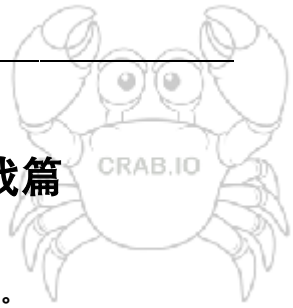


大脚蟹快速开发平台学习教程《四》：MiCO 实战篇

上一节，我们学习了《高级编程篇》，本篇将学习硬件控制和 **MiCO** 模块实战应用。

本文默认所有的工具已经准备就绪中。

本文以庆科公司的 **EMW3166** 模块为教程模块。

本文建议大家使用 **SublimeText 3.0**，不仅界面清爽美观，而且功能强大好用。

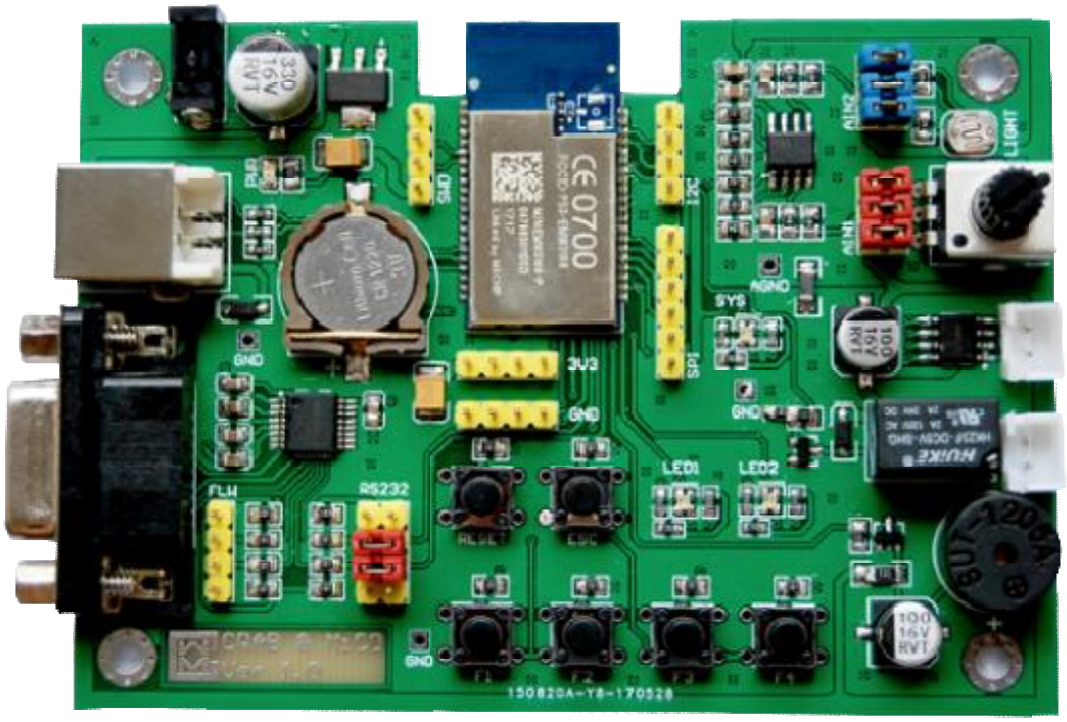
一、EMW3166 简介

特性：

- 集 **ARM-Cortex M4, WLAN MAC/BB/RF** 于一体
  - **100MHz** 的 **Cortex-M4 MCU**
  - **256KB RAM**
  - **1MB** 片内 **Flash**，**2MB** 片外 **SPI Flash**
- **WiFi** 相关特性
  - 支持 **802.11b/g/n** 标准
  - 支持 **Station, SoftAP** 模式
  - 支持 **EasyLink** 配网
- 宽幅工作电压：**2.3V-3.6V**
- 主接口：**UART**
- 尺寸：**16.0*32.0mm**
- 工作温度：**-30℃ to +85℃**

1）准备工作

本篇是以应用为主，所以默认你已经拿到这个具有 **EMW3166** 核心模块的开发板，并且准备好各种连接线。（详情请参阅第一篇《环境搭建篇》）



2）关於固件

上面的开发板，已经烧录好 **CRAB for MiCO** 固件，如果你需要重新烧录，或者如果你是自己 **DIY** 的开发板，请记得先烧录 **CRAB** 固件，再继续下面的章节。

如果你需要自己订制和自己修改固件，请自行下载源代码修改和编译（详情请参阅《系统移植篇》）。

固件和源代码下载地址：**http://www.wisearm.com/crab**

3）主板简介

- 本开发板以 **EMW3166** 为核心模块，提供以下硬件资源
- **USB** 接口：**CRAB** 的主要数据接口，用于应用程序下载，数据交互，调试日志监控。
- **RS232** 接口：用于 **EMW3166** 底层调试日志输出。
- **DC4.0** 接口：用于外部 **DC** 电源，注意电源的电压是 **5V**，电流要求提供 **1A** 或以上。
- 按键：共有 **6** 个，其中 **RESET** 是复位键，**ESC** 是系统键，**F1-F4** 是用户按键。
- 光照传感器：用于感应光线的强弱。
- 可调电位器：用于获得用户旋转的旋扭位置。
- 电机接口：接直流电机，可以设置电机转动方向和速度。

- 继电器接口：可做为控制灯具等开关性质的功能。
- 蜂鸣器：可以用一定的频率和设置，来控制蜂鸣器的声音响度和次数。
- **LED**：共有三个 **LED**，一个是系统用来显示当前状态，另外两个是用户使用，可以显示红色或蓝色。
- **SPI** 接口：用于外扩 **SPI** 模块，比如控制器等。
- **I2C** 接口：用于外扩 **I2C** 模块，比如三轴传感器等。
- **RTC** 电池：用来保持 **EMW3166** 模块的实时时钟在断开电源的时候能正常运行。

二、硬件实验例子

1）全世界通用的 HelloWorld

这个程序向控制台（调试日志窗口）输出一个“**HelloWorld"**的字符串。

不管什么类型的核心硬件，只要它支持 **CRAB** 虚拟机，那就肯定能成功执行并输出。

所以，这个最最最简单的程序，一般就是用来测试开发板的核心模块能否正常运行的手段之一。

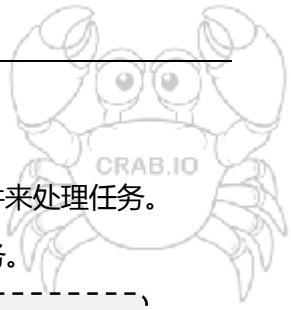
```
main
{
  PrintLn('Hello World');
}
```

2）LED 控制

CRAB 支持控制 **LED** 的三种形态：

- 熄灭（**LED_OFF**）：**LED** 不显示任何灯光，同样亦表示该端口处于非输出（高阻）状态。
- 低电平（**LED_BLUE**）：**LED** 显示蓝色，同样亦表示该端口处于低电平状态。
- 高电平（**LED_RED**）：**LED** 显示红色，同样亦表示该端口处于高电平状态。

```
main
{
  // 点亮左边的LED为蓝色
  Board.LED1 = LED_BLUE;

  // 点亮左边的LED为红色
  Board.LED1 = LED_RED;

  // 关闭LED灯
  Board.LED1 = LED_OFF;
}
```



3）按键读取和处理

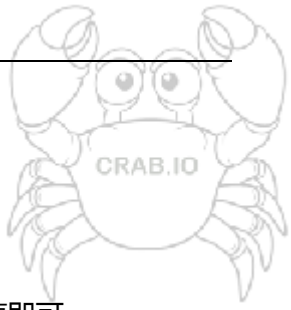
CRAB 支持两种按键读取方式，第一种是直接读取按键码，第二种是通过按键事件来处理任务。

A.直接读取按键码，这种方式适合循环读取按键，再通过按键判断来做相应的任务。

```
main
{
  ushort Key = 0;

  //不间断的重复
  repeat
  {
    //读取开发板按键码
    Key = Board.Key;

    //如果有按键发生，则按键码不为0
    if (Key != 0)
    {
      //显示该按键码
      PrintLn('Key Press: ' # Key);
    }
  }
  until error;

}
```

B.通过按键事件来处理任务，这是我们推荐的方式。通过事件，可以减少主程序的检查和判断的时间，而且还可以快速反应按键任务。

```
//当使用按下F1按键的时候，将会进入下面的事件函数
//事件函数：OnKey1          这个函数名是可选的。
//事件标志：KEY_PRESS_F1    每一种事件都有自己惟一的标志
event OnKey1 : KEY_PRESS_F1
{
  board.LED2 = LED_RED;
  Delay(500); //这里延时500毫秒，仅仅是为了看清LED是否已经点亮
  board.LED2 = LED_OFF;
}

main
{
  //打开事件驱动机制
  OpenEvent();

  //主程序需要循环，否则会因为主程序的结束而导致整个应用程序终结
  repeat
  {
  }
  until error;

  //关闭事件驱动机制
  CloseEvent();
}
```



3）光照传感器

光照传感器是一个感应光线强弱的传感器。根据光线强弱，其值在 **0~100** 之间。

光线越强，其值越低。反之光线越弱，其值越高。

在硬件驱动的底层，光照传感器已经自动激活，所以应用程序只需要读取相应的值即可。

```
main
{
  ushort Light;

  //读取光照传感器的值
  Light = Board.LIGHT;

  //显示当前光照传感器的值
  PrintLn("Board LIGHT: " # Light);
}
```

4）可调电位器

可调电位器用于获得用户旋转的旋扭位置，其值在 **0~100** 之间。

刻度指针的位置越向左，其值越小，反之越向右，其值越大。

在硬件驱动的底层，电位传感器已经自动激活，所以应用程序只需要读取相应的值即可。

```
main
{
  ushort Twist;

  //读取电位传感器的值
  Twist = Board.TWIST;

  //显示当前电位传感器的值
  PrintLn("Board TWIST: " # Twist);
}
```

5）蜂鸣器

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电。蜂鸣器又分为无源他激型与有源自激型。

本开发板采用的是无源蜂鸣器，它的工作发声原理是：方波信号输入谐振装置转换为声音信号输出。

无源蜂鸣器还有另一个作用，就是可以做简易音乐播放器。

```
main
{
  //蜂鸣器鸣响功能
  //声音频率为1400Hz，这个频率声音比较大
  //鸣响次数为3次
  Board.Beep(1400, 3);
}
```

6）继电器接口

继电器相当一个开关，当你打开开关的时候，继电器接口将会有电流输出。如果关闭开关，电流将会停止输出。它适合作为控制灯具等简单操作。

```
main
{
  //打开继电器开关
  Board.RETY = RETY_ON;

  //关闭继电器开关
  Board.RETY = RETY_OFF;
}
```

7）电机接口

电机接口是控制小型直流电机的，它可以启动或关闭电机，也可以设置电机脉冲频率（**0~100000**），转动方向（**0** 正向，**1** 反向）和转动速度（**0~100**）。注意电机的额定电压和电流。本接口仅支持 **5V/1A** 的直流电机，比如玩具车小马达，玩具小风扇等。

电机在启动的时候，需要先给出一个比较高的速度（**80** 或以上）。电机在换向的时候，同样也需要比较高的速度，和启动时一样。

```
//事件：当用户按下F2按键的时候
event OnKey2 : KEY_PRESS_F2
{
  if (!Motor1.Active) return;

  if (Motor1.Polar == 0)
  {
     //马达反转
    Motor1.Polar = 1;
 }
  else
  {
    //马达正转
    Motor1.Polar = 0;
  }
}

main
{
    //打开马达
    Motor1.Active = true;

    //马达的驱动频率为10000
    Motor1.Frequ = 10000;

    //马达的速度为80（最大100）
    Motor1.Speed = 80;
}
```

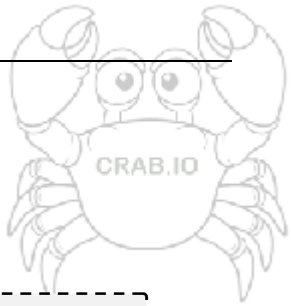
8）实时时钟

所有支持 **CRAB** 虚拟机的模块都支持实时时钟。如果开发板有安装 **RTC** 电池，则不管开发板有没有电源，实时时钟都将时刻在运行，直到电池没电。

```
main
{
  date CurDate;
  time CurTime;

  //读取当前日期
  CurDate = SystemCalendar;

  //读取当前时间
  CurTime = SystemClock;

  //打印当前日期和时间
  PrintLn("Current Date: " # CurDate);
  PrintLn("Current Time: " # CurTime);
}
```

实时时钟可以读取，也可以设置后回写。一般在重新安装电池之后，就需要重新设置时间。

```
main
{
  //设置当前日期
  SystemCalendar = $"2016-06-15";

  //设置当前时间
  SystemClock = $"20:30:00";

  //打印当前日期和时间
  PrintLn("Current Date: " # SystemCalendar);
  PrintLn("Current Time: " # SystemClock);
}
```



三、网络实验例子

1）扫描 **WIFI** 热点

此函数未能正确运行，待查证。

```
main
{
  //扫描WIFI热点
  Net.Scan();
}
```

2）连接到 **WIFI** 接入点

EMW3166 在使用网络功能之后，必须先连接到接入点，或者是自身成为接入点。本函数是让模块连接到外界路由器，作为终端设备。

函数原型：

```
boolean Connect(string SSID, string Password);
```

函数参数：

- **SSID**: 连接点名称
- **Password**: 连接密码

函数返回:

- **true -** 连接成功。
- **false -** 连接失败，请检查 **SSID** 和密码是否正确，或者路由器是否提供连接服务。

```
main
{
  boolean status;

  //连接到WIFI接入点
  //SSID: "WU-Family"
  //Password: "12345678"
  status = Net.Connect("WU-Family", "12345678");

  if (status)
  {
    //连接成功
    PrintLn("Connect Success.");
  }
  else
  {
    //连接失败
    PrintLn("Connect Faild.");
  }
}
```

3）建立 **WIFI** 接入点

EMW3166 在使用网络功能之后，必须先连接到接入点，或者是自身成为接入点。本函数是让模块成为接入点，相当为一个路由器。

函数原型：

```
boolean SoftAP(string SSID, string Password);
```

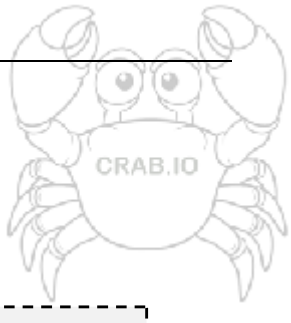
函数参数：

- **SSID**: 连接点名称
- **Password**: 连接密码

函数返回:

- **true -** 建立成功。
- **false -** 建立失败，当前的 **WIFI** 环境可能所有的频道都很繁忙，或者模块本身有故障。

```
main
{
  boolean status;

  //建立WIFI接入点
  //SSID: "WU-Family"
  //Password: "12345678"
  status = Net.SoftAP("WU-Family", "12345678");

  if (status)
  {
    //建立成功
    PrintLn("Connect Success.");
  }
  else
  {
    //建立失败
    PrintLn("Connect Faild.");
  }
}
```



4）取得网络信息

获取当前的网络连接信息或是热点信息。

函数原型：

```
string[] Info(boolean AddCaption);
```

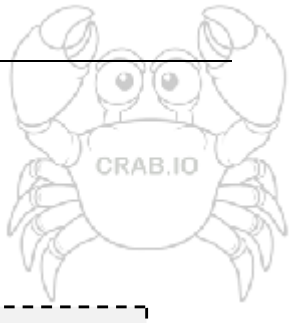
函数参数：

- **AddCaption**: 是否在每种信息内容前面加上标题。

函数返回:

- **string[]**：这是一个信息列表，需要独行读出。

```
main
{
  string[] Info;
  string   Line;

  //获取网络信息，每条信息都加上标题
  Info = Net.Info(true);

  //逐行打印信息
  foreach (Line in Info)
  {
    PrintLn(Line);
  }
}
```



5）获取连接状态

获取当前的网络连接状态。

函数原型：

```
boolean LinkStatus();
```

函数参数：

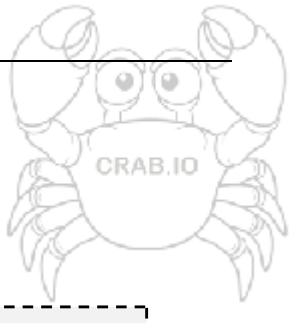
l 无。

函数返回:

true - 已经连接到 **WIFI**。

false - 未连接，或是连接失败。

```
main
{
  boolean Active;

  //获取连接状态
  Active = Net.LinkStatus();

  if (Active)
  {
    //连接成功
    PrintLn ("Connect Success.");
  }
  else
  {
    //连接失败，或未连接
    PrintLn("Connect Faild.");
  }
}
```



6）域名转为 **IP** 地址

根据域名获取相对应的 **IP** 地址。

函数原型：

```
string GetDns(string Domain);
```

函数参数：

- **Domain**: 需要转换的域名。

函数返回:

- **string**：如果正确执行，则返回 **IP** 地址，如果执行失败，而返回空字符串（**null** 值）。

```
main
{
  string IP;
  string Domain = "ntp.shu.edu.cn";

  //获取域名所对应的IP地下
  IP = Net.GetDns(Domain);

  //如果IP地址有效，则打印出来
  if (IP != null)
  {
    PrintLn(Domain # ' = ' # IP);
  }
}
```

7）获取本机 **IP** 地址

如果模块已经连接到热点或路由器，则模块会自动获得路由器所分配的相对应的 **IP** 地址。而本函数是将获取这个 **IP** 地址。

函数原型：

```
string LocalIP();
```

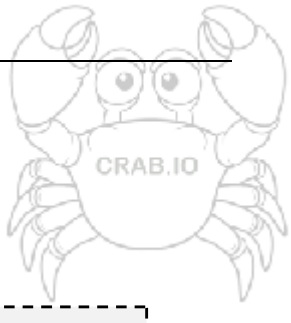
函数参数：

- 无。

函数返回:

- **string**：如果正确执行，则返回 **IP** 地址，如果执行失败，而返回空字符串（**null** 值）。

```
main
{
  string IP;

  //获取本机IP地址
  IP = Net.LocalIP();

  //如果IP地址有效，则打印出来
  if (IP != null)
  {
    PrintLn("Local IP: " # IP);
  }
}
```



8）获取时间服务器数据

根据时间服务器所提供的时间和日期。

函数原型：

```
datetime ServerDatetime(string Domain, int DateOffset, int TimeOffset);
```

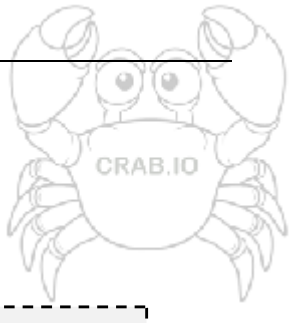
函数参数：

- **Domain**：时间服务器的域名。
- **DateOffset**：日期偏移值，一般为 **0.**
- **TimeOffset**：时间偏移值，这个跟时区有关，比如中国的时区是**+8**

函数返回:

- **datetime**：如果获取成功，则服务器将会返回一个完整的当前时间值。如果获取失败，则返回值是无效的 **0** 值（其时间表达式为**$"0000-00-00 00:00:00.000"**）。

```
main
{
  datetime Now;

  //获取时间服务器的当前时间
  //时间服务器: "ntp.shu.edu.cn"
  //日期偏移: 0
  //时间偏移: 8
  Now = Net.Util.ServerDatetime("ntp.shu.edu.cn", 0 ,8);

  //打印时间，前面是日期，后面是时间
  PrintLn("Server Datetime: " # Now.Date # " " # Now.Time);
}
```



9）建立 **UDP** 单播

建立 **UDP** 单播端，并打开相应的端口。

函数原型：

```
boolean Open(uint LocalPort);
```

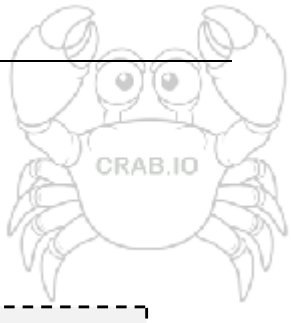
函数参数：

- **LocalPort**：单播端口。

函数返回:

- **true**：建立成功。
- **false**：建立失败，可能是未连接到网络，或者是该端口已经存在。

```
main
{
  //UDP端口
  const Unicast_port = 20000;
  boolean Status;

  //建立UDP单播
  //UDP端口: 20000
  Status = Net.Udp.Open(Unicast_port);

  if (Status)
  {
    //建立成功
    PrintLn("Open Udp@ " # Unicast_port # " success.");
  }
  else
  {
    //建立失败
    PrintLn("Open Udp@ " # Unicast_port # " faild.");
  }
}
```



10）关闭 UDP 单播

关闭 **UDP** 单播端，并关闭相应的端口。

函数原型：

```
boolean Close();
```

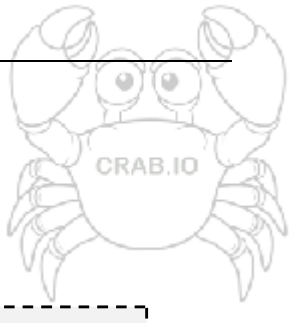
函数参数：

- 无。

函数返回:

- **true**：关闭成功。
- **false**：关闭失败，可能 **UDP** 单播未曾打开，或是未连接到网络。

```
main
{
  boolean Status;

  //关闭UDP单播
  Status = Net.Udp.Close();

  if (Status)
  {
    //关闭成功
    PrintLn("Close Udp success.");
  }
  else
  {
    //关闭失败
    PrintLn("Close Udp faild.");
  }
}
```



11）获取 UDP 单播状态

获取 **UDP** 单播状态。

函数原型：

```
boolean Status();
```

函数参数：

- 无。

函数返回:

- **true**：已经建立 **UDP** 单播。
- **false**：未建立 **UDP** 单播，或是未连接到网络。

```
main
{
  boolean Status;

  //获取UDP单播状态
  Status = Net.Udp.Status();

  if (Status)
  {
    //关闭成功
    PrintLn("Udp Active.");
  }
  else
  {
    //关闭失败
    PrintLn("Udp Deactive.");
  }
}
```

11）获取 **UDP** 单播所接收的字符串

如果 **UDP** 的其它设备端发送信息到本设备，本设备将会将所接收的字符串信息保存在缓冲区，而本函数，就是获取缓冲区里的字符串，并清空缓冲区。

函数原型：

```
string Receive();
```

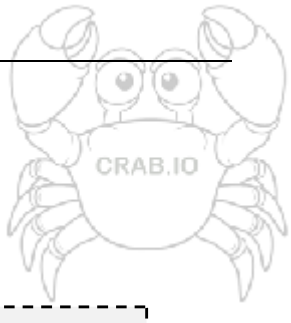
函数参数：

- 无。

函数返回:

- **string**：如果正确执行，则返回当前已接收的字符串，如果执行失败，而返回空字符串（**null** 值）。

```
main
{
  string Text;

  //获取UDP单播所接收的字符串
  Text = Net.Udp.Receive();

  //如果IP地址有效，则打印出来
  if (Text != null)
  {
    PrintLn("Receive Text: " # Text);
  }
}
```



12）发送字符串到 **UDP** 单播

获取 **UDP** 单播所接收的字符串。

函数原型：

```
boolean Send(string Text);
```

函数参数：

- **Text**：待发送的字符串信息。

函数返回:

- **true**：发送成功。
- **false**：发送失败，可能是未建立 **UDP** 单播，或是未连接到网络。

```
main
{
  boolean Status;

  //发送字符串到UDP单播
  Status = Net.Udp.Send("Hello");

  if (Status)
  {
    //发送成功
    PrintLn("Send Success.");
  }
  else
  {
    //发送失败
    PrintLn("Send Faild.");
  }
}
```

13）获取 **UDP** 单播远端 **IP**

当 **UDP** 单播的其它设备连入 **UDP** 单播，并发送字符串信息给本设备的时候，模块会自动获得对方相对应的 **IP** 地址。而本函数是将获取这个 **IP** 地址。

函数原型：

```
string RemoteIP();
```

函数参数：

l 无。

函数返回:

l **string**：如果正确执行，则返回 **IP** 地址，如果执行失败，而返回空字符串（**null** 值）。

```
main
{
  string IP;

  //获取远端设备IP地址
  IP = Net.Udp.RemoteIP();

  //如果IP地址有效，则打印出来
  if (IP != null)
  {
    PrintLn("Remote IP: " # IP);
  }
}
```

13）获取 **UDP** 单播远端端口

当 **UDP** 单播的其它设备连入 **UDP** 单播，并发送字符串信息给本设备的时候，模块会自动获得对方相对应的端口。而本函数是将获取这个端口。

函数原型：

```
uint RemotePort();
```

函数参数：

- 无。

函数返回:

- **uint**：如果正确执行，则返回端口，如果执行失败，而返回 **0**。

```
main
{
  uint Port;

  //获取UDP单播远端Port
  Port = Net.Udp.RemotePort();

  if (Port > 0)
  {
    //获取成功
    PrintLn("Port: " # Port);
  }
  else
  {
    //获取失败
    PrintLn("Get Port Faild.");
  }
}
```