Click to verify

Using S7 Net Plus for PLC Programming: A Step-by-Step Guide 首先，在使用的地方引用必要的NuGet 包：S7NetPlus。官网可以访问在 ��该软件环境使用的是.net core 控制台程序。 要开始，需要引用以下所需的类： ``` using ConsoleApp1; using Microsoft.VisualBasic; using S7.Net; using S7.Net.Types; using System; using System.ComponentModel.Design; using System.Text; ``` 然后可以创建一个新的PLC对象，使用CPU类型S71500和IP地址192.168.43.14。 ```csharp Plc plc = new Plc(CpuType.S71500, "192.168.43.14" + "", 0, 1); ``` 在这里，机架号通常为0和1，插槽号也是0和1。 下一步是打开PLC连接： ```csharp plc.Open(); if (plc.IsConnected) { Console.WriteLine("PLC连接成功"); } ``` 该方法会建立一个TCP连接。 要关闭PLC连接，可以使用以下代码： ```csharp plc.Close(); ``` 在阅读数据之前，需要了解PLC中类型与C#之间的转换规则。以下是PLC中的不同数据类型: - byte => byte - word => ushort - DWord => uint - Int => short - DInt => int - Real => float - LReal => double - String => string - DateTimeLong=>datetime - s7wstring=>string 在使用这些类型时需要注意转换规则。 以下是一个简单的示例，展示了如何阅读和写入PLC数据: ```csharp var a = plc.Read("DB1.DBX0.0"); plc.Write("DB1.DBX0.0", true); ``` 在实际开发中，最好使用以下方案来节省资源： ```csharp int db = 1; var boolDemo = plc.Read(DataType.DataBlock, db, 0, VarType.Bit, 1); Console.WriteLine("bool值打印：" + boolDemo); plc.Write(DataType.DataBlock, db, 0, false); var boolDemoRead = plc.Read(DataType.DataBlock, db, 0, VarType.Bit, 1); Console.WriteLine("修改后bool值打印：" + boolDemoRead); ``` 在途中，第一个参数标识数据块类型，第二个参数标志数据块。 以下代码是其他值类型的示例： ```csharp var wordDemo = plc.Read(DataType.DataBlock, db, 2, VarType.Int, 1); Console.WriteLine("word值打印：" + wordDemo); plc.Write(DataType.DataBlock, db, 2, (ushort)9); var wordDemoRead = plc.Read(DataType.DataBlock, db, 2, VarType.Int, 1); Console.WriteLine("修改后word值打印：" + wordDemoRead); // Dword 类型 var DwordDemo = plc.Read(DataType.DataBlock, db, 4, VarType.Int, 1); Console.WriteLine("Dword值打印：" + DwordDemo); plc.Write(DataType.DataBlock, db, 4, 8); var DwordDemoRead = plc.Read(DataType.DataBlock, db, 4, VarType.Int, 1); Console.WriteLine("修改后Dword值打印：" + DwordDemoRead); // Int 类型 var intDemo = plc.Read(DataType.DataBlock, db, 8, VarType.Int, 1); Console.WriteLine("int值打印：" + intDemo); short readInt = 9; plc.Write(DataType.DataBlock, db, 8, readInt); var intDemoRead = plc.Read(DataType.DataBlock, db, startByteAdr: 8, VarType.Int, 1); Console.WriteLine("修改后int值打印：" + intDemoRead); // Dint 类型 var DintDemo = plc.Read(DataType.DataBlock, db, 10, VarType.Int, 1); Console.WriteLine("Dint值打印：" + DintDemo); plc.Write(DataType.DataBlock, db, 10, 100); var DintDemoRead = plc.Read(DataType.DataBlock, db, 10, VarType.Int, 1); Console.WriteLine("修改后Dint值打印：" + DintDemoRead); // Real 类型 var RealDemo = plc.Read(DataType.DataBlock, db, 14, VarType.Int, 1); Console.WriteLine("Real值打印：" + RealDemo); plc.Write(DataType.DataBlock, db, 14, (float)99.9); var DrealDemoRead = plc.Read(DataType.DataBlock, db, 14, VarType.Int, 1); Console.WriteLine("修改后Real值打印：" + DrealDemoRead); // LReal 类型 var LRealDemo = plc.Read(DataType.DataBlock, db, 18, VarType.Int, 1); Console.WriteLine("LReal值打印：" + LRealDemo); plc.Write(DataType.DataBlock, db, 18, 88.88); var LrealDemoRead = plc.Read(DataType.DataBlock, db, 18, VarType.Int, 1); Console.WriteLine("修改后Lreal值打印：" + LrealDemoRead); // byte 类型 var byteDemo = plc.Read(DataType.DataBlock, db, 290, VarType.Byte, 1); Console.WriteLine("byte值打印：" + byteDemo); plc.Write(DataType.DataBlock, db, 290, (byte)2); var byteDemoRead = plc.Read(DataType.DataBlock, db, 290, VarType.Byte, 1); Console.WriteLine("修改后byte值打印：" + byteDemoRead); // DateTime 类型 var dateDemo = plc.Read(DataType.DataBlock, db, 282, VarType.DateTime, 1); Console.WriteLine("data值打印：" + dateDemo); plc.Write(DataType.DataBlock, 2, 282, System.DateTime.Now); var dateDemoRead = plc.Read(DataType.DataBlock, db, 282, VarType.DateTime, 1); Console.WriteLine("修改后data值打印：" + dateDemoRead); // String 类型 var charDemo = plc.Read(DataType.DataBlock, db, 4, VarType.String, 1); Console.WriteLine("char值打印：" + charDemo); plc.Write(DataType.DataBlock, db, 4, "a"); var charDemoRead = plc.Read(DataType.DataBlock, db, 4, VarType.Int, 1); ```### 动态链接库（DLL）的优点和应用 动态链接库是C++程序中一个重要的概念，它使得程序可以在运行时加载和卸载所需的功能，从而增加了程序的灵活性和可扩展性。下面是关于动态链接库的优点和应用： ### 优点 1. **减小程序大小**: 多个程序可以共享同一个 DLL 文件，导致总体程序大小的缩小。 2. **增加灵活性**:Programs可以在运行时添加或移除功能，根据实际需求来调整其性能和实用性。 ### 应用 1. **游戏开发**: 动态链接库有助于实现游戏中的动态加载、卸载和更新。 2. **系统管理**: 动态链接库用于管理系统的资源和服务，可以灵活地扩展或减少系统的功能性。 3. **数据库连接**: 动态链接库使得程序可以轻松连接到不同的数据库，实现多数据库的兼容性。 ### 综合性 1. **应用于多平台**: 动态链接库可以在不同操作系统上运行，从而实现跨平台开发。 2. **提高编程效率**: 动态链接库降低了程序的复杂度和开销，提高了编程效率。 ### 总结 动态链接库在C++程序中扮演着至关重要的角色，它使得程序更加灵活、可扩展并且能够减小其大小。通过了解 DLL 的优点和应用，我们可以更好地利用这种技术来提高我们自己的编程实践。 ###参考资源 * [Visual Studio Documentation](* [C++ Documentation](* [CSDN Resources](Paraphrased text: The article discusses how to create and use a dynamic link library (DLL) in C++ and Visual Studio. The DLL is used to store reusable code that can be shared among multiple applications. To verify that everything is working correctly, the user should compile the DLL. To do this, they can select "Build" > "Rebuild Solution" from the menu bar. The compiled library and related compiler output will be placed in the "Debug" folder of the solution directory. If a release version is created, the output will be placed in the "Release" folder. The user then needs to copy the DLL file into the client application's project directory. This can be done by setting up the "Additional Include Directories" and "Additional Dependencies" properties for the client project. Next, the user needs to add the DLL import library to the client project. This is done by setting up the "Additional Dependencies" property in the linker settings. The user also needs to set up the "Additional Library Directories" property to point to the location of the DLL file. Finally, the user needs to configure the post-build event for the client project to copy the DLL file into the output directory. This is done by setting up a command in the "Post-Build Event" settings that uses the xcopy command to copy the DLL file from its current location to the output directory. With these steps complete, the client application should be able to successfully compile and run, and it will be able to use the functions and variables exported by the DLL. paraphrased text here The author of the article was struggling to get their single-board computer's audio hardware working. They had written a program that used the I2C protocol to communicate with the chip, but it wasn't producing any output. After some trial and error, they discovered that adding an pull-up resistor on the IO pin of the DS1302 chip resolved the issue. However, when they tried to add a 4.7K pull-up resistor as recommended by some online sources, it actually caused problems. The author suspected that the long length of the connection between the single-board computer and the DS1302 chip was the cause of the problem, but further investigation showed that this was not the case. Finally, they tried re-wiring the connection to ensure that there were no loose connections or broken wires. This proved to be the solution, and the author's program was able to read data from the DS1302 chip correctly. The author notes that they are surprised by how often issues like this arise in electronics projects, and hopes that their experience will save others from similar problems in the future.