Application-relevant changes in PLCnext Technology firmware releases 2025 and newer

Application notes

PHOENIX CONTACT

Table of Contents

Revision history for this PDF file

Date	Revision	Changes
2026-03-16	111781_en_04	Addition to the Migration guidance: After updating, a reset type 1 is recommended! Addition to the Introduction: Firmware recommendation for AXC F 2152 and AXC F 1152. Changes with 2026.0 LTS added: Constructor added in chapter 2.3.1.2 The library definition.
2025-07-22	111781_en_03	Addition to RSC changes in chapter 2.4.3.
2025-06-06	111781_en_02	Addition to RSC changes in chapter 2.4.3.
2025-05-09	111781_en_01	Migration guidance added; changes and additions in chapters 2.3.3 up to 3.1 incl. subchapters; improved description of signed applications in chapter 4.5.
2024-09-18	111781_en_00	Published with *Preview* 2024.7

1 Introduction

This documentation specifies changes of the PLCnext Control firmware and SDK regarding the release 2025.0 that may be relevant to user-generated applications.

The major change is the non-compliance of the binary compatibility to former SDK s. A recompile of user code that is based on any pre-2025 SDK is mandatory. Furthermore, some code adjustments might become necessary, which are mostly indicated through *deprecated* warnings. This documentation has to be used together with the firmware's Change Notes of the respective PLC type.

Recommendation for using released PLCnext Control firmware versions for the AXC F 2152 and AXC F 1152

As part of our continuous quality assurance and product maintenance, we would like to inform you that we currently recommend using the **2024.0.16 LTS** firmware version for the PLCnext Control AXC F 2152 and AXC F 1152. This version has been successfully in use in the field for an extended period and fully complies with our current release and validation guidelines.

Once the upcoming long-term supported firmware version **2026.0 LTS** is officially released, we recommend switching to this version as part of your regular maintenance or update cycles. This ensures that you continuously benefit from the latest advancements and improvements of our platform.

Please note that this recommendation is part of our standard lifecycle and product strategy and does not constitute an assessment of specific applications or operational conditions. For project-specific requirements or individual evaluations, we will gladly support you. Thank you for your cooperation and your continued trust in our products.

Firmware 2025.0 comes with some fundamentally updated parts of the underlying Linux® operating system and adapts the PLCnext Runtime System to those changes. This is intended to improve the long-term compatibility of binary files from C++ or MATLAB®/Simulink® programs, as well as components and apps.

The updates to the Linux® system will facilitate future updates to the kernel, the installation of new technologies and, above all, the security hardening of the system to be prepared for the Cyber Resilience Act in the European Union.

In parallel, the current 2024.0 LTS based on the familiar foundation of PLCnext Technology will get security-related fixes for an extended period (details see Schedule).

1.1 General advantages

The 2025.0 release will implement the following optimizations and lays the foundations for later implementation:

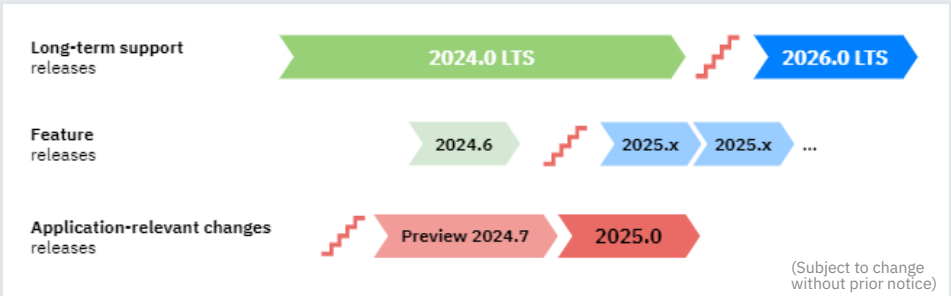
- Increased stability due to adjustments of internal system functions
- Implementing long-term binary-compatible interfaces so we can continuously develop new RSC services
- Simplified implementation of user applications by means of standardized interfaces
- Several Linux® adaptations:
  - Increased security by splitting into separate processes and better use of Linux® capabilities
  - Removal of Linux® packages with high potential for security vulnerabilities
  - Switch to Linux® `systemd` for a better adaptation to new technology
  - Faster boot process due to parallel start of Linux® services
- New system monitoring for CPU, RAM, flash memory, and processes
- Implementation of interfaces to add WBM configuration pages for user components
- Support for versioning of user components
- Improved diagnostics by splitting into different log files
- Deep and secure integration of OCI containers for firmware functions and apps
- Newly designed Web-based Management:
  - single sign-on with OAuth2 allows for easy but secure access
  - modern layout based on Angular.io that adapts to display sizes ("responsive")

1.2 Changes regarding application development

The substantial changes from developers' view are:

- The C++ standard is elevated to the more modern C++ 20 (formerly C++ 17) and standardized data types
- Library `Arp.System.Core` is substituted by the new library `Arp.Base.Core`
- The public classes of `Arp.System.Acf` were moved to the new library `Arp.Base.Acf.Commons`
- Refactoring of `Logging` substruction
- Removed `RTLD_GLOBAL` flag while loading shared libraries
- Update of several external libraries, e.g.:
  - `CppFormat` upgrade to libfmt 10.2.1
  - `boost` update to version 1.84 (but see boost version for details)

1.3 Schedule



The 2024.0 LTS firmware release stays available for all PLCnext Control device types, and will be maintained for an extended period of 2 years without integration of any of the changes relevant to developing applications. This way, customers can benefit from security fixes and patches without having to adapt the productive applications to the upcoming changes for another 1 ½ years. But there won't be an LTS successor on the that code base.

Another feature release was derived from the well-known code base.

A public *Preview* 2024.7 contained most of the 2025.0 changes. Developers were invited to inspect that code base and begin testing their applications in that environment. This *Preview* 2024.7 obsolete now and needs to be deinstalled.

The 2025.0 firmware release for all PLCnext Control hardware is not considered to be an LTS version. Of course, security updates will be provided.

New features are built upon the 2025.0 code base.

The 2026.0 LTS and all further firmware releases will be based on the improved foundation of PLCnext Technology.

1.4 Migration guidance

Note: Updating to firmware versions 2025.0 and newer from older firmware versions may have unexpected effects on the existing application or specific user settings.
It is strongly recommended to observe the following points before rolling out the firmware into productive operation:

– Intensive study of change notes regarding firmware release 2025.0 or newer to your hardware, and of the documentation in this PDF file.
– Thorough backup the current status; a downgrade is possible, but an automatic restoration of all original configurations cannot be guaranteed.
– Validation of the existing applications with regard to functionality and compatibility after the firmware update.

For updating, follow these steps:

- In case you tried the *Preview* 2024.7, uninstall it.
- Back up the projects and configuration data.
- Reset your controller to default settings type 1 (see the respective hardware user manual for the proper procedure).
- Check the current firmware version:

Coming from devices running on 2019.0 LTS up to 2021.9 firmware, first update to the 2022.0 LTS release, then proceed.
Coming from 2022.0 LTS or newer, update your controller to 2025.0 (e. g. via Web-based Management).
↪ On rebooting after a successful update, a script converts the network configuration.

- **Important:** Reset your device (type 1) after updating.
- Enter the Web-based Management and set up your configurations (IP address, Date and Time, active/inactive system services).
- Review your applications, adapt and recompile C++ programs.
  Note: RAM usage and CPU load may be higher than before, depending on active services.
- If you're using PLCnext Technology Apps, check the PLCnext Store for versions compatible with firmware 2025.0 or newer. Due the changes regarding C++ code, many of the PLCnext Technology apps need to be adapted and recompiled, too.
- Download your applications to your controller.
- Validate the applications' functionality.

2 C++ standard, libraries and classes

2.1 Update to C++ 20

With firmware release 2025.0 and higher, the C++ language standard used in the SDK is updated to C++ 20 to enable the usage of new features provided by the language. Since features of C++ 20 are used in the SDK's header files, all compilation units including headers from the PLCnext Technology SDK must be recompiled using at least C++ 20.

The new features of C++ 20 which are now available with PLCnext Technology can be found in the cppreference.com for C++ 20.

The main improvements regarding `Arp` code are described in the following subsections.

Simplified namespace syntax

The syntax of namespace declaration was simplified as shown in the following example.

Before C++ 20:

```
namespace Arp { namespace System { namespace Commons
{

// code here

}}} // end of namespace Arp::System::Commons
```

From C++ 20:

```
namespace Arp::System::Commons
{

// code here

} // end of namespace Arp::System::Commons
```

The new namespace syntax should be preferred in future development.

2.2 Library Arp.Base.Core

(formerly Arp.System.Core)

The entire code of the `Arp.System.Core` library is re-implemented in the `Arp.Base.Core` library. The purpose of this change is to provide a source-compatible migration path to ensure binary compatibility in future releases. `Arp.Base.Core` implements some techniques to further separate the interface from the implementation to enable new features and easier fixes. Some inconsistencies in naming and behavior have also been fixed.

Even though the location of the library in the SDK has changed, it does not cause any adjustments of the include directives nor any type names:

– All include directives of `Arp.System.Core` files are delegated to the corresponding header file in `Arp.Base.Core`

– All types of the `Arp.Base.Core` library are defined in namespace `Arp::Base::Core`, but are imported into the `Arp` namespace, as was in the `Arp.System.Core` library.

All (non-template) classes in the `Arp.Base.Core` library implement the `PImpl` pattern now. This is to avoid any binary incompatibilities in future when extensions or modifications become necessary.

The `Arp.System.Core` library was built as archive and linked statically, which causes some trouble with the `AppDomain` singleton pattern, so that the `AppDomain` instance had to be injected into any shared library. This issue is now fixed, since the `Arp.Base.Core` library is built as a shared library and linked dynamically. Therefore, dependencies do not propagate to the client's link process anymore.

There are only a few minor changes regarding the usage of `Arp.Base.Core`, which are mostly indicated through *deprecated* warnings during compilation, while the warning message gives a hint how to fix it. All deprecated operations will be removed in future (but not yet in firmware release 2025.0).

2.2.1 Mapping Arp::byte to std::byte

C++ 17 introduced a new datatype called `std::byte`. This type might be seen as a real binary type of size 'one byte', which was lacking in C++ for the time being.

The characteristics of this datatype are:

– It's not an integral datatype.
– It's not an arithmetic datatype.
– It only supports logical bit operations.
– Initialization with integral datatypes is not possible; therefore, in namespace `Arp` two string literal operators were defined: `_b` and `_B` (see examples in the code block below).

PLCnext Technology firmware release 2021.6 switched to the C++ 17 standard already. At that time, the C++ compiler may have regarded the datatype `byte` as ambiguous, so that our Change Notes stated:

"C++ 17 introduces the datatype `std::byte` which is unfortunately not compatible with `Arp::byte`. Thus, if the namespaces `std` and `Arp` are both active, the compilation results in an error. In this case existing C++ sources have to be adjusted, so that they explicitly use `Arp::byte` (e.g. by adding `using byte = Arp::byte;`)."

With the 2025.0 firmware, the `Arp::byte` datatype is switched from `std::uint8_t` to `std::byte`. Up to now it was just an alias of `Arp::uint8` which maps to `unsigned char`. Thus, if user code generates compile errors related to the new `Arp::byte` datatype, it is necessary to just switch to `Arp::uint8` to avoid them.

Another approach would be to adapt the code to the new datatype if it should represent a plain byte buffer. To get familiar with the new `byte` datatype, examine the following code examples, which demonstrate the usage of it:

String literal operator for byte datatype:

```
byte b = 0;                     // distinct type => compile error!
byte b1 = 0_b;
byte b2 = 123_B;
byte b3 = 0xff_B;
byte b4 = 0xFE_b;
byte b5 = 333_B;
```

Zero initialization:

```
byte b = static_cast<byte>(0);  // static_cast operator
byte b0 = (byte)0;              // explicit C cast
byte b1 = byte(0);              // explicit C++ cast
byte b2 = 0_b;                  // use of string literal operator _b
byte b3{ 0 };                   // {}-Initialization with implicit cast
byte b4{ 0_b };                 // {}-Initialization (initialize first field by 0,
                                // all others are default initialized)
byte b5{ };                     // Default {}-Initialization (value is zero'ed)
byte b6[6]{ };                  // Default array {}-Initialization (array is zero'ed)
byte b7[7]{ 0_b };              // Explicit array {}-Initialization with zero
byte b8[]{ 0_b, 0_b, 0_b };     // Initialization through initializer list (size==3)
byte b9[9] { 0 };               // distinct type => compile error!
```

Number initialization:

```
byte b = 123;                   // distinct type => compile error!
byte b = static_cast<byte>(1);  // static_cast operator
byte b0 = (byte)123;            // explicit C cast
byte b1 = byte(0x9A);           // explicit C++ cast
byte b2 = 111_b;                // use of string literal operator _b
byte b3{ 255 };                 // {}-Initialization with implicit cast
byte b4{ 0xA4_b };              // {}-Initialization
byte b5[5]{ 5_b };              // Partial array Initialization through
                                // initializer list => {5, 0, 0, 0, 0}
byte b6[6]{ 1_b, 2_b, 3_b };    // Partial array Initialization through
                                // initializer list => {1, 2, 3, 0, 0}
byte b7[]{ 1_b, 2_b, 3_b };     // Initialization through
                                // initializer list (size==3)
byte b8[8] { 8 };               // distinct type => compile error!
```

Byte format operations defined by Arp:

```
byte b0{};
byte b1 = 1_b;
byte b2 = 43_B;                 // ==0x2B
byte b3 = 0xff_B;

Assert::AreEqual("00", String::Format("{}", b0));
Assert::AreEqual("01", String::Format("{}", b1));
Assert::AreEqual("2b", String::Format("{}", b2));
Assert::AreEqual("ff", String::Format("{}", b3));
```

Bit operations:

```
byte b11111110 = 254_b;
byte b00001111 = 1_b | 2_b | 4_b | 8_b;         // bitwise | operator
byte b00001110 = b00001111 & b11111110;         // bitwise & operator
byte b01111111 = static_cast<byte>(~0x80_b);    // bitwise ~ operator

Assert::AreEqual(0x0F_b, b00001111, "Bitwise | operator.");
Assert::AreEqual(0x0E_b, b00001110, "Bitwise & operator.");
Assert::AreEqual(0x7F_b, b01111111, "Bitwise ~ operator.");
```

Shift operation:

```
constexpr byte b1 = 0x01_b;
uint16 u16 = uint16(b1 << 7);
static_assert(u16 == 128, "Shift operation of byte in range.");
```

If shift operation fails:

```
byte b1 = 0x01_b;
uint16 u16 = uint16(b1 << 8);    // fails, correct code would be:
uint16 u16 = uint16(b1) << 8     // ok!
Assert::AreEqual(u16, 256, "Shift operation of byte as uint16.");
// The failing shift operation would be okay
// for any 1 byte integral data type like uint8,
// but not for std::byte,
// because std::byte is not cast implicitly to int
```

Converting string from and to bytes:

```
String actual("abc");
std::vector<byte> bytes = actual.ToBytes();
String expected(bytes);

Assert::AreEqual(expected, actual);
```

2.2.2 class AppDomain

The following operations are deprecated:

– `AppDomain::GetCurrent()` → use `AppDomain::GetInstance()` instead.
– `AppDomain::Assign(AppDomain&)` → it's not required any more.

2.2.3 class AppDomainSingleton

The following operation is deprecated:

– `AppDomainSingleton::GetInstancePtr()` → use `AppDomainSingleton::IsCreated()` to check if it was created, and use `&AppDomainSingleton::GetInstance()` to get access to the instance pointer.

The following operation changes its behavior:

– `AppDomainSingleton::CreateInstance(...)` → throws an exception now if the singleton was already created.

The `AppDomainSingleton` class is deprecated.

It should not be used any more. The main goal of this class was to fix the issue, that the `Arp::Singleton<T>` base class has to be implemented as template, so that it only works within shared-library scope, but not within process-wide scope.

The `GlobalSingleton` pattern should be used instead of the `AppDomainSingleton` implementation. The following demo code might be seen as a template to be copied into custom projects, with replacing the class name by the desired custom class name.

Header file of the GlobalSingleton pattern:

```
#pragma once
#include "Arp/Base/Core/Arp.hpp"
#include <memory>

namespace Apps::Demo::Extension::Internal
{

class GlobalSingleton
{
public: // usings
    using Instance = GlobalSingleton;
    using InstancePtr = std::unique_ptr<Instance>;

public: // construction
    // If the constructor shall get parameters
    // then the CreateInstance(..) operation has to be adjusted accordingly
    GlobalSingleton(void) = default;

public: // copy/move/assign/destruction
    GlobalSingleton(const GlobalSingleton& arg) = delete;
    GlobalSingleton(GlobalSingleton&& arg) = delete;
    GlobalSingleton& operator=(const GlobalSingleton& arg) = delete;
    GlobalSingleton& operator=(GlobalSingleton&& arg) = delete;
    ~GlobalSingleton(void) = default;

public: // static singleton operations
    static Instance&     CreateInstance(void);
    static bool          IsCreated(void);
    static void          DisposeInstance(void);
    static Instance&     GetInstance(void);

private: // static fields
    static InstancePtr instancePtr;
};

//////////////////////////////////////////////////////////////////////////////
// inline methods of class GlobalSingleton

} // end of namespace Apps::Demo::Extension::Internal
```


...continuation see next page...

Source file of GlobalSingleton pattern:

```
#include "Apps/Demo/Extension/Internal/GlobalSingleton.hpp"
#include "Arp/Base/Core/TypeName.hxx"
#include "Arp/Base/Core/Exception.hpp"

namespace Apps::Demo::Extension::Internal
{

using namespace Arp;

// initializing of static fields
GlobalSingleton::InstancePtr GlobalSingleton::instancePtr;

GlobalSingleton::Instance& GlobalSingleton::CreateInstance()
{
  if (IsCreated())
  {
     throw Exception("Singleton instance of type '{}' was created yet!", TypeName<Instance>());
  }
  instancePtr = std::make_unique<Instance>();
  return *instancePtr;
}

bool GlobalSingleton::IsCreated()
{
  return instancePtr != nullptr;
}

GlobalSingleton::Instance& GlobalSingleton::GetInstance()
{
  if (!instancePtr)
  {
    throw Exception("Singleton instance of type '{}' was not created yet!", TypeName<Instance>());
  }
  return *instancePtr;
}

void GlobalSingleton::DisposeInstance()
{
  instancePtr.reset();
}

} // end of namespace Apps::Demo::Extension::Internal
```


If the constructor of the `GlobalSingleton` class requires some arguments then the `CreateInstance(..)` operation must be adjusted accordingly.

Note: The `GlobalSingleton` pattern does not work from within static libraries.

2.2.4 class Singleton<T>

The `Singleton<T>` class implemented some operations only intended to be used by the `AppDomain` and `AppDomainSingleton` classes. Since they got refactored, these functions are not needed any more.

The following operation is deprecated:

– `Singleton<T>::GetInstancePtr(void)` → use `Singleton<T>::IsCreated()` to check if it was created; use `&Singleton<T>::GetInstance()` to get access to the instance pointer.

The following protected operation is deprecated:

– `Singleton<T>::SetInstance(T* pInstance)` → not required anymore.

The following operation changes its behavior:

– `Singleton<T>::CreateInstance(...)` → throws exception now if singleton was already created.

2.2.5 class ArpVersion

The `ArpVersion` combined multiple functionalities in a single class: a version number and additional information like state and name. To simplify the usage of `ArpVersion`, this class was divided into two classes in the `Arp.Base.Core` library:

- `Version` → a simple version class consisting of four integer properties: `Major`, `Minor`, `Patch`, and `Build`
- `ArpVersion` → providing the build version, the build name and the build state

The following operations and properties of class `ArpVersion` are therefore deprecated:

- `ArpVersion::Current`
  → use `ArpVersion::GetCurrent()` instead
- `ArpVersion::GetMajor()`
  → use `ArpVersion::GetBuildVersion().GetMajor()` instead
- `ArpVersion::GetMinor()`
  → use `ArpVersion::GetBuildVersion().GetMinor()` instead
- `ArpVersion::GetPatch()`
  → use `ArpVersion::GetBuildVersion().GetPatch()` instead
- `ArpVersion::GetBuild()`
  → use `ArpVersion::GetBuildVersion().GetBuildNumber()` instead
- `ArpVersion::operator<`
  → use `ArpVersion::GetBuildVersion()` for comparison instead
- `ArpVersion::operator>`
  → use `ArpVersion::GetBuildVersion()` for comparison instead
- `ArpVersion::operator<=`
  → use `ArpVersion::GetBuildVersion()` for comparison instead
- `ArpVersion::operator>=`
  → use `ArpVersion::GetBuildVersion()` for comparison instead

The following constructors and operations of the `Version` class are deprecated and were only added for code compliance:

- `Version(Value major, Value minor, Value patch, Value build, const String& state, const String& name)`
  → use `ArpVersion` class instead
- `Version::GetName()`
  → use `ArpVersion` class instead
- `Version::GetState()`
  → use `ArpVersion` class instead

2.2.6 class Exception

The following protected operations are deprecated:

- `Exception::Exception(String&&, const Exception::Ptr&)`
  → use Exception(ExceptionTypeId, String&&, const Exception&)instead
- `void Exception::Format(int indentLevel, bool withInnerException)const`
  → use `void Exception::Format(bool withInnerException)const` instead
- `uint32 Exception::GetTypeCodeInternal(void)const`
  → not required anymore

– The return value of `Exception::GetInnerException()` changed from `Exception::Ptr` to `const Exception&`
→ not a deprecation warning but a compile error occurs when using this operation.

2.2.7 class TypeName<T> and CommonTypeName<T>

The classes `TypeName<T>` and `CommonTypeName<T>` provided a public member variable `Value`. This can cause issues regarding compatibility in the future. `TypeName<T>` contained a function to generate the `CommonTypeName<T>`. This violates the *Single Responsibility* principle.

2.2.7.1 class TypeName<T>

The following operations and properties are deprecated:

– `TypeName<T>::Value`
→ use operation `TypeName<T>::GetFullName(void)` instead
– `TypeName<T>::GetCommonName(void)`
→ use class `CommonTypeName<T>` instead

2.2.7.2 class CommonTypeName<T>

The following properties are deprecated:

– `CommonTypeName<T>::Value`
→ use operation `CommonTypeName<T>::GetFullName()` instead

2.2.8 class DateTime

The class `DateTime` provided public member variables. This can cause compatibility issues in the future.

Currently, `DateTime` only implements UTC-based timestamps. When support for local time is added in the future, `DateTime` objects with an unspecified time zone will cause problems. Therefore, these are not tolerated anymore.

The following properties are deprecated:

– `DateTime::MinTicks`
→ use `DateTime::GetMinTicks()` instead
– `DateTime::MaxTicks`
→ use `DateTime::GetMaxTicks()` instead

The following operation changes its behaviour:

– `DateTime` constructor throws an exception now if the `DateTimeKind` parameter is not set to `DateTimeKind::Utc`

2.2.9 class String

Some member functions of `String` were renamed for more clarity and for conformance to the C++ standard library.

The following properties are deprecated:

- `String::GetBaseString()`
  → use `String::GetStdString()` instead
- `String::StartWith(..)`
  → use `String::StartsWith(..)` instead (C++ 20 introduced `std::basic_string<…>::starts_with()`).
- `String::EndWith(..)`
  → use `String::EndsWith(..)` instead (C++ 20 introduced `std::basic_string<…>::ends_with()`).

Since C++ 20, the C++ standard defines the `std::span` type to determine the size of a static array automatically by the compiler. Thus, the explicit specified parameter `delimitersCount` is not required any more, and shall be omitted.

Therefore, the following operation change its signature and causes compile errors if not altered:

- `String::Split(const CharType delimiters[], size_t delimitersCount, bool trimTokens, bool removeEmptyTokens)`
  → the new signature is:
  `String::Split(std::span<const CharType> delimiters, bool trimTokens, bool removeEmptyTokens)const;`

The following operations change their behavior and would cause compile errors, which is caused by the upgrade of `libfmt` library:

- When using `String::Format(..)` the formatting of some primitive types changes:
  - `Arp::int8` is formatted as a number now (formerly as a character)
  - `Arp::uint8` is formatted as a number now (formerly as a character)
  - `Arp::byte` is formatted as a 2-digit hex number number now (formerly as a character)

- `String::Format(..)` does not accept following types any more:
  - raw enums
    → cast the values to `int` instead
  - enum classes which do not implement the `Arp` enum pattern
    → cast the values to `int` instead
  - Variables of type `std::atomic<T>`
    → format the raw value by calling `std::atomic<T>::load()` instead
  - `std::thread::id`
    → no suggestions so far
  - smart pointers like `std::shared_ptr<T>`
    → obtain the raw pointer by calling `std::shared_ptr<T>::get()` and cast it to `void*`

- The requirements on custom types to be formatted using `String::Format` have changed.
  - Instead of providing an `operator<<(std::ostream&, T)` function, a `fmt::formatter<T>` specialization has to be provided.
  - If the `ostream` operator `operator<<` is defined already, this is just a matter of adding an appropriate `using` declaration on global namespace level: `template<> struct fmt::formatter<T> : public fmt::ostream_formatter {};` where `T` is the fully qualified name of the custom type (see example below).

Template specialization of String formatter:

```
template<> struct fmt::formatter<Arp::Base::Core::String> : public fmt::ostream_formatter {};
```

2.3 Library Arp.Base.Acf.Commons

As mentioned above, the public code of the `Arp.System.Acf` library has moved to the new library `Arp.Base.Acf.Commons`. This was caused by the fact, that the `Acf` did not separate the interfaces from the implementation code, which violates the *Inversion of Dependencies* principle.

Even though the location of the library in the SDK has changed, it does not cause any adjustments of the `include` directives nor any type names:

– All include directives of `Arp.System.Acf` files are delegated to the corresponding header file in `Arp.Base.Acf.Commons`
– All types of the `Arp.Base.Acf.Commons` library are defined in namespace `Arp::Base::Acf::Commons`, but are imported also into the former namespace `Arp::System::Acf` to avoid code adaptation

But in contrast to the `Arp.Base.Core` library, the revised `Acf` code causes some major code changes:

– All user code based on the `Acf` classes `ComponentBase` and `LibraryBase` must be adjusted to fit to the new interfaces.

2.3.1 The library implementation of a custom project

2.3.1.1 The library declaration

The project's library class is derived by `LibraryBase` from `Acf` and implements a default constructor as well as an operation to obtain the singleton.

Header file:

```
#pragma once
#include "Arp/Base/Core/Arp.hpp"
#include "Arp/Base/Acf/Commons/LibraryBase.hpp"

namespace Apps::Demo::Sdk
{

using Arp::Base::Acf::Commons::ILibrary;
using Arp::Base::Acf::Commons::LibraryBase;

class SdkLibrary : public LibraryBase
{
public: // construction/destruction
    SdkLibrary(void);

public: // static singleton operations
    static ILibrary& GetInstance(void);
};

} // end of namespace Apps::Demo::Sdk
```

Changes:

- The former code also derives the library class by `Singleton<SdkLibrary>` → remove this
- The signature of the constructor was formerly `SdkLibrary(AppDomain& appDomain)` → remove the parameter

2.3.1.2 The library definition

The implementation of the project's library shall look like this:

```
#include "Apps/Demo/Sdk/SdkLibrary.hpp"
#include "Apps/Demo/Sdk/SdkComponent.hpp"
#include "Arp/Base/Core/TypeName.hxx"

namespace Apps::Demo::Sdk
{

SdkLibrary::SdkLibrary()
    : LibraryBase(/* User defined version: */ ArpVersion(1,2,3))
{
    this->AddComponentType<SdkComponent>();
    // Add more component types here if required
}

ILibrary& SdkLibrary::GetInstance()
{
    static SdkLibrary instance;
    return instance;
}

extern "C" ARP_EXPORT ILibrary& Apps_Demo_Sdk_MainEntry()
{
    return SdkLibrary::GetInstance();
}

} // end of namespace Apps::Demo::Sdk
```


Changes:

- The former code passed the `ARP_VERSION_CURRENT` macro to the constructor of `LibraryBase`. This is no longer necessary. The version of the SDK is passed to the library using another mechanism.
- The version parameter to the constructor of `LibraryBase` is a version number intended for the user. It can be used to track a version of a library. The new `Arp.System.Acf.Services.ISystemInfoService` provides methods to query information about the loaded components. For each component the `Arp` SDK version and the user version is provided by the service.
- The former implementation of the library constructor registers the provided components through accessing the protected `componentFactory` member variable directly. This is now replaced by the call of `LibraryBase::AddComponentType<T>` operation, where `T` is the component type to register.
- The singleton is implemented by the project's library itself through the `GetInstance()` operation, using a local static instance (see Scott Meyers (1996), *More Effective C++*, Addison-Wesley, pp. 146 ff.).
- The name of the library's entry function was static so far and always called `ArpDynamicLibraryMain`. From now on it depends on the project's name (or library's name, respectively). It consists of the safe name of the shared library omitting the extension .so as well as the default Linux lib prefix and appends a static suffix called `_MainEntry`. The safe name of the library replaces all special characters by underscores, e.g. if the project/library is called libArp.Plc.Gds.so, the main entry function shall be called: `Arp_Plc_Gds_MainEntry`.

– In the ACF configuration files (*.acf.config) the new optional attribute `mainEntry` of the `Library` element may be used to specify an alternative name for the `mainEntry` function.

From firmware release 2026.0 LTS on:

– The constructor needs this addition:

```
#if ARP_ABI_VERSION_MAJOR >= 2 && ARP_ABI_VERSION_MINOR >= 1
    this->SetSdkBuildInfo(ARP_VERSION_BUILT, ARP_ABI_VERSION_NAME, ARP_TARGET_IDENTIFIER);
#endif
```

2.3.2 The component implementation of a custom project

2.3.2.1 The component declaration

The project's component class is derived by `ComponentBase` from `Acf`.

Header file:

```
#pragma once
#include "Arp/Base/Core/Arp.hpp"
#include "Arp/Base/Acf/Commons/ComponentBase.hpp"

namespace Apps::Demo::Sdk
{

using namespace Arp;
using namespace Arp::Base::Acf::Commons;

class SdkComponent : public ComponentBase
{
public: // construction/destruction
    SdkComponent(ILibrary& library, const String& name);

public: // IComponent operations
    void Initialize(void)override;
    void SubscribeServices(void)override;
    void LoadSettings(const String& settingsPath)override;
    void SetupSettings(void)override;
    void PublishServices(void)override;
    void LoadConfig(void)override;
    void SetupConfig(void)override;
    void ResetConfig(void)override;
    void Dispose(void)override;
    void PowerDown(void)override;

public: // properties

public: // operations

private: // methods

private: // fields

private: // static fields
};

} // end of namespace Apps::Demo::Sdk
```


Changes:

– The signature of the constructor changed from `SdkComponent(IApplication& application, const String& name)` to `SdkComponent(ILibrary& library, const String& name)`
– The static `SdkComponent::Create(...)` operation having the same signature as the former constructor might be removed.

– All special member functions (copy and move operations, destructor) can be removed, if not required by the implementation of the derived class.
– All canonical constructors and assignment operators can be removed.
– The function `IComponent::GetVersion()` has been removed, since it is now ambiguous. It can be replaced using `GetLibrary().GetBuildVersion()`. Note the new function `ILibrary::GetLibraryVersion()`.

2.3.2.2 The component definition

The implementation of the project's component shall look like this:

```
#include "Apps/Demo/Sdk/SdkComponent.hpp"
#include "Apps/Demo/Sdk/SdkLibrary.hpp"

namespace Apps::Demo::Sdk
{

SdkComponent::SdkComponent(ILibrary& library, const String& name)
    : ComponentBase(library, name, ComponentCategory::Custom, GetDefaultStartOrder())
{
}

void SdkComponent::Initialize()
{
    // register components, initialize singletons
    // subscribe notifications or events here
}

void SdkComponent::SubscribeServices()
{
    // subscribe the services used by this component here
}

void SdkComponent::LoadSettings(const String& settingsPath)
{
    // load firmware settings here
}

void SdkComponent::SetupSettings()
{
    // setup firmware settings here
}

void SdkComponent::PublishServices()
{
    // publish the services of this component here
}

void SdkComponent::LoadConfig()
{
    // load project config here
}

void SdkComponent::SetupConfig()
{
    // setup project config here
}

void SdkComponent::ResetConfig()
{
    // implement this inverse to SetupConfig() and LoadConfig()
}

void SdkComponent::Dispose()
{
    // implement this inverse to SetupSettings(), LoadSettings() and Initialize()
}

void SdkComponent::PowerDown()
{
    // implement this only if data must be retained even on power down event
}

} // end of namespace Apps::Demo::Sdk
```

Changes:

- The signature of the constructor of class `ComponentBase` changed
  from `ComponentBase(IApplication&, ILibrary&, const String&, ComponentCategory, size_t, bool)`
  to `ComponentBase(ILibrary&, const String&, ComponentCategory, size_t)`
  Thus, the first parameter of type `IApplication` was removed. Pass the component's constructor arguments accordingly to `ComponentBase` class as shown in the example above. The start order of components must now be passed explicitly.
- Use `GetDefaultStartOrder()` for a sensible default value for the start order. The value shall only be changed to resolve dependencies of multiple custom components.

2.3.3 TraceController from Arp.System.Commons

The class `Arp::System::Commons::Diagnostics::TraceController` is removed from the SDK. The functionality is offered by the `Arp.Services.TraceController` component and its `ITraceControllerServiceRSC` service.

2.4 Library Arp.Base.Rsc.Commons

(formerly Arp.System.Rsc.Services)

2.4.1 Class RscVariant<N>

When calling RSC services in C++ with a parameter of type `RscVariant<N>`, then compile errors might occur because the interface of `RscVariant` has changed moderately. E.g., the constructors of `RscVariant` are declared as `explicit`
now, so that some notification sending calls using raw payload has to be adjusted.

```
nm.NonBlockingSendNotification(
    Registration2.GetNotificationNameId(),
    Timestamp,
   { NameID, Name, Timestamp.ToBinary(), int(Severity) });
```

has to be modified to:

```
nm.NonBlockingSendNotification(
        Registration2.GetNotificationNameId(),
        Timestamp,
       {       RscVariant<512>(NameID),
               RscVariant<512>(Name),
               RscVariant<512>(Timestamp.ToBinary()),
               RscVariant<512>(int(Severity))
       });
```

Furthermore, some constructors to create array or struct variants were made private and shall be replaced by static factory operations:

- `RscVariant<N>::CreateStructVariant(...)`
- `RscVariant<N>::CreateArrayVariant(...)`

2.4.2 Class RscString<N>

Related to template class `RscVariant<N>` and template class `RscString<N>`:
The template parameter `N` specifies the maximal length of the string. Formerly the effective length was `N-1`, due to the `NUL` terminator; now it is `N`. This issue was claimed frequently, so that it was fixed with the Application-relevant Changes release.

2.4.3 Further RSC changes

- If any RSC services were implemented (non-public feature), the service implementation code has to be re-generated by the `RscGenerator`. In some rare cases, the code of the service implementation has to be adjusted.
- The class `RscStreamAdapter` in C++ was renamed to `RscStream`. Thus, if the compiler claims a non-overridden `Service-Impl` operation, which uses `RscStreamAdapter`, just rename it to `RscStream`.
- RscStructReader.hxx and RscStructWriter.hxx have been replaced with RscStructReader.hpp and RscStructWriter.hpp (respectively). The template classes `RscStructReader<MaxStringSize>` and `RscStructWriter<MaxStringSize>` have been replaced with the non-template classes `RscStructReader` and `RscStructWriter` (respectively). The latter classes implement template operations which deduce the template arguments automatically.
- The following `RscType` entries have been removed: `Utf8String` (replaced by `String`), `AnsiString`, `Utf16String`.
- The class `SecureString` has been renamed to `RscSecureString`.
- When using RSC services from within C++, it is nowadays required to link to the according .so library, which contains the service implementation. This was not necessary before, because the de-/serialization code of structures or enumerations has been implemented inline inside the header files. The reason for this change is, that it was not possible before to apply changes or fixes to the inline serialization code.

2.5 Library Arp.Base.Commons

(formerly Arp.System.Commons)

2.5.1 Namespace Exceptions

A couple of exception classes have been moved from namespace `Arp::System::Commons::Exceptions` to namespace `Arp::Base::Commons::Exceptions`. This has an impact regarding the collecting header files

- Arp/System/Commons/Exceptions.h
- Arp/System/Commons/Exceptions/Exceptions.h

in a way that some of the imports into namespace `Arp` inside that file had to be disabled. Otherwise, it would cause name clashes between the exceptions from `Arp.System.Commons` and `Arp.Base.Commons` libraries.

The handling suggestion is:

- The collecting header file will be removed in future, when moving to library `Arp.Base.Commons`. Thus, do not use any of the collecting header files any more.

– If the compiler claims a missing exception type, use the new implementation from Arp/Base/Commons/Exceptions. The exceptions from the new library are all imported into the root namespace `Arp` by including the related header file, so that there is no need for an import declaration.

2.6 Library Arp.System.Commons

2.6.1 Class IpAddress from namespace Net

The operation `IpAddress::GetIpv4Value()` was marked as deprecated and is replaced by operation `IpAddress::GetNetworkValue()` according to the *host-to-network* (hton) and *network-to-host* (ntoh) functions.

2.6.2 Use of Logging API instead of class Console

The class `Arp::System::Commons::Console` was removed. Use Logging API instead (see next chapter).

2.6.3 Class StackTrace was renamed to Stacktrace (fixed casing)

The class `Arp::System::Commons::Runtime::StackTrace` was renamed to `Arp::System::Commons::Runtime::Stacktrace` to fix the erroneous casing. The name of the include file was adjusted accordingly.

3 Miscellaneous

3.1 Logging

3.1.1 Logging API

The logging substruction was refactored completely.

Nevertheless, the public API did not change, except of the following:

- The static logging implementation using macros was removed because it was not used.
- The logging implementation using streams was removed because it was not used.
- Static class `Log` was moved (see next section).

3.1.2 Static class "Log" (root logger)

The static class Log from namespace `Arp::System::Commons::Diagnostics::Logging` was moved to library `Arp.Base.Commons` into namespace `Arp::Base::Commons::Logging`.

The new `include` path and `using` declaration is:

```
#include "Arp/Base/Commons/Logging/Log.hpp"
using Arp::Base::Commons::Logging::Log;
```

There is a single major change when using this class. The static `Log` class shall be initialized explicitly by a reasonable logger name; e.g., the name of the application: `Log::Initialize("MyApp");`

If the initialization is omitted, the logging will be disabled.

3.1.3 Logging on class level

If logging on class level is wanted, use class `Arp::System::Commons::Diagnostics::Logging::Loggable` by deriving your own class from it.

This way, a logger instance called log is inherited by your class where the logger name is the full qualified type name of the class.

See the following examples.

Normal class

```
class Demo : private Loggable<Demo>
{
}
```

This code example shows how to use the inherited logger.

```
void Demo::Method(void)
{
   log.Debug("Any log"); // logs: "<TimeStamp> Demo   DEBUG - Any log"
}
```

Singleton class

For singleton classes inherit from class `Loggable` as follows:

```
class Demo : private Loggable<Demo, true>
{
}
```

Pure static class

For pure static classes inherit from class `Loggable` as follows:

```
class Demo : private Loggable<Demo, false, true>
{
}
```

For pure static classes, it is required to initialize the log member explicitly by once calling `Demo::InitializeLogger()`.

3.1.4 Log file separation

To filter and find log messages more easily, the Output.log will be split into multiple log files.

The main log file is renamed to Arp.log:

– Size: 16 MB + 1 backup file
– All loggers starting with Arp, Eclr and CommonRemoting log to this file (see exceptions below).

For the following components, separate log files will be created:

– PROFINET: Arp.Io.ProfinetStack.log
– SPNS: Arp.Services.SpnsProxy.log
– eHMI: Arp.Services.Ehmi.log
– Size: 2 MB each + 1 backup file

An additional log file will be created to store identification and version information at system start:

– Arp.Init.log
– Size: 1 MB + 1 backup file
– The purpose of this file is to store persistently some important information that would otherwise be overwritten by log file rotation. These include:

- Firmware version
- Vendor, article name, article number, hardware revision, serial number
- FPGA version
- SPNS firmware version
- PCIe extensions: vendor, article name
- File system: size, free space
- External SD card: present, enabled, free space
- Function and location, initial value and changes
- Network interfaces (no IP address configuration, since it is too volatile)

An additional log file will be created for customer messages, that is every logger whose name does not start with Arp (see exceptions above):

- Custom.log
- Size: 2 MB + 1 backup file

The separated files can be merged using the tool arp-merge-logs on the controller. For information, run `arp-merge-logs --help` in the console.

3.2 External libraries update

3.2.1 CppFormat library replaced by libfmt 10.2.1

The `CppFormat` library has moved to `libfmt`, and parts of the `libfmt` library were adapted to the C++ standard.

PLCnext Technology does not use the `std::format` due to lacking compiler support and compatibility issues with already present `operator<<` implementations for user-defined types. But PLCnext Technology uses the `libfmt` code directly, which was now updated to version 10.2.1.

There are no compatibility issues expected except those listed in class String .

3.2.2 boost libraries updated to version 1.84

The previously used version 1.63 of the `boost` libraries is quite old and does not support C++ 20. To make newer features available, the `boost` libraries are updated to version 1.84 from the 2025.0 firmware release on.

Some of the `boost::filesystem` classes have minor changes regarding their behavior, but this was adapted in the PLCnext Technology firmware code. Thus, there are no compatibility issues expected, as long as the `boost` code was not used directly by custom code.

3.3 CMake adjustments

The PLCnext Technology SDK now provides CMake export configuration files. A virtual package `Arp` is provided that includes all the packages included in the SDK. This defines the target `Arp::ALL` that encompasses all relevant libraries and sets the basic usage requirements for the libraries. These include C++ 20 and linker flag `--no-undefined`.

Link at least one of the `Arp` libraries to enable these preparations.

In CMakeLists.txt:

```
find_package(Arp REQUIRED)
target_link_libraries(YourTarget PRIVATE Arp::ALL)
```

Individual libraries are exported in the namespace `Arp`. For example, to explicitly link the GDS, use `Arp::Arp.Plc.Gds`.

For cross-compiling on Windows®, the `mingw` SDK includes a CMake toolchain file in sysroots\x86_64-w64-mingw32\usr\share\cmake\OEToolchainConfigStandalone.cmake.

The function `arp_add_tracing` from cmake/ArpTracing.cmake was moved to the export package. It is available after `find_package(Arp)`.

3.4 API documentation

The API documentation is provided as an HTML-based online help system for each firmware release from 2020.0 LTS up to the latest releases. Each online help system is publicly available at PLCnext Technology C++ API documentation.

3.5 PLCnext CLI adaption

The templates of PLCnext CLI 2025.0 (download at PLCNEXT TECHNOLOGY TOOLCHAIN | Phoenix Contact) have been reworked to support the application-relevant changes.

Therefore, in order to re-use an existing project, a new project needs to be created by means of PLCnext CLI 2025.0. All `Components` and `Programs` have to be created newly, too. Their old source code needs to be merged into the new files. All other files of the src folder may be copied into the new project.

Due to changes in the `Library` and `Component` class, the existing code has to be merged manually (member/method by member/method).

3.6 Changing the initialization system

Every Linux® system has an initialization system which is responsible for starting the daemons and the system configuration (e.g., network configuration) when the system boots. Up to now, PLCnext Technology firmware has used `SysV` initialization. Due to the increasing complexity of the PLCnext Runtime System and the growing importance of container technology (e.g., Podman or Docker®), it is necessary to switch to the more modern and complex `systemd`.

Users who intervene directly in the Linux® OS to start their own daemons, or to integrate additional scripts into the system start, will have to convert their start scripts from `SysV` init to `systemd` service files. Since container apps from the PLCnext Store currently initialize and start their containers via a special init script, these apps must also be adapted accordingly.

Another impact is the network configuration, as `systemd` has its own integrated network management. Therefore, the Linux® tools `ifup` and `ifdown` no longer exist, and the file /etc/network/interfaces will also no longer exist.

This affects all users who manually intervene in the network configuration via the /etc/network/interfaces file. If a user only uses `Arp` mechanisms for network configuration (e.g., PLCnext Engineer or the Web-based Management), no adjustments are necessary.

3.7 Rework of network management

With the change of the Linux® initialization system (details see Changing the initialization system), the Linux® network management is also changed from the `ifup`/`ifdown` procedure with /etc/network/interfaces to the network management integrated in `systemd` via `networkd`.

This change allows to implement functions such as multiple IPs per interface, multiple gateways, VLANs, and more in the medium term. It also increases the stability of the system when the IP is changed using Linux® tools.

The network configuration is now handled by the `networkd` daemon, which is part of `systemd`. The Linux® tools `ifup` and `ifdown` no longer exist, and the file /etc/network/interfaces will no longer be evaluated. This affects all users who manually intervene in the network configuration via the /etc/network/interfaces file. How the conversion of the IP configuration is to be done can be found in the change notes for firmware 2025.0.

If a user only uses `Arp` mechanisms for network configuration (e.g., PLCnext Engineer or the Web-based Management), no adjustments are necessary.

3.8 Activation of Usrmerge

Since 2012, some Linux® distributions have been implementing the project `Usrmerge` which has the goal to remove the folders /bin, /lib and /sbin by "merging" their contents into the folders /usr/bin, /usr/lib and /usr/sbin. The reason for this is that this duplication of directories is historical and no longer necessary today. The result is a clearer and simpler directory structure. For compatibility reasons, only symbolic links are created for /bin, /lib and /sbin. In addition, the home directory of the `root` user is no longer /home/root but just /root.

Since 2022, major Linux® distributions like Debian have activated `Usrmerge`, and since 2024 also the Yocto Project® which is used to generate the PLCnext Linux OS did that. Newer versions of `systemd` now require `Usrmerge`, so that there is no longer an alternative to activation.

The folders /bin, /lib and /sbin do not exist anymore in the PLCnext Linux OS. Instead, symbolic links are created:

```
/bin --> /usr/bin
/lib --> /usr/lib
/sbin --> /usr/sbin
```

Special care should be taken by users who have stored files in one of the directories mentioned, as these could then overlay the symbolic links in the file system after an update to firmware version 2025.0 which would lead to unpredictable behavior.

3.9 Web-based Management 2 (WBM 2)

The previous Web-based Management (WBM) is completely replaced by a new development, called WBM 2. The reasons for the new development are, on the one hand, a new work standard for the definition of web interfaces for Phoenix Contact devices, which describes the design in a new way and considers contemporary features such as mobile views or responsive design. As well, the previous web technology used internally was no longer up to date.

The WBM 2 is set up with the help of a framework (Angular.io), offering many more options and futureproofing. In addition, it will be possible for users to seamlessly integrate their own pages into the controller's WBM, e.g., via apps from the PLCnext Store.

The look and feel as well as the user operation of the WBM 2 are new. Functionally, little to nothing has changed on the pages.

3.10 Removal of Linux packages/tools/libraries

3.10.1 Removal of the vim editor

In addition to the `nano` editor, the `vim` editor is also integrated in PLCnext Linux.

In the recent past, security vulnerabilities (CVEs reported to the Phoenix Contact Product Security Incident Response Team (PSIRT)) have frequently been discovered in `vim`, so that the editor had to be updated frequently. As the security reports continue to be more frequent, the `vim` editor is to be removed from the system. This will make the system more secure.

Only users who intervene directly in the Linux system and use the `vim` editor there are affected by this change. These users must switch to the `nano` editor, or alternatively translate the `vim` editor themselves with the SDK and integrate it into PLCnext Linux. It would also be conceivable to provide an app that contains the `vim` editor.

3.10.2 Removal of the busybox package

Every Linux® system has a set of elementary basic tools such as `cp`, `mount`, or `grep`. These tools are provided by program collections such as GNU `core-utils`, `util-Linux` or `net-tools`. One collection that is optimized in terms of resource requirements is `busybox`, which was originally developed with a focus on use on embedded systems. The resource requirements have been reduced by omitting features of the tools.

However, many advanced Linux® technologies (e.g., container engines) require the basic tools in their full configuration, which is only provided by the mentioned collections. Over the years, PLCnext Linux has seen a difficult to understand coexistence of tools from collections such as `core-utils` or `util-Linux` on the one hand and `busybox` on the other.

To regain order and future-proof the basic tools, it was decided to no longer integrate `busybox` but to rely exclusively on the standard tools of the larger collections.

The impact on the user should be very small or even imperceptible. All tools that were previously provided via `busybox` can still be found in the system under the same name. As the `busybox` tools each support a subset of the features of the "big" tools, previous calls should continue to work as expected.

Restriction: If users have developed parsers that evaluate the output on `stdout` of the tools, these parsers will no longer work in many cases, as the output of the `busybox` tools differs from that of the tools from the larger collections. Users who call the `busybox` binary directly are also affected, although this should very rarely be the case.

3.10.3 Change of NTP daemon

The currently used `ntpd` provides a basic set of functions. However, for more complex time synchronization tasks like those mandatory for the implementation of Time-Sensitive Networking (TSN) this function set is not sufficient anymore.

Therefore, the `ntpd` daemon will be replaced by the `chrony` daemon. All configuration files will be invalidated or removed from the system.

All configuration files will be invalidated/removed from the system. This change will only affect users who intervene with the Linux® system by means of the `ntpd` daemon itself and/or its configuration.

3.10.4 Removal of strongSwan legacy configuration interface

The strongSwan Team declared the `stroke` plug-in as deprecated and it is disabled by default in strongSwan 6.0:

"The legacy `stroke` management interface has been deprecated for many years and has been replaced by the versatile `vici` management interface. Thus with strongSwan 6.0, the `stroke` plugin is not enabled by default anymore and has to be built separately."

To avoid a forced removal of this deprecated plug-in in later PLCnext Technology firmware versions, it is removed from the firmware release 2025.0 and newer. Existing configuration files (ipsec.conf) from 2024.6 or earlier firmware must be migrated to the new swanctl.conf syntax (read the strongSwan wiki article).

Also the `ipsec` script was removed. To control an `ipsec` connection, the swanctl commands must be used.

4 Security-related changes

4.1 Prevent using RTLD_GLOBAL when loading shared libraries

When loading shared libraries (e.g. ACF or PLM components) the flag `RTLD_GLOBAL` was used for the `dlopen()` system call. This causes the symbols of a loaded library to become globally visible and may result in executing an unintended function.

The `RTLD_GLOBAL` flag is omitted now when loading shared libraries. Compatibility issues are not expected regarding this issue.

4.2 Removing unprivileged folders from ld.so.conf

Up to firmware 2024.6, the following folders or entries have been put into the file /etc/ld.so.conf :
`/usr/local/lib include /opt/plcnext/appshome/ld.configs/*.conf`

This was used to find user programs or *.so files that have dependencies on other *.so files but are not in the system. *.so files integrated into the system in this way are made known system-wide. They could therefore also be loaded by processes running under `root` privileges by mistake and thus cause all kinds of damage to the system. So that configuration needed to be removed.

From firmware release 2025.0 on, programs and *.so files that have further dependencies but are not present in PLCnext Technology must enter a fixed search path (`rpath`). This happens at the time of creation (linking) of the program or of the *.so file, or later by using `chrpath`.

4.3 Redesign of remoting to platform and security requirements

Security mechanisms in remoting do not work under all circumstances. In particular, security is restricted if there is more than one process running. From a security point of view, splitting processes is necessary. Firmware 2025.0 contains further preparations for a future splitting into several processes.

The only currently known effect is that multiple remoting sessions (logins) within one TCP connection are no longer possible.

4.4 ACF can restrict capabilities and UID/GID of processes

These measures are intended to achieve a better separation of system functions and user applications, so that applications cannot cause any damage to the system:

Up to firmware 2024.6, the entire `ARP` framework ran within a few processes. In addition, all processes ran with the same rights in the system. For example, an application also received all rights from the user that are actually only required for system functionalities. This needed to be reworked and firmware 2025.0 is prepared for such additional restrictions.

In special cases, authorizations that have been possible for applications before may no longer be available in the system with firmware 2025.0 or newer.

4.5 Verification of signed application update containers

PLCnext Engineer Application updates should be protected against modification so that only correctly signed PLCnext Engineer application update containers are accepted. In order to use this feature, users must sign such containers and upload certificates into the Code signing Trust List on the controller via the Security → Certificate management WBM 2 page. Then, in the Security → Project integrity WBM 2 page, the check needs to be enabled and configured.

4.6 App part types Linux Daemon and Shared Library

The app part types Linux Daemon and Shared Library are no longer supported. The reason given for this is the security risks that can emanate from these app part types.

Linux Daemon:

The Linux Daemon app part contained in the PLCnext Technology App is integrated into the system by the `AppManager` in such a way that it is started with `root` privileges by the initialization system at system startup. Since the `AppManager` cannot check the Linux Daemon itself, malicious code in the form of processes, one-time programs or scripts can be infiltrated in this way and called or started with `root` privileges. From firmware 2025.0, the OCI container app part type is supported as a successor.

Shared Library:

The Shared Library app part contained in a PLCnext Technology App is integrated into the system by the `AppManager` and made known to the entire system by calling `ldconfig`. The attack vector here is the possibility of replacing existing Shared Libraries app parts with the help of an app, and thus, injecting any programs with manipulated Shared Libraries that potentially contain malicious code without being noticed.

Phoenix Contact GmbH & Co. KG
Flachsmarktstr. 8
32825 Blomberg, Germany
Phone: +49 5235 3-00
Email: info@phoenixcontact.com
phoenixcontact.com

111781_en_04