions from `Running` to `Standby`.
2. The RCU service transitions to the `Deactivating` rack system state.
3. The RCU service transitions to the `Inactive` state, then proceeds through the `Init1` and `Init2` states to the `Operational` state.
4. All errors or warnings detected by the RCU service are cleared.
5. All RCU service resources are released during termination and re-initialized when the service runs.
6. The gRPC server is terminated and restarted.
7. The RCU service creates a new status log file.
8. The RCU service reinitializes the rack.

Full Restart Actions

During a full restart, the embedded RCU operating system is rebooted. The restart sequence and effects are as follows:

1. The sequence and all effects of the service-level restart occur.
2. The RCU processor is reset.
3. All firmware in the rack is restarted.
4. The network connection is disconnected and reinitialized.

Related reference:

- Triggering a Full Restart or Service-Level Restart

Credential Management

The remote communication channel between a host and the RCU is encrypted using TLS. Each ATECCGEN2 rack has a default private key and a public certificate. To communicate with the rack the host must have the TLS public certificate, which can be obtained from the RCU via FTP service. The RCU is also password protected, meaning a password is required for communication via API. Each rack has a unique default password, which you should change for security purposes.

Default Host Name

An application uses NI-ATE Core to control a rack identified by the host name or IP address of the rack. The rack host name is `ni-rmx-1010x-<serial>` where `<serial>` is the serial number of the RCU. If the serial number has less than eight characters, `0` characters are inserted before the serial number to produce the minimum length of eight characters.

The example 7-digit serial number `217D902` produces a host name of `ni-rmx-1010x-0217d902`

Default Password

The default password is unique and generated based on the serial number of the RCU. The default password consists of eight hexadecimal characters (`0` to `9`, `a` to `f`, lower case). If the serial number has less than eight characters, `0` characters are inserted before the serial number to produce the minimum length of eight characters.

The example 7-digit serial number `217D902` produces a default password of `0217d902`.

Changing the Password

Call `ChangeRackPasswordAsync` to change the password. You must provide the existing password and the new password. The new password takes effect once the existing password is verified.

Remembering Passwords

When connecting to a target rack for the first time using the `CreateAteCoreSessionAsync` function, you can instruct the NI-ATE Core driver to save the password you use to connect by setting the `rememberPassword` parameter to `true`. After you set `rememberPassword` to `true`, NI-ATE Core uses the previously saved password when connecting to the target if you pass an empty string as the password parameter on subsequent connections to the target.

Note The saved password is stored securely on the host machine and is not accessible by other users of the same machine or other host machines.

You can also use the `PasswordManager` class function `SetSavedPasswordAsync` to save a password for a specified target or update the target password if you already saved one.

Setting or Changing a Saved Password

```
PasswordManager passwordManager = new PasswordManager();
try
{
        // Change saved password
        await passwordManager.SetSavedPasswordAsync(hostname, password);
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (passwordManager != null)
        {
                // To close the session.
                passwordManager.Dispose();
        }
}
```

Deleting Saved Passwords

You can delete a password saved to a target rack by calling the `PasswordManager`

class function `DeleteSavedPasswordAsync`. You must provide a password when connecting to the target rack using `CreateAteCoreSessionAsync` after deleting the saved rack password.

Deleting a Saved Password

```
PasswordManager passwordManager = new PasswordManager();
try
{
        // Delete saved password
        await passwordManager.DeleteSavedPasswordAsync(hostname);
}
catch(Exception ex)
{
        Console.WriteLine(ex.Message); //Print the exception message.
}
finally
{
        if (passwordManager != null)
        {
                // To close the session.
                passwordManager.Dispose();
        }
}
```

Generating TLS Private Key and Public Certificate Pairs

Call `GenerateAndApplyCertificateKeyPairAsync` to generate new key pairs. The RCU generates and applies a new private key and public certificate pair and returns the new public certificate to the host. You must reboot the rack and create a new session for the new key pairs to take effect.

Manually Retrieving the Effective Certificate

In the event of a key pair mismatch you can copy the effective certificate to replace the public certificate on the host. You must also complete this process when you have generated a key pair and you want to control the rack from a new host, or you want to reinstall NI-ATE Core driver software.

1. Retrieve the effective certificate from the RCU via FTP at the following path:

`/upload/tls_cert.pem`.
2. Rename the certificate to `ni_ate_core_cert.pem`.
3. Move the certificate in the following folder: `C:\ProgramData\National Instruments\NI-ATE Core\`.
4. Delete `ni_ate_core_cert.pem.new` from the same location, if it exists.

You can now use the API to generate a new key pair.

Related reference:

- Creating and Disposing of Sessions

Log File Location and Behavior

Log files can be accessed via FTP at the following path: `/upload/log`. Log files are stored in the rack system storage (MicroSD card), which is separate from RCU OS storage.

Log File and Folder Size Limitations

The maximum log file size is 5 MB. When a log file size reaches the maximum file size, a new log file is created and logging continues in the new log file. The maximum size of the log folder is 2 GB. When the log folder reaches the maximum folder size, the oldest log file is replaced.