EE1301：计算系统简介

物联网实验室4

光子上的简单多任务处理

作者：Kia Bazargan、David Orser和John Sartori



感谢学生、助教和教职员工不断改进本文件。我们一起做更好的实验！

请将意见和建议发送至orser@umn.edu

版权所有2018

# 背景

到目前为止，您为光子板编写的程序都是一次执行一个任务。例如，一个典型的程序可能在一个循环中执行以下任务：从传感器读取输入值，然后执行一些计算，然后将输出值写入执行器。当所有的任务都需要以相同的速度执行时，这个简单的顺序流就可以了。但是，要实现更复杂的项目，可能需要以不同的速率执行多个任务，在开始下一个任务之前等待一个任务完成可能是不可接受的。例如，如果希望程序每秒闪烁一次LED并检查用户是否按下按钮，则不应等到按下该按钮后再继续闪烁LED；否则LED可能不会以每秒一次的速率闪烁。同样，您不应该等到LED闪烁时才检查按钮是否已按下；否则，用户可能在等待LED闪烁的1秒延迟期间按下按钮，并且可能会错过按钮按下。在这个实验中，我们将学习当任务可能以异步（即随机时间）或不同速率发生时，如何同时处理多个任务。

# 目的

在这个实验室中，您将学习一些关于Photon固件如何处理执行用户代码（在setup（）和loop（）函数中）和其他功能（如wifi连接、到终端的串行通信和与云的通信）的基本知识。您将学习使用状态变量来记住程序的执行状态。最后，您将学习使用计时器定期触发事件。

# 实验室前要求

这个实验室的新材料数量大大少于实验室2和3。然而，这些概念的应用更为复杂。你应该仔细阅读这个实验室，并考虑如何在你的最终项目中使用这些概念。

|  |
| --- |
| 实验前检查表  阅读millis（）函数的粒子文档  阅读关于使用输入、输入下拉和输入下拉的颗粒文件 |

## 所需组件：

|  |
| --- |
| 按钮开关  去耦电容器 |

# 光子背景过程

光子板上的处理器带有一个预先编程的“固件”，它是负责通过USB端口、wifi连接和云连接进行串行通信的底层软件。如前面的实验所述，固件在调用setup（）函数之后

它首先连接到wifi网络。然后，固件每隔1毫秒重复调用loop（）函数，每次迭代loop（）函数后，固件执行后台任务以保持wifi连接，并与云端检查是否有设备的消息（例如，是否有新的用户软件可用于闪到设备上，或是否有云事件设备订阅的已发布）。如果的迭代，设备将很长一段时间无法执行这些后台任务，并且可能会发生不好的事情，例如失去wifi或云连接。你将在下面的实验中看到这一点。**loop（）函数花费的时间远远超过1毫秒**

接通下面的电路，并将随后的代码闪到光子上。代码应该是采样的按钮和闪烁的车载LED在一个固定的速率。Pull-down Resistors
The keyword INPUT_PULLDOWN asks Photon to use its internal pull-down resistors. Using a pull-down resistor will weakly connect the pin to GND (0V). This will result in a value of LOW whenever the button is not pressed. Since the other side of the button is connected to 3.3V, the button will read HIGH when it is pressed.


**注意**我们将按钮输入定义为一个下拉式输入端，而不是一个简单的输入端。你的助教会解释为什么你需要一个下拉电阻（你学习这个很重要！）.

|  |  |
| --- | --- |
| Lab4_Button_bb.png | Lab4_Button_schem.png |

|  |
| --- |
| **//此程序将每秒钟打开和关闭车载D7 LED。**  **//按住这个按钮，你就可以暂时停止闪烁，并显示一个恒定的蓝色**  **//点亮D7 LED。但是，代码的编写方式不正确。它停滞不前**  **//当按下按钮时的loop（）函数，如果按下按钮**  **//时间太长，会导致超时。**  **//试试看：按住按钮大约20秒，你就会看到**  **//董事会失去云连接，开始呼吸绿色，而不是呼吸青色。**    **//首先，定义我们要使用的管脚。**  **int led=D7；//D7车载led**  **int button=D3；//输入按钮（我们将用内部**  **//下拉电阻器）**    **//当设备启动或重置时，设置功能仅运行一次**  **无效设置（）{**  **//将D7引脚设置为输出（车载LED），并**  **//D3作为按钮输入，下拉**  **pinMode（led，输出）；**  **pinMode（button，INPUT\_PULLDOWN）；//我们使用INPUT\_PULLDOWN代替INPUT**  **//启动内部下拉电阻器。当按钮没有**  **//按下后，按钮销通过下拉连接到低（GND）**  **//电阻器。当按下按钮时，按钮销连接到**  **//高（3.3伏）。**  **}**    **//在设备之后尽可能频繁地重复调用循环函数**  **//穿上靴子。固件交错与WiFi+云相关的后台任务**  **//通过在每个循环之后执行任务，在循环函数中使用代码的活动**  **//函数迭代。**  **//注意：阻塞（暂停）太长时间（如超过5秒）的代码可能**  **//不好的事情会发生（比如失去云连接）。因此，while循环**  **//下面的代码可能会导致故障，因为它会阻止循环函数完成**  **//任意长的时间，导致设备失去其云连接。**  **//**  **//注2：**  **//内置的延迟函数可以安全使用，即使是任意长的延迟。**  **//它的实现方式（通过粒子）是安全地交织所需的**  **//执行循环函数的后台活动。**  **无效循环（）{**  **而（digitalRead（button）==HIGH）{/，只要按下按钮，**  **//在这里等着什么也不做。这称为阻塞循环，并且**  **//总的来说不是个好主意。**  **}**    **digitalWrite（led，LOW）；//关闭led**  **延迟（1000）；//等待1000mS=1秒**  **digitalWrite（led，HIGH）；//打开led**  **延迟（1000）；//等待1000mS=1秒**  **}**    **//如果你想等待按钮被释放而不导致光子丢失**  **//云连接，将上面的while循环替换为下面的循环。**  **//**  **//while（digitalRead（button）==HIGH）{/，只要按下按钮**  **///下来，在这里等着什么也不做。**  **//Particle.process（）；///允许固件执行一些后台任务**  **////例如，在**  **////loop正在等待按钮被释放**  **// }** |

当您尝试在光子上运行代码时，请注意以下行为。如果不按这个按钮，代码将反复切换LED——打开一秒钟，关闭一秒钟。但是，如果按住按钮几秒钟，闪烁将停止，蓝色LED将保持点亮。

按住按钮几秒钟，然后松开。只要按住按钮，就会看到闪烁停止。接下来，试着按住按钮20秒以上。您将看到，该板将停止呼吸青色（这表示云连接），并将开始呼吸绿色（仅wifi连接）。原因是您已在while循环中暂停处理器（while（digitalRead（button）==HIGH）{}），它无法完成循环的此迭代并签入云。（记住，签入cloud是在循环函数的迭代完成后进行的。）使用的while循环称为阻塞循环。一般来说，阻塞循环不是一个好主意，除非您确定您的特定应用程序可以无限期地停止。

现在，根据loop（）函数下面蓝色注释中的建议更改代码，然后重复实验。您将看到，无论您按住按钮多久，光子板都不会失去云连接（即，它继续呼吸青色）。原因是该功能允许光子的固件进行后台处理，如wifi令牌更新和在while循环执行时进行云消息检查。**粒子处理（）**

尽管代码做了它应该做的事情（即只要用户按住按钮就停止闪烁），但它的结构并不适合所有需要执行多任务的应用程序。我们将在下一节中查看此类应用程序的示例。

# 基于先前按钮状态检测按钮按下（来自物联网实验室2）

前面的示例使用while循环来检测用户何时按下了按钮。如上所述，在某些不需要执行其他操作的应用程序中，这可能是可以接受的，但是如果希望在等待按钮被按下或释放时能够执行其他任务，则应使用避免使用while循环轮询按钮状态的方法。通过使用if语句而不是while循环检查按钮的状态，我们可以避免使用while循环。

如果你还记得的话，我们在物联网实验室2：“传感器-人的输入设备”。您可能还记得，如果我们只是在一个特定的循环函数迭代中检查按钮的值，那么每次按下按钮时都会有多个按钮按下，因为循环函数在释放按钮之前执行多次。我们的解决方案是比较两个连续循环迭代中按钮的值。我们添加了一个状态变量来记录上一个循环迭代中按钮的状态，并比较当前和上一个按钮的值，以检查自上一个循环迭代（只是在两个调用之间按下）以来按钮是否从低变为高。此状态变量需要初始化为全局变量或静态局部变量，以便其值从一个循环迭代持续到下一个循环迭代。我们在下面的代码中选择了全局变量选项，因为它更容易理解，但是使用静态变量是首选方法。有关静态局部变量的更多信息，请参阅附录。

下面是我们将如何在下一个版本的代码中实现按钮检查的示例。通读代码并确保您理解它——特别是使用状态变量来记录按钮的前一个状态的概念。如果您有任何问题，请在这里停下来，与邻居或助教讨论后再继续。

|  |
| --- |
| **//初始化代码——prevButton声明为全局变量**  **int prevButton=LOW；//button为高或低时要存储的状态变量**  **//上次我们检查的时候**    **//内部循环（）代码。。。**  **int curButton=数字读取（按钮）；**  **if（抑制按钮==高&&prevButton==低）{**  **//发生了转换——用户刚刚按下了按钮**  **//响应此代码块中的按钮按下**  **}**  **prevButton=curButton；//记录下一轮按钮的状态** |

# 将闪烁速率存储在状态变量中，并使用按键更改闪烁速率

现在，我们将使用改进的按键代码来更改LED的闪烁速率。闪烁速率应该在两个不同的速率之间循环（我们称之为速率0和速率1），每次按下按钮时都会发生变化。当LED以0的速率闪烁并按下按钮时，LED应以1的速率开始闪烁，反之亦然。要决定按下按钮时如何设置闪烁速率，可以使用另一个状态变量来跟踪LED的当前闪烁速率。根据状态变量的值，您将知道如何在按下按钮时设置新的闪烁速率。

下面的程序将使车载LED（D7）闪烁。通过按下按钮（D3），您可以在0.25赫兹（2秒打开，2秒关闭）和0.5赫兹（1秒打开，1秒关闭）之间切换LED闪烁的速率。看看下面的代码。

|  |
| --- |
| **//此程序以不同速率闪烁车载LED（D7**  **//当按下按钮时，闪烁的速率改变。**  **//使用状态变量存储上一个按钮值和当前闪烁率**    **//定义要使用的管脚**  **int led=D7；//D7车载led**  **int button=D3；//这是输入按钮（使用内部下拉电阻器）**    **int ledferqstate=0；//0.50赫兹设为0，0.25赫兹设为1**  **int prevButton=LOW；//state变量，用于存储上次button是高还是低**    **无效设置（）{**  **pinMode（led，输出）；**  **pinMode（button，INPUT\_pull down）；//带内置下拉电阻器的输入模式**  **Serial.begin（9600）；//使用串行端口进行调试**  **}**    **无效循环（）{**  **int curButton=数字读取（按钮）；**  **if（抑制按钮==高&&prevButton==低）{**  **//发生了转换——用户刚刚按下了按钮**  **Serial.println（“检测到按键。”）；//调试打印输出**    **//更改LED闪烁率的状态**  **ledferqstate++；**  **如果（ledferqstate>1）{//请确保它从0-->1-->0-->更改。。。**  **LedFreqState=0；**  **}**  **}**    **prevButton=curButton；//下次跟踪上一个按钮状态**  **Serial.print（curButton）；//调试打印输出**    **if（ledferqstate==0）{//Rate 0:0.5hz（开1秒，关1秒）**  **数字写入（led，低）；**  **延迟（1000）；**  **数字写入（led，高）；**  **延迟（1000）；**  **}否则{//速率1:0.25Hz（2秒打开，2秒关闭）**  **数字写入（led，低）；**  **延迟（2000年）；**  **数字写入（led，高）；**  **延迟（2000年）；**  **}**  **}** |

在上面的代码中，存储闪烁率的新状态变量是。当检测到按钮按下时，此状态变量将更新（）。还检查该状态变量，以确定在关闭和打开LED之间是否延迟1秒或2秒。**LedFreqStateLedFreqState++…如果（LedFreqState==0）{…}其他{…}**

把上面的程序闪到你的光子上试试！当你测试程序时，看看你是否能观察到任何异常行为。首先，按住按钮一段时间，您应该观察到闪烁速率按预期变化。现在，尝试在LED熄灭时快速按下并松开按钮（当频率为0.25Hz时更容易做到这一点）。你将看到程序没有检测到你的按钮按下。这是因为光子正忙于执行使LED闪烁的if/else语句的一个代码块。由于较短的代码块至少需要2秒，较长的代码块至少需要4秒，如果您不按住按钮几秒钟，则可能会错过按钮按下。

LED闪烁是使用阻塞码处理的，这使得在切换LED时无法检测到按钮按下。如前所述，当您希望异步或以不同的速率执行多个任务（如检查按钮按下和LED闪烁）时，阻塞代码不是一个好的解决方案。我们将在后面的“使用计时器”一节中的代码中解决此缺陷，但首先，通过添加额外的闪烁率，为当前代码添加一些功能，如下所述。

# 练习1——更多LED闪烁率

在继续之前，向助教演示你可以在上面的程序中添加一个状态。更改LED闪烁程序，使您可以循环通过三种状态：1Hz、0.5Hz和0.25Hz。和以前一样，当按下按钮时，程序应该改变状态。

# 使用计时器知道何时在不使用delay（）函数的情况下切换LED

如前所述，我们试图在代码中修复的问题是，当LED被切换时，程序无法检测到按钮按下。这是因为函数阻止光子继续执行loop（）函数，直到它完成[1]。它类似于while循环，在经过指定的延迟量之前，它一直在迭代，什么也不做。与前面一样，为了使代码不被阻塞，我们将使用if语句来检查何时切换LED，而不是使用blocking delay（）函数。if语句将检查经过的时间是否正确。如果有，我们将切换LED；如果没有，我们将继续执行其余代码。**延迟（）**

我们用来实现这一点的编程结构称为计时器。定时器是一种特殊的硬件，它从你的光子打开的那一刻起就一直在计数。您可以使用函数millis（）访问计数器的当前值，该函数返回设备打开后经过的时间（以毫秒为单位）。以下是指向millis（）上光子帮助页的链接：

https://docs.particle.io/reference/firmware/photon/#millis（毫秒）-

我们将使用一个变量来确定何时切换LED。每次切换LED时，我们将通过读取计时器的当前值并添加LED下次应切换的时间量来计算和存储下次切换的时间。然后，每次通过循环函数，我们都可以检查当前计时器值是否大于告诉我们下次切换的存储变量。

例如，以下程序将以0.5Hz的速率闪烁车载LED（D7）。它通过检查当前时间并将其与保存的状态变量（存储下一个要切换的时间）进行比较来完成此操作。如果当前时间大于状态变量，将切换LED，并计算下一次切换时间并保存在状态变量中。下面的程序是非阻塞的，因为它无法阻止循环函数的迭代在很长一段时间内完成。

|  |
| --- |
| **//此程序将使车载D7 LED以0.5 Hz的频率闪烁。**  **//它不使用delay（）函数**  **//注意，有两个状态变量——一个用于存储下一次切换的状态变量**  **//LED和一个用于存储当前LED状态（开/关）**    **int led=D7；//D7车载led**    **//将状态变量声明为全局变量。**  **//您还可以在循环内将状态变量声明为局部静态变量**  **//功能（首选）。**    **bool LED\_state=错误；**    **//此变量存储下次切换LED的时间。**  **//在循环函数的第一次迭代中，它将被初始化为1秒**  **无符号long int timetologgled=0；**    **无效设置（）{**  **pinMode（led，输出）；**  **数字写入（led，低）；**  **}**    **无效循环（）{**  **unsigned long int current time=millis（）；//获取当前时间**    **//检查是否是时候切换LED“我们到了吗？我们到了吗？..."**  **if（当前时间>TimeToToLoggled）{**  **//是时候切换了！**  **LED状态=！LED状态；**  **数字写入（led，led状态）；**    **//计算并存储下次切换LED的时间**  **TimeToToLoggled+=1000；**  **}**    **//剩下的光子代码在这里**  **}** |

该语句将下次切换LED的时间设置为1000毫秒（1秒）。把上面的代码闪到你的光子上并测试出来。**TimeToToLoggled+=1000**

|  |
| --- |
| **未签名长直觉长直觉**：在这些示例中，用于存储设备重置后的时间量（以毫秒为单位）。在光子上，有32位。关键字表示变量只存储正值，不能存储负值。因此，未签名的32位变量可以容纳的最大值是232 - 1＝4294967295。这意味着计数器在4294967295毫秒（大约49.7天）后将再次翻转到0。在本课程的大部分内容中，您可以安全地忽略翻滚。但是，如果您的项目使用计时器，并且每次运行时间超过49天，请与讲师或助教讨论如何处理滚动。 |

# 练习——使用非阻塞代码检查按钮按下和更改闪烁速率

现在，根据你在上面的例子中所学到的，创建一个非阻塞程序来检测按钮按下并改变LED的闪烁速率。您的程序需要满足以下条件：

●无阻塞码

●至少有3种不同的状态，每种状态具有不同的闪烁速率或模式

●无延迟（）函数调用

如果您愿意，可以在本练习中尝试使用RGB led或添加其他传感器/执行器。

|  |
| --- |
| 暗示-使用预处理器指令可以非常方便地启用/禁用调试代码，如下例所示：  **D:\document\convert_tasks\transweb\2665852_2678521\2665852.docx.files\image005.gif**  ifdef和endif行中的所有代码仅在定义了调试时编译为可执行文件。要启用调试代码，请使用语句define debug定义调试。要禁用调试代码，只需注释表示define debug的行。然后，将不定义调试，并且不执行串行打印语句。 |

# 实验室报告

对这个实验室的工作做一个非常简短的报告。它应该包括以下内容：

●描述在光子设备上使用阻塞码的一些潜在问题。

描述如何使用状态变量创建非阻塞代码。

●实验室助教必须观看电路演示。你离开实验室前一定要让他检查你的工作。

# 附录一：静态变量

正如您在讲座中所学到的，函数中的局部变量只有在函数的代码块结束时才可见，即变量的作用域是函数的局部。另一方面，全局变量可以被所有函数看到，也就是说，顾名思义，它的作用域是全局的。请看下面的代码示例（它是为通用平台编写的，不一定是为光子板编写的）。

|  |
| --- |
| **积分gVar=1；**    **//爱丽丝在X公司写了这个函数。**  **无效f1（）**  **{**  **积分lVar=0；**  **gVar++；**  **lVar++；**  **回报；**  **}**    **//鲍勃在X公司写了这个函数。**  **无效的f2（）**  **{**  **gVar=0；//假设这是一个错误。鲍勃不是有意重置全球价值链的。**  **回报；**  **}**    **int主（）{**  **gVar=5；**  **f1（）；//1次调用**  **f1（）；//2次调用**  **二层（F）；**  **返回0；**  **}** |

当程序启动时，全局变量gVar初始化为1，然后main（）开始执行。main（）的第一行将gVar的值更改为5。从这一点开始，每个函数，包括f1（）和f2（）都将看到gVar的值为5。

main（）第一次调用f1（）时，函数f1（）创建一个局部变量lVar并将其初始化为零。gVar增加到6，lVar增加到1。记住这些价值观。函数f1（）在返回main之前放弃对包含lVar的内存位置的控制（将其视为破坏lVar）。第二次main（）调用f1（）时，函数f1（）首先再次创建局部变量lVar，并使用值0初始化它。注意，上一次调用的lVar的上一个副本丢失，这将是一个新的初始化变量。然而，gVar++将增加gVar的前一个值，这将导致gVar被赋值为7。局部变量lVar在被销毁之前递增为1。请注意，gVar保留了上次调用f1（）时的值。

当f1（）返回时，main将看到gVar的值为7。main（）中的下一行调用f2（）。假设编写f2（）的Bob不小心，重置gVar时出错。编写f1（）的爱丽丝无法控制其他人弄乱gVar的值，这可能对f1（）的正确操作至关重要。

总而言之，全局变量的优势在于它们“记住”了以前调用函数时发生的事情，但其主要缺点是它们不受其他函数访问的保护。由于编码通常是一个协作过程，使用全局变量可能会有问题。

为了解决这个问题，C语言引入了静态变量。静态变量是函数在作用域上的局部变量，这意味着没有其他函数可以访问该变量。但是，静态变量“记住”它们以前调用函数时的值，这类似于全局变量的行为方式。以下示例说明了这一点：

|  |
| --- |
| **无效f1（）**  **{**  **静态积分sVar=3；**  **sVar++；**  **回报；**  **}**    **int主（）{**  **f1（）；//1次调用**  **f1（）；//2次调用**  **返回0；**  **}** |

当main（）第一次调用f1（）时，sVar被创建为一个int变量并初始化为3。从这一点开始，以后每当调用f1（）时，将跳过sVar的初始化。在第一次调用中，在初始化之后，sVar在返回main（）之前递增为4。与常规局部变量不同，sVar不会被破坏，并将保留其值。

main（）中的下一行再次调用f1（）。在第二个调用中，sVar未初始化为3。相反，它保留了4的值。下一行f1（）将sVar增加到5。

[[一]](" \l "_ftnref1" \o ")如前所述，delay（）函数允许执行固件中的后台任务，但不允许执行用户代码，例如检查按钮的状态并将其与按钮的前一个状态进行比较。