**一、类介绍**

■ Point

[Serializable]

public class Point

{

public double x { set; get; }

public double y { set; get; }

public double z { set; get; }

public Point()

{

this.x = -1d;

this.y = -1d;

this.z = -1d;

}

public Point(double x, double y, double z)

{

this.x = x;

this.y = y;

this.z = z;

}

public Point(double x, double y)

{

this.x = x;

this.y = y;

}

}

Point类是一个坐标类，该类有两个double成员变量x、y、z，默认这三个成员变量为-1d,当然我们在生成这个类的对象时可以对两个变量初始化。

Point类还有一个特殊的地方是可以序列化，这样我们就可以将坐标讯息保存到硬盘上。

注：我们在初始化基站坐标及计算Tag的坐标时都是使用这里的Point类，而没有选择系统中的System.Drawing.Point;

■ TagInfo和BsInfo

public class TagInfo

{

public byte[] Id = new byte[2];

public double height = -1d;

}

public class BsInfo

{

public byte[] ID = new byte[2];

public byte[] GroupID = new byte[2];

public Point Place = new Point();

}

TagInfo类是Tag的基本讯息类，主要用来设置Tag的高度。而BsInfo类是基站的基本讯息类，用来设置基站的高度、坐标(含高度)以及所属组别。它们的主要作用在后面介绍函数InitBasicStations(BsInfo[] BStations)和InitTag (TagInfo [] tginfos)时会介绍。

■ TagPlace

[StructLayout(LayoutKind.Sequential)]

public struct TagPlace

{

[MarshalAs(UnmanagedType.ByValArray, SizeConst = 2)]

public byte[] ID;//卡片ID

[MarshalAs(UnmanagedType.ByValArray, SizeConst = 2)]

public byte[] GroupID;//组别ID

public double X, Y, Z;//X,Y,Z坐标，其中Z表示高度，当选择三点定位时，是没有高度的

public byte Battery;

public int index;

public byte LocalType;

public ushort NoExeTime;

public ushort SleepTime;

[MarshalAs(UnmanagedType.ByValArray,SizeConst = 2)]

public byte[] ReferID1;

[MarshalAs(UnmanagedType.ByValArray,SizeConst = 2)]

public byte[] ReferID2;

[MarshalAs(UnmanagedType.ByValArray,SizeConst = 2)]

public byte[] ReferID3;

[MarshalAs(UnmanagedType.ByValArray,SizeConst = 2)]

public byte[] ReferID4;

[MarshalAs(UnmanagedType.ByValArray,SizeConst = 2)]

public byte[] ReferID5;

public double Dis1, Dis2, Dis3, Dis4, Dis5;

public int SigQuality1, SigQuality2, SigQuality3, SigQuality4, SigQuality5;

public double ResidualValue1, ResidualValue2, ResidualValue3, ResidualValue4, ResidualValue5;

}

向上位抛Tag的消息结构体

■ BasicReport

public struct BasicReport

{ public byte[] ID;

public ushort SleepTime;

public uint Version;

}

向上位抛基站的消息结构体

■ TPPID

/// <summary>

/// 协议标识

/// </summary>

public class TPPID

{

public const byte Head = 0xFC;

public const byte End = 0xFB;

public const byte TagType = 0x01;

public const byte RouterType = 0x02;

public const int TagPackLen = 18;

public const int RouterPackLen = 13;

public const byte RouterVersionLen = 6;

public const int WPARAM\_TYPE = 0;

public const int WBASIC\_TYPE = 1;

public const int WTAG\_TYPE = 0;

public const int WBIGVERSION = 0;

public const int WSMALLVERSION = 1;

public const int USER = 0x0400;

public const int WM\_TAG\_PACK = USER + 500;

public const int WM\_DEVICE\_DIS = USER + 501;

public const int WM\_VERSION\_ERR = USER + 502;

}

TPPID类是协议标识类，由于在DLL中已经把处理协议的相关讯息封装好了，所以我们在这个类中只需要知道WPARAM\_TYPE（包括）后面的常量的作用即可。

在处理上报数据中，我们只要计算出Tag的坐标讯息就会向上位应用抛出消息，我们需要在上位窗体中重写protected override void WndProc(ref Message m)函数用于处理消息。

m.Msg用于区分不同的消息类型:

WM\_TAG\_PACK:Tag的定位包消息及基站上报自身数据包消息

WPARAM\_TYPE：定位包消息

WBASIC\_TYPE：基站上报自身数据包

WM\_DEVICE\_DIS:检测到设备断开的消息

WBASIC\_TYPE：基站断开连接的消息

WTAG\_TYPE:Tag断开连接的消息

WM\_VERSION\_ERR:检测到基站设备版本不同时抛出的消息

WBIGVERSION:基站的大版本不同时抛出的消息

WSMALLVERSION: 基站的小版本不同时抛出的消息

**二、函数介绍**

■ bool InitBasicStations(List<BsInfo> BStations)

初始化基站讯息，包括基站的Id、坐标、基站所属组别以及高度。

这个操作是在我们进行连接之前就需要执行，在进行数据处理时，我们只有知道基站(至少3个基站)在地图上的实际坐标再结合它们的距离才能计算出Tag当前的位置，从而抛给上位应用。

这里需要高度是为了更准确知道Tag距离基站的水平距离，下位上报上来的距离实际上是Tag距离基站的距离，若Tag和基站不在一个水平面上时，我们若知道Tag与基站之间的高度差，就可以计算出它们的水平距离了。

之所以需要基站所属组别讯息，是因为在进行多区域定位时，我们需要根据下位上报上来的基站讯息和设置的基站所属区域来确定当前卡片所属区域，从来进行精准定位。

■ bool InitTag(TagInfo[] tginfos)

初始化Tag的讯息，包括Tag的高度和Id。

我们在计算Tag距离基站的水平距离时，使用Tag距离基站的距离的平方 - Tag与基站的高度差的平方 再开方即为Tag与基站的水平距离。

Tag距离基站的水平距离：L

Tag的高度：TH

基站的高度为：BH

Tag距离基站的距离：D

L = Sqrt(D\*D - (TH-BH)\*(TH-BH))

若我们进行了初始化基站讯息而没有初始化Tag的讯息时，我们会把基站的高度当做Tag与基站的高度差来计算Tag与基站的水平距离。

InitTag函数一定要与InitBasicStations函数一起使用才有效。

■ bool InitNet(IPAddress ip, int Port)

初始化网络数据，包括ip及端口，只有网络数据初始化完成后，我们才能连接数据。

■public static bool Start(IntPtr CurHandler, ReportMode CurReportMode,PosititionMode mPositionMode,AfewDPos adpos)

连接开始即开始监控指定的端口，在这个函数中有两个参数，分别是CurHandler和CurReportMode。

CurHandler是当前调用函数的窗体句柄指针。

CurReportMode是连接方式（ImgMode,ListMode）

图形模式(ImgMode)，则只有计算出Tag的坐标时才上报讯息。

列表模式(ListMode),不用计算出Tag的坐标，每次上报得到最优的3的基站的讯息就向上位窗体上报讯息。

mPositionMode

SigQuality：信号质量优先。

Closestdistance：最近距离优先。

Adpos是选择是二维定位还是三维定位

Pos3Dim: 三维定位(至少需要四个基站)

Pos2Dim：二维定位(至少需要三个基站)

■ void Stop()

断开连接即取消端口的监听。

■ public static bool GetHGLobj(IntPtr handler,out TagPlace tpk)

将非托管内存指针处保存的结构体取出。

■ void FreeHGLOBAL(IntPtr Hander)

释放非托管区内存函数，向上位抛消息我们是使用非托管的内存区，即将需要向上位抛的消息结构体放到一个非托管的内存区，得到该非托管内存区的一个指针，再将这个指针抛给上位软件，上位软件收到后，将消息结构体重新生成，再将该块内存区释放掉即可，这个函数及是释放非托管内存的函数。

1. **Dll具体使用介绍**

**Dll函数调用流程:**

初始化基站讯息 => 初始化网络讯息（Ip及端口）=> 开始端**口**监听=>上位应用在重写的protected override void WndProc(ref Message m)函数中接收消息=>判断消息类型进行不同的处理=>释放非托管内存区=>断开连接

在上面的流程图中基本都是调用对应的函数,所以我们主要就讲解如何在WndProc(ref Message m)函数中接收消息及处理不同的消息类型

我们都知道Message结构体中包含msg,LParam及WParam等参数，

这里我们将不同的消息类型用msg区分，每一个消息类型中我们又区分基站的和Tag.

protected override void WndProc(ref Message m)

{

switch(m.Msg)

{

case PrecisePositionLibrary.TPPID.WM\_TAG\_PACK:

if((int)m.WParam==PrecisePositionLibrary.TPPID.WPARAM\_TYPE)

{

PrecisePositionLibrary.PrecisePosition.TagPlace tp;

if (!PrecisePositionLibrary.PrecisePosition.GetHGLobj(m.LParam, out tp))

{

return;

}

PrecisePositionLibrary.PrecisePosition.FreeHGLOBAL(m.LParam);

}else if((int)m.WParam == PrecisePositionLibrary.TPPI D.WBASIC\_TYPE)

{

PrecisePositionLibrary.PrecisePosition.BasicReport br = (PrecisePositionLibrary.PrecisePosition.BasicReport)Marshal.PtrToStructure(m.LParam, typeof(PrecisePositionLibrary.PrecisePosition.BasicReport)); PrecisePositionLibrary.PrecisePosition.FreeHGLOBAL(m.LParam);

}

break;

case PrecisePositionLibrary.TPPID.WM\_VERSION\_ERR:

if((int)m.WParam==PrecisePositionLibrary.TPPID.WBIGVERSION)

{

}else

if((int)m.WParam==PrecisePositionLibrary.TPPID.WSMALLVERSION)

{

}

break;

case PrecisePositionLibrary.TPPID.WM\_DEVICE\_DIS:

if ((int)m.WParam==PrecisePositionLibrary.TPPID. WBASIC\_TYPE)

{//基站类型

PrecisePositionLibrary.PrecisePosition.BasicReport br = (PrecisePositionLibrary.PrecisePosition.BasicReport)Marshal.PtrToStructure(m.LParam, typeof(PrecisePositionLibrary.PrecisePosition.BasicReport)); PrecisePositionLibrary.PrecisePosition.FreeHGLOBAL(m.LParam);

Console.WriteLine("BasicStation DisConnect!");

}

else if((int)m.WParam==PrecisePositionLibrary.TPPI D.WTAG\_TYPE)

{//Tag类型

PrecisePositionLibrary.PrecisePosition.TagPlace tp = (PrecisePositionLibrary.PrecisePosition.TagPlace)Marshal.PtrToStructure(m.LParam, typeof(PrecisePositionLibrary.PrecisePosition.TagPlace)); PrecisePositionLibrary.PrecisePosition.FreeHGLOBAL(m.LParam);

Console.WriteLine("Tag DisConnect!");

}

break;

}

}

1. Tag坐标点过滤

要想对Tag的坐标点(x,y,z)进行过滤，我们必须创建两个Kalman的对象，一个表示X的坐标，表示Y的坐标，另外一个表示高度Z。

在创建Kalman对象时，我们需要提供4个变量，分别是X，P，Q，R。

其中X是坐标点的X,Y,Z坐标

P是状态预测的协方差(一般选择一个较小的数如：0.1)

Q是过程噪声协方差（默认0.2即可）

R是测量噪声(这个一般是根据Tag上报的频率来确定，频率较快时我们可以选择一个较大的数，反之选择一个较小的数)

如当第一次抛上来Tag的坐标(x0,y0)时，我们创建了两个Kalman对象，分别为KalmanX,KalmanY;

KalmanX = new Kalman(x0,0.1,0.2,0.5);

KalmanY = new Kalman(y0,0.1,0.2,0.5);

KalmanZ = new Kalman(y0,0.1,0.2,0.5);

后面每次上报上来的坐标我们都可以直接调用

KalmanX.Kalman\_Filter(x);

KalmanY.Kalman\_Filter(y);

KalmanY.Kalman\_Filter(z);

即可获得滤波过后的坐标X = KalmanX.X,

Y = KalmanY.X;

Z = KalmanZ.X;

这种过滤坐标点的方法也有不足的地方，如存在连续丢包的情况下，若任然使用这种方法就会使得到的坐标点与实际坐标点偏离较远。

为了避免这种情况，需要我们根据序列号来判断是否存在丢包的情况如：

上一个的序列号为56，而当前包的序列号为60，假设Tag的休眠时间为1s，那么中间就丢了3包，也就是有3s的时间没有接收到数据，此时我们再使用Kalamn\_Filter()函数过滤坐标点显然不合适。

此时，我们就需要用实际上报上来的坐标为此时的Kalman坐标,即重新以当前上报上来的坐标创建Kalman新的坐标，方法同一开始的创建是一样的。