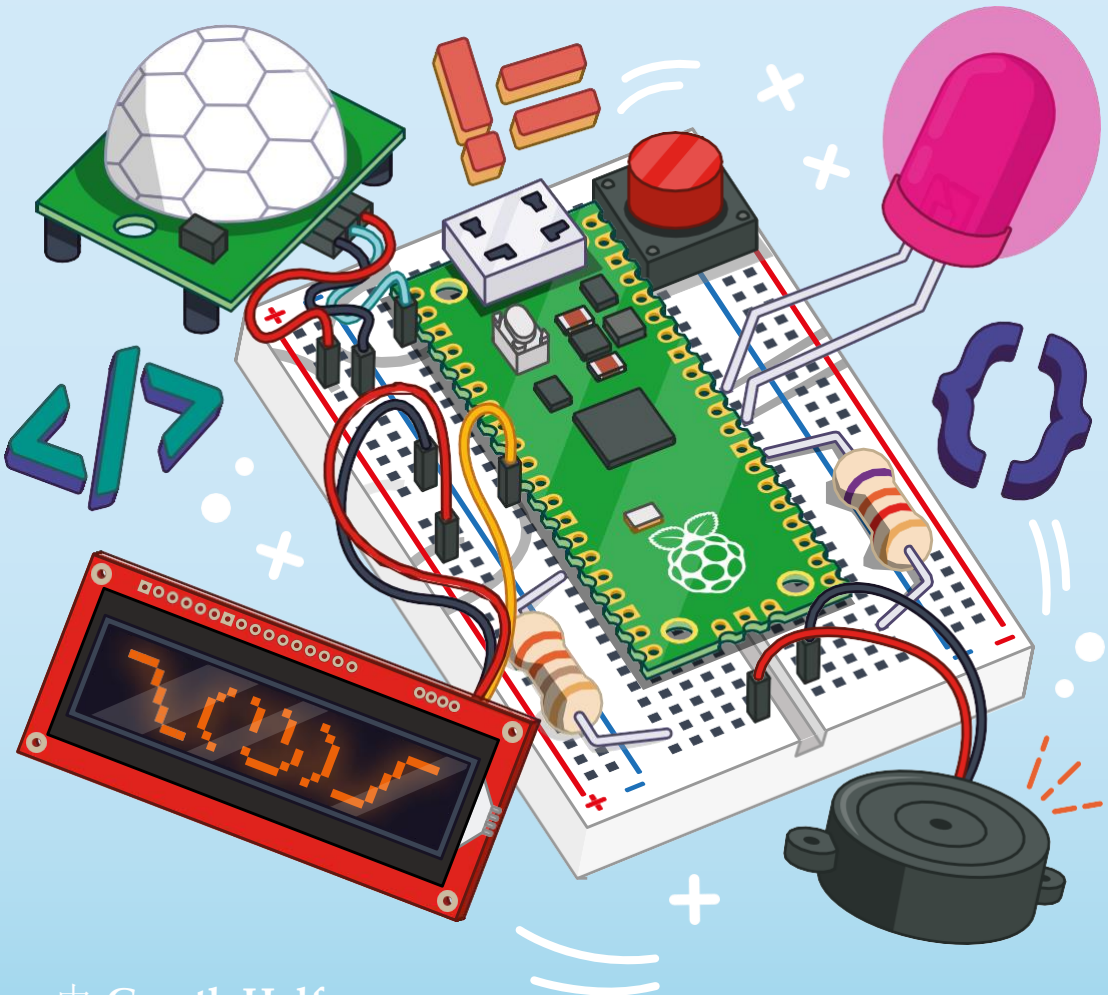
官方RASPBERRY PI PICO指南

开始 MicroPython 在Raspberry Pi Pico上



由 Gareth Halfacree
和 Ben Everard编著

由造控科技译著

Raspberry Pi PRESS

开始
MicroPython
在Raspberry Pi Pico上

首次出版于2021年 Raspberry Pi Trading Ltd, Maurice Wilkes Building, St. John's Innovation Park, Cowley Road, Cambridge, CB4 0DS

出版总监：Russell Barnes • Editor: Phil King • Sub Editor: Nicola King Design: Critical Media • Illustrations: Sam Alder
CEO: Eben Upton

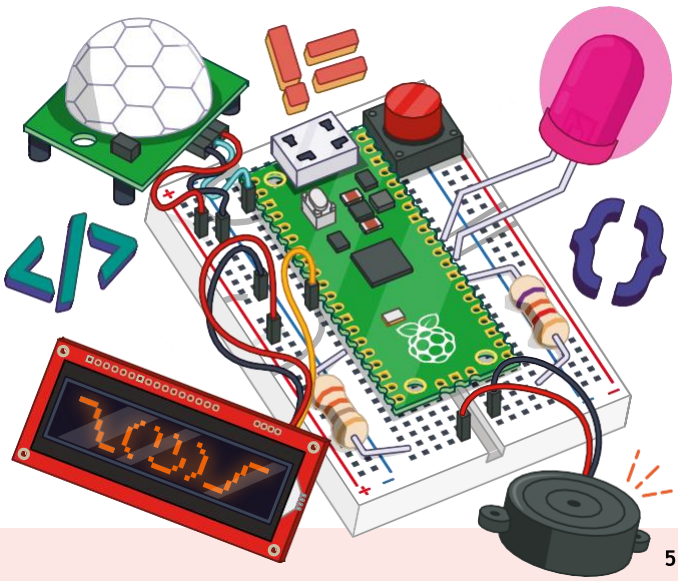
ISBN: 978-1-912047-86-4

欢迎你

可能会认为计算机是您粘在桌上并打字的东西，这肯定是一类计算机，但这不是唯一的类型。 在本书中，我们正在研究微控制器-带有少量存储器的小型处理单元，擅长控制其他硬件。您的房屋中可能已经有很多微控制器。您的洗衣机很有可能由微控制器控制； 也许你的手表是；您可能会在咖啡机或微波炉中找到一个。当然，所有这些微控制器都已经有其程序，制造商很难更改在其上运行的软件。

另一方面，可以通过USB连接轻松地对Raspberry Pi Pico进行重新编程。在本书中，我们将介绍如何开始使用该硬件以及如何与其他电子组件一起使用。到本书结尾，您将知道如何创建自己的可编程电子设备。 您对他们的处理取决于您自己。

Ben Everard



关于作者

Gareth Halfacree 是一名自由科技杂志记者、作家和前教育部门系统管理员。由于对开源软件和硬件充满热情，他也是树莓派平台的早期采用者，并撰写了几篇关于其功能和灵活性的文章。可以在他Twitter上**@ghalfacree**或通过他的网站**freelance.halfacree.co.uk**上找到。

Ben Everard是一个创客，他偶然涉足了可以让他玩新硬件的职业。作为HackSpace杂志（**hsmag.cc**）的编辑，他花费的时间比他尝试使用最新（而不是最新）的DIY技术的实际时间要多。他与妻子和两个女儿住在布里斯托尔的一所房子里，那里正慢慢装满电子设备和3D打印机。

内容

第 1 章：了解你的 Raspberry Pi Pico 008

完全熟悉你强大的新微控制器，并学习如何连接引脚排针并安装 MicroPython 来编程

第 2 章：用 MicroPython 编程 020

连接计算机并开始使用MicroPython语言为你的Raspberry Pi Pico编写程序

第 3 章 ：物理计算 034

了解你的Raspberry Pi Pico上的引脚与你可以连接和控制的电子器件

第 4 章：使用Raspberry Pi Pico 进行物理计算 044

开始将基本电子元件连接到你的Raspberry Pi Pico，并编写程序来控制和感知它们

第 5 章：交通灯控制器 058

使用多个 LED 和按钮创建自己的迷你行人过路系统

第 6 章： 反应游戏 068

使用 LED 和按钮为一个或两个玩家构建一个简单的反应定时游戏

第 7 章： 防盗警报 080

使用运动传感器检测入侵者，并用闪烁的灯光和警笛发出警报

第 8 章： 温度计 092

使用你Raspberry Pi Pico 里内置的模数转换器来转换模拟输入信号，并读取其内部的温度传感器

第 9 章： 数据记录器 104

将 Raspberry Pi Pico 变成一个温度数据记录设备，然后把它和电脑分开，使它完全便携

第 10 章：数字通信协议：I2C 和 SPI 116

探索这两个流行的通信协议，并使用它们在 LCD 上显示数据

附录

附录 A： Raspberry Pi Pico 规格 124

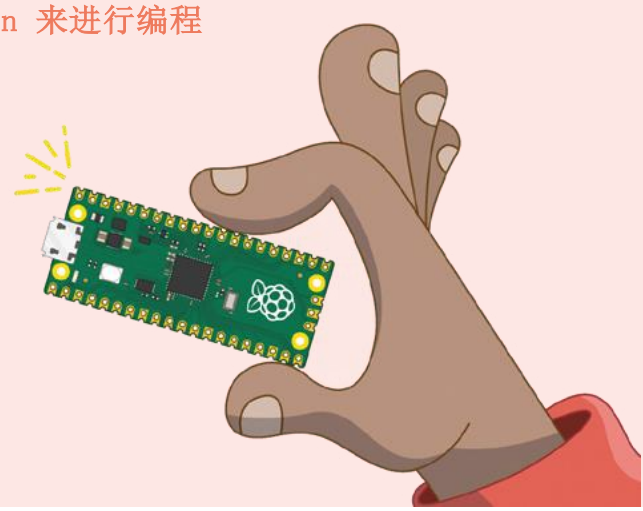
附录 B： 引脚分配 指南 128

附录 C： 可编程 IO 130

第一章

了解你的 Raspberry Pi Pico

充分了解你强大的新微控制器板
学习如何连接引脚排针
安装 MicroPython 来进行编程



Raspberry Pi Pico 是一个小奇迹，将构成智能家居系统到工业工厂的任何事物同样的技术放在你的手掌中。

无论你是希望了解 MicroPython 这一编程语言，在物理计算方面迈出第一步，还是想要构建硬件项目，Raspberry Pi Pico 及其神器的社区都将支持你在这条路上的每一步。

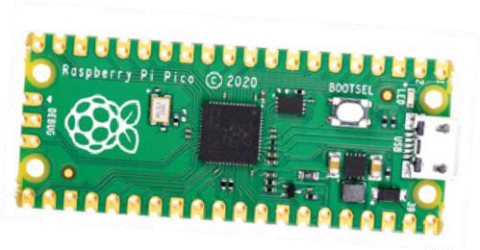
Raspberry Pi Pico被称为微控制器开发板，简单来说，它是一个印刷电路板，其中装有专为物理计算而设计的一种特殊类型的处理器：微控制器。 Raspberry Pi Pico像口香糖一样大小，它的功率惊人，这要归功于板子中央的RP2040微控制器芯片。

Raspberry Pi Pico并非旨在取代Raspberry Pi，后者是另一类称为单板计算机的设备。 尽管您可能会使用Raspberry Pi来玩游戏，编写故事和浏览网络，但Raspberry Pi Pico却是为物理计算项目而设计的，它可以控制从LED和按钮到传感器，电机甚至其他微控制器的任何东西。

在整本书中，您将学习有关Raspberry Pi Pico的全部知识，但是您所学到的技能也将应用于基于RP2040微控制器的任何其他开发板，甚至其他设备，只要它们与MicroPython编程语言兼容即可。

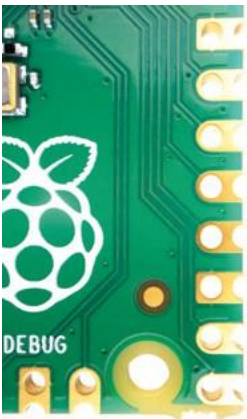
Raspberry Pi Pico导览

Raspberry Pi Pico（简称“ Pico”）甚至比Raspberry Pi Zero（Raspberry Pi单板计算机家族中体积最小）更小得多。 尽管如此，它仍包含许多功能-只需使用板边缘的引脚即可访问所有功能。



▲图 1-1： 板的顶部

图1-1显示了从上方看的Raspberry Pi Pico。如果您查看较长的边缘，看到金色的部分，看起来有点像小太空人。这些引脚为RP2040微控制器提供了与外界的连接-称为输入/输出（IO）。



▲图1-2： 种姓

Pico上的引脚与Raspberry Pi上的通用输入/输出（GPIO）接头连接器的引脚非常相似-但是，尽管大多数Raspberry Pi单板计算机都已连接了物理金属引脚，但是Pico 没有。 这样做有一个很好的理由：查看电路板的外边缘，会发现是不平整的，带有小的圆形切口（图1-2）。

这些圆形的小切口形成了所为的”城堡形电路板”，该电路板可以通过焊接连接到其他电路板的顶部，而无需安装任何物理金属销-这有助于降低高度，从而减小了成品尺寸。如果你买一个现成的由Raspberry Pi Pico驱动小工具，它几乎肯定会使用”城堡形电路板”的安装方式。

从半孔槽向内的孔用于2.54毫米公引脚接头连接器-与 Raspberry Pi的GPIO接头连接器使用的引脚类型相同。 通过将它们向下焊接到适当的位置，您可以将Pico推入无焊面包板中，以使连接和断开新硬件的过程变得尽可能容易-非常适合进行实验！

▲图1-3：RP2040芯片

Pico中心的芯片（图1-3）是RP2040微控制器。 这是一种定制的集成电路（IC），由Raspberry Pi的工程师专门设计和制造来驱动你的Pico和其他基于微控制器的设备。

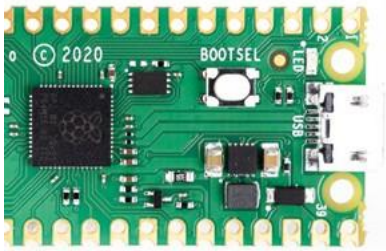
如果你把注意力放到芯片中心的反光部分，则会看到

Raspberry Pi标志蚀刻在芯片的顶部，以及一系列的字母和数字，使工程师可以跟踪制造芯片的时间和地点。

▲图1-4： 微型USB端口

在你的Pico顶部是一个micro *USB端口*。（**图1-4**）。它不仅可以提供电源使你的Pico运行，也可以让Pico通过其USB端口与你的树莓派或其他计算机通信并使用它们上传程序到你的Pico。

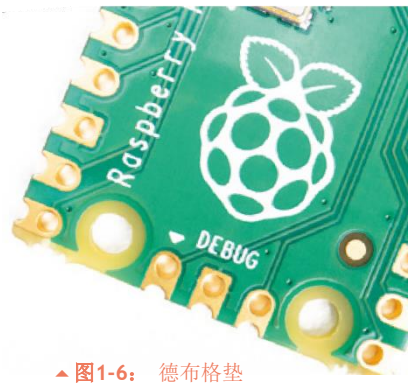
如果你拿着你的Pico使其micro USB端口朝上从正面看，则将看到它是底部变窄，顶部变宽的形状。用一根micro USB连接线，你会看到它的形状与是相同的。



▲图1-5：引导选择开关

micro USB电缆仅能通过向上对齐插入Pico的micro USB端口这一方向。 连接时，请确保将正确的窄边和宽边对齐，否则以错误的方式堵塞micro USB电缆可能会损坏Pico!

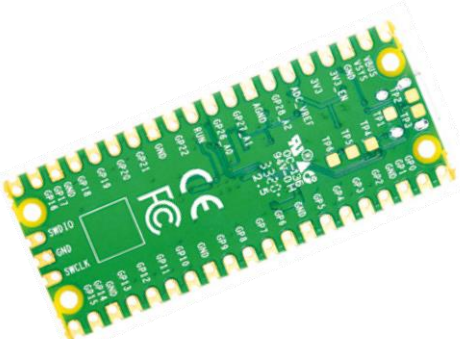
在micro USB端口下面是一个小按钮标记“BOOTSEL”(图1-5)。“BOOTSEL”是“boot selection”(启动选择)的缩写，它可以在你的Pico第一次启动时在两种启动模式之间切换。在为使用MicroPython编程准备好Pico之后，您将使用到引导选择按钮。



▲图1-6： 德布格垫

在Pico的底部有三个较小的金色焊盘，上面有*"DEBUG"*字样（**图1-6**）。它们是为在Pico上运行的程序中使用一种称为调试器的特殊工具调试或查找错误而设计的，虽然你在本书中不会用到调试引脚，但是在当你写比较大型的程序或者较复杂的程序是你会发现它非常有用。

把你的Pico翻转过来，你会看到底面有如（**图1-7**）的内容展现在上面。这些内容被称为*丝印层*，上面而且标记着每个引脚与它的核心功能。你将看到"GP0"和"GP1"，"GND"，"RUN"和"3V3"等内容。如果你忘了哪个引脚是哪个，这些标签会告诉你-但当你把Pico安装到面包板上你不能看到他们，所以你需要找到印刷在这本书中完整的*引脚图*，以方便参考。



▲图 1-7： 标签底面

您可能已经注意到，并不是所有的标签都与它们的引脚对齐:板顶部和底部的小孔是安装孔，设计用于允许您使用螺钉或螺母和螺栓更永久地将Pico连接到项目上。在这些孔妨碍标签的地方，标签被进一步向上或向下推移：看右上角，“VBUS”是左边的第一个针，“VSYS”是第二个，“GND”是第三个。

你还会看到一些标有“TP”和数字的金色平板。这些都是为工程师设计的测试点，以便在Raspberry Pi Pico在工厂组装完成后，快速检查它是否工作正常。根据测试焊盘，工程师可能会使用一个叫做万用表或示波器的工具来检查你的Pico是否正常工作，然后将其打包并运送给你。

最后，你会看到一个带有条形码的小贴纸。这是你Pico的序列号还有版本信息和它的制造日期。

焊接排针

当你拆开你的Raspberry Pi Pico时，你会发现它是完全平坦的：并没有金属别针从两侧伸出来，就像你发现你的树莓派的GPIO排针一样。这是因为你可能想使用城堡式PCB连接到另一个电路板，或者直接焊接导线。

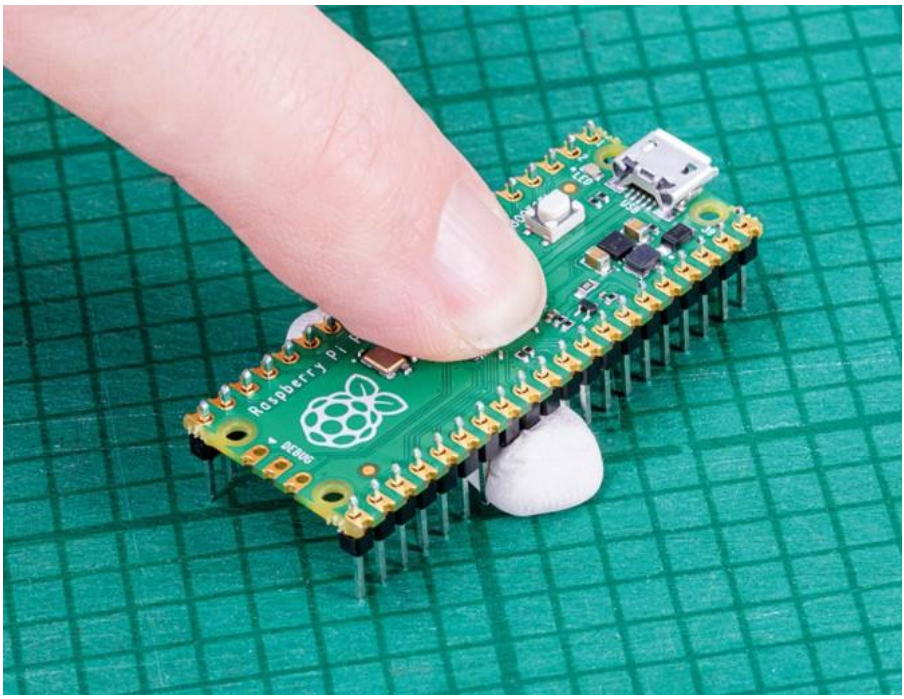
不过，使用Pico的最简单方法是将其安装连接到面包板上， 为此，你需要附加引脚头。你需要一个焊接铁与支架，一些焊工，清洁海绵，你的Pico，和两个20针2.54毫米单排针。如果你已经有一个无焊接面包板，你可以用它来使焊接过程更容易。

有时，2.54毫米的排针以超过 20 针的条带提供。如果你的更长，只需从一端数20针，看看20号和21号针之间的塑料：你会看到它有一个小凹痕在两边。这是一个*折断点*：把你的左手和右手的拇指放在贴近这个折断点的两侧然后进行弯曲，它会干净利落地断裂，留给你整整20的引脚的排针。如果剩余的排针长于 20 引脚，则再次做相同的工作，以便你有两个 20 针条。

警告

电烙铁不是玩具：它们会变得非常非常热，在拔下后也会长时间保持热。 如果你是一个年纪比较小的学习者，确保你有成人监督：无论你是年幼还是年长，确保你不使用它时把电烙铁放在支架上，永远不要碰金属部件

– 即使在拔下插头后的一段时间内也是如此。



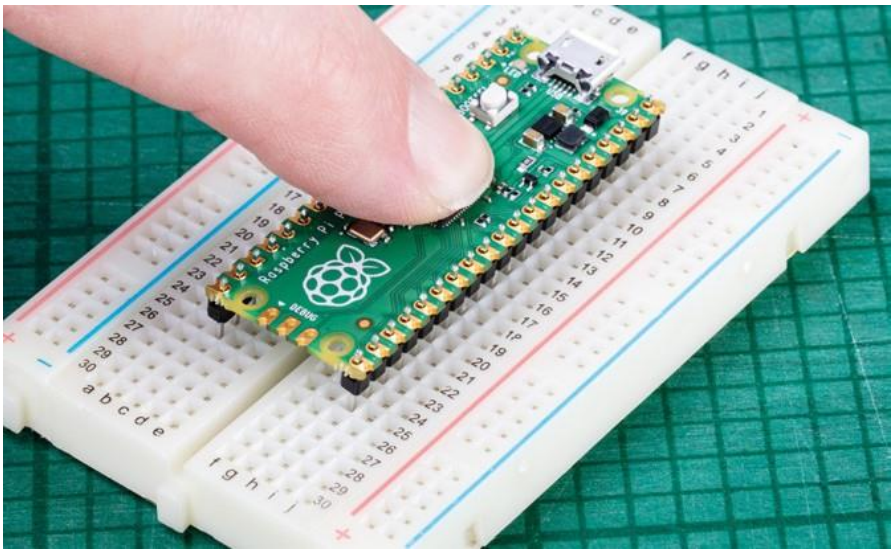
▲图1-8： 在焊接之前，你可以用粘性腻子将排针进行固定

将Pico倒置，以便你可以在底部看到丝网引脚数和测试点。取两个排针中的一个，轻轻将其推入Pico左侧的针孔中。确保它正确插入孔中，而不仅仅是贴在城堡式PCB半孔上，并且所有20 个引脚都到位，然后取另一个排针并将其插入右侧。完成后，引脚上的塑料块应向上推至Pico的电路板上并使其与电路板贴合。

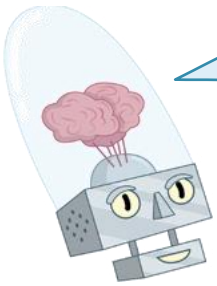
在Pico两侧捏着抬起Pico并同时捏紧两个排针，这个时候不能松手，否则排针会掉下来的！如果你还没有面包板，你需要一些方法来保持头到位，而你焊接-不要用你的手指去触碰到他们，否则它们会烫伤你。你可以用小鳄鱼夹，或一小斑点的蓝斑或其他粘腻子（**图1-8**）保持标固定位置。先焊接一个引脚，然后检查对齐：如果引脚是在一个角度，融化焊接，你仔细调整他们使其对齐。

如果你有一个面包板，只需把你的Pico倒过来-记住保持捏住排针-并推动排针和你的Pico到你的面包板上的孔。继续推，直到你的Pico与夹在你的Pico和你的面包板（**图1-9**）之间的排针上的塑料块是平整的。

看到Pico的顶部：你会看到每个排排针的一小段长度伸出过孔。这是你要焊接的部分 - 这意味着需要加热Pico上的排排针和焊盘，并融化少量的特殊金属焊接到他们身上。



▲图 1-9： 或者使用面包板将排针保持固定位置进行焊接



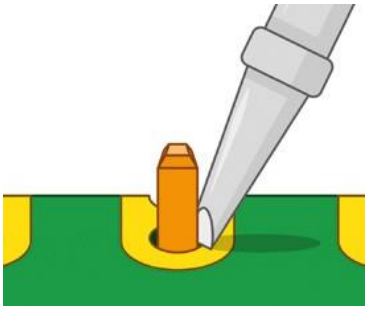
警告

焊接是一个很好的进行学习的技能，但它确实需要多加练习。在你打开你的焊接铁之前，仔细和完整的阅读指示，并记住慢慢地和小心地进行焊接。也避免使用过多的焊件：在焊件太少的接头中添加更多焊件很容易，但要带走多余的焊件可能更加困难——特别是当它溅到Pico的其他部位时。

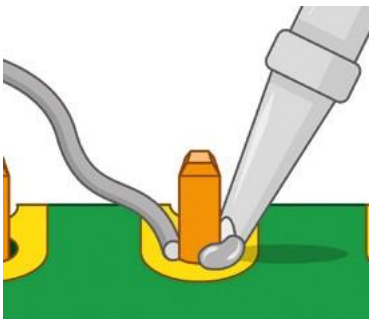
将焊接铁放在支架上，确保金属尖端不会靠在任何东西上，并将其插入。熨斗尖需要几分钟才能变热：当你等待展开一小段焊料时——大约是食指长度的两倍。你应该能够通过拉扯和扭曲来打破焊接：它是一种非常柔软的金属。

警告

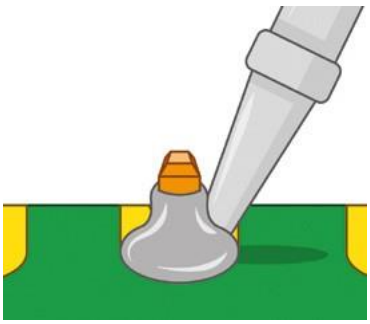
虽然现代焊锡丝是无铅制成的，但由于一种叫做助焊剂的特殊物质，它仍然有毒。这是一种腐蚀性凝剂，在焊接时会腐蚀接触点上的污垢。如果你把它弄到你的的手指上它不会伤害你，但是如果你吃把它吃进去了，它可能会对你有害。只有当你正确的处理焊锡丝时你才能进行焊接，然后保持勤洗手的习惯——尤其是在你吃任何东西之前。



▲图1-10： 加热排针和焊盘



▲图1-11： 添加一点焊锡丝



▲图1-12： 现在移除电烙铁

▲图1-13： 一个焊接好的排针

如果你的焊接架有一个清洁海绵，把海绵到水槽，并放一点点冷水在它上面，使其软化。从海绵中挤出多余的水，这样它很潮湿，也不会滴水，然后放回支架上。如果你使用的是用黄铜丝制成的清洁球，则不需要任何水。

拿起你电烙铁的手柄，确保防止在移动电烙铁时电缆不会绊倒其他物品。像握铅笔一样握住它，但要确保你的手指只接触塑料或橡胶手柄区域：**金属部件，甚至实际铁尖前的轴，将非常热，可以很快烧伤你。**

在你开始焊接前，清洁电烙铁的烙铁头：沿着你的海绵或卷线清洁器刷烙铁头。把你的需要焊接的一定长度的焊锡丝，在一端举行，并推动另一端到你电烙铁的烙铁头上：它会迅速融化成一个锡珠。如果没有，让电烙铁停留加热更长的时间-或尝试给烙铁头在清洁海绵上进行一次清洁。

把焊料滴在烙铁上的过程叫做镀锡。焊接中的助焊剂有助于燃烧烙铁头上的任何污垢，并将准备好待下次使用。再次用海绵或清洁导线擦拭熨斗，以清除多余的焊锡;烙铁头应该保持干净光亮。

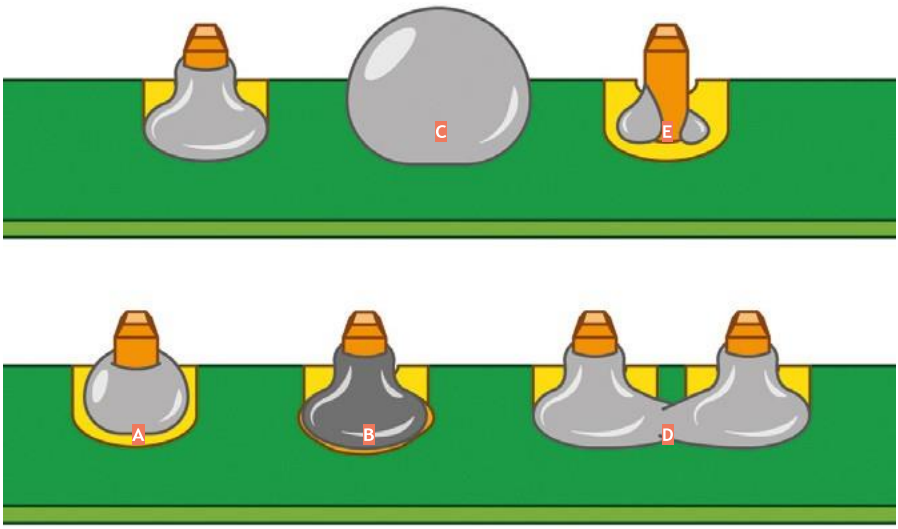
把电烙铁放回支架上，它应该总是在那里，除非你需要很频繁地使用它，移动你的Pico放到在你面前。一只手拿起电烙铁，另一只手拿起焊锡丝。将铁尖压在离你最近的排针上，以便同时接触到Pico上的垂直排针和金色焊盘（**图 1-10**）。

重要的是，排针和焊盘都需要加热，所以保持你的电烙铁同时接触这两部分数三下。当你数到三时，仍然保持电烙铁的位置，将你一定长度的焊锡丝末端轻轻地按到排针和焊盘上，但你的烙铁头是在另一边位置（**图1-11**）。就像你在焊烙铁头上镀锡一样，焊锡应该很快熔化并开始流动。

焊料会围绕着引脚和焊盘流动，但不会进一步流动:这是因为你的Pico的电路板被涂上了一层叫做阻焊剂的涂层，它能将焊料保持在需要的位置。确保不要用太多焊料：一点点就够了。

将焊料的剩余部分从焊接处拿开时，确保电烙铁还是保持在该位置。如果你先把铁拔出，焊料就会变硬，你就不能把你手里的焊锡丝取出来;如果发生这种情况，只要把铁放回原处，让它再次熔化。一旦熔化的焊料在引脚和焊盘周围扩散开来（**图1-12）**，这应该只需要一秒钟左右的时间，取出烙铁。祝贺你:你焊接了你的第一个排针!

用海绵或铜丝清洁熨斗的尖端，然后把它放回架子上。拿起你的Pico，看一下你的焊点:它应该填满焊盘，然后向上平稳地与销子会合，看起来有点像火山的形状，销子填充了熔岩所在的洞，如**图1-13**所示.



▲图1-14： 焊接问题示例

如果焊锡丝粘在引脚上，但不粘在铜焊盘上，例如**图1-14**中的 示例 A 所示，然后导致焊盘加热不够。别担心，它很容易解决：拿起你的电烙铁，把它放在焊盘和排针相接触的地方，确保它这一次都能压住。几秒钟后，焊锡就会回流，形成良好的焊点。

另一方面，如果焊接器太热，它不会流动良好，你会得到一个过热的焊点与一些烧焦的助焊剂（例如**图1-14**中的示例 B 所示）。这可以通过用刀尖、牙刷和少量异丙醇小心地刮去。

如果焊接器完全覆盖了引脚，例如**图1-14**中的 C 示例，则使用了太多。这不一定会引起问题，尽管它看起来不太吸引人：只要没有焊锡接触它周围的任何排针，它仍然会工作。如果它触摸其他引脚（如**图1-14**的 D 示例），你导致他们连锡了，将导致短路。

同样，连锡也很容易修复。首先，试着在你正在制作的焊点上回流焊锡：如果这不起作用，用烙铁头抵住连锡的另一边，让它流一些进去，如果仍然有太多的焊料，你需要在使用你的Pico之前清除多余的焊料:你可以购买脱焊编织，你压在熔化的焊料上吸出多余的焊料，或一个吸锡器物理上吸出熔化的焊锡。

另一个常见的问题是焊料太少:如果你还能看到铜焊盘，或者在引脚和焊盘之间有一个没有填充焊料的空隙，那么你用的太少了(**图1-14**中的示例 **E**)。把铁放回针和垫，数到三，并添加一点更多的焊料。焊料太少总是比太多更容易修复，所以要记得在焊接时放轻松一点!

一旦你对第一个引脚感到满意，对你的Pico上的所有40个引脚重复这个过程——让底部的三个引脚的“DEBUG”头为空。提示:先焊接好四个角上的排针。慢慢来，不要着急，记住错误总是可以补救的。记得在焊接时也要定期清洁熨斗的尖端，如果你发现事情变得困难，熔化一些焊料在尖端重新镀锡。确保焊锡的长度也要保持新鲜:如果太短，手指太靠近烙铁头，很容易烫伤自己。

当你完成后，你检查了所有的引脚是否有良好的焊接点，并确保它们没有连接到附近的任何引脚上，在把它放回支架上并拔下它之前，最后一次清洁和锡烙铁的尖端。把电烙铁放凉后再放好:拔下电烙铁后，它的热度伤你可能会让保持很长一段时间!

最后，记住一定要洗手 - 并庆祝你掌握焊接能手的新技能!

安装 MicroPython

现在，你已经将排针焊接到Pico（**图1-15**）上，只有一件事需要完成：在它上面安装MicroPython。首先，将micro USB 电缆插入Pico上的micro USB 接口，确保它是正确的插入方向，然后再轻轻地将其推入。

▲ **图1-15：** 所有引脚正确焊接

警告

为了要将 MicroPython 安装到您的Pico上，您需要从网上下载它。如果您的树莓派上没有互联网连接，您需要将Pico连接到具有互联网连接的系统，以便完成设置。你只会必须这样做一次：MicroPython安装后，它将留在您的Pico上，除非您决定在将来用其他东西取代它。

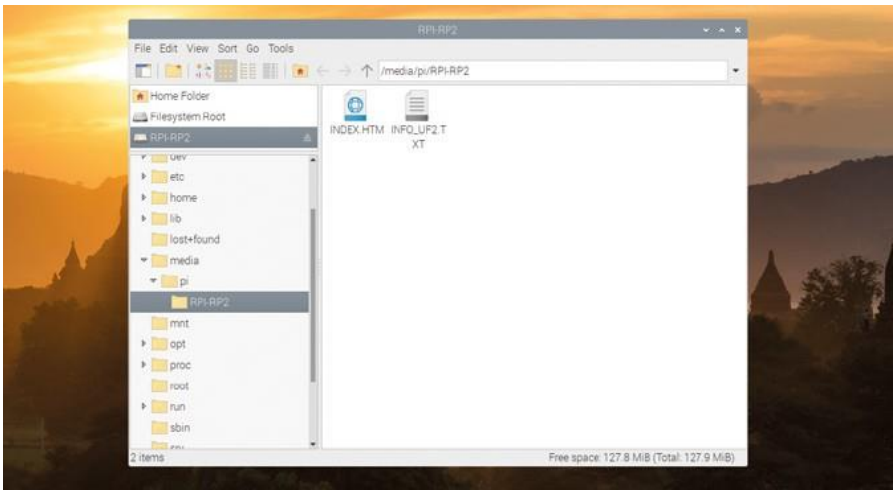
按住你的Pico顶部的“BOOTSEL”按钮;然后，按住鼠标，将micro USB线的另一端连接到树莓派或其他电脑的USB接口上。数到三，然后松开“BOOTSEL”按钮。

再过几秒钟，你应该会看到你的Pico作为一个可移动驱动器出现——就像你连接了一个USB闪存驱动器或外部硬盘驱动器。在你的树莓派上，你会看到一个弹出的询问是否要在文件管理器中打开驱动器。

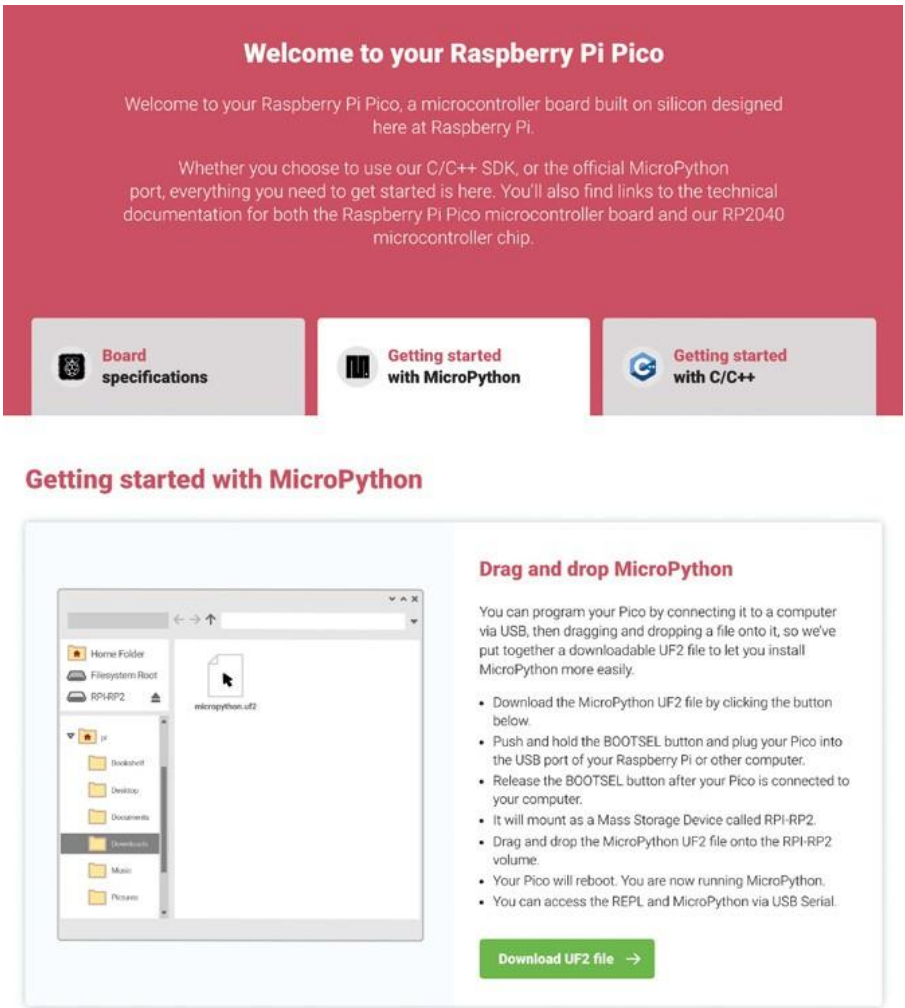
在文件管理器窗口中，你将看到Pico上的两个文件（**图 1-16）：INDEX。 HTM** 和 **INFO_UF2。TXT**.第二个文件保存有关你的Pico的信息，例如它当前运行的 *引导加载程序*的版本。第一个文件，**INDEX。HTM，**是你想要的：将鼠标指针指向它并在浏览器中双击它来打开它。

当浏览器打开时，您将看到一个欢迎页面，告诉您有关您的Pico的所有信息。阅读页面上的信息，然后单击MicroPython选项卡来加载页面中特定于MicroPython的部分。

点击“Download UF2 file”按钮(图1-17，后面一页中)下载MicroPython固件-一个包含MicroPython的小文件，供您的Pico使用。下载时间不会太长。



▲ **图1-16：** 你会看到你的Raspberry Pi Pico两个文件

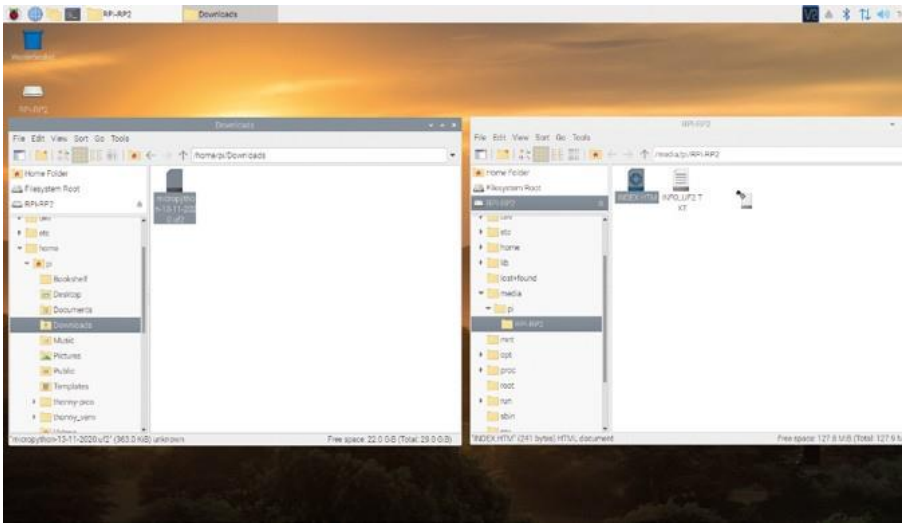


▲ 图1-17： 单击按钮下载 MicroPython 固件

-这是一个非常小的文件。下载后，只需单击右上角的交叉图标，即可关闭浏览器窗口。

通过单击树莓图标菜单、前往附件和单击文件管理器来打开新的文件管理器窗口。使用文件管理器窗口左侧的文件夹列表查找"**下载**"文件夹。你可能需要滚动列表才能找到它，这取决于你在树莓派上有多少个文件夹。打开**下载**文件夹，找到你刚刚下载的文件-它将被称为"MicroPython"，然后是日期和扩展名"uf2"。

点击并按住UF2文件上的鼠标按钮，然后将它拖到另一个文件管理器窗口上，打开您的Pico的可移动存储驱动器。将其悬停在窗口上，松开鼠标按钮将文件拖放到Pico上（**图 1 - 18**）。



▲图 1-18：将MicroPython固件文件拖动到你的Raspberry Pi Pico

几秒钟后，你会看到你的Pico从文件管理器消失，你也可能看到一个警告，驱动器被删除，而不会被弹出：别担心，这是正常且应该发生的！当您将MicroPython固件文件拖到Pico上时，您告诉它将固件烧录到其内部存储上。要做到这一点，您的Pico将退出您通过“BOOTSEL”按钮将其放入的特殊模式，烧录新的固件，然后加载它——这意味着您的Pico现在正在运行MicroPython。

恭喜你:你现在已经准备好在你的Raspberry Pi Pico上开始使用MicroPython了！

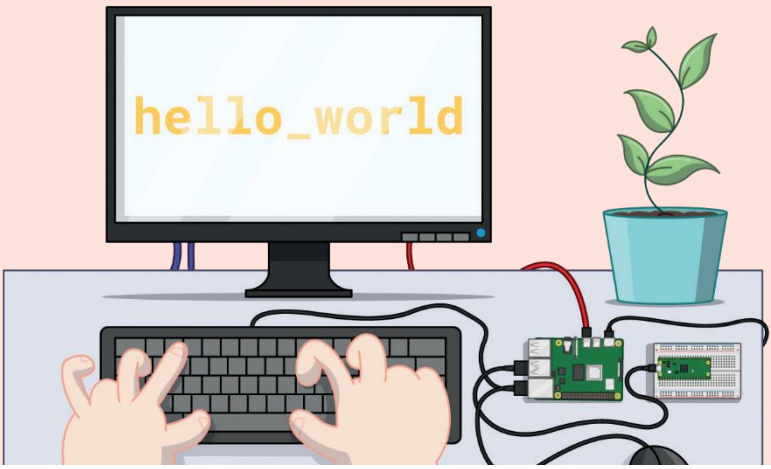
进一步阅读

从指数链接的网页。HTM不仅仅是一个下载MicroPython的地方，它还包括大量的额外资源。单击选项卡并滚动以访问指南、项目和数据书集 -涵盖从RP2040 微控制器的内部工作到 Python 和 C/C++ 语言编程等各种详细技术文档的书架。

第二章

使用MicroPython编程

连接计算机，并开始使用MicroPython语言为你的Raspberry Pi Pico编写程序

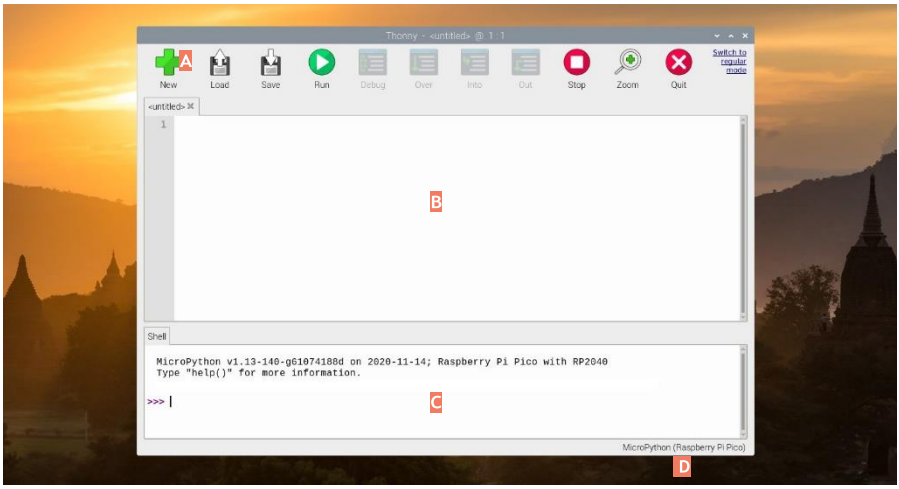


自它在1991年推出以来，Python编程语言——得名于著名的喜剧团体Monty Python，而不是蛇——已经成长为世界上最受欢迎的语言之一。虽然它很流行，但这并不意味着没有改进的余地-特别是如果你工作在微控制器上。

Python 编程语言是为台式机、笔记本电脑和服务器等计算机系统开发的。像Raspberry Pi Pico这样的微控制器板更小、更简单、内存也更小，这意味着它们不能运行与大型Python语言相同的Python语言。

这就是MicroPython的用处。MicroPython最初由Damien George开发，于2014年首次发布，是一种专门为微控制器开发的python兼容编程语言。它包含了许多主流Python的特性，同时添加了一系列新的特性，旨在利用Raspberry Pi Pico和其他微控制器板上灯可用的设备。

如果你以前与 Python 编程过，你会立即发现 MicroPython 很熟悉。如果没有，不要担心：这是一个学习起来友好的语言！



▲图 2 - 1：汤尼 Python Ide

Thonny Python Ide 介绍

A **Toolbar** – 工具栏提供基于图标的快速访问系统，用于常用的程序功能，如保存、加载和运行程序。

B **脚本区域** – 脚本区域是编写 Python 程序的位置。它被分割成你的程序的主要区域和显示行数的少量侧边距。

C **Python控制台** – Python控制台允许你键入单个指令，当您按下ENTER键时，这些指令就会运行，并提供有关运行程序的信息。这也称为***REPL***，用于"读取、评估、打印和循环"。

D **解释器** –Thonny 窗口的右下角显示并允许您更改当前的Python解释器——用于运行您的程序的Python版本。

Thonny 连接到Pico

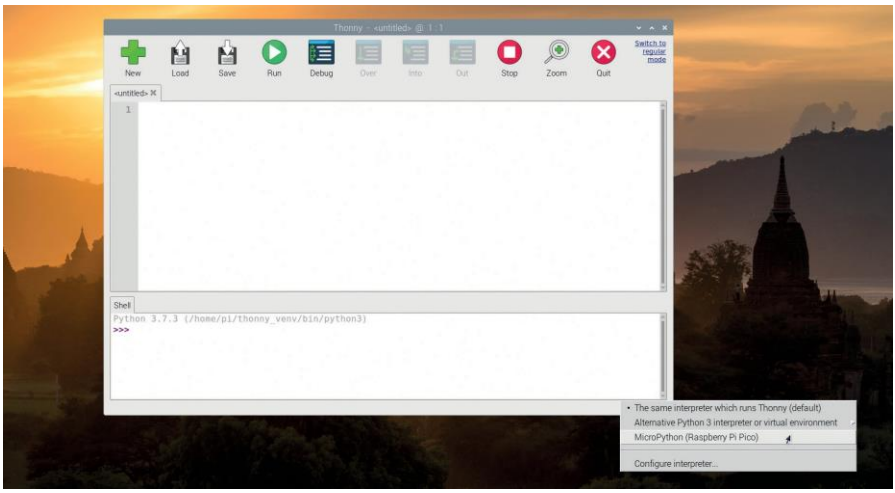
在开始用MicroPython为Pico编程之前，需要设置所谓的集成开发环境(IDE)。Thonny，一个流行的Python和MicroPython IDE，已经预装在Raspberry Pi OS上;如果您在另一个Linux发行版、Windows或macOS上使用Pico，请打开web浏览器，访问**thonny.org**，并单击页面顶部的下载链接，下载用于您的操作系统的Thonny和Python包安装程序。

作为一个集成的开发环境，Thonny将你需要编写*或开发*的所有不同工具聚集在一起或*集成*到单个用户界面或*环境中*。有许多不同的 IDE：有些允许你使用多种不同的编程语言进行开发，而另一些则像Thonny一样专注于单一语言。

首先加载Thomny：在树莓派操作系统上，你可以通过点击屏幕左上角的Raspberry菜单，移动鼠标到编程部分，然后单击Thomny来加载它。如果你还没有这样做，拿起你的Pico，在它和树莓派的任一个USB接口之间连接一根micro USB线。

当你的Pico连接到树莓派时，点击单词"Python"，然后在Thonny窗口的右下角加上版本编号-这个区域在**图2-1中**标记为**D**。这显示了你当前的解释器，它负责接受你输入的指令，并将它们转换成计算机或微控制器可以理解和运行的代码。通常解释器是在Raspberry Pi上运行的Python的副本，但是为了在你的Pico上运行你的程序，需要对它进行修改。

在显示的列表中，查找"MicroPython（Raspberry Pi Pico）"**（图2-2），**然后单击它。如果你在列表中看不到它，请仔细检查你的Pico是否正确插入micro USB 电缆，以及micro USB电缆是否正确插入到树莓派或其他计算机。



▲图2-2： 选择 Python 解释器

PYTHON的专业人员

如果您在《官方树莓派初学者指南》中浏览学习过 Python 章节，你将会对这一章中的大部分内容非常熟悉。尽管如此，通过前两个例子，以适应在Raspberry Pi的Python解释器和在Pico MicroPython解释器中运行程序的不同，然后自由地跳到下一章。

没有RASPBERRY PI PICO选项？

Raspberry Pi Pico解释器只在最新版本的Thonny提供。如果你运行的是一个较老的版本并且不能更新它，寻找“MicroPython (generic)”代替。如果你的版本更老，在窗口的右下角没有解释器选项，你不能更新它，点击右上角的“切换到常规模式”，重新启动Thonny，点击运行菜单，然后点击“选择解释器”。点击“运行Thonny (默认)的同一解释器”旁边的下拉箭头，点击列表中的“MicroPython(通用)”，然后点击下拉箭头旁边的'端口'和点击'板在FS模式'在该列表在单击OK以确认更改之前。

看看Thonny窗口底部的 Python控制台： 你会看到它现在输出显示 'MicroPython'， 并告诉你，它正在运行在 'Raspberry Pi Pico' 。恭喜你：你已准备好开始编程。

你的 第一个 MicroPython 程序：Hello, World!

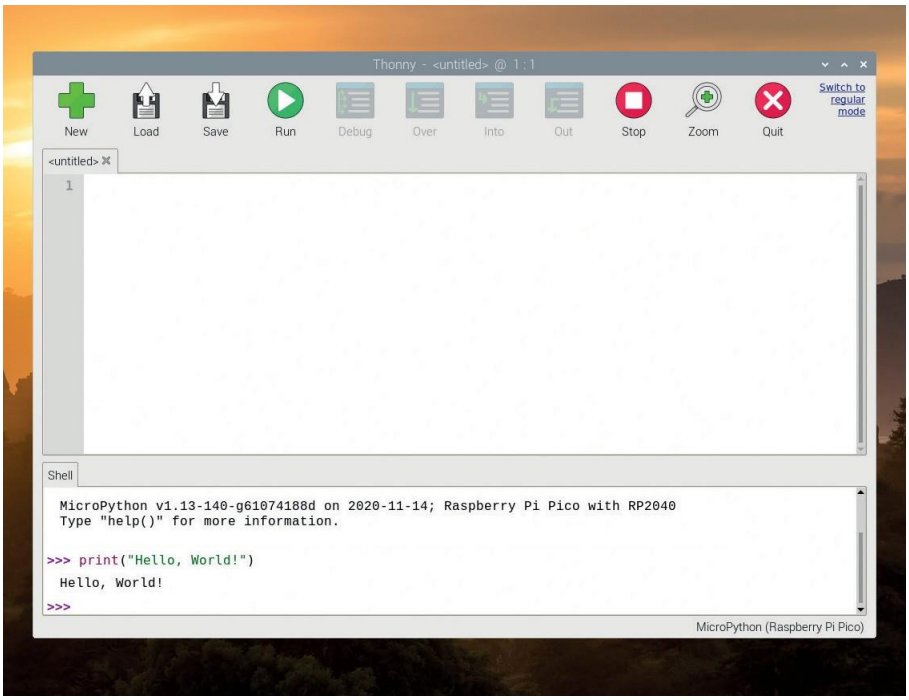
要开始编写第一个程序时，请单击 Thonny窗口底部的 Python Shell区域，只需在底部"+++"符号的右侧，并在按下**ENTER**键之前键入以下指令。

```
print("Hello, World!")
```

当你按下 **ENTER**时，你会看到你的程序立即开始运行：Python将在相同的Shell区域响应，并发出“Hello, World!”（**图2-3**），这是因为Shell与运行在您的Pico上的MicroPython解释器直接相连，后者的工作是查看您的指令并解释它们的含义。这就是所谓的互动模式，你可以把它想象

语法错误

如果你的程序没有运行，而是在Shell区域打印了一个“syntax error”消息，说明你写的程序有错误。MicroPython需要以一种非常特定的方式来编写指令:漏掉一个括号或引号，拼写“print”错误或给它一个大写的P，或者在指令的某个地方添加额外的符号，它就不能运行。再试着输入一遍指令，并在按回车键前确认它与这本书的版本相匹配!



▲ 图2-3： MicroPython 在Shell地区打印"你好，世界！

成与某人面对面的对话:你刚说完话，对方就会做出回应，然后等着你接下来说什么。

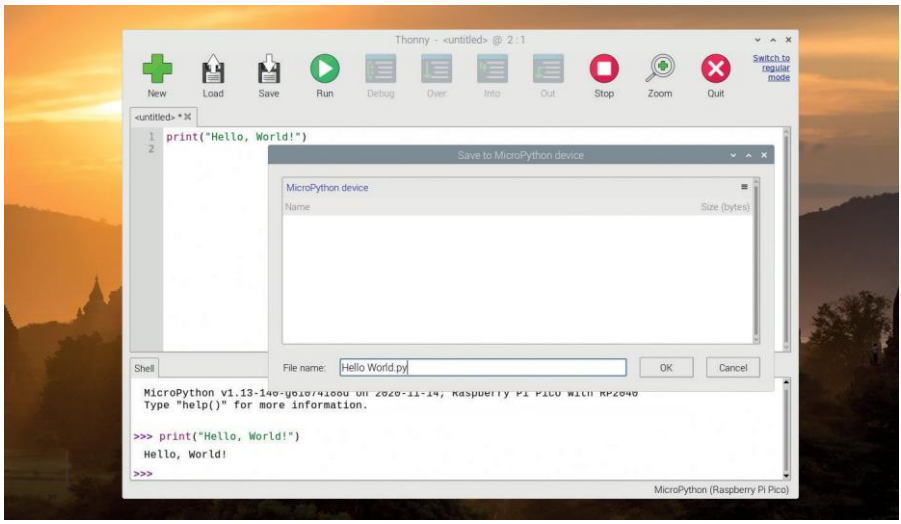
使用Shell为Pico编程有点像电话对话：当您按下 **ENTER** 键时，您的指令将通过micro USB线发送到运行在Pico上的MicroPython解释器;解释器查看你的指令，做它被告知的任何事情，然后通过micro USB线把结果发送回给Thonny。

不过，您不必通过Shell在交互模式下编程。点击Thonny窗口中间的脚本区域，然后再次输入你的程序:

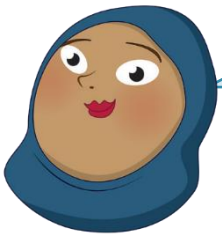
```
print("Hello, World!")
```

当您这次按下 **ENTER** 键时，除了在脚本区域中获得新的空白行外，什么都没发生。要使此版本的程序工作，你必须单击 Thonny 工具栏中的"运行"图标，或单击"运行"菜单，然后单击"运行当前脚本"。

现在点击运行图标:你会被询问是否要保存程序到“这台电脑”，即Raspberry Pi，或“MicroPython设备”，即Pico，如**图 2-4**所示。单击“MicroPython device”，然后键入一个描述性名称，如Hello_World.py，然后单击OK按钮。



▲图2-4：保存程序到Pico



文件名称

将在将MicroPython文件保存到您的Pico时，始终记住键入文件扩展名:文件末尾后跟字母’ p ’和’ y ’(表示’ Python ’)的句号。这可以帮助您记住每个文件都是一个程序，并防止它们与您可能保存在Pico上的任何其他文件混淆。

你可以为你的程序使用任何你喜欢的名字，但是尽量让它描述程序的功能——不要叫它**boot.py**或**main.py**，因为这些是特殊的文件名，您将在本书后面了解更多。

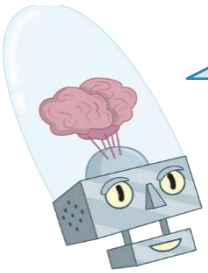
一旦你的程序保存好，它就会自动在你的Pico上运行。你会看到两条消息出现在Shell区域的底部的thony窗口:

```
>>> %Run -c $EDITOR_CONTENT
Hello, World!
```

其中第一行是 Thonny 的指示，告诉 Pico 上的 MicroPython 翻译运行脚本区域的内容 - "EDITOR_CONTENT"。第二个是程序的输出——你让MicroPython打印的信息。恭喜你：现在你已经以交互模式和脚本模式写了两个MicroPython程序，并成功地在你的Pico上运行了它们！

还有一件事需要解决:重新加载你的程序。按窗口右上角的X键关闭Thonny，然后再次加载它。这次不是编写一个新程序，而是点击Thonny工具栏中的“Open”图标。就像当你保存你的程序时，你会被询问是否要保存到' This computer ' or your ' MicroPython device ' -点击' MicroPython device '，你会看到你已经保存到你的Pico的所有程序的列表。

在列表中查找Hello_World.py—如果您的Pico是新的，它将是唯一的文件—并单击它选择它，然后单击OK。你的程序将加载到Thonny，准备好被编辑或你再次运行它。



可以装满程序的Pico

当你告诉索尼将你的程序保存在MicroPython设备上时，这意味着程序存储在Pico设备上。如果你拔掉你的Pico插头，把它带到你的朋友家、学校活动或编程俱乐部，并把它插到他们的电脑上，你的程序仍然会在你保存它们的地方——在你自己的Pico上。

挑战：新消息

你能改变Python程序输出的消息吗?如果要添加更多消息，是使用交互模式还是脚本模式?如果您从程序中删除括号或引号，然后尝试再次运行它，会发生什么?

下一步：循环和代码缩进

与标准Python程序一样，MicroPython程序通常从上到下运行:它依次运行每一行，在进入下一行之前通过解释器运行它，就像您在Shell中逐行输入它们一样。

但是，仅仅逐行运行指令列表的程序并不是很聪明——因此，MicroPython就像Python一样，有自己的方法来控制程序运行的顺序:缩进。

通过单击Thonny工具栏中的新图标创建一个新程序。你不会失去现有的程序;相反，Thonny将在脚本上方的区域创建一个新的标签。输入以下两行开始你的程序:

```
print("Loop starting!")
for i in range(10):
```

第一行向Shell输出一条简单的消息，就像您的Hello World程序一样。第二行开始一个定义的循环，它将重复-循环-一个或多个指令的一组次数。一个变量i被赋值给循环，并给出一系列数字——range指令，它被告知从数字0开始，向上工作，但永远不会到达数字10——进行计数。冒号(:)告诉MicroPython循环本身从下一行开始。

变量是强大的工具：顾名思义，变量是可以改变的值-或在程序的控制下随时间变化。最简单的说，一个变量有两个方面:它的名称和它所存储的数据。在循环的情况下，变量的名字是' i '，其数据由range指令设置——从0开始，每次循环结束并重新开始时增加1。

为了在循环中真正包含一行代码，它必须缩进——从脚本区域的左侧移动进来。下一行以四个空格开始，当您按下第二行回车后，Thonny将自动添加这些空格。现在输入:

```
    print("Loop number", i)
```

与程序中的其他行相比，这四个空格将这行推入内部。缩进是MicroPython区分循环外部指令和循环内部指令的方式:缩进的代码构成了循环内部，被称为嵌套代码。

您将注意到，当您在第三行末尾按下ENTER时，Thonny会自动缩进下一行——假设它是循环的一部分。要删除这个缩进，只需在输入第四行之前按下BACKSPACE键一次:

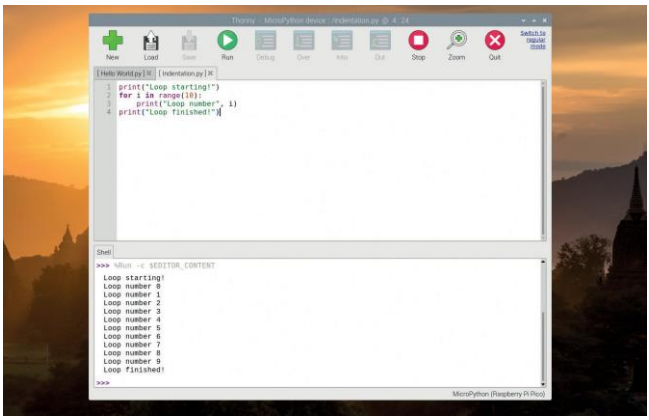
```
print("Loop finished!")
```

您的四行程序现在已经完成。第一行位于循环之外，并且只运行一次;第二行设置循环;第三个循环位于循环内部，每次循环运行一次;第四行还是在循环外面。

```
print("Loop starting!")
for i in range(10):
    print("Loop number", i)
print("Loop finished!")
```

单击Run（运行）图标，选择通过点击"MicroPython设备"选择将程序保存在Pico上，并将其命名为**Indentation.py**。该程序将在保存后立即运行:查看Shell区域的输出（**图2-5**，后一页）。

```
Loop  starting!
Loop  number  0
Loop number 1
Loop number 2
Loop number 3
Loop number 4
Loop number 5
Loop number 6
Loop number 7
Loop number 8
Loop number 9
Loop finished!
```



▲ 图2-5：执行循环

从零开始计数

Python是一种0索引语言——这意味着它从0开始计数，而不是从1开始计数。这就是为什么程序打印数字0到9而不是1到10的原因。如果您愿意，您可以通过将range(10)指令切换为range(1,11)或其他任何您喜欢的数字来改变这种行为。

缩进是MicroPython的一个强大部分，也是程序不能按预期工作的最常见原因之一。当寻找程序中的问题时，一个称为调试的过程，总是反复检查缩进-特别是当您开始在循环中嵌套循环时。

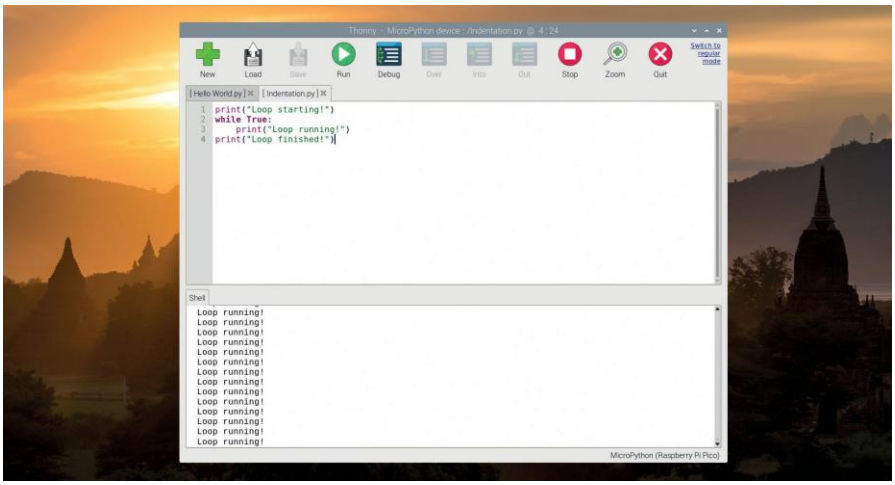
MicroPython还支持无限循环，它可以无限循环。要将程序从确定循环变为无限循环，请编辑第2行：

```
while True:
```

：如果您现在单击Run图标，您将得到一个错误:名称“i”没有定义。这是因为你已经删除了为变量i创建并赋值的行。要修复这个问题，只需编辑第3行，这样它就不再使用变量了：

```
print("Loop running!")
```

再次点击Run（运行）图标，如果你动作快，你会看到“循环开始!”消息后面跟着一串永无休止的“Loop running”消息(图2-6)。“循环完成!'消息将永远不会打印，因为循环没有结束:每次Python打印完' loop running!消息时，它返回到循环的开始处并再次输出



▲ **图2-6：** 一个无限循环，它一直持续到你停止程序

点击Thonny工具栏上的Stop图标，告诉程序停止正在做的事情——即中断程序——并重新启动MicroPython解释器。您将看到Shell区域中出现一条消息，程序将停止，而不会到达第4行。

挑战：对循环进行循环

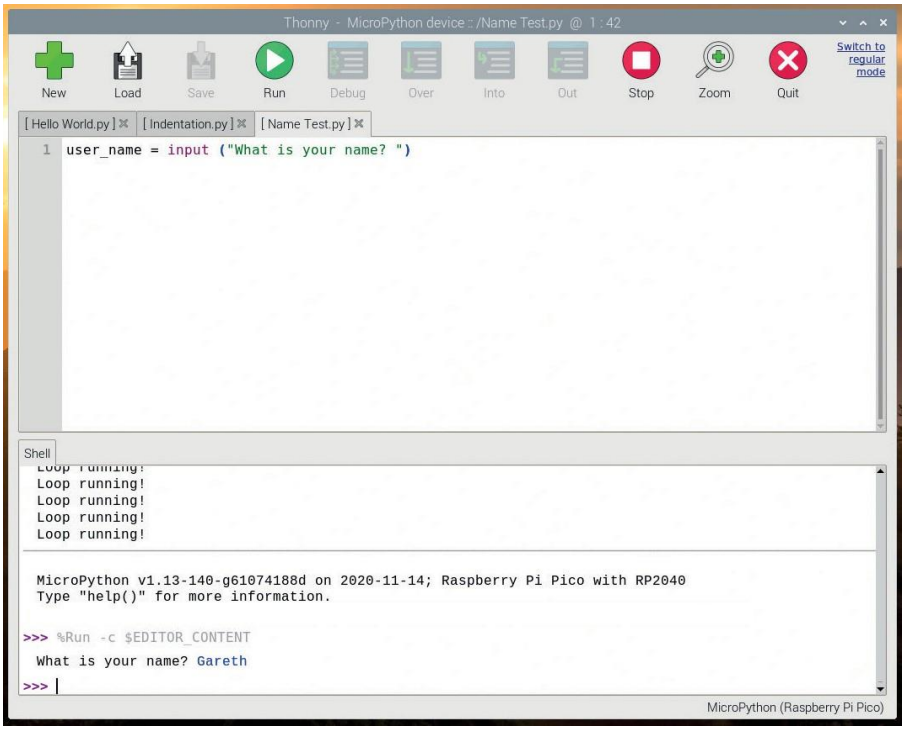
你能把这个循环再变成一个确定的循环吗?你能给程序添加第二个明确的循环吗?你会如何在循环中添加循环，你期望它如何工作?

条件和变量

和所有编程语言一样，MicroPython中的变量的存在不仅仅是为了控制循环。点击Thonny工具栏上的新图标启动新程序，然后在脚本区域输入以下内容:

```
user_name = input ("What is your name? ")
```

点击Run图标，通过点击“MicroPython device”选择将程序保存在Pico上，并将其命名为Name_Test. py，并观察Shell区域中发生了什么:将询问您的姓名。在Shell区域中输入您的名字，然后输入ENTER。因为这是你的程序中唯一的指令，其他的都不会发生(图2-7，背面)。如果想要对放入变量中的数据进行实际操作，则需要在程序中增加更多行。



▲ 图2-7：输入功能允许你向用户询问一些文本输入

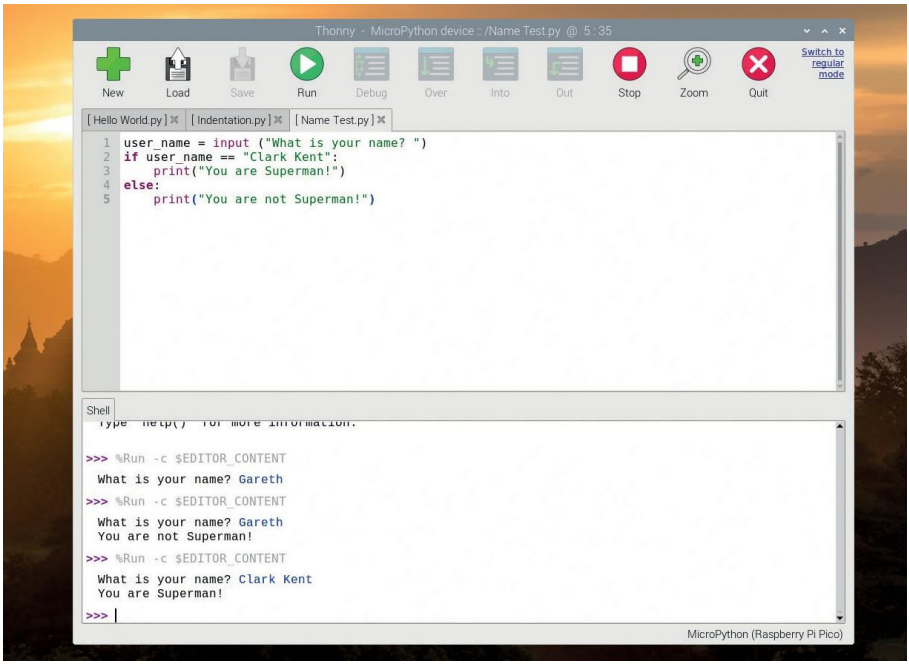
为了让你的程序用这个名字做一些有用的事情，从第2行开始输入一个条件语句：

```
if user_name == "Clark Kent":
    print("You are Superman!")
else:
    print("You are not Superman!")
```

记住，当Thonny看到你的代码需要缩进时，它会自动这么做——但它不知道什么时候你的代码需要停止缩进，所以你必须自己删除空格。

单击Run图标并在Shell区域中键入您的名字。除非你的名字恰好是克拉克·肯特，否则你会看到“你不是超人!”再次点击Run，这次输入Clark Kent的名字——确保写的和程序中完全一样，用大写的C和k。这次，程序承认你实际上是超人（**图2-8**）。

==符号告诉Python进行直接比较，查看变量user_name是否与程序中的文本(即字符串)匹配。如果你处理的是数字，你还可以进行其他的比较>查看一个数字是否大于另一个数字，



▲图 2 - 8： 你不应该出去拯救世界吗？

<查看它是否小于，>=查看它是否大于或等于，<=查看它是否小于或等于。还有!=，意思是不等于，是==的反义词。这些符号在技术上称为比较运算符。

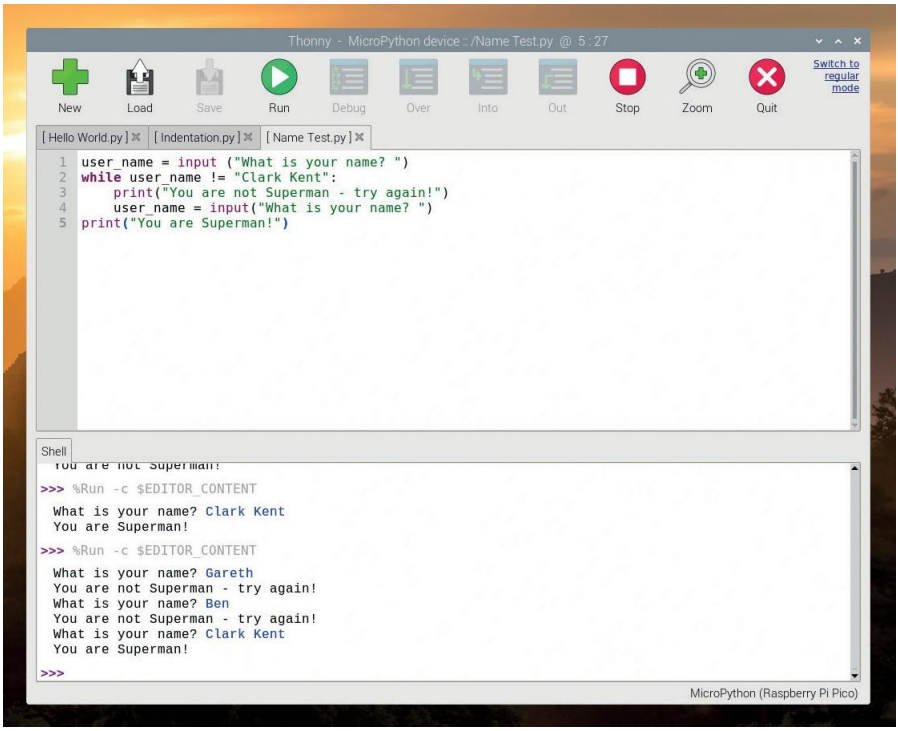
使用=和==

使用变量的关键是了解=,与==两者的区别。记住：=的意思是'使这个变量等于这个值'，而==的意思是'检查变量是否等于这个值'。把它们混在一起肯定会导致程序无法工作!

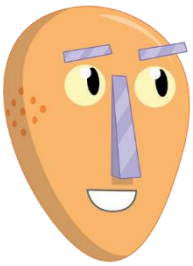
比较操作符也可以在循环中使用。删除第2行到第5行，然后在它们的位置输入以下内容:

```
while user_name != "Clark Kent":
    print("You are not Superman - try again!")
    user_name = input("What is your name? ")
print("You are Superman!")
```

再次点击Run（运行）图标。这一次，程序不会退出，而是会不断询问你的名字，直到确认你是超人(图2-9)——有点像一个非常简单的密码。要退出循环，可以在脚本区域输入“Clark Kent”，或者单击Thonny工具栏上的Stop（停止）图标。祝贺您:您现在知道如何使用条件语句和比较操作符了!



▲图2-9:程序会一直问你的名字，直到你说它是“Clark Kent”



挑战：添加更多问题

你可以改变程序来问多个问题，把答案存储在多个变量中吗?你能制作一个程序，使用条件运算符和比较运算符来打印用户输入的数字是大于还是小于5吗?

第三章

物理计算

了解你可以连接和控制的Raspberry Pi Pico上的引脚和电子元件

当人们想到“编程”或“编码”时，他们通常很自然地想到软件。然而，编码不仅仅是关于软件:它可以通过硬件影响现实世界。这就是所谓的物理计算。顾名思义，物理计算就是用你的程序控制现实世界中的事物:硬件，而不是软件。

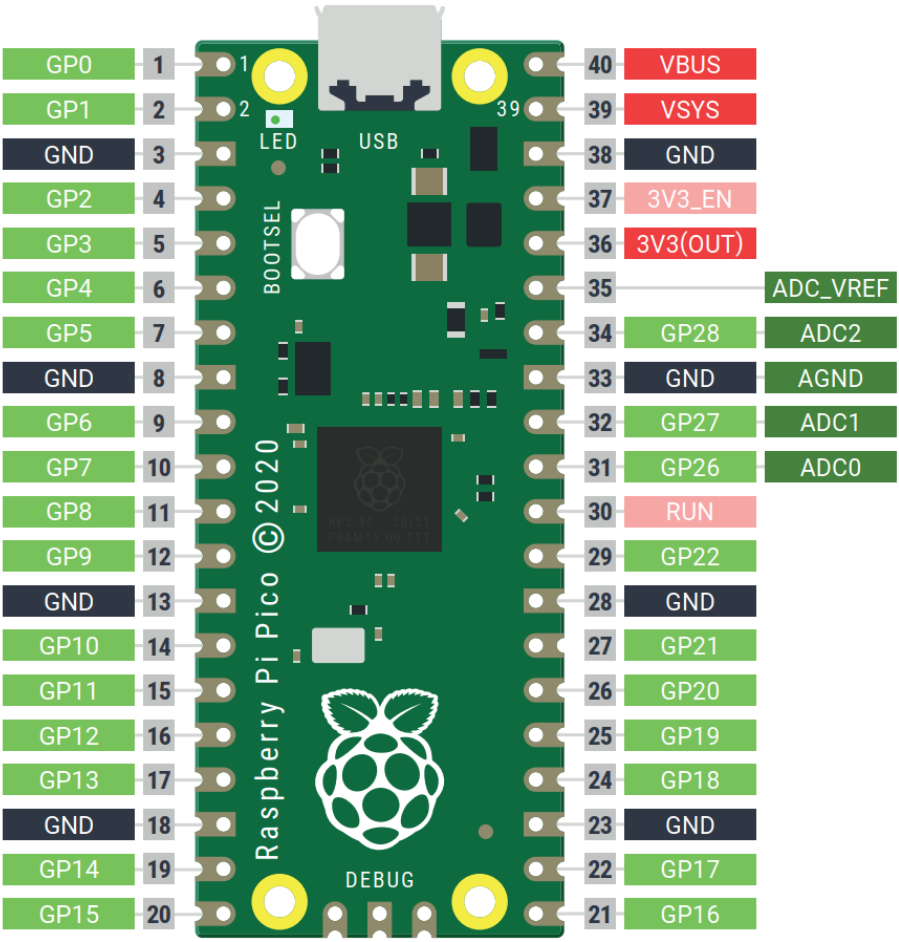
当你在你的洗衣机上设置程序，改变你的可编程恒温器上的温度，或按下一个按钮交通信号灯安全过马路，你是使用物理计算。

这些设备通常是由单片机控制的非常像一个在你的Raspberry Pi Pico-这是完全有可能为你创建自己的控制系统通过学习利用你的“微小”的能力,很容易当你学会了编写软件运行在你的“微小”。

你的Pico的引脚

您的Pico通过其边缘一系列的引脚与硬件通信。这些引脚大多是作为通用输入/输出(GPIO)引脚工作，这意味着它们可以被编程作为输入或输出，并没有自己的固定用途。有些引脚有额外的功能和与更复杂的硬件通信的替代模式;另一些则有一个固定的目的，比如供电和提供连接的功能。

Raspberry Pi Pico的40个引脚被标记在板的底部，3个也在板的顶部被标记有他们的数字:Pin 1, Pin 2, Pin 39。这些顶级标签帮助你记住编号是如何排序的: 引脚1是在左上方，当你从上面看板与micro USB端口到上面的一边，引脚19、20在左下角，引脚21、22在右下角，引脚39的右上方与未标记的引脚40在右上角。



▲图 3-1: 从上面看Raspberry Pi Pico的引脚图

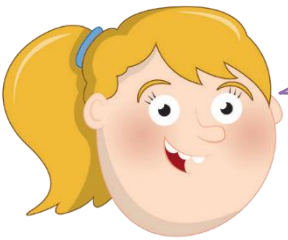
比起使用物理引脚的序号，更常见的是根据其在引脚拥有的功能来使用的(见图3-1)。下面有有几个类别的pin类型，每一个都有一个特定的功能：

3V3	3.3V 电源	3.3 V的电源，与Pico内部运行的电压相同，由VSYS输入产生。可以使用上面的3V3_EN引脚打开和关闭这个电源，它也会关闭您的Pico。
VSYS	~2-5V 电源	一个直接连接到Pico内部电源的引脚，如果没有将Pico关闭，就不能将其关闭。
VBUS	5V 电源	从你的Pico的micro USB端口获取的5 V电源，用于为需要3.3 V以上的硬件供电。
GND	0V接地	一种接地连接，用于完成电路与电源的连接。几个这样的引脚点缀在您的Pico上，使布线更容易。
GP*xx*	通用输入/输出 引脚编号"xx"	你的程序可以使用的GPIO引脚，标记为'GP0'到'GP28'。
GP*xx*_ADC*x*	通用输入/输出引脚号码xx 模拟输入号码x	以ADC和数字结尾的GPIO引脚既可以用作模拟输入，也可以用作数字输入或输出，但不能同时用作两者。
ADC_VREF	模数转换器参考电压	为任何模拟输入设置参考电压的一种特殊输入引脚。
AGND	模数转换器 0V 参考电压	一种特殊的接地方式 ADC_VREF引脚。
RUN	启用或禁用您的Pico	其他微控制器可以通过排针

几个GPIO引脚有额外的功能，你将在本书后面了解到。有关包括这些额外功能的完整引脚，请参阅附录B。

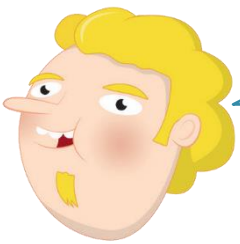
引脚GP0

就像在Python中计数一样，你的Pico的GPIO引脚从数字0开始而不是数字1。标记在板的底部，它们从0到29，尽管有些没有引出作为物理引脚。



缺失的引脚

Pico上的通用输入/输出引脚基于其主控芯片(RP2040微控制器)的引脚进行编号。然而，并不是RP2040上所有可用的引脚都被带到您的Pico的引脚上——这就是为什么在最后一个基本通用引脚GP22和第一个可模拟引脚GP26_ADC0之间的编号存在差距的原因。

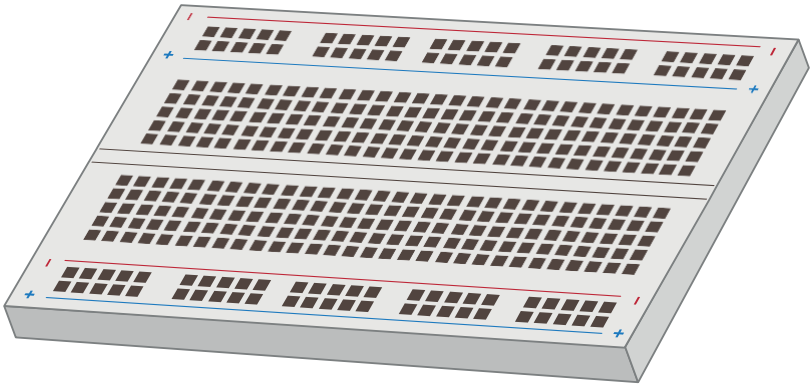


警告

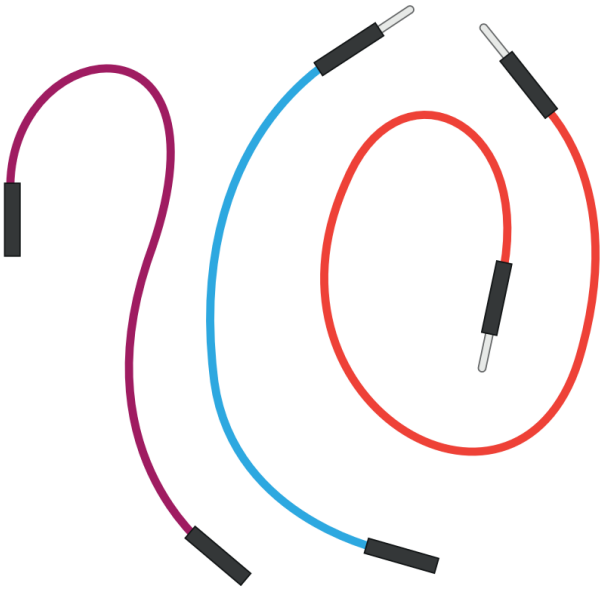
您的Pico的引脚被设计成一种有趣且安全的物理计算实验方式，但始终要小心对待。注意不要弯曲别针，特别是当你把你的Pico插入面包板的时候。永远不要将两个引脚意外或故意直接连接在一起，除非你在项目说明中被告知这样做:这被称为短路，根据引脚的不同，可能会永久性地损坏你的Pico。

电子元件

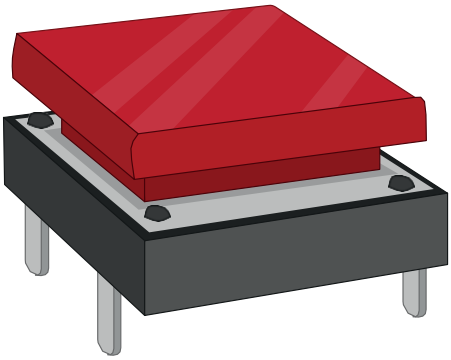
您的Pico只是您开始使用物理计算所需的一部分;另一半由电子元件组成，你可以通过Pico的GPIO管脚来控制这些设备。有数千种不同的组件可用，但大多数物理计算项目都使用以下通用元件。



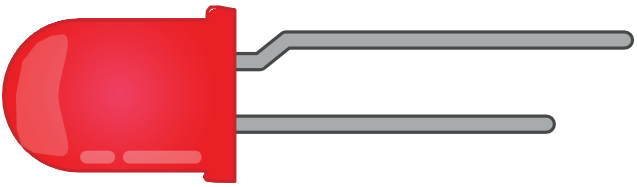
面包板，也称为无焊料面包板，可以使物理计算项目变得容易得多。面包板不是一堆需要用电线连接的独立组件，而是让你插入组件并通过隐藏在表面下的金属轨道将它们连接起来。许多电路板还包括电源分配的部分，使你更容易建立电路。您不需要一个breadboard来开始物理计算，因为您可以使用特殊的线将组件直接连接到您的Pico的GPIO引脚上，但它肯定会使事情更简单和更稳定。



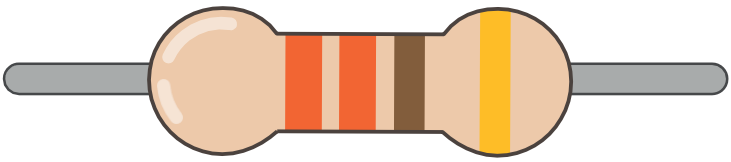
跳线，也被称为跳线引线，在你不使用面包板时将你的Pico与连接组件连接到彼此。它们有三个版本:公对母线(M2F);母对母线(F2F)，如果您不使用面包板，可以使用它将单个组件连接到您的Pico;以及公对公(M2M)，用于将面包板的一部分连接到另一部分。根据您的项目，您可能需要所有三种类型的跳线;如果你使用的是面包板，你通常可以用M2F和M2M跳线。



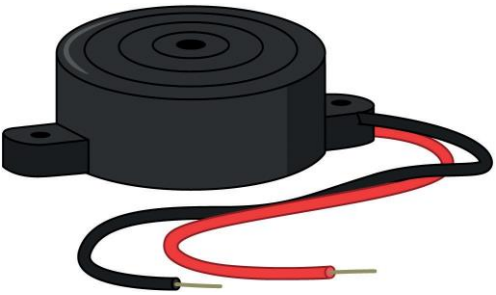
按钮开关，也称为*瞬时开关*，是用于控制游戏机的开关类型。通常有两条或四条脚可供选择——任何一种类型都可以与Pico配合使用——按钮是一种输入设备：你可以告诉程序注意它是否被按下，然后执行任务。另一种常见的开关类型是锁定开关：而按钮仅在按住按钮时处于活动状态，而*锁定开关*（就像在光开关中发现一样）在切换一次按钮时激活，然后保持活动状态，直到再次切换它。



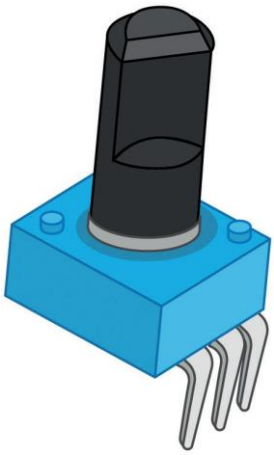
*发光二极管（LED）*是输出设备：你可以直接从你的程序控制它。当LED灯亮着的时候，你会发现你的房子里到处都是LED灯，从让你知道你的洗衣机开机的小灯到让你的房间亮起来的大灯。led有各种各样的形状、颜色和尺寸，但并非所有LED 都适合与Pico一起使用：避免使用任何表示它们专为 5V 或 12V电源设计的LED。



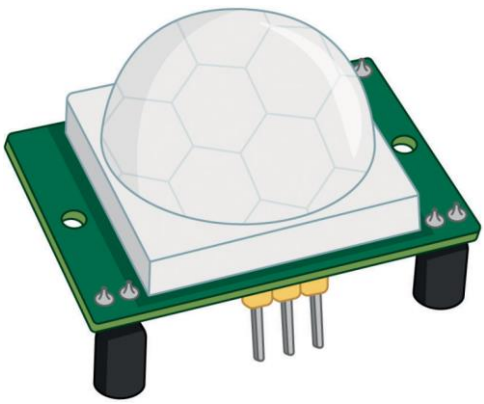
*电阻器*是控制电流流动的组件，可使用称为*ohms（Ω）*的单位进行测量不同的值。ohms 的数量越大，提供的阻力就越大。对于Pico物理计算项目，它们最常见的用途是防止LED产生过多的电流并损坏自己或Pico：为此，你希望电阻器的额定值约为330 Ω，尽管许多电气供应商销售的方便电阻包含许多不同的常用值，这为你提供了更大的灵活性。



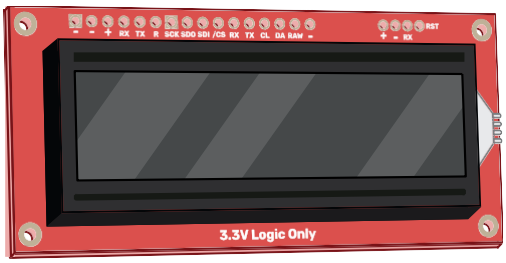
压电式蜂鸣器，通常被称为蜂鸣器或声音器，是另一种输出设备。虽然LED能发光，但蜂鸣器却会发出噪音——实际上是嗡嗡声。蜂鸣器的塑料外壳内是一对金属板;当接通电源时，这些平板会相互振动，产生嗡嗡声。蜂鸣器有两种:主动蜂鸣器和被动蜂鸣器。确保有一个活跃的蜂鸣器，因为这是最容易使用的。



电位器是一种你可以在音乐播放器上找到音量控制的组件，可以作为两个不同的组件工作。当它的三个引脚中的两个连接起来时，它就充当了可变电阻或压敏电阻，这种电阻可以通过拧旋钮随时调节。当三个引脚适当地连接起来，它成为一个分压器，并根据旋钮的位置就可以输出从0V到全电压输入的任何东西。



无源红外传感器（PIR）是被称为的多种传感器输入设备之一，旨在报告所监视内容的变化。 对于PIR传感器来说，它监测运动:传感器观察塑料镜片覆盖区域的变化，当它检测到运动时就发送信号。PIR传感器通常在防盗警报器中看到，以发现在黑暗中移动的人。



*I2C 显示屏*是一个屏幕，通过称为*集成电路（I2C） 总线*的特殊通信系统与你的Pico进行通信。此总线允许你的Pico控制显示面板，发送从文字到图片的所有内容以供其显示。有很多类型的显示可供选择，其中比较流行的一种是SparkFun的SerLCD，它同时具有I2C和SPI接口。注意，一些显示器只使用串行外围接口(SPI)总线，而不是I2C;它们仍然可以与您的Pico一起工作，但需要更改程序。

其他常见的电气组件包括电机，它需要一个特殊的驱动元件才能连接到你的Pico，电流传感器可以检测电路使用的功率，跟踪运动和方向的惯性测量单元 （IMUs）， 以及光敏电阻（LDRs）-通过检测光而不是发射它像反向LED 一样运行的输入设备。

世界各地的卖家都提供了很多与树莓派物理计算的组件，无论是作为单独的部件或套件，都能在你开始时提供需要的一切。要找到卖家，请访问**rpf.io/products**，点击树莓派4，你会得到树莓派合作伙伴在线商店（批准经销商）为你的国家或地区的列表。

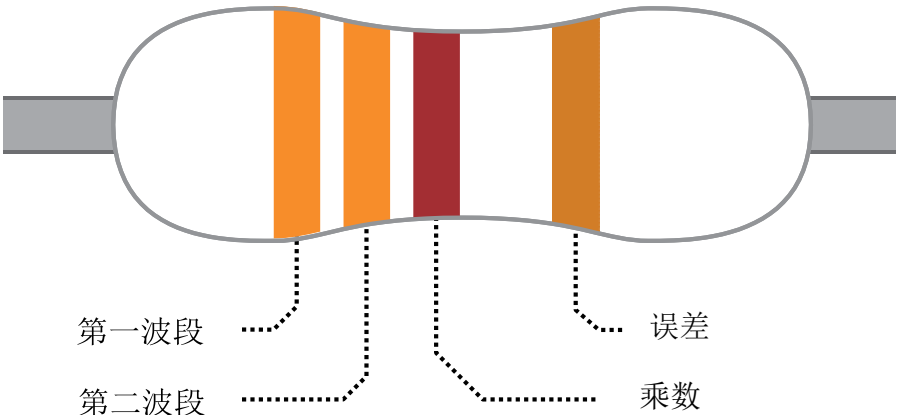
要完成本书中的项目，你至少应该拥有：

- 焊接好排针的Raspberry Pi Pico
- micro USB 数据线
- 面包板
- 树莓派或其他用于编程的计算机
- 公对母杜邦线（M2F） 和公对公杜邦线 （M2M）
- 3 ×单色LED灯： 红色、绿色、黄色或琥珀色
- 1 ×有源压电蜂鸣器
- 1 × 10 kΩ线性电位器
- 3 × 330 Ω电阻
- 1 × HC-SR501红外传感器
- micro USB 数据线
- 1 ×串行LCD屏
- WS2812B LEDs

你也会发现买一个有多个隔层的廉价储物箱很有帮助，这样你就可以把你在项目中不用的组件保持安全和整洁。如果可以的话，试着找一个同样适合面包板的，这样每次你做完的时候就可以把所有东西都收拾干净。

通过读取电阻上的颜色码获取阻值

电阻器的值范围很广，从零电阻版本(实际上只是几根电线)到发电站使用的脚大小的版本。但是，很少有电阻器的数值是以数字的形式打印出来的:取而代之的是，电阻器使用一种特殊的代码，即电阻器周围的彩色条纹或条带。



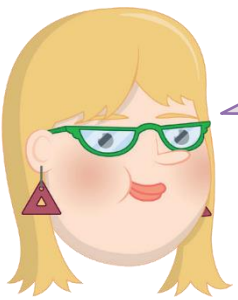
	1st/2nd Band	Multiplier	Tolerance
Black	0	$\times 10^{0}$	-
Brown	1	$\times 10^{1}$	±1%
Red	2	$\times 10^{2}$	±2%
Orange	3	$\times 10^{3}$	-
Yellow	4	$\times 10^{4}$	-
Green	5	$\times 10^{5}$	±0.5%
Blue	6	$\times 10^{6}$	±0.25%
Violet	7	$\times 10^{7}$	±0.1%
Grey	8	$\times 10^{8}$	±0.05%
White	9	$\times 10^{9}$	-
Gold	-	$\times 10^{-1}$	±5%
Silver	-	$\times 10^{-2}$	±10%
None	-	-	±20%

要读取电阻器的值，将其定位为一组带在左边，一个带在右边。从第一个波段开始，在表的“第一/第二波段”列中查找它的颜色，以得到第一个和第二个数字。本例有两个橙色的条带，它们都表示值为“3”，总共为“33”。如果你的电阻器有四个分组波段，而不是三个，记下第三波段的值也-为5或6波段的电阻器看到 **rpf.io/5-6band**

移到最后一组波段-第三或第四，取决于你的电阻-在“倍增器”列中看它的颜色。这告诉你，你需要用当前数乘以什么数才能得到电阻器的实际值。这个例子有一个棕色的条带，意思是“$\times 10^1$”。这看起来可能让人困惑，但它只是一个简单的科学符号:“$\times 10^1$”的意思只是“在你的数字后面加一个零”。如果它是蓝色的，如×106 '，它的意思是'在你的数字的最后加上六个零'。

橙色带是33，加上棕色带是0，得到330，这是电阻的值，单位是欧姆。最后一个波段，在右边，是电阻的容差。这仅仅是它可能有多接近其评级价值。便宜的电阻可能有一个银带，表明它可以高于或低于其额定值的10%，或根本没有最后一个带，表明它可以是20%的高或低;最昂贵的电阻有一个灰色带，表明它将在其额定值的0.05%之内。对于大多数业余爱好者的项目来说，准确性并不是最重要的。

如果你的电阻值超过 1000 ohms（1000 Ω），则通常以千欧（kΩ）级：如果它超过一百万欧姆，那些是兆欧（MΩ）级。2200 Ω电阻器将写成2.2kΩ;2200000Ω电阻器将被写成2.2 MΩ。



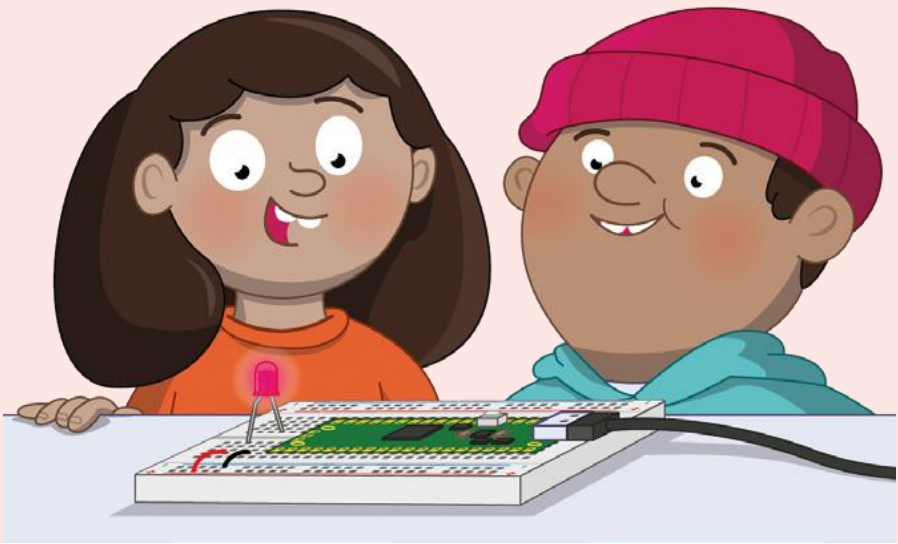
你能解决吗？

100 Ω电阻有什么颜色的带?2.2 MΩ电阻有什么颜色? 如果你想为你的项目找到最便宜的电阻，你会找到什么样的颜色误差范围带呢?

第四章

使用 Raspberry Pi Pico 进行物理计算

开始将基本电子元件连接到Raspberry Pi Pico，并编写程序来控制和观察它们



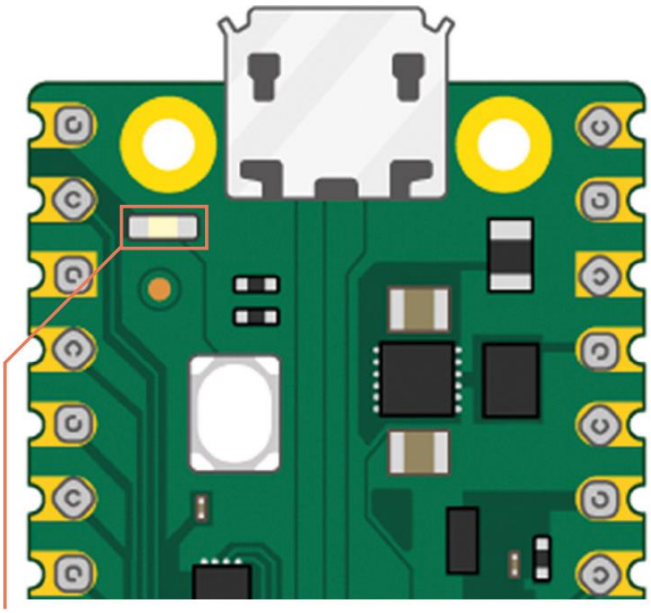
Raspberry Pi Pico的RP2040微控制器在设计时考虑到了物理计算。它的众多通用输入/输出(GPIO)引脚让它与一系列组件对话，允许你建立项目，从照明led到记录你周围世界的数据。

物理计算并不比传统计算更难学习:如果你能遵循**第二章**的例子，你将能够建立你自己的电路，并对它们进行编程来执行你的命令。

你的第一个物理计算程序：Hello, LED!

就像在屏幕上打印“Hello, World”是学习编程语言的第一步，让LED发光是学习物理计算的传统入门。你也可以在没有任何额外组件的情况下开始:你的Raspberry Pi Pico在顶部有一个小的LED，被称为贴片(表面安装SMD) LED。

首先找到LED:它是板顶部micro-USB端口左侧的一个小矩形组件**(图4-1)**，标签上写着“LED”。这个小LED的工作原理和其他的一样:当它被通电时，它会发光;当它关闭电源，它就会熄灭。



▲图 4-1：板载LED位于micro-USB连接器的左下侧

板载LED连接到RP2040的一个通用输入/输出引脚GP25上。您可能还记得，这是RP2040上“缺失”的GPIO引脚之一，但并没有在您的Pico边缘上出现一个物理引脚。虽然你不能将任何外部硬件连接到这个引脚上，但除了板载LED，它可以被当作与你程序中的任何其他GPIO引脚一样的处理——而且它是一种不需要任何额外组件就可以向你的程序添加输出的好的途径。

加载Thonny，如果您还没有这样做，请将其配置为连接到您的Pico，如第2章所示。点击进入脚本区域，用以下一行启动你的程序:

```
import machine
```

这一小行代码是在您的Pico上使用MicroPython的关键:它加载或导入MicroPython代码集合，称为库——在本例中是“machine”库。machine library包含MicroPython与Pico和其他兼容MicroPython的设备通信所需的所有指令，为物理计算扩展了语言。如果没有这一行，您将无法控制任何Pico的GPIO引脚，也无法使机载LED发光。

选择性导入

在MicroPython和Python中，都可以导入库的一部分，而不是整个库。这样做可以使您的程序使用更少的内存，并允许您混合和匹配来自不同库的函数。本书中的程序导入了整个库;在其他地方，你可能会看到程序有像from machine import Pin;这告诉MicroPython只从' machine '库中导入' Pin '函数，而不是整个库。

machine library公开了所谓的应用程序编程接口(API)。这个名称听起来很复杂，但是准确地描述了它的功能:它为您的程序(或应用程序)提供了一种通过接口与Pico通信的方法。

程序的下一行提供了machine library的API示例:

```
led_onboard = machine.Pin(25, machine.Pin.OUT)
```

这一行定义了一个名为led_onboard的对象，它提供了一个友好的名称，您可以使用它在程序的后面引用板载LED。从技术上讲，这里可以使用任何名称-例如susan、gupta或fish_sandwich-但最好还是使用描述变量用途的名称，这样可以使程序更容易阅读和理解。

这行代码的第二部分调用machine library中的Pin函数。顾名思义，这个函数是为处理你的Pico的GPIO引脚而设计的。目前，没有一个GPIO引脚-包括GP25，连接到板载LED的引脚-知道他们应该做什么。第一个参数，25，是你设置的pin号码;第二,machine.Pin。OUT，告诉Pico该引脚应该被用作输出而不是输入。

这行代码只起到了设置引脚的作用，它并不会点亮LED。要做到这一点，您需要告诉Pico打开pin。在下一行输入以下代码:

```
led_onboard.value(1)
```

它可能看起来不像，但这一行也使用了machine library的API。前面的一行创建了对象led_onboard作为pin GP25上的输出;这一行创建了对象并将其值设置为1(表示“on”)-它也可以将值设置为0(表示“off”)。

引脚编号

当谈到您的Pico上的GPIO引脚时，它们通常使用它们的全名:例如，GP25表示连接到板载LED的引脚。但是，在MicroPython中，字母G和P被省略了——所以确保在程序中写的是' 25 '而不是' GP25 '，否则它将不起作用!

单击Run按钮并将程序保存在您的Pico上为Blink.py。你会看到LED灯亮起来。祝贺您——您已经编写了您的第一个物理计算程序!

然而，你会注意到，LED仍然是亮着的:这是因为你的程序告诉Pico打开它，但从来没有告诉它关闭LED。你可以在程序的底部添加另一行:

```
led_onboard.value(0)
```

然而，这次运行程序，LED似乎永远不会点亮。这是因为Pico的工作速度非常非常快——比你用肉眼看到的速度快得多。LED正在发光，但在如此短的时间内，它似乎保持熄灭。要解决这个问题，您需要通过引入延迟来降低程序的速度。

转到程序的底部，并单击led_onboard.value(1)行末尾，然后按 **ENTER** 键插入新行。类型:

```
Import utime
```

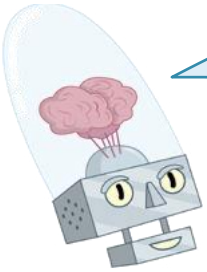
与**import machine**类似，这一行将一个新的库导入**MicroPython: ' utime '**库。这个库处理与时间有关的所有事情，从度量时间到向程序中插入延迟。

转到程序的底部，并单击**led_onboard.value(1)**行末尾，然后按ENTER键插入新行。类型:

```
utime.sleep(5)
```

这将调用来自**utime**库的**sleep**函数，该函数将使您的程序暂停数为您输入的秒数——在本例中为5秒。

再次单击Run按钮。这次你会看到你的Pico上的LED灯亮了，保持亮5秒钟-试着一直数-然后再出去。



UTIME VS TIME

如果你以前用过Python编程，你会习惯使用' time '库。utime库是为像Pico这样的微控制器设计的一个版本——“u”代表“μ”，希腊字母“mu”是“micro”的缩写。如果您忘记并使用了导入时间，不要担心:MicroPython将自动使用utime库。

最后，是时候让LED闪烁了。为此，您需要创建一个循环。重写你的程序，使它与下面的匹配:

```
import machine
import utime

led_onboard = machine.Pin(25, machine.Pin.OUT)

while True:
    led_onboard.value(1)
    utime.sleep(5)
    led_onboard.value(0)
    utime.sleep(5)
```

记住，循环中的行需要缩进四个空格，这样MicroPython就知道它们构成了循环。再次点击Run图标，你会看到LED打开5秒，关闭5秒，然后再次打开——无限循环地不断重复。LED将继续闪烁，直到你点击停止图标取消你的程序和重置你的Pico。

还有另一种方法来处理同样的工作:使用toggle，而不是显式地将LED的输出设置为0或1。删除程序的最后四行，并替换它们，所以看起来像这样:

```
import machine
import utime

led_onboard = machine.Pin(25, machine.Pin.OUT)

while True:
    led_onboard.toggle()
    utime.sleep(5)
```

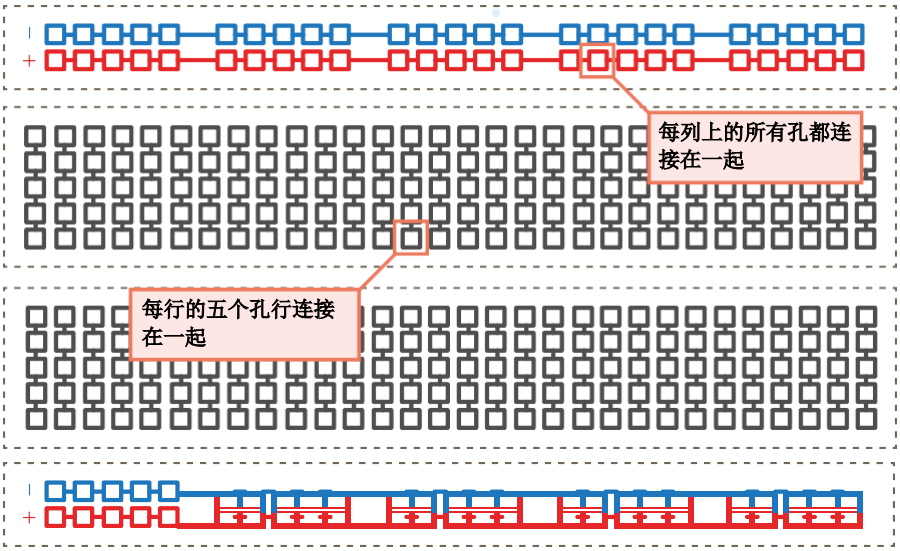
再次运行程序。您将看到与之前相同的活动:机载LED将点亮5秒，然后熄灭5秒，然后在无限循环中再次点亮。不过，这一次，您的程序缩短了两行:您已经优化了它。toggle()可用于所有数字输出引脚，它只是在开和关之间切换:如果引脚当前是开的，toggle()将其关闭;如果它是关闭的，toggle()将它打开。

挑战： 更长的发光时间

你会如何改变你的程序来让LED持续更长的时间?多休息一会儿怎么样?当你还能看到LED开关的时候，最小的延迟是多少?

使用面包板

如果你使用面包板来保存组件并进行电气连接，那么本章中接下来的项目将会更容易完成。



▲ **图4-2：** 面包板上的内部连接

面包片覆盖着孔-间隔，以匹配组件，2.54毫米分开。在这些孔下是金属条，它就像你一直使用到现在的跳线。这些横行运行，大多数板有一个差距，在中间，以分裂成两半。

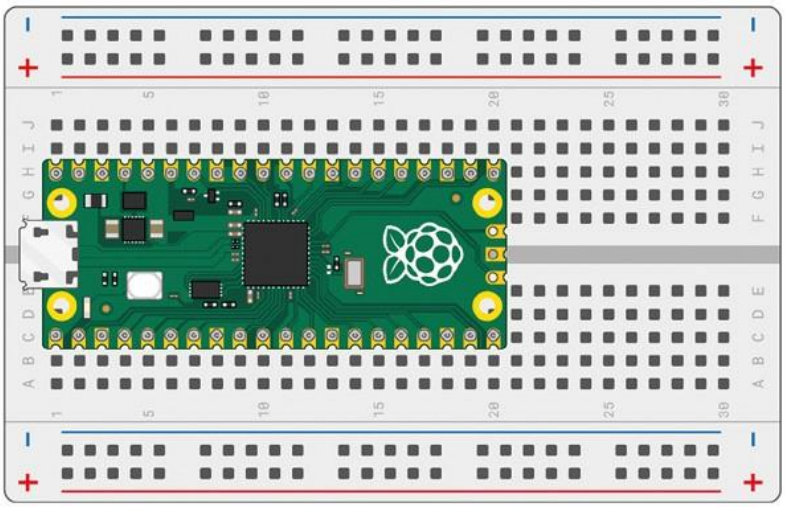
许多面包板顶部都有字母，两侧都有数字。这些可以让你找到一个特定的洞:A1是左上角，B1是直接右边的洞，而B2是从那里向下的一个洞。A1通过隐藏的金属条连接到B1上，但是没有一个标记为1的孔连接到任何标记为2的孔上，除非你自己添加跳线。

更大的面包板在两侧也有条纹状的洞，通常用红色和黑色或红色和蓝色条纹标记。这些是rails的权力,是为了简化布线:你可以从你的一个连接一个单线Pico的地面别针的权力rails——通常标有蓝色或黑色条纹和a -象征——提供一个共同点很多案板上的组件,你可以做同样的如果你的电路需要3.3 V或5 V电源。注:所有用条纹连接的孔都是连接的;间隙表示中断。

把电子元件加到面包板上很简单:只要把它们的引线(伸出来的金属部件)与孔对齐，然后轻轻推，直到元件就位。除了面包板为你做的那些连接，你可以使用公对公(M2M)跳线;从面包板到外部设备的连接，比如树莓派皮科，使用公对母(M2F)跳线。

永远不要尝试将多个元件引线或跳线塞入电路板上的任何一个孔中。记住:洞是成排连接的，除了中间的裂口，所以A1中的一个元件引线是电连接到B1, C1, D1和E1上的任何东西。

把你的Pico推到面包板上，使它跨过中间的空隙，micro USB接口在最上面(如**图4-3**所示)。如果您的面包板有编号，那么左上角的pin 0应该在面包板行中标记为1。在把你的微管压下去之前，确保所有的头销都是正确的位置-如果你弯曲一个销，它可能很难在不折断的情况下再次拉直它。



▲图4-3： 你的Pico被设计为可以合适的装配到面包板上

轻轻向下推Pico，直到排针的塑料部分接触到面包板。这意味着排针的金属部分完全插入，并与面包板有良好的电气接触。

后续步骤：外接 LED

到目前为止，您一直在使用自己的Pico——在其RP2040微控制器上运行MicroPython程序，并打开和关闭板上LED。不过，微控制器通常与外部组件一起使用——您的Pico也不例外。

对于这个项目，你需要一个面包板，公对公(M2M)跳线，一个LED，和一个330 Ω电阻-或接近330 Ω，你有。如果你没有面包板，你可以使用母对母(F2F)跳线，但电路将是不稳定的，容易接触不良导致连接断开。

电阻是至关重要的

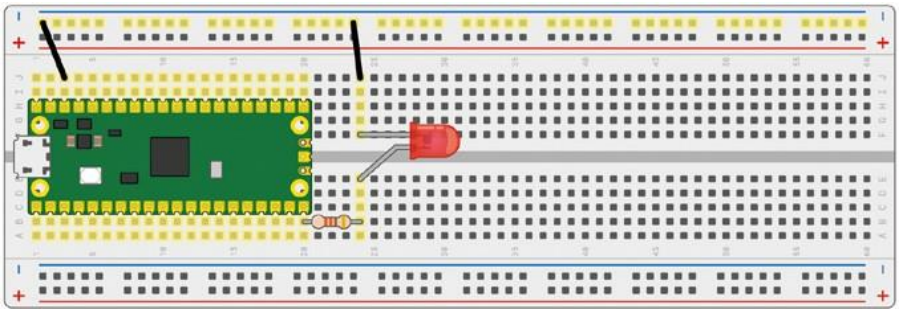
电阻器在这个电路中是一个至关重要的组成部分:它通过限制LED可以引出的电流量来保护你的Pico和LED。如果没有它，LED可能会拉出太多的电流而烧毁自己——或者你的微缩输出。当这样使用时，电阻器称为限流电阻器。确切的电阻值取决于你所使用的LED，但是330 ω对大多数常见的LED都适用。数值越高，LED越暗;数值越低，LED就越亮。

如果没有电流限制电阻，千万不要把LED连接到你的Pico上，除非你知道LED有一个合适的内置电阻。

用手指握住LED:你会看到其中一根导线比另一根长。较长的引线称为正极;较短的引线是负极。正极需要通过电阻连接到你的Pico的GPIO管脚;负极需要连接到一个接地脚。

从连接电阻器开始:取一端(哪一端无关紧要)，并将它插入到面包板的同一行，与你的Pico的GP15引脚在左下角-如果你使用一个编号的面包板，你的Pico插入在最顶部，这应该是第20行。将另一端插入面包板下面的一个空闲行中——我们将使用第24行。

拿起LED，把较长的支脚——正极——推到电阻器的末端所在的同一排。把较短的“负极”放到同一行，但要穿过面包板中间的空隙，这样它就能对齐，但不能与较长的“负极”通电，除非通过“负极”领导本身。最后，在LED短脚的同一行插入一根M2M跳线，然后将它直接连接到你的Pico的接地针上(通过另一个孔)，或者连接到你的面包板的电源孔的负极。如果你把它连接到动力轨道上，完成通过将轨道连接到你的一个微型接地针来实现电路。你的成品电路应该外观如**图4-4**(背面)所示。



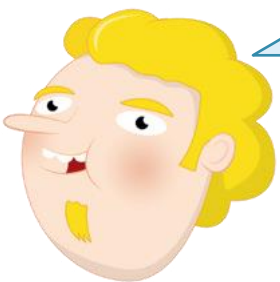
▲ 图4-4： 带LED和电阻器的成品电路

在MicroPython中控制一个外部LED和控制你的Pico内部LED没有什么不同:只是pin号改变。如果你关闭了Thonny，重新打开它并从本章的前面加载**Blink.py**程序。找到:

```
led_onboard = machine.Pin(25, machine.Pin.OUT)
```

编辑引脚号，将其从25(连接到您的Pico内部LED的引脚)改为15(连接外部LED的引脚)。还要编辑您创建的名称:您不再使用车载LED了，所以让它改为led_external。你还必须在程序的其他地方更改名称，直到它看起来像这样:

```
import machine
import utime

led_external = machine.Pin(15, machine.Pin.OUT)

while True:
    led_external.toggle()
    utime.sleep(5)
```



命名规范

您实际上不需要更改程序中的名称:如果您将其命名为led_onboard，它将照样运行，因为真正重要的只有pin号码。但是，当您稍后回到这个程序时，如果有一个名为led_onboard的对象，它会点亮一个外部LED，这将非常令人困惑-所以要养成这样的习惯，确保你的名字与目的相符!

挑战：同时使用多个发光二极管

你能修改程序同时点亮板上和外部led吗?你能写一个程序，当外部LED被关闭时，点亮板上的LED，反之亦然?你能将电路扩展到包含一个以上的外部LED吗?记住，你将需要一个电流限制电阻为每个你使用的LED！

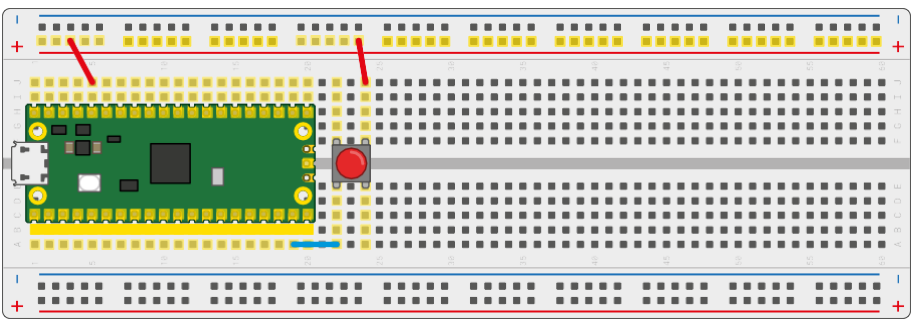
输入：读取按键

输出类似led操作是一回事，但是“GPIO”的“输入/输出”部分意味着你也可以使用引脚作为输入。对于这个项目，您将需要一个面包板，公对公跳线，和一个按钮开关。如果你没有面包板，你可以使用母对母(F2F)跳线，但是按钮将会很难按下，而不会意外地断开电路。

除了Pico之外，从面包中拆除任何其他组件，并从添加按钮开关开始。如果你的按钮只有两个引脚，确保它们在你的Pico下方的面包板上有不同的编号行。如果它有四个引脚，那就把它翻过来，让脚的两边沿着面包板的一排，没有脚的两边在顶部和底部，然后再把它推进去，让它跨在面包板的中间。

将面包板的正功率导轨连接到Pico的 3V3销，并从那里连接到开关的一条脚：然后将另一条脚连接到Pico上的GP14-它就在你用于LED项目的引脚上方，并且应该位于面包板的第19行。

如果你使用带四条脚的按钮，只有使用正确的一对脚，你的电路才会工作：双脚成对连接，因此你需要使用同一侧的两条脚或（如**图 4-5**中所见）对角对立的双脚。



▲ 图4-5： 将四脚按钮连接到 GP14

隐藏的内置电阻

与LED不同，按钮开关不需要限流电阻。它仍然需要一个电阻，尽管：它需要一个所谓的上拉或下拉电阻，这取决于你的电路如何工作。如果没有上拉或下拉电阻，一个输入被称为浮动-这意味着它有一个“噪声”信号，即使你没有按下按钮也会触发。

那么电路中的电阻器在哪里?藏在你的Pico芯片里。就像它有一个板载LED，你的Pico包括一个板载可编程电阻连接到每个GPIO管脚。这些可以在MicroPython中设置下拉电阻或上拉电阻，根据您的电路要求。

有什么区别呢?一个下拉电阻将引脚连接到地面，这意味着当按钮没有被按下时，输入将是0。一个上拉电阻连接引脚到3V3，这意味着当按钮没有按下，输入将是1。

本书中所有的电路都使用下拉模式下的可编程电阻。

加载Thonny (如果你还没有加载过)，然后在新的一行启动一个新程序:

```
import machine
```

接下来，你需要使用machine API设置一个引脚作为输入，而不是输出:

```
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

这与您的LED项目的工作方式相同:创建一个名为“button”的对象，它包括引脚号- GP14，在本例中-并将其配置为一个电阻设置为下拉的输入。不过，创建对象并不意味着它会自己做任何事情——就像之前创建LED对象并不能使LED发光一样。

要真正读取按钮，您需要再次使用machine API——这一次使用value函数读取，而不是设置pin的值。输入以下一行:

```
print(button.value())
```

单击"Run"图标并将程序保存为Button.py-记住以确保它保存在你的Pico MicroPython设备里。你的程序将打印出一个数字：GP14的输入值因为输入端使用的是一个下拉电阻，这个值将是0 -让你知

道按钮没有被按下。

用手指按住按钮，并再次按下运行图标。这一次，您将看到值1打印到Shell：按下按钮就电路导通并改变了从pin读取的值。

要连续读取按钮，您需要在程序中添加一个循环。编辑程序，使其如下所示：

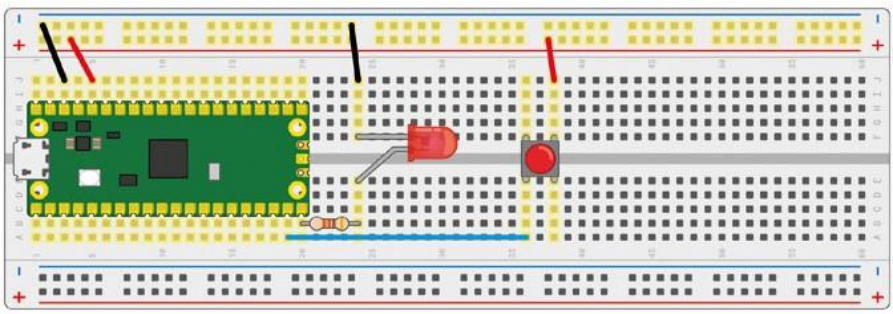
```
import machine
import utime

button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)

while True:
    if button.value() == 1:
        print("You pressed the button!")
        utime.sleep(2)
```

再次单击Run按钮。这一次，在您按下按钮之前不会发生任何事情;当您这样做时，您将看到一条消息打印到Shell区域。同时，延迟也很重要:请记住，您的Pico运行速度要比您的阅读速度快得多，并且没有延迟，即使是简单地按一下按钮，也可以向Shell打印数百条消息!

每次按下按钮你都会看到打印出来的消息。如果你按住按钮超过两秒的延迟，它会每两秒打印一条消息，直到你松开按钮。

输入和输出：同时使用

大多数电路都有不止一个元件，这就是为什么你的Pico有这么多GPIO管脚。现在是时候把你学到的所有东西结合在一起，制作一个更复杂的电路了：一个用按钮开关LED的装置。



▲图 4-6： 带按钮和 LED 的成品电路

实际上，这个电路将前面的两个电路组合成一个。你可能记得你使用pin GP15来驱动外部LED, pin GP14来读取按钮;现在重建你的电路，使LED和按钮同时在面包板上，仍然连接到GP15和GP14(见**图4-6**)。别忘了LED的电流限制电阻!

在Thonny启动一个新程序，并开始导入你的程序需要的两个库：

```
import machine
import utime
```

接下来，设置输入和输出引脚：

```
led_external = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

然后创建一个读取按钮值的循环：

```
while True:
    if button.value() == 1:
```

不过，这一次您将根据输入引脚的值切换输出引脚和连接到它的**LED**，而不是将消息打印到**Shell**。键入以下，记住它将需要缩进8个空格-这是Thonny应该自动处理时，你按下回车在上面的行结束：

```
        led_external.value(1)
        utime.sleep(2)
```

这足以让**LED**亮起来，但当按钮没有被按下时，你还需要再次关闭它。添加以下新行，使用**BACKSPACE**键删除8个空格中的4个——这意味着这行将不是**if**语句的一部分，而是无限循环的一部分：

```
    led_external.value(0)
```

你完成的程序应看起来像这样：

```
import machine
import utime

led_external = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

```
while True:
    if button.value() == 1:
        led_external.value(1)
        utime.sleep(2)
    led_external.value(0)
```

单击Run图标并将程序保存为您的Pico上的Switch.py。起初，什么都不会发生;不过，按下按钮，你就会看到LED灯亮起来。松开按钮;两秒钟后，指示灯再次熄灭，直到你再次按下按钮。

祝贺你：你已经建立了你的第一个电路，它根据一个输入引脚控制另一个输出引脚——我们在构建模块程序的路上跨越了一大步!

挑战：构建起来

你能修改你的程序，使它既点亮LED又向Shell输出状态信息吗?你需要做什么改变，才能在按钮未按下时让LED保持亮着，在按钮按下时让LED关闭?你能在电路中增加更多的按钮和led吗?

第五章

交通灯控制器

创建自己的迷你人行横道系统使用多个LED和按钮

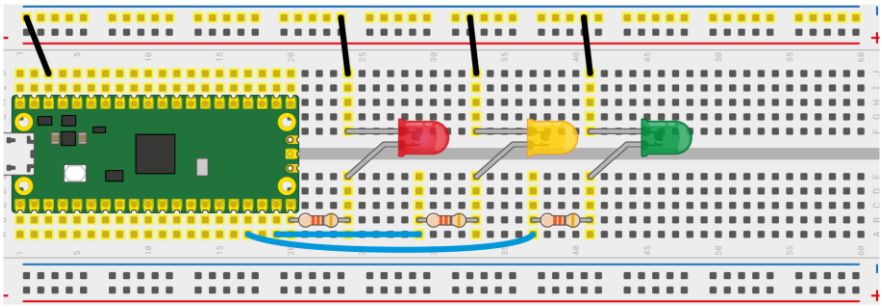
控制器几乎存在于你日常使用的所有电子产品中，包括交通灯。交通灯控制器是专门建造系统,改变定时器的灯,为行人寻求交叉手表,甚至可以调整灯光的时机取决于有多少流量,与附近的交通灯系统,以确保整个交通网络保持顺畅。

虽然构建大型交通管理系统是一个非常高级的项目，但构建由Raspberry Pi Pico驱动的微型模拟器本身就很简单。通过这个项目，您将看到如何控制多个led，设置不同的时间，以及如何在程序的其余部分继续运行时监控一个按钮输入，使用一种称为线程的技术。

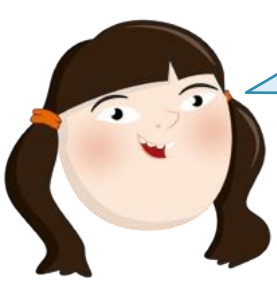
对于这个项目，你需要你的Pico，面包板，红色与黄色或琥珀色和绿色的LED，三个 330 Ω电阻器，一个有源压电蜂鸣器，一组公对公（M2M）跳线的选择。你还需要一根微型USB 电缆，并将Pico连接到你的树莓派或其他运行 Thonny MicroPython IDE 的计算机。

一个简易的交通灯

首先建立一个简单的红绿灯系统电路，如**图5-1**所示。拿起你的红色LED灯，把它插到面包板上，让它跨过中间的鸿沟。使用一个330 Ω电阻，如果你需要做一个较长的连接，跳线从你的Pico左下角的引脚连接到LED较长的引脚——正极上，从顶部看到的micro USB电缆，GP15。如果您正在使用一个编号的面包板，并将Pico插入到最顶部，那么这将是面包板第20行。



▲图5-1： 基本的三色红绿蓝交通灯系统



警告

请始终记住，一个LED需要一个限流电阻才能连接到你的Pico。如果你连接一个没有电流限制电阻的LED，最好的结果是LED会烧坏，不再工作;最坏的结果是，它可能对您的Pico造成同样的后果。

拿一根跳线，把红色LED的短脚——负极——连接到你面包板的ground接地引脚。取另一跟跳线，并将接地轨连接到您的Pico的接地(GND)引脚-在**图5-1**中，我们已经在面包板的第三行使用了接地引脚。

你现在有一个LED连接到你的Pico，但一个真正的红绿灯至少再加两个总共三个指示灯：红灯告诉交通停止，琥珀色或黄色灯告诉交通灯即将改变，绿色LED告诉交通可以同行。

把你的琥珀色或黄色LED和连接线以红色LED同样的安装方式连接到你的Pico，确保短脚连接到面包板的接地端，你已经有了330 Ω电阻可以保护到它。不过，这一次，通过电阻器将更长的脚连接到连接红色LED 的GP15引脚旁边的GP14。

最后，把绿色的LED用同样的方式连接起来-记的要使用330 Ω电阻连接到引脚GP13。这次连接的引脚GP13不是GP14旁边的引脚——那个引脚是地面(GND)引脚，如果你仔细观察你的Pico，你可以看到:接地引脚的焊盘都是方形的，而其他引脚的焊盘都是圆形的。

完成后，你的电路应与**图5-1**相匹配：红色、黄色或琥珀色以及绿色LED，全部通过单独的 330 Ω电阻器连接到Pico上的不同GPIO 引脚，并通过面包板的地线连接到共同的接地端。

要编程你的红绿灯，连接你的Pico到你的树莓派或其他计算机并且加载Thomny。创建一个新的程序，然后开始导入machine 库，以便你可以控制你的GPICO引脚：

```
import machine
```

你还需要导入utime 库，以便你可以在亮灯与灭灯之间添加延迟：

```
import utime
```

在你控制你Pico的GPIO口的任何程序里，在使用前都你将需要对它进行设置：

```
led_red = machine.Pin(15, machine.Pin.OUT)
led_amber = machine.Pin(14, machine.Pin.OUT)
led_green = machine.Pin(13, machine.Pin.OUT)
```

这些线将GP15、GP14和GP13引脚设置为输出，每个引脚都有一个描述性的名称，以便于阅读代码:' led '，这样你就知道这些引脚控制一个led，然后是led的颜色。

真正的红绿灯不会一闪而过然后停下来，它们会不停地开着，即使那里不塞车，但为了让你的程序做同样的事情，你需要建立一个无限循环：

```
while True:
```

下面的每一行都需要缩进四个空格，这样MicroPython就知道它们是循环的一部分;当你按下回车键时，Thonny会自动为你缩进行。

```
    led_red.value(1)
    utime.sleep(5)
    led_amber.value(1)
    utime.sleep(2)
    led_red.value(0)
    led_amber.value(0)
    led_green.value(1)
```

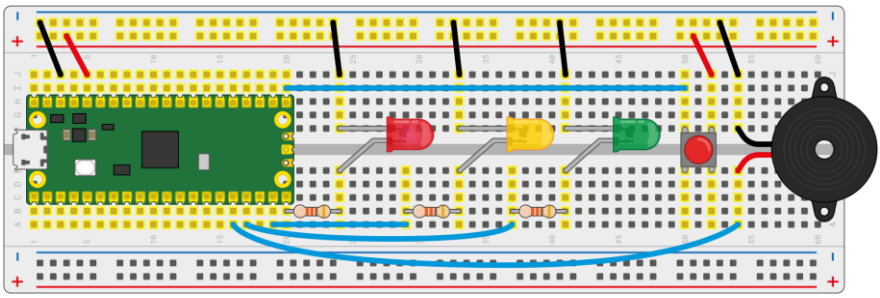
```
utime.sleep(5)
led_green.value(0)
led_amber.value(1)
utime.sleep(5)
led_amber.value(0)
```

单击Run图标并将程序保存到Pico中，文件名为Traffic_Lights.py。观察LED灯:首先红色的LED灯会亮起来，告诉交通停止;接下来，琥珀色LED将会亮起，警告司机信号灯即将改变;接下来，两个LED都关闭，绿色LED亮起，让车辆知道它可以通过;然后绿色的LED熄灭，黄色的LED亮起来，警告司机信号灯又要变了;最后，琥珀色LED熄灭，回路从开始重新启动，红色LED亮起。

这个显示状态会一直循环直到你按下停止按钮，因为它形成了一个无限循环。它是基于现实世界中英国和爱尔兰交通控制系统中使用的交通灯模式，但它是加速的——现实中给汽车5秒钟的时间通过交通灯是不能让交通畅通无阻的!

然而，真正的红绿灯不仅仅是道路车辆的红绿灯：它们也在那里保护行人，让他们有机会安全地穿过繁忙的道路。在英国，最常见的类型这些灯被称为*行人操作用户友好型智能交叉口*或*海雀交叉口*。

要将红绿灯变成海雀交叉口，你需要两样东西：一个按钮开关，这样行人就可以要求车灯让他们过马路：一个蜂鸣器，所以行人知道什么时候轮到他们穿越。将这些连接到你的面包板，如**图5-2**，与开关连接固定GP16和3V3的面包板通道，和蜂鸣器连线固定在GP12和你的面包板的地线通道。



▲ **图5-2**：带蜂鸣的红绿灯系统

如果你再次运行你的程序，你会发现按钮和蜂鸣器没有什么反应。那是因为你还没有告诉你的程序如何使用它们。在 Thonny中，请返回编辑控制LED 的行，并在下面添加以下两条新行：

```
button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)
buzzer = machine.Pin(12, machine.Pin.OUT)
```

这将引脚GP16上的按钮设置为输入，控制蜂鸣器引脚的引脚GP12上的设置为输出。请记住，您的Raspberry Pi Pico有内置的可编程电阻的输入，我们为本书中的项目设置为下拉模式。这意味着引脚的电压被拉到0 V(它的逻辑电平是0)，除非它连接到3.3V电源(在这种情况下，它的逻辑电平将是1，直到断开)。

接下来，你需要一种方法来让程序不断监控按钮的价值。以前，你的所有程序都通过一系列说明一步一步地工作-一次只做一件事。你的红绿灯程序没有什么不同：当它运行时，MicroPython会一步一步地浏览你的指示，打开和关闭 LED。

对于一套基本的红绿灯，这就足够了;不过，对于海雀交叉口，你的程序需要能够记录按钮是否以不中断红绿灯的方式按下。要做到这一点，你需要使用到一个新的库：_thread。返回到程序的开始部分，在那里你导入machine和 utime库，并导入_thread库：

```
import _thread
```

一个线程或执行线程实际上是一个小的、部分独立的程序。您可以将前面编写的控制灯的循环视为程序的主线程，并使用_thread库创建一个同时运行的额外线程。

可视化线程的简单方法是将每个线程都想象成厨房里的独立人员：当厨师准备主菜时，其他人正在制作酱汁。 目前，你的程序只有一个线程 -控制红绿灯的线程。然而，为Pico提供算力的RP2040微控制器有两个处理核心——意思是，就像厨房中的厨师和副厨师一样，您可以同时运行两个线程来完成更多的工作。

在制作另一个线程之前，你需要一种方法让新线程将信息传回主线程-并且你可以使用*全局变量*做到这一点。在此之前，你一直在处理的变量称为*局部变量*，并且仅在程序的一个部分工作：一个全局变量在任何地方都有效，这意味着一个线程可以更改值，另一个线程可以检查它是否已更改。

首先，您需要创建一个全局变量。在buzzer = 行下面，添加以下内容：

```
global button_pressed
button_pressed = False
```

这将把button_pressed设置为一个全局变量，并给它一个默认值False——意思是当程序启动时，按钮还没有被按下。下一步是定义

你的线程，通过添加下面的行-添加一个空行，如果你想，使你的程序更具有可读性:

```
def button_reader_thread():
    global button_pressed
    while True:
        if button.value() == 1:
            button_pressed = True
```

添加的第一行定义了线程，并给它一个名称来描述它的用途:读取按钮输入的线程。就像在编写循环时，MicroPython需要将线程中包含的所有内容缩进4个空格——这样它就知道线程从哪里开始和结束。

下一行让MicroPython知道您将更改全局button_pressed变量的值。如果您只想检查该值，则不需要这一行—但是没有这一行，您就不能对变量进行任何更改。

接下来，您设置了一个新的循环——这意味着接下来需要一个新的四空格缩进，总共是八个缩进，这样MicroPython就知道循环是线程的一部分，下面的代码也是循环的一部分。这个嵌套代码在多个水平MicroPython缩进是很常见的,和Thonny将尽最大的努力来帮助你通过自动添加一个新的水平在每次需要的,但由你记得删除空间增加当你完成一个特定的项目。

下一行是一个条件语句，用于检查按钮的值是否为1。因为你的Pico使用一个内部下拉电阻，当按钮没有被按下时，读取的值是0—表示在条件下的代码永远不会运行。只有当按钮被按下时，线程的最后一行才会运行:这一行将button_pressed变量设置为True，让程序的其余部分知道按钮已被按下。

您可能注意到，在线程中没有任何东西可以将button_pressed变量重置回当按钮被按下后释放时为False。这是有原因的:虽然你可以在交通灯周期的任何时候按下海雀过马路的按钮，但它只有在交通灯变红、你可以安全过马路时才会生效。你的新线程需要做的就是在按钮被按下时改变变量;当行人安全过马路时，主线程会将其重置为False。

定义一个线程并不会使它运行:可以在程序中的任何地方启动一个线程，并且需要明确地告诉_thread库何时启动线程。与运行普通代码不同，运行线程不会停止程序的其余部分:当线程启动时，MicroPython将继续运行程序的下一行，即使它运行新线程的第一行。

在你的线程下面新建一行，删除所有Thonny为你自动添加的缩进，如下所示:

```
_thread.start_new_thread(button_reader_thread, ())
```

这告诉_thread库启动前面定义的线程。此时，线程将开始运行，并迅速进入循环——每秒检查按钮数千次，看它是否被按下。与此同时，主线程将继续执行程序的主要部分。

现在点击Run按钮。你会看到交通灯和以前一样，没有延迟或停顿。但是，如果您按下按钮，什么也不会发生——因为您还没有添加代码来实际响应按钮。

转到你的主循环的开始，在True:的正下方，添加以下代码——记住要注意嵌套缩进，并删除Thonny添加的缩进，当它不再需要时：

```
    if button_pressed == True:
        led_red.value(1)
        for i in range(10):
            buzzer.value(1)
            utime.sleep(0.2)
            buzzer.value(0)
            utime.sleep(0.2)
        global button_pressed
        button_pressed = False
```

这段代码检查button_pressed全局变量，以查看自上次循环运行以来，按钮开关是否在任何时候被按下。如果有，就像你之前的按钮阅读线程报告的那样，它开始运行一段代码，首先打开红色LED灯来阻止交通，然后蜂鸣器响十次——让行人知道该过马路了。

最后，最后两行将button_pressed变量重置为False—因此下次循环运行时，除非再次按下按钮，否则不会触发行人过马路代码。您将看到，在条件语句中不需要使用global button_pressed来检查变量的状态;只有当您想要更改变量并使该更改影响程序的其他部分时，才需要使用它。

你完成的程序应该像这样：

```
import machine
import utime
import _thread

led_red = machine.Pin(15, machine.Pin.OUT)
led_amber = machine.Pin(14, machine.Pin.OUT)
```

```
led_green = machine.Pin(13, machine.Pin.OUT)
button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)
buzzer = machine.Pin(12, machine.Pin.OUT)

global button_pressed
button_pressed = False

def button_reader_thread():
    global button_pressed
    while True:
        if button.value() == 1:
            button_pressed = True

_thread.start_new_thread(button_reader_thread, ())

while True:
    if button_pressed == True:
        led_red.value(1)
        for i in range(10):
            buzzer.value(1)
            utime.sleep(0.2)
            buzzer.value(0)
            utime.sleep(0.2)
        global button_pressed
        button_pressed = False
    led_red.value(1)
    utime.sleep(5)
    led_amber.value(1)
    utime.sleep(2)
    led_red.value(0)
    led_amber.value(0)
    led_green.value(1)
    utime.sleep(5)
    led_green.value(0)
    led_amber.value(1)
    utime.sleep(5)
    led_amber.value(0)
```

点击Run图标。首先，程序将正常运行：交通灯将按照通常的模式亮和关。按下按钮开关:如果程序当前在它的循环过程中，在到达终点并再次循环之前不会发生任何事情，这时红灯会

变亮，蜂鸣器会提示你可以在道路上安全通过了。

过马路的条件部分代码运行在前面编写的代码的灯打开和关闭在循环模式:完成之后,模式将开始像往常一样的红色LED点亮待进一步点燃了五秒的时间时,蜂鸣器。这模仿了真正的海雀过马路的工作方式:即使蜂鸣器不再响了，红灯仍然亮着，所以任何在蜂鸣器发出时开始过马路的人都有时间在车辆允许通行之前到达另一边。

让交通灯再循环几次，然后再按下按钮触发另一个路口。恭喜你:你已经建立了你自己的海雀交叉口!

挑战：你能改进它吗？

你能改变程序，让行人更长时间通过吗？您能否找到有关其他国家红绿灯模式的信息，并重新编程以匹配您的交通信号灯？你能加一个第二个按钮， 这样路另一边 的行人也可以提示他们想过马路吗？

第六章

反应游戏

使用LED和按钮为一两个玩家构建一个简单的反应定时游戏

微控制器不仅存在于工业设备中:它们为包括玩具和游戏在内的许多家庭电子产品提供算力。在这一章中，你将创建一个简单的反应计时游戏，看看你的朋友中谁会在灯熄灭时第一个按下按钮。

对反应时间的研究被称为心理计时学，虽然它是一门硬科学，但它也是许多基于技能的游戏的基础——包括你即将制作的游戏。你的反应时间——你的大脑来处理的时间需要做一些和发送信号,使事情发生——以毫秒计:人类的平均反应时间大约是200 - 250毫秒,但有些人喜欢得更快的反应时间,给他们一个真正的优势在游戏中!

对于这个项目，你需要你的Pico;试验板;任何颜色的LED灯;单个330 Ω电阻;两个按钮开关;和一组公对公(M2M)跳线。您还需要一根micro USB线，并将您的Pico连接到您的树莓派或其他运行Thony micropython IDE的计算机。

单人游戏

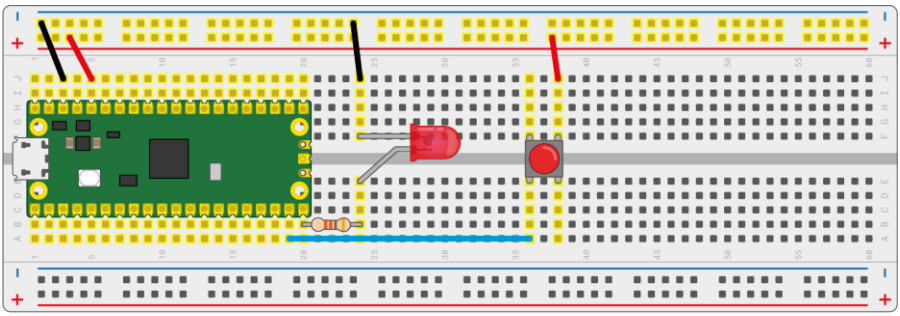
首先，把LED灯放在面包板上，这样它就会横跨中间的分界线。记住，led只有在正确的位置才能工作:确保你知道哪条引脚是较长的阳极，哪条引脚是较短的阴极。

使用一个330 Ω的限流电阻，保护LED和你的Pico，将LED的长引脚连接到你的Pico左下角的GP15，从顶部看，micro USB电缆最上面。如果您正在使用一个编号的面包板，并将Pico插入到最顶部，那么这将是面包板第20行。

警告

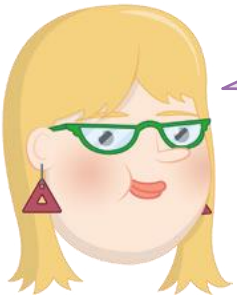
值得重复的是，LED 始终需要电流限制电阻器才能连接到您的Pico。如果没有电阻器，LED 可能会烧毁-或者您的Pico可能会损坏。

拿一根跳线，把LED的短引脚连接到面包板的地面轨道上。取另一个，并将接地轨连接到您的Pico的接地(GND)引脚-在图6-1中，我们已经在面包板的第三行使用了接地引脚。



▲图 6-1： 单人反应游戏

接下来，添加如图6-1所示的按钮开关。拿一个跳线和连接其中一个按钮的开关引脚GP14，右边的引脚你用你的LED。用另一根跳线连接另一条引脚——如果你用的是四条引脚的按钮开关，那就用另一条引脚连接到面包板的动力栏上。最后，拿最后一根跳线，把电源轨连接到你的Pico的3V3引脚上。



为什么要连接3 v3?

请记住，开关，像led一样，需要电阻器才能正确工作，而且你的Pico上有可编程电阻器GPIO管脚。在这本书的项目中，我们将它们设置为下拉电阻，这意味着pin必须被拉高当按钮开关被按下-这就是是为什么接线开关要通过面包板的电源通道连接到3V3 pin。

现在你的电路已经具备了作为一个简单的单人游戏所需要的一切:LED是输出设备，取代了你通常使用的电视和游戏机;按钮开关为控制器;而你的Pico是游戏主机，尽管它比你通常看到的要小得多!

现在你需要真正编写游戏。和往常一样，把你的皮科和覆盆子连起来Pi或其他电脑和加载Thonny。创建一个新程序，通过导入机器库来启动它，这样你就可以控制你的Pico的GPIO引脚了:

```
import machine
```

你还需要 utime 库:

```
import utime
```

此外，你将需要一个新的库:urandom，它处理创建随机数-使游戏变得有趣的关键部分，并在这个游戏中使用，以防止曾经玩过它的玩家简单地倒数一个固定的秒数，从点击运行按钮。

接下来，设置您正在使用的两个引脚:GP15用于LED, GP14用于按钮开关

```
led = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

在前面的章节中，您已经在主程序或单独的线程中处理了按钮切换。不过，这一次您将采用一种不同的、更灵活的方法:中断请求(IRQs)。

这个名字听起来很复杂，但其实很简单:想象一下，你正在一页一页地阅读一本书，有人走过来问你一个问题。那个人在执行一个打断请求:要求你停止正在做的事情，回答他们的问题，然后让你继续读你的书。

一个MicroPython中断请求以完全相同的方式工作:它允许某些东西(在这种情况下是按下按钮开关)中断主程序。在某些方面，它和线程很相似，在主程序之外有一段代码。然而，与线程不同的是，代码不是持续运行的:它只在中断被触发时运行。
首先定义中断的处理程序。这个被称为回调函数的代码在中断被触发时运行。与任何嵌套代码一样，处理程序的代码-第一行之后的所有内容-需要缩进四个空格;Thonny会自动为你做这个。

```
def button_handler(pin):
    button.irq(handler=None)
    print(pin)
```

这个简单的两行处理程序首先关闭中断，这样它只触发一次，然后打印有关触发中断的pin的信息。现在这不是很重要-你只有一个引脚配置为输入，GP14，所以中断总是来自那个引脚-但让你可以很容易地测试你的中断。
继续你下面的程序，记住删除thony自动创建的四行缩进-下面的代码不是处理程序的一部分:

```
led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
```

这段代码你马上就会熟悉:第一行将LED，连接到GP15引脚上，打开;下一行暂停程序;最后一行再次关闭LED灯——玩家按下按钮的信号。然而，它并没有使用固定的延迟，而是利用urandom库将程序暂停5到10秒——“均匀”部分指的是这两个数字之间的均匀分布。
然而，目前还没有什么东西在等待着按钮被按下。您需要为此设置中断，方法是在b处输入以下一行

```
button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

设置中断需要两个东西:触发器和处理程序。触发器告诉你的Pico它应该寻找什么作为中断它正在做的事情的有效信号;那个联络人，你在您的程序前面定义的，是中断被触发后运行的代码。
在这个程序中，你的触发器是IRQ_RISING:这意味着当中断被触发时由于内置的下拉电阻，引脚的默认状态为低，它的值从低上升到高，当按下连接3V3的按钮时。触发IRQ_FALLING会做相反的事情:当引脚从高到低触发中断。在你的电路中，IRQ_RISING

一旦按下按钮就会触发;IRQ_FALLING只在按钮被触发时触发之后发布的。

IRQS 的上升和下降

如果你需要写一个程序，在一个引脚改变时触发一个中断，而不关心它是上升还是下降，你可以使用管道或垂直条符号(|)组合两个触发器:

button.irq(trigger=machine.Pin.IRQ_RISING |

machine.Pin.IRQ_FALLING, andler=button_handler)

你的程序应该看起来像这样：

```
import machine
import utime
import urandom

led = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)

def button_handler(pin):
    button.irq(handler=None)
    print(pin)

led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

单击"运行"按钮，并将程序保存到你的Pico**Reaction_Game.py**。你会看到LED灯亮起来：这是你的信号，准备与你的手指在按钮上。当 LED 熄出时，尽可能快地按下按钮。

当你按下按钮时，它会触发你之前编写的处理程序代码。查看Shell区域：你将看到Pico打印了一条消息，确认中断是由 pin GP14 触发的。你还会看到另一个细节：**mode=IN** 告诉你引脚被配置为输入。不过，这个信息并没有给游戏造成多大的反响：为此，你需要一种方法来加快玩家的反应速度。首先从按钮处理程序中删除打印(pin)行-你不再需要它了。

转到程序的底部并添加一条新行，就在你设置中断的位置的正上方：

```
timer_start = utime.ticks_ms()
```

这创建了一个名为timer_start的新变量，并填充了 utime.ticks_ms（）函数的输出，该函数计算自utime 库开始计数以来已过的毫秒数。这提供了一个参考点：在 LED 熄灭之后和中断触发器准备好读取按钮之前的时间。

接下来，回到你的按钮处理程序，添加以下两行，记住它们需要缩进四个空格，这样MicroPython就知道它们是嵌套代码的一部分：

```
    timer_reaction = utime.ticks_diff(utime.ticks_ms(), timer_start)
    print("Your reaction time was " + str(timer_reaction) +
" milliseconds!")
```

第一行创建了另一个变量，这一次是中断实际触发的时候——换句话说，就是按下按钮的时候。但是，它不像以前那样简单地从utime.ticks_ms()中读取数据，而是使用utime.ticks_diff()——这个函数提供了触发这行代码的时间与变量timer_start中保存的参考点之间的差异。

他的第二行打印结果，但使用连接来很好地格式化它。文本(或字符串)的第一个比特告诉用户后面的数字是什么意思;+表示接下来的内容应该与该字符串一起打印。在本例中，接下来是timer_reaction变量的内容——取计时器的参考点与按下按钮和触发中断之间的差值(以毫秒为单位)。

最后，最后一行连接了另一个字符串，这样用户就知道这个数字是以毫秒为单位计算的，而不是以秒或微秒等其他单位计算的。注意空格:您将看到在' was '之后和第一个字符串的结束引号之前有一个尾随空格，在第二个字符串的开始引号之后和单词' milliseconds '之前有一个前导空格。如果没有这些，连接字符串将打印类似“你的反应时间是323毫秒”的内容。

你的程序现在应该看起来像这样：

```
import machine
import utime
import urandom

led = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)

def button_handler(pin):
    button.irq(handler=None)
    timer_reaction = utime.ticks_diff(utime.ticks_ms(), timer_start)
```

```
        print("Your reaction time was " + str(timer_reaction) +
" milliseconds!")

    led.value(1)
    utime.sleep(urandom.uniform(5, 10))
    led.value(0)
    timer_start = utime.ticks_ms()
    button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

再次点击运行按钮，等待LED熄灭，然后按下按钮。这一次，你将看到一条线，告诉你按下按钮的速度，而不是触发中断的针的报告，这是对你反应时间的测量。再次点击运行按钮，看看你是否可以按下按钮更快的这次-在这个游戏中，你正在尝试尽可能低的分数!

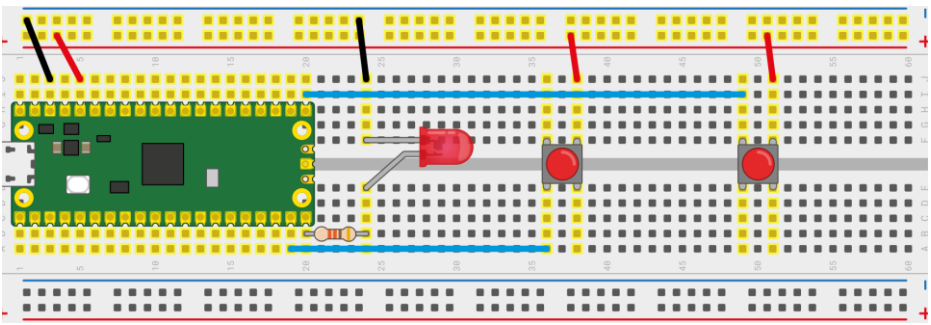
挑战： 定制

你能否调整游戏，让LED长时间保持亮着?在短时间内保持光亮怎么样?你可以个性化的消息打印到Shell区域，并添加第二条消息祝贺玩家吗?

双人游戏

单人游戏很有趣，但是让你的朋友参与进来会更好。你可以先邀请他们玩你的游戏，比较你的高分或低分，看看谁的反应最快。然后，你可以修改你的游戏，让你的头对头!

首先在你的电路中添加第二个按钮。和第一个按钮一样将它连接起来，一条引脚连接到你的面包板的电源栏，另一条引脚连接到GP16——从连接LED的GP14到整个板的GP16，在你的Pico的另一个角落。



▲ **图 6-2：** 双人反应游戏的电路

确保两个按钮之间有足够的距离，以便玩家能够将手指放在按钮上。完成后的电路如图6-2所示。

虽然第二个按钮现在已经连接到Pico，但它还不知道如何使用它。回到你在Thonny的程序中，找到你设置第一个按钮的地方。在这一行下面，添加：

```
right_button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

您将注意到，名称现在指定了您正在使用的按钮：面包板上的右手边按钮。为了避免混淆，请编辑上面的一行，这样你就可以清楚地看到，原来黑板上唯一的按钮现在变成了左边的按钮：

```
left_button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

你还需要在程序的其他地方进行相同的更改。转到按钮处理器功能并更改行：

```
button.irq(handler=None)
```

...因此，它读取：

```
left_button.irq(handler=None)
```

接下来，为第二个按钮添加一个新行——记住，像处理程序中的所有代码一样，它需要缩进四个空格，这样MicroPython就知道它是函数的一部分：

```
right_button.irq(handler=None)
```

向下滚动到程序的底部，并更改设置中断触发器的行，以便它进行读取：

```
left_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

同样，在它下面添加另一行，以在新按钮上设置中断触发器：

```
right_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

你的程序现在应该看起来像这样：

```
import machine
import utime
```

```
import urandom

led = machine.Pin(15, machine.Pin.OUT)
left_button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
right_button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)

def button_handler(pin):
    left_button.irq(handler=None)
    right_button.irq(handler=None)
    timer_reaction = utime.ticks_diff(utime.ticks_ms(), timer_start)
    print("Your reaction time was " + str(timer_reaction) +
" milliseconds!")

led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
right_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
left_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

点击运行图标，等待LED熄灭，然后按下左边的按钮开关:你会看到游戏和之前一样，将你的反应时间打印到Shell区域。再次点击运行图标，但这一次，当LED熄灭时，按右边的按钮:游戏将照常工作，打印你的反应时间。

中断和中断处理函数

您创建的每个中断都需要一个处理程序，但单个处理程序可以处理任意数量的中断。在这个程序中，有两个中断都指向同一个处理程序——这意味着无论触发哪个中断，它们都将运行相同的代码。不同的程序可能有两个处理程序，让每个中断运行不同的代码——这完全取决于您需要您的程序做什么。

为了让游戏更精彩一点，你可以让它报告两个玩家中哪一个是第一个按下按钮的。回到程序的顶部，就在下面，你可以设置 LED 和两个按钮，并添加以下内容：

```
fastest_button = None
```

这将设置一个新变量fastest_button，并将其初始值设置为None——因为还没有按下任何按钮。接下来，到按钮处理程序的底部，删除处理计时器和打印的两行，然后用以下代码替换它们：

```
    global fastest_button
    fastest_button = pin
```

记住，这些行需要缩进四个空格，以便MicroPython知道它们是函数的一部分。这两条线让你的函数的变化,而不仅仅是阅读,fastest_button变量,并将其设置为包含的细节销触发中断——相同的细节你的游戏印刷外壳区域本章早些时候,包括触发销的数量。

现在直接转到程序的底部，并添加以下两个新行：

```
while fastest_button is None:
    utime.sleep(1)
```

这创建了一个循环，但它不是一个无限循环:这里，您告诉MicroPython只有在fastest_button变量仍然为0时才在循环中运行代码——这个值是在程序开始时初始化的。实际上，这会暂停程序的主线程，直到中断处理程序更改了变量的值。如果两个玩家都没有按下按钮，程序就会暂停。

最后，您需要一种方法来确定哪位选手获胜，并向他们表示祝贺。在程序的底部输入以下代码，确保删除了Thonny在第一行为你创建的四个空格缩进——这些行不构成循环的一部分：

```
if fastest_button is left_button:
    print("Left Player wins!")
elif fastest_button is right_button:
    print("Right Player wins!")
```

第一行设置了一个“if”条件，用于查看fastest_button变量是否为left_button——这意味着IRQ是由左手按钮触发的。如果是这样，它将打印一条消息——下面的一行缩进四个空格，以便MicroPython知道只有当条件为真时才应该运行它——祝贺左边的玩家，他的按钮连接到GP14。

下一行不应该缩进，它将条件扩展为' elif ' - ' else if '的缩写，意思是'如果第一个条件不成立，检查这个条件下'。这一次，它将查看fastest_button变量是否为right_button——如果是，则打印一条消息祝贺右边的玩家，该玩家的按钮已连接到GP16。

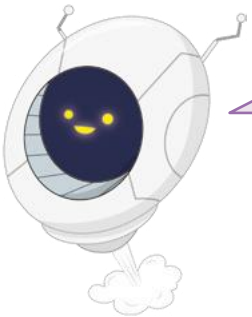
你完成的程序应看起来像这样：

```
import machine
import utime
import urandom

led = machine.Pin(15, machine.Pin.OUT)
left_button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
right_button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)

fastest_button = None

def button_handler(pin):
    left_button.irq(handler=None)
    right_button.irq(handler=None)
    global fastest_button
    fastest_button = pin

led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
left_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
right_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
while fastest_button is None:
    utime.sleep(1)
if fastest_button is left_button:
    print("Left Player wins!")
elif fastest_button is right_button:
    print("Right Player wins!")
```

按下运行按钮，等待LED熄灭-但不要按下任何一个按钮开关。您将看到Shell区域仍然是空白的，并且不会返回>>>提示符;这是因为主线程仍在运行，处于您创建的循环中。

现在按左手按钮，连接到pin GP14。您将看到一条祝贺您的消息打印到Shell上——您的左手是获胜者!再次单击Run，并尝试在LED熄灭后按下右手按钮:您将看到另一条消息打印出来，这一次祝贺您的右手。再次点击Run，这次每个按钮上都有一个手指:同时按下它们，看看你的右手还是左手更快!

现在你已经创造了一个双人游戏，你可以邀请你的朋友一起玩，看看你们谁的反应速度最快!



挑战：时间安排

你能修改打印的消息吗?你是否可以添加第三个按钮，这样三个人可以同时玩游戏?你可以添加的按钮数量是否有上限?你能否将计时器添加到程序中，以便告诉获胜玩家他们的反应时间有多快?

第七章

窃贼报警器

使用运动传感器检测入侵者和用闪烁的灯光和警笛发出警报

微控制器在现实世界的另一个用途是在报警系统中。从早上叫你起床的闹钟到火灾警报、防盗警报，甚至是核电站出现问题时发出的警报，微控制器帮助保护我们所有人的安全。

在这一章你要构建自己的防盗报警器,以完全相同的方式工作作为一个商业版本:一个特殊的运动传感器不断关注任何人进入房间时,他们不应该,和闪烁光同时发出警报,提醒人们入侵。无论你是在保护银行金库，还是只是想防止兄弟姐妹们出现在你的房间里，防盗报警器肯定会派上用场。

对于这个项目，你需要你的Pico;试验板;任何颜色的LED灯;一个330 Ω电阻;有源压电蜂鸣器;一个或多个HC-SR501被动红外(PIR)传感器;和一对公对公(M2M)和公对母(M2F)跳线的选择。您还需要一根微型USB线，并将您的Pico连接到您的树莓派或其他运行thonymicropython IDE的计算机。

HC-SR501 PIR 传感器

在前面的章节中，您已经使用了按钮开关形式的简单输入组件。这一次，你将使用一种专门的输入被称为被动红外传感器或PIR。有数百种不同的PIR传感器可用;HC-SR501是低成本，高性能，并与您的Pico完美地工作。

如果你选择了不同型号的传感器，看看它的文档，仔细检查哪个是哪个针脚;还要确保它运行在3V3逻辑级别，就像你的Pico -如果你连接一个使用更高电压的传感器，如12v传感器，它会损坏你的Pico无法修复。一些传感器可能需要一个小开关或跳线来改变逻辑电压电平之间的移动;这将在文档中说明。

被动式红外传感器被设计用来检测运动，特别是人和其他生物的运动。它的工作原理有点像照相机，但它不是捕捉可见光，而是寻找以红外辐射的形式从活体发出的热量。它被称为被动红外传感器，而不是主动红外传感器，因为就像相机传感器一样，它自己不发出任何红外信号。

实际的传感器被埋在一个塑料镜头下，通常形状像半个球。镜头在技术上并不是传感器工作所必需的，但可以提供更宽的视野(FOV);如果没有透镜，PIR传感器只能看到传感器正前方一个非常窄的角度的运动。这种镜头可以从更宽的角度吸收红外线，这意味着一个PIR传感器就可以观察到房间里大部分的移动情况。

在商业防盗报警系统中，PIR传感器只是使用的传感器之一;其它的传感器还包括:能告知窗户何时被打碎的碎玻璃传感器、能监测门是开着还是关着的磁传感器、能捕捉窃贼脚步声的声音传感器以及能告知门锁是否被撬开的振动传感器。然而，一个简单的PIR传感器对于低安全级别的地方就足够了——想想接待室，而不是银行金库。

现在拿起你的HC-SR501传感器看一看。首先要注意的是，它有自己的电路板，很像你的Pico -只是更小。除了传感器和镜头，还有其他几个组件:驱动传感器的黑色小集成电路(IC)、一些电容和微小的表面安装电阻。你也可以看到一些小的电位器，你可以用螺丝刀拧来调整传感器的灵敏度，以及当触发时它保持激活的时间;暂时让这些保持原样。

你还会看到三个公排针，和你的Pico底部的别针一模一样。但是，您不能直接将它们放入面包板，因为面包板上的组件会碍事。相反，取三根公对母(M2F)跳线，将母线端插入HC-SR501的引脚上。

接下来，拿起公的一端，把它们连接到面包板和你的Pico上。在将传感器连接到Pico时，需要检查传感器的文档:有很多不同之处公司生产HC-SR501传感器，它们并不总是为同一种传感器使用相同的引脚目的。对于图7-1所示的传感器(背面)，引脚设置为GND在底部，信号或触发

引脚在中间，电源引脚在右边;您的传感器可能需要电线放置在不同的顺序!

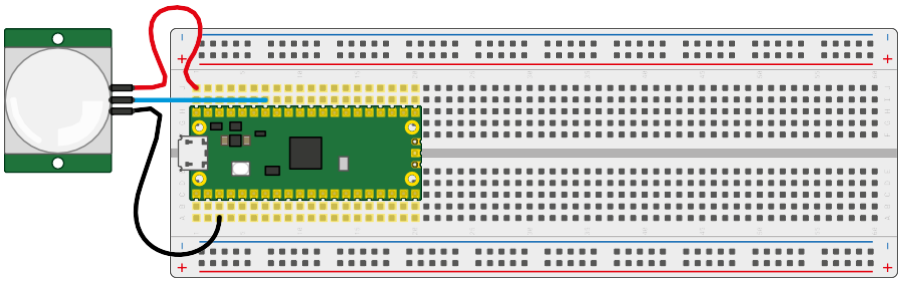


图 7-1：将HC-SR501 PIR 传感器与Pico连接

从接地线开始:这需要连接到您的Pico的任何接地针。在图7-1，它连接到面包板第三行第一个接地引脚。接下来，连接信号线:这应该连接到你的Pico的GPIO引脚GP28。

最后，你需要连接电源线。不过，不要把这个连接到你的Pico的3V3引脚上:HC-SR501是一个5v的设备，这意味着它需要5伏的电压才能工作。如果你把传感器连接到你的Pico的3V3引脚上，它就不起作用了——这个引脚不能提供足够的电力。

为了给你的传感器提供所需的5v电源，将它连接到你的Pico - VBUS的最右上角的引脚上。这个引脚连接到你的Pico上的micro USB端口，在转换为3.3 V以运行你的Pico微处理器之前，插入USB 5v电源线。所有三个HC-SR501引脚现在都应该连接到你的Pico:地线，信号线和电源线。

编程你的闹钟

你需要对Pico进行编程才能识别传感器。方便的是，这并不比读取一个按钮更难——事实上，你可以使用相同的代码。首先创建一个新的程序并导入machine库，这样你就可以配置你的Pico的GPIO pin:

```
import machine
```

然后设置你连接HC-SR501传感器的引脚，GP28:

```
sensor_pir = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

就像你做的反应游戏一样，防盗报警器的输入应该作为一个中断——停止程序正在做的任何事情，并在传感器被触发时做出反应。和前面一样，首先定义一个回调函数来处理中断:

```
def pir_handler(pin):
```

```
    print("ALARM! Motion detected!")
```

记住，第二行需要缩进四个空格，这样MicroPython就知道它是函数的一部分;当你在第一行后按回车键时，Thonny会自动输入这些空格。

最后，设置中断本身。记住，这不是handler函数的一部分，所以你需要删除Thonny自动插入的四个空格:

```
sensor_pir.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)
```

这就足够了:请记住，不管程序的其余部分在做什么，中断都是活跃的，因此不需要添加无限循环来保持程序运行。单击Run图标并将程序保存到您的Pico中，作为Burglar_Alarm.py。

在PIR传感器上挥一挥你的手:一条信息将打印到外壳区域，确认传感器看到了你。如果你一直挥动你的手，信息将继续打印-但是在每次打印之间有一个延迟。

缩小范围

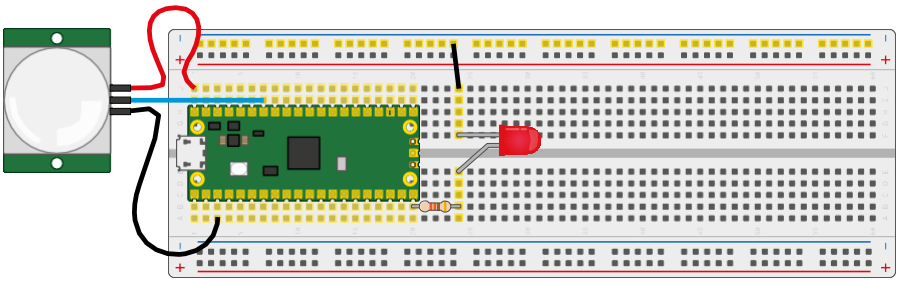
PIR传感器被设计成覆盖尽可能宽的视野(FOV)，这样窃贼就不会简单地在房间边缘四处乱窜。如果你发现你的传感器在你不希望的时候触发了，有一个简单的方法来解决它:从厕纸上拿一个硬纸板内胎，把传感器放在底部。这个管子就像马的眼罩一样，阻止传感器看到侧面的东西，这样它就能集中注意力看前面的东西。

这种延迟不是你程序的一部分，而是内置在HC-SR501硬件本身:当检测到运动时，传感器将触发信号发送到你的Pico的GPIO引脚，并保持信号打开几秒钟后再丢弃。在大多数HC-SR501传感器上，延迟是通过一个小电位器来调整的:你可以插入一个螺丝刀，把它朝一个方向转动来减少延迟，朝另一个方向转动来增加延迟。请检查传感器的文档，以确定哪个电位器控制延迟。

因为你的中断触发器被设置为在信号的上升边缘触发，当PIR传感器发送它的信号1或“高”时，消息就会被打印出来。即使检测到更多的运动，中断也不会再次触发，直到内置延迟过去并且信号返回到0或“低”。

向Shell打印一条消息就足以证明您的传感器正在工作，但它并不足以发出警报。真正的防盗警报器有灯和警报器来提醒周围的人有问题了——你也可以把同样的东西加到你自己的警报器上。

首先将一个LED(任何颜色的)连接到你的Pico上，如图7-2所示。较长的支架，也就是阳极，需要通过一个330 Ω的电阻连接到引脚GP15 -记住，如果没有这个电阻在适当的地方来限制通过LED的电流量，你可以损坏LED和你的Pico。较短的引脚，阴极，需要连接到你的Pico的一个接地针上-使用面包板的接地线和两根公对公(M2M)跳线



▲图 7-2： 在防盗报警器中添加 LED

这一次，您将处理程序中的延迟，而不是依赖内置在PIR传感器中的延迟。到程序的顶部，在导入machine这一行的下面，添加以下内容:

```
import utime
```

接下来，在你设置PIR传感器pin的地方添加一行新代码:

```
led = machine.Pin(15, machine.Pin.OUT)
```

这足以配置LED，但你需要让它亮起来。添加以下新行到你的中断处理函数-记住，像函数中的所有行一样，它将需要缩进四个空格，这样MicroPython就知道它是嵌套代码的一部分:

```
for i in range(50):
```

在这行末尾按下ENTER，您将注意到，Thonny自动添加了另外四个空格，以形成一个8个空格的缩进。这是因为您刚刚创建了一个有限循环，它将运行50次。字母i代表一个递增值，这个值在循环每次运行时递增，并由指令范围(50)填充。

给你的新循环做一些事情，记住这些行将需要缩进8个空格-这是Thonny将自动完成的-因为他们形成了你刚刚打开的循环和中断处理函数的一部分:

```
led.toggle()
```

```
    utime.sleep_ms(100)
```

这两个说明与您在前面章节中使用的略有不同。第一个是机器库的特性，它可以让你简单地改变一个输出引脚的值，而不是设置一个值-所以如果引脚当前是1，或高，切换它将它设置为0，或低;如果引脚已经是0，或低，切换它将它设置为高。

第二行使用utime库在程序中插入一个暂停，但不是以秒为单位，而是以毫秒为单位。utime.sleep_ms()函数非常适合于少于一秒的短延迟;如果你暂停了一秒钟或更长时间，请使用utime.sleep()。

这两行代码加在一起，LED忽明忽暗的延时为100毫秒(十分之一秒)。结果是完全一样的LED闪烁程序你写在第四章,但是你以前的程序需要四行代码——一个将领导一个,另一个暂停,再关掉了,最后一个暂停之前再次循环——这只需要两个。

值与切换

有时，在设置值时使用切换功能，，金通的导引LED时， 但请确保您已经考虑了您首先尝试实现的目标。如果您的项目取决于在给定时间（如警示灯或排干水箱的泵）的输出，则始终明确设定值，而不是依赖切换。

你的程序现在看起来像这样：

```
import machine
import utime

sensor_pir = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_DOWN)
led = machine.Pin(15, machine.Pin.OUT)

def pir_handler(pin):
    print("ALARM! Motion detected!")
    for i in range(50):
        led.toggle()
        utime.sleep_ms(100)

sensor_pir.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)
```

点击运行按钮，然后在PIR传感器上再次挥舞你的手:你会看到通常的警报信息打印到外壳区域，然后LED将开始快速闪烁作为一个视觉警报。等待LED停止闪烁，然后再次在PIR传感器上挥挥手:信息将再次打印，LED将重复它的闪烁模式。

为了让防盗报警器更具震慑力，你可以让它在没有察觉到任何动静的情况下缓慢闪烁——以此警告潜在的入侵者，你的房间已经被人监视了。到程序的最底部，添加以下几行:

```
while True:
    led.toggle()
    utime.sleep(5)
```

再次点击Run，但是不要碰PIR传感器:你会看到LED现在打开5秒，然后关闭5秒。只要传感器没有被触发，这种模式就会持续下去;在PIR传感器上挥动你的手，你会看到LED再次快速闪烁，然后回到慢闪模式。

这突出了线程和中断之间的一个关键区别:如果您使用线程编写这个程序，您的程序将仍然试图以5秒的间隔打开和关闭LED，即使您的PIR处理程序以100毫秒的间隔打开和关闭LED。这是因为线程是并行运行的。

中断,中断处理程序所运行的主程序暂停,——所以你fivesecond切换代码停止,直到处理程序完成LED闪烁,然后拿起从那里离开。您的代码是否需要暂停或继续运行的关键是你是否需要使用线程或中断,并将完全取决于你的项目正试图做什么。

输入和输出：将所有输入和输出放在一起

你的防盗报警器现在有一个闪烁的LED来警告入侵者离开，并且有一种方法可以看到它什么时候被触发，而不必观察外壳区域来获取信息。现在，它只需要一个警报器，或者至少是一个压电蜂鸣器，它发出的声音不会把你的邻居震聋。

根据你购买的型号不同，你的压电蜂鸣器底部会有插针，或者边上有短电线。如果蜂鸣器有引脚，把它们插到你的面包板上，这样蜂鸣器就会跨在中间隔板上;如果它有电线，把这些放在面包板上，简单地把蜂鸣器放在面包板上

如果你的蜂鸣器有足够长的电线，你可以把它们连接到你的Pico的GPIO引脚旁边的面包板上;如果蜂鸣器没有安装，则使用M2M (male-to-male)跳线连接蜂鸣器，如图7-3所示。红色的电线，或者标记为+符号的正极引脚，应该连接到您的Pico左下角的引脚GP14，就在您正在使用的LED引脚上方。黑色的电线，或标记有负极(-)符号或字母GND的负极针，需要连接到面包板的接地通道上。

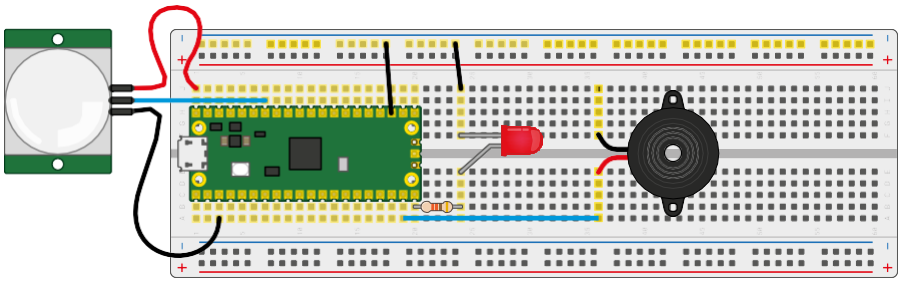


图 7-3： 连接双线压电蜂鸣器

如果你的蜂鸣器有三个引脚，连接标记有减号(-)或字母的引脚GND到您的面包板的接地轨，标记为S的针或信号将GP14针到您的和其余的引脚-通常是中间的引脚-到你的Pico的3V3别针。

如果你现在运行你的程序，什么都不会改变:蜂鸣器只有在从你的Pico的GPIO引脚接收到电源时才会发出声音。回到程序顶部，在设置LED的下方设置蜂鸣器

```
buzzer = machine.Pin(14, machine.Pin.OUT)
```

接下来，改变你的中断处理程序，在led.toggle()下面添加一个新行——记住，因为它是循环和处理函数的一部分，它需要缩进8个空格:

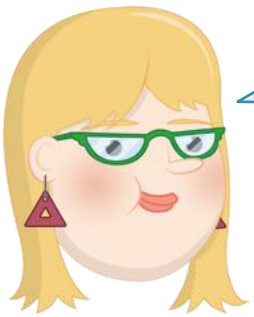
```
buzzer.toggle()
```

你的程序现在看起来像这样：

```
import machine
import utime

sensor_pir = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_DOWN)
led = machine.Pin(15, machine.Pin.OUT)
buzzer = machine.Pin(14, machine.Pin.OUT)

def pir_handler(pin):
    print("ALARM! Motion detected!")
    for i in range(50):
        led.toggle()
        buzzer.toggle()
        utime.sleep_ms(100)

sensor_pir.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)
```

```
while True:
    led.toggle()
    utime.sleep(5)
```

点击运行并在PIR传感器上挥动你的手:LED会像之前一样快速闪烁，但这一次它将伴随着蜂鸣器发出的哔哔声。祝贺你:这应该足以吓走洗劫你秘密收藏的糖果的入侵者!



警告

当使用一个主动蜂鸣器，它将继续声音，只要它连接的引脚是高的-换句话说，有一个值

1. 因为你的循环运行偶数次，它结束时蜂鸣器关闭;改变循环运行一个奇数次,尽管如此,它仍然将完成与蜂鸣器声音,按下停止按钮不会关掉它。如果发生这种情况,只需拔掉Pico的微型USB电缆和插头它回去,然后改变你的计划所以它不会再次发生!

如果你发现你的蜂鸣器在点击，而不是发出蜂鸣声，那么你使用的是被动蜂鸣器而不是主动蜂鸣器。主动蜂鸣器内部有一个被称为振荡器的部件，它可以快速移动金属板，发出嗡嗡声;被动蜂鸣器缺少这个组件，这意味着您需要用自己的一些代码来替换它。

如果你使用的是被动蜂鸣器，尝试这个版本的程序代替-它切换连接到蜂鸣器的引脚非常快地开和关，模仿有源蜂鸣器振荡器的效果:

```
import machine
import utime

sensor_pir = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_DOWN)
led = machine.Pin(15, machine.Pin.OUT)
buzzer = machine.Pin(14, machine.Pin.OUT)

def pir_handler(pin):
    print("ALARM! Motion detected!")
    for i in range(50):
        led.toggle()
        for j in range(25):
            buzzer.toggle()
```

```
            utime.sleep_ms(3)

sensor_pir.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)

while True:
    led.toggle()
    utime.sleep(5)
```

注意，控制蜂鸣器的新循环不使用字母i来跟踪增量;这是因为你已经在外部回路中使用了这个字母——所以它使用了字母j。与此同时，只有3毫秒的短延迟，意味着连接蜂鸣器的引脚打开和关闭的速度足以让蜂鸣器发出嗡嗡声。

尝试将延迟时间从3毫秒改为4毫秒，你会发现蜂鸣器的声音音调更低。改变延迟会改变蜂鸣器振荡的频率:延迟越长，蜂鸣器振荡的频率就越低，声音的音调就越低;延迟越短，它的振动频率就越高，声音的音调也就越高。

延长闹钟

你需要一个HC-SR501传感器来覆盖你想要覆盖的每个区域;在本例中，您将再添加一个传感器，总共添加两个，但您可以继续添加所需的多个传感器。

你需要一个HC-SR501传感器，用于你想要覆盖的每个区域;在此示例中，你将添加一个传感器，总共两个传感器，但你可以继续添加尽可能多的传感器。

你的两个传感器都需要5v电源才能工作，但你已经在你的Pico为第一个传感器。如果你的面包板有足够的空间，你可以把一个公对母(M2F)的跳线与第一个传感器相连，用于第二个传感器;不过，一种更简洁的方法是使用面包板上的电源通道。

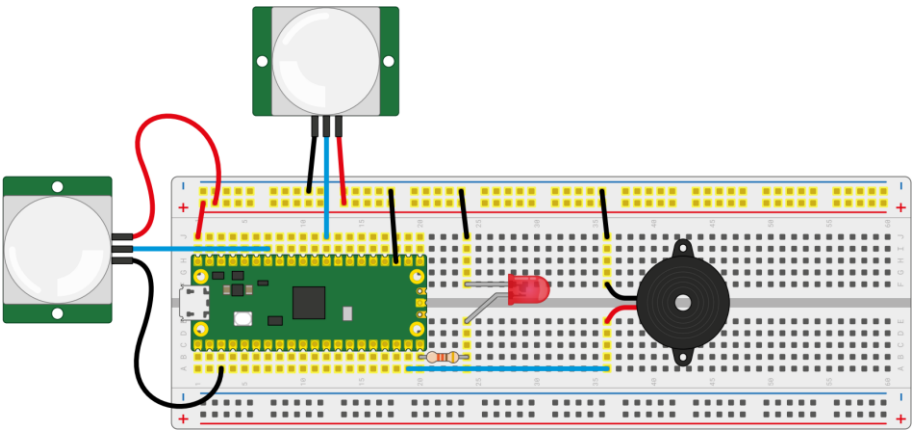
断开第一个传感器的电源线与面包板端，并将其插入红色或标记为加号(+)的电源轨。取一个公对公(M2M)跳线，并将同一电源轨连接到您的Pico的VUSB引脚。接下来，采取一个男性到女性(M2F)跳线，并连接电源轨到您的第二个PIR传感器的电源输入引脚。

最后，像之前一样连接第二个PIR传感器的接地引脚和信号引脚——但这次将信号引脚连接到Pico上的GP22引脚，如图7-4(背页)所示。你的电路现在有两个传感器，每一个连接到一个单独的引脚。

将程序设置为读取第二个传感器和第一个传感器，只需添加两行即可。首先设置第二个传感器，在设置第一个传感器的地方添加一行新代码:

```
sensor_pir2 = machine.Pin(22, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

然后创建一个新的中断，同样在第一个中断的下面:



▲图 7-4：添加第二个 PIR 传感器以覆盖另一个房间

```
sensor_pir2.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)
```

请记住，一个处理程序可以有多个中断，因此不需要更改程序的这一部分。

点击运行，并在第一个PIR传感器上挥一挥你的手:你会看到警报信息，LED闪光，蜂鸣器的声音和往常一样。等他们完成，然后在第二个PIR传感器上挥挥手:你会看到你的防盗警报以完全相同的方式响应。

为了使闹钟变得非常智能，你可以根据哪个引脚负责中断来自定义消息 - 它的工作方式与你之前编写的双人反应游戏中完全相同。

回到你的中断处理程序并修改它，使其看起来像：

```
def pir_handler(pin):
    if pin is sensor_pir:
        print("ALARM! Motion detected in bedroom!")
    elif pin is sensor_pir2:
        print("ALARM! Motion detected in living room!")
    for i in range(50):
        led.toggle()
        buzzer.toggle()
        utime.sleep_ms(100)
```

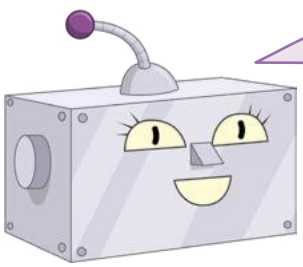
就像在第6章的反应游戏项目中，这段代码使用了一个事实，中断报告它被哪个引脚触发:如果PIR传感器连接到引脚GP28是负责任的，它将打印一条消息;如果它是PIR传感器连接到pin GP22，它将打印另一个。

你完成的程序将看起来像这样：

```
import machine
import utime

sensor_pir = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_DOWN)
sensor_pir2 = machine.Pin(22, machine.Pin.IN, machine.Pin.PULL_DOWN)
led = machine.Pin(15, machine.Pin.OUT)
buzzer = machine.Pin(14, machine.Pin.OUT)

def pir_handler(pin):
    if pin is sensor_pir:
        print("ALARM! Motion detected in bedroom!")
    elif pin is sensor_pir2:
        print("ALARM! Motion detected in living room!")
    for i in range(50):
        led.toggle()
        buzzer.toggle()
        utime.sleep_ms(100)

sensor_pir.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)
sensor_pir2.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)

while True:
    led.toggle()
    utime.sleep(5)
```

如果你使用的是被动的蜂鸣器，而不是主动的蜂鸣器，记住你需要将蜂鸣器切换到一个循环来让它发出蜂鸣声。单击Run并在一个传感器上挥一挥手，然后在另一个传感器上挥一挥手，就可以看到两个消息打印到Shell区域。

祝贺您:您现在知道如何构建一个模块化的防盗报警器，可以覆盖您需要的所有区域!



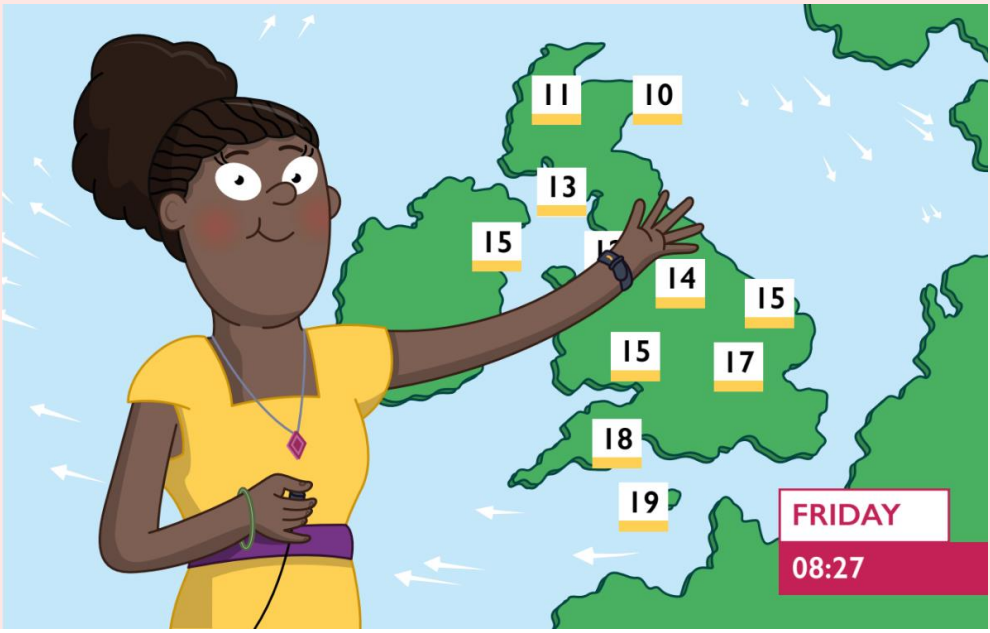
挑战：定制

你能用另一个红外传感器把防盗报警器加长吗?再加一个LED灯或蜂鸣器怎么样?你能改变打印的信息以匹配每个传感器所覆盖的区域吗?你能让蜂鸣器发出的声音更长或更短吗?除了红外传感器外，你能想到其他能很好地用于防盗报警器的传感器吗?

第八章

温度计

使用Raspberry Pi Pico的 内置ADC转换模拟输入，并读取其内部温度传感器



在之前的章节中，你已经在你的Raspberry Pi Pico上使用了数字输入。数字输入是开的或关的，A改变一个引脚从低，关，高，开;当被动红外传感器检测到运动时，二进制状态。当一个按钮开关被按下时，它会做同样的事情。

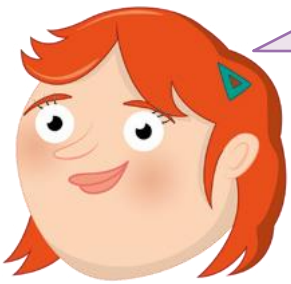
但是，您的Pico可以接受另一种类型的输入信号:模拟输入。而数字信号只能打开或关闭，而模拟信号可以从完全关闭到完全打开——一个可能值的范围。模拟输入被用于从音量控制到气体、湿度和温度传感器的所有东西，它们通过一个被称为模拟-数字转换器(ADC)的硬件工作。

在本章中，您将学习如何使用Pico上的ADC，以及如何接入其内部温度传感器来构建一个数据记录热测量小工具。您还将学习创建类似模拟输出的技术。为此，您需要使用Pico;任何颜色的LED及330 Ω电阻;一个10k Ω电位器;和一组公对公(M2M)跳线。您还需要一根微型USB线，并将您的Pico连接到您的树莓派或其他运行Thonny micropython IDE的计算机。

模数（模拟到数字）转换器

Raspberry Pi Pico的 RP2040微控制器是一种数字设备,像所有的主流微控制器:它是建立成千上万的晶体管,微型开关类设备无论是打开或关闭。因此,没有办法为你Pico真正理解一个模拟信号,它可以是任何东西之间的光谱完全,完全不依赖额外的硬件:analogue-to-digital转换器(ADC)。顾名思义，一个模拟-数字转换器接受一个模拟信号并将其转换为数字信号。你不会在你的Pico上看到ADC，无论你看得多近:它内置在RP2040中。许多微控制器有他们自己的ADC，就像RP2040，和那些不可以使用外部ADC连接到一个或多个他们的数字输入。

ADC有两个关键特征:它的分辨率(以数字位来衡量)和它的通道(或如何测量)它可以同时接受和转换许多模拟信号。你的Pico中的ADC的分辨率是12位，这意味着它可以将模拟信号转换为数字信号作为一个数字范围从0到4095——尽管这在MicroPython中被转换为16位范围的数字从0到65,535，这样它的行为与其他MicroPython微控制器上的ADC相同。它有三个通道带出GPIO引脚:GP26、GP27和GP28，这也是已知的作为GP26_ADC0, GP27_ADC1和GP28_ADC2的模拟通道0,1，和2。还有一个第四ADC通道，连接到RP2040内置的温度传感器;你会发现这一章后面会有更多的内容。



为什么是65，535?

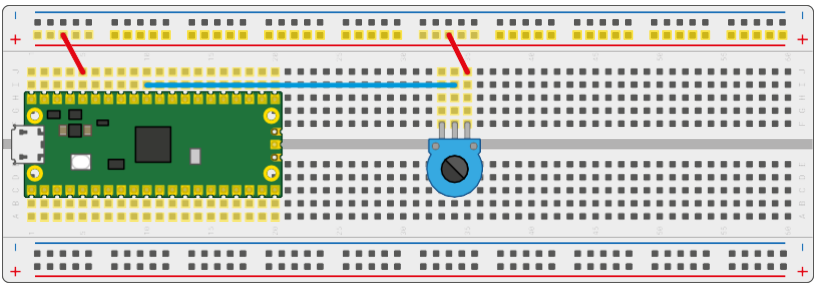
乍一看，数字“65,535”看起来很奇怪——为什么会这样，为什么不只是0-100呢?答案与你的事实有关Pico在二进制数字系统上工作，其中数字的唯一可能值是0或1。16位二进制数由16位组成，最大值为16位:1111111111111111. 如果您将其转换回十进制数(人类使用的0-9计数系统)，则得到65,535。

阅读一个电位计

连接到您的Pico的模拟数字转换器的每个引脚也可以用作简单的数字输入或输出;要把它用作模拟输入，你需要一个模拟信号——你可以很容易地用电位器做一个。

有各种类型的电位器可供选择:有些电位器，如您在第7章中使用的HC-SR501被动红外传感器上的电位器，设计为用螺丝刀调整;其他的，通常用于音量控制和其他输入，有旋钮或滑块。最常见的类型有一个小的，通常是塑料的，旋钮从顶部或前面伸出来:这被称为旋转电位器。

拿起你的电位器，把它翻过来:你会看到它有三个插脚，适合面包板。根据你如何连接这些引脚，电位器有两种不同的工作方式。首先将电位器插入你的面包板中，小心不要把针弄弯。使用M2M跳线将中间引脚连接到Pico上的GP26_ADC0，如图8-1所示。如果你的Pico插入面包板的最顶端，它将在第10行。最后，拿两个更多的跳线和电线电位器的外引脚之一-哪一个不重要-只要连接到你面包板的电源通道和到你的Pico 3v3通道上就可以。



▲图 8-1： 两个引脚相连的电位器

打开 Thonny 并开始一个新的程序：

```
import machine
import utime
```

与数字通用输入/输出 （GPIO）引脚一样，模拟输入引脚由machine库处理-就像数字引脚一样，它们需要设置，然后才能使用它们。继续编写你的代码：

```
potentiometer = machine.ADC(26)
```

这配置引脚GP26_ADC0在模拟数字转换器中作为第一通道ADC0。为了读取引脚，设置一个循环:

```
while True:
    print(potentiometer.read_u16())
    utime.sleep(2)
```

在这个循环中,读取和打印值通过在这一行中执行：这是一个更紧凑的选择读值到一个变量,然后打印变量,而且你不需要做其他工作除了把值打印出来——这正是这个项目需要。

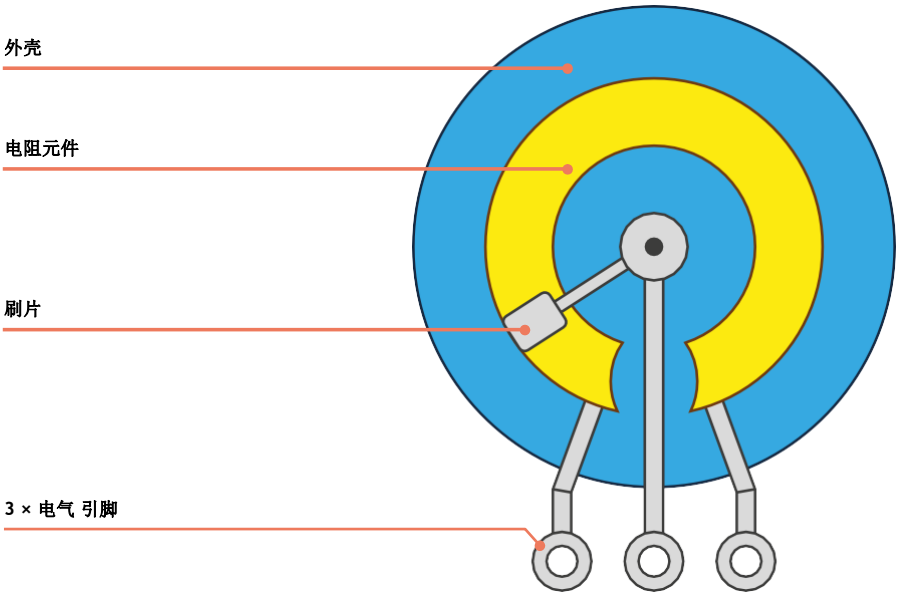
读取模拟输入与读取数字输入几乎完全相同，除了一点:读取数字输入时使用read()，但读取这个模拟输入时使用read_u16()。最后一部分u16只是警告您，接收到的不是二进制0或1的结果，而是一个无符号的16位整数——一个0到65,535之间的整数。

点击运行图标并将程序保存为potentiometer.com .py。观察Shell:您将看到程序打印出一个很大的数字，可能超过60,000。试着在一个方向上转动电位器：数字将上升或下降取决于你转动旋钮的方向和你在电路中使用的外部引脚。反过来:数值会向相反的方向改变。

不管你怎么转动它，它永远不会接近0。这是因为电位器只有两条引脚相连，就像一个被称为可变电阻器或变阻器的组件。一个变阻器是一个你可以改变值的电阻器-在一个10k Ω的电位器的情况下，在0 Ω和10,000 Ω之间。电阻越高，电压越低，3V3引脚到达你的模拟输入-所以数字下降。电阻越低，到达模拟输入的电压就越多——所以数字就会上升。

电位器的工作原理是有一个内部导电条，连接到两个外部引脚，一个雨刷或电刷连接到内部引脚(图8-2)。当你转动旋钮时，雨刷就会靠近雨刷带的一端，而远离另一端。从你连接到你的Pico的3V3引脚的那一端，刮水器得到的距离越远，电阻就越高;距离越近，电阻就越低。变阻器是非常有用的组件，但有一个缺点:无论如何你都会注意到

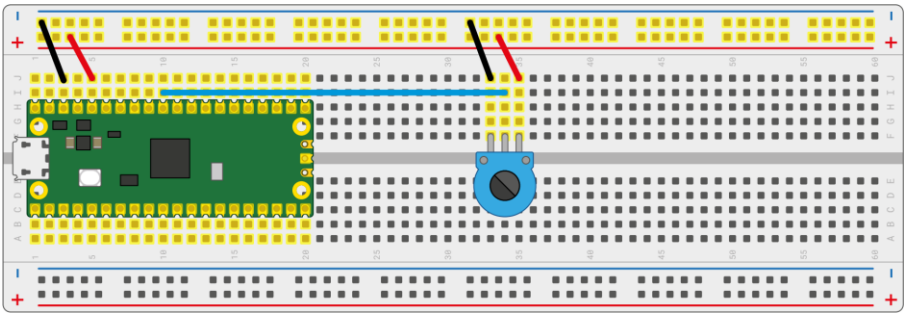
无论你在哪个方向转动旋钮，你永远都不会得到0或者接近0的值。这是因为一个10k Ω的电



▲ 图 8-2： 电位器的工作原理

阻不足以使3V3引脚的输出降到0v。你可以寻找一个更大的电位器与更高的最大电阻，或者你可以简单地把你现有的电位器连接起来作为分压器。

作为分压器的电位器



电位器上未使用的引脚并不是用来显示的:在电位器上加一个引脚连接到电路上完全改变了电位器的工作方式。点击停止图标来停止你的程序，并抓住两个男对男(M2M)跳线。如图8-3所示，将电位器未使用的引脚连接到面包板的接地轨上。拿起另一个，将接地轨连接到Pico上的GND pin。

▲图 8-3： 把电位器布接线后作为分压器

单击Run图标重新启动程序。再次转动电位器旋钮，一路一个方向然后一路另一个方向。注意打印到Shell区域的值:与以前不同，它们现在从接近0到接近完整的65,535—但是为什么呢?

添加的另一端接地电位计的导电带创造了一个分压器:而前电位计只是充当之间的电阻3 v3销和模拟输入插口,现在是3.3 V输出之间的电压除以3 v3销和0 V的接地针。将旋钮完全转向一个方向，你将得到3.3V的100%;完全反过来，0%。

零是最难的数字

如果您不能让您的Pico的模拟输入准确地读取0或准确地读取65,535，不要担心——您没有做错任何事情!所有的电子元件都有一个公差，这意味着任何声称的价值都不会是精确的。在电位器的情况下，它可能永远不会完全达到它的输入的0或100% -但它会让你非常接近!

你在外壳上看到的数字是模拟-数字转换器的原始输出的十进制表示，但这不是最友好的方式来看到它，特别是如果你忘记了65,535意味着“全电压”。

有一个简单的方法可以解决这个问题:一个简单的数学方程。回到你的程序，在你的循环上面添加以下内容:

```
conversion_factor = 3.3 / (65535)
```

这就建立了一种数学方法，将模拟-数字转换器给出的数字转换成与实际电压相当的近似值。第一个数字是引脚可以期望的最大可能电压:3.3 V，从你的Pico的3V3引脚;第二个数字是模拟输入读数所能达到的最大值，65,535。

总的来说，转换因子是由‘3.3除以65,535’产生的数字——最大可能电压除以模拟数字转换器报告的值范围，这反过来是其以位为单位的分辨率的一个特征。

设置好转换因子后，只需在程序中使用它。回到你的循环，并编辑它为阅读:

```
while True:
    voltage = potentiometer.read_u16() * conversion_factor
    print(voltage)
    utime.sleep(2)
```

循环中的第一行通过模拟输入引脚读取电位器的读数，并将其乘以(*符号)之前在程序中设置的转换因子，将结果存储为可变电压。然后，该变量将被打印到Shell中，代替您之前使用的原始读取。

你完成的程序将看起来像这样:

```
import machine
import utime

potentiometer = machine.ADC(28)

conversion_factor = 3.3 / (65535)

while True:
    voltage = potentiometer.read_u16() * conversion_factor
    print(voltage)
    utime.sleep(2)
```

LINEAR VS LOG

如果你发现在一个极限和另一个极限之间缓慢转动电位器会使数字起初变化缓慢，然后开始快速变化，或者相反，你几乎可以肯定是使用对数电位器或对数电位器。而线性电位器平滑地改变其整个范围，对数电位器开始做小的变化，然后迅速增加变化的速度。对数电位器通常用于放大器的音量控制，而线性电位器更常用于基于微控制器的设备，如Pico。

点击Run图标。从一个方向转动电位器，然后再从另一个方向转动。观察被打印到外壳区域的数字:你会看到，当电位器一路都是一种方式，数字非常接近于零;如果是另一种情况，则非常接近3。3。这些数字代表了引脚所读取的实际电压，当你转动电位器的旋钮时，你就可以平滑地将电压在最小值和最大值之间划分，0 V到3.3 V。

祝贺你:你现在知道了如何将电位器既作为压敏电阻又作为分压器连接起来，以及如何将模拟输入既作为原始值又作为电压读取!

测量温度

您的Raspberry Pi Pico的RP2040微控制器有一个内部温度传感器，它在第四模拟数字转换器通道读取。像电位器一样，传感器的输出是一个可变的电压:随着温度的变化，电压也随之变化。

启动新程序，导入machine和utime 库：

```
import machine
import utime
```

再次设置模拟-数字转换器，但这一次不是使用一个引脚的编号，而是使用连接到温度传感器的通道编号:

```
sensor_temp = machine.ADC(4)
```

您将再次需要转换因子，将传感器的原始读数转换为电压值，因此添加:

```
conversion_factor = 3.3 / (65535)
```

然后设置一个循环从模拟输入中获取读数，应用转换因子，并将它们存储在一个变量中：

```
while True:
    reading = sensor_temp.read_u16() * conversion_factor
```

虽然不是直接打印读数，但你需要进行第二次转换——取模拟数字转换器报告的电压并将其转换为摄氏度：

```
temperature = 27 - (reading - 0.706)/0.001721
```

这是另一个数学方程，是RP2040中特定于温度传感器的方程。这些值来自一份称为数据表或数据手册的技术文件:所有电子元件都有一份数据表，通常可以根据制造商的要求提供。您可以在rptl的Pico文档中查看RP2040的数据表。io/rp2040-get-started -它充满了关于微控制器如何工作的信息，尽管它是针对工程师的，所以是深度技术。

最后，完成循环：

```
print(temperature)
utime.sleep(2)
```

你的程序现在看起来像这样：

```
import machine
import utime

sensor_temp = machine.ADC(4)

conversion_factor = 3.3 / (65535)

while True:
    reading = sensor_temp.read_u16() * conversion_factor
    temperature = 27 - (reading - 0.706)/0.001721
    print(temperature)
    utime.sleep(2)
```

单击Run图标并将程序保存为Temperature.py。观察外壳区域:你会看到打印的数字代表传感器报告的温度，单位是摄氏度。

RP2040和其产生的热量

如果您使用传统的温度计，您可能会看到您的Pico报告的数字稍高一些:这是因为温度传感器位于Pico的RP2040芯片内，该芯片正忙于运行您的程序。当微控制器通电时，它会自己产生热量——而这些热量足以扭曲结果。对于像这样一个简单的程序，倾斜度可能不会太高;如果您的程序进行了大量复杂的计算，那么偏差可能会更高。

试着轻轻按压你的指尖到RP2040，最大的黑色芯片在你的Pico中间，并保持它在那里:你的手指的温暖应该使芯片更温暖，温度将上升。把手指从芯片上拿开，温度就会再次下降。

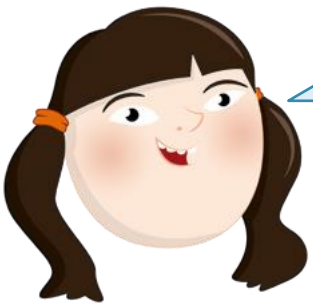
恭喜你 —— 你已经把你的Pico变成了温度计!

用 PWM 褪色 LED

Pico中的模拟-数字转换器只能以一种方式工作:它将模拟信号转换为微控制器可以理解的数字信号。如果你想走另一种方式，并让你的数字微控制器创建一个模拟输出，你通常需要一个数字到模拟转换器(DAC) -但有一种方法来“伪造”模拟信号，使用所谓的脉宽调制或PWM。

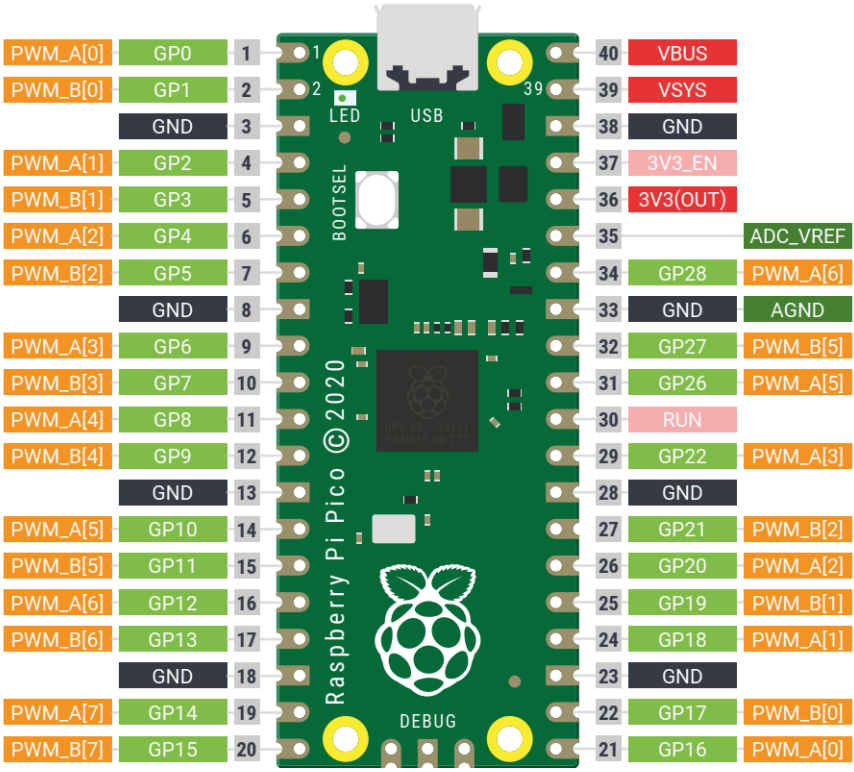
一个微控制器的数字输出只能是on或off，0或1。打开和关闭数字输出被称为脉冲，通过改变引脚打开和关闭的速度，你可以改变或调制这些脉冲的宽度——因此称为“脉宽调制”。

你的Pico上的每个GPIO管脚都能够进行脉宽调制，但是微控制器的脉宽调制块是由八个切片组成的，每个切片有两个输出。看看图8-4:你会看到每个pin都有一个字母和一个数字。数字表示连接到该引脚的PWM片;字母表示使用的片的输出。



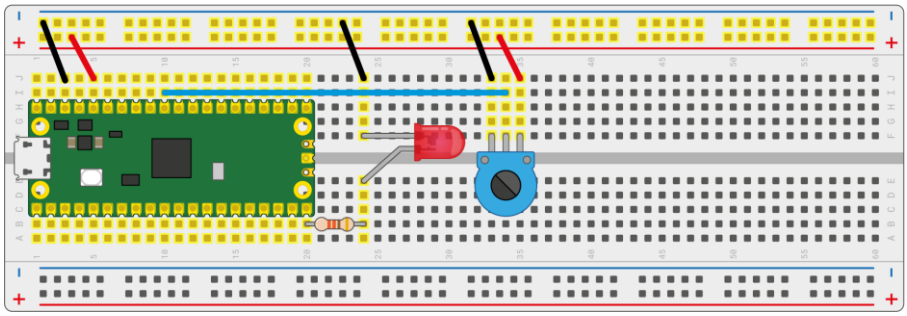
PWM冲突

如果你不小心使用了相同的PWM输出两次，你就会知道，因为每次你改变一个引脚上的PWM值，它也会影响冲突的引脚。如果发生这种情况，看看图8-4的引脚图和你的电路，找到一个你还没有使用过的PWM输出。



▲图 8-4： 脉宽调制（PWM）引脚

如果这听起来令人困惑，不要担心:这意味着您需要确保跟踪您正在使用的PWM切片和输出，确保只连接到您还没有使用过的字母和数字组合的引脚。如果您在引脚GP0上使用PWM_A[0]，在引脚GP1上使用PWM_B[0]，事情将正常工作，并且如果您在引脚GP2上添加PWM_A[1]，将继续工作;但是，如果您尝试在引脚GP0和引脚GP16上使用PWM通道，您会遇到问题，因为它们都连接到PWM_A[0]。



▲图 8-5： 添加 LED

取一个LED和一个330 Ω的限流电阻，将它们放入如图所示的电路板中图8 - 5。连接LED的较长支脚，阳极，通过330 Ω电阻引脚GP15，并将较短支脚连接到您的Pico的接地引脚。

通过点击Thonny工具栏下的标签回到你的第一个程序;如果您已经关闭了它，请单击Open图标并从您的Pico加载电位计.py。在将电位器设置为模拟数字输入的下面输入:

```
led = machine.PWM(machine.Pin(15))
```

这在GP15引脚上创建了一个LED对象，但有一个区别:它激活了引脚上的脉宽调制输出，通道B[7] -第8片的第二个输出，从零开始计数。

您还需要设置频率，这是您可以更改的两个值之一，以控制或调制脉冲宽度。在下面再加一行:

```
led.freq(1000)
```

这设置了1000赫兹的频率——每秒1000周。接下来，到程序的底部，在添加以下代码之前删除print(voltage)和utime.sleep(2)行，记住要让它缩进四个空格，这样它就成为循环内嵌的代码的一部分:

```
led.duty(potentiometer.read_u16())
```

这行从连接到电位器的模拟输入中获取原始读数，然后使用它作为脉宽调制的第二个方面:占空比。占空比控制引脚的输出:0%占空比使引脚在每秒1000次脉冲中处于关闭状态，并有效地关闭引脚;100%占空比使引脚处于开启状态，达到每秒1000次脉冲，在功能上相当于将引脚作为固定数字输出打开;50%占空比的引脚有一半脉冲是开的，一半脉冲是关的。

点击运行，转动电位器时观察LED:转动电位器时LED会变亮，转动电位器时LED会变暗。这是因为从连接到电位器的模拟引脚读取的数据被转换为PWM信号占空比的值:低占空比就像模拟输出上的低电压，使LED暗淡;高占空比就像高电压，使LED发光。

不过，你会注意到，LED在电位器旋钮达到停止点之前就已经达到最大亮度，非常轻微的移动就会引起亮度的巨大变化。这是因为模拟读数是一个介于0到65,535之间的数字，一个16位整数，但占空比为0%，设置为0,100%，值仅为1024。任何高于该值的值都将被忽略，并将其视为1024。

为了解决这个问题，并使之能够正确地控制LED的亮度，您需要将模拟输入的值映射到PWM片能够理解的范围。最好的方法是告诉MicroPython您将占空比值传递为无符号16位整数，与您从Pico的模拟输入引脚接收到的数字格式相同。
在你的程序中找到以下行：

```
led.duty(potentiometer.read_u16())
```

编辑它，以便它进行读取：

```
led.duty_u16(potentiometer.read_u16())
```

你完成的程序将看起来像这样：

```
import machine
import utime

potentiometer = machine.ADC(28)
led = machine.PWM(machine.Pin(15))
led.freq(1000)

while True:
    led.duty_u16(potentiometer.read_u16())
```

点击运行图标，试着把电位器一路转到一边，然后一路转到另一边。注意LED:这一次，除非你用的是对数电位器，否则你会看到LED的亮度平滑地从电位器旋钮的一端完全关闭到另一端完全点亮。

祝贺您:您不仅掌握了模拟输入，而且现在可以使用脉宽调制创建等效的模拟输出!

挑战：定制

你能把你的两个程序结合起来，让LED的亮度由车载温度传感器读取的温度来控制吗?你还记得你的Pico有多少模拟输入吗?PWM输出呢?尝试在你的Pico中添加另一个模拟传感器——类似于光依赖电阻(LDR)、气体传感器或气压计——并让你的程序读取它而不是电位器。

第九章

数据记录器

将Raspberry Pi Pico转换为温度数据记录设备，然后从计算机中分离它，使其完全便携

在这本书中，你一直在使用你的树莓派Pico连接到你的树莓派或其他电脑通过其微型USB端口。不过，与所有微控制器一样，没有理由让你的Pico必须以这种方式绑定：它是一个功能齐全的独立系统，具有处理能力、内存和它自己工作所需的一切。

在本章中，您将学习如何使用文件系统来创建、写入和读取文件，允许您将Pico放在任何您喜欢的地方，并让它记录数据以供以后访问——将其转换为所谓的数据记录器。这样你只需要你的Pico，如果你想使用它远离树莓派，一个微型USB充电器或电池组;一旦你完成了本章，如果你想扩展你的项目，你可以连接额外的模拟传感器。

文件系统

文件系统是Pico存储您编写的所有程序的地方。它的功能相当于树莓派的microSD卡，或笔记本电脑或台式机的硬盘或固态硬盘:它是一种非易失性存储，这意味着即使你拔掉Pico的micro USB线，你保存在那里的任何东西也不会丢失。

连接你的Pico到你的树莓派和加载Thonny，如果你还没有打开。

单击“打开”图标，然后在弹出的窗口中单击MicroPython设备。您将看到到目前为止所编写的所有程序的列表，这些程序存储在您的Pico文件系统中。您现在不会打开一个，所以单击Cancel。

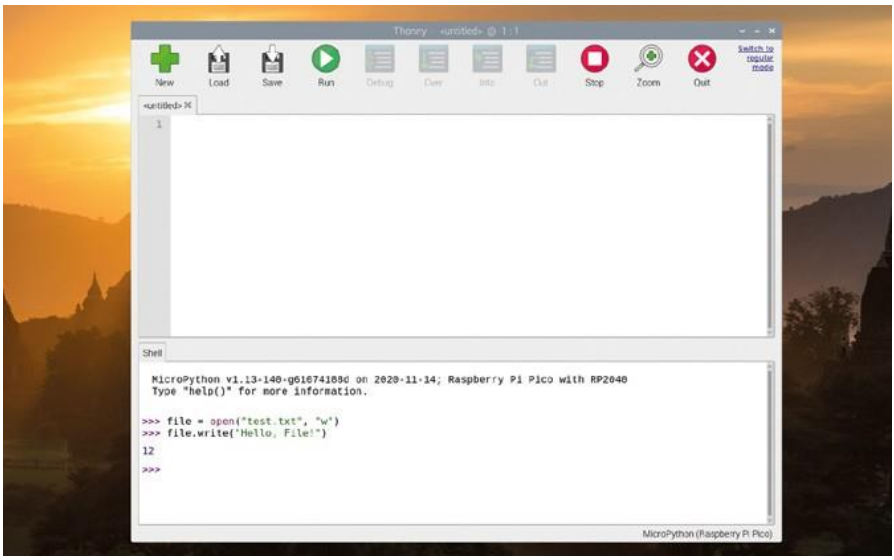
单击Shell区域的底部，开始在交互模式下使用Pico。类型:

```
file = open("test.txt", "w")
```

这告诉MicroPython打开一个名为test.txt的文件进行写入——指令的“w”部分。当您在行尾按下ENTER时，您不会看到任何打印到Shell区域的内容，因为尽管您已经打开了文件，但还没有对它做任何操作。类型:

```
file.write("Hello, File!")
```

当你按下这行末尾的ENTER键时，你会看到数字12出现(图9-1)。这是MicroPython向您确认它已经向您打开的文件写入了12个字节。计算你写的消息中字符的数量:包括字母、逗号、空格和感叹号，共有12个字符——每个字符占一个字节。



▲图 9-1： 您写入的数据的大小将打印到Shell区域

当您写入一个文件时，您需要关闭它——这确保您所告知的数据

要写的MicroPython实际上是写到文件系统的。如果你不关闭文件，数据可能还没有被写入——有点像在LibreOffice Writer或其他文字处理器上写一封信，却忘记保存它。类型：

```
file.close()
```

您的文件现在安全地存储在您的Pico的文件系统中。单击sonny工具栏上的“打开”图标，单击MicroPython设备，滚动文件列表，直到找到test.txt。点击它，然后点击确定打开它：你会看到你的消息弹出在一个新的Thonny标签。

不过，你不必使用Open图标来读取文件：你可以直接在MicroPython中读取。点击回到外壳区域的底部，并输入：

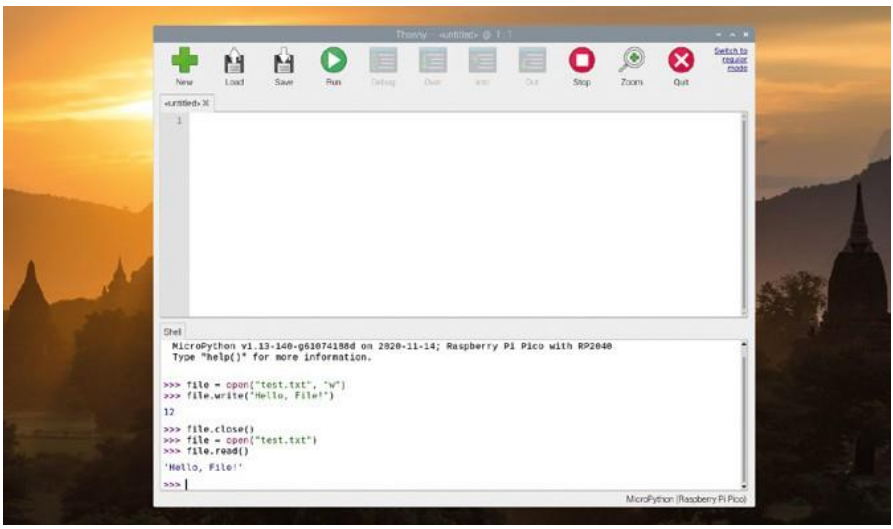
```
file = open("test.txt")
```

您将注意到，这次没有“w”：这是因为您将读取文件，而不是写入文件。你可以用“r”代替“w”，但是MicroPython默认以读模式打开文件-所以简单地保留这部分指令是没问题的。

加下来，编辑以下代码：

```
file.read()
```

您将看到写入文件的消息打印到Shell区域(图9-2)。祝贺您：您可以在您的Pico的文件系统上读写文件！



▲ 图9-2： 将存储的消息打印到Shell区域

在你完成之前，记得关闭文件——在读取文件后关闭文件并不像在写入文件时关闭文件那么重要，但无论如何这都是一个好习惯：

```
file.close()
```

记录温度

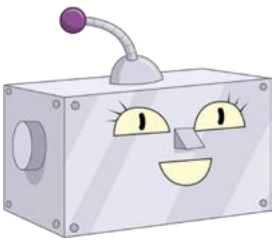
现在您知道了如何打开、写入和读取文件，您已经具备了在Pico上构建数据记录器所需的一切。点击新的图标开始一个新的程序在Thonny，并开始您的程序输入：

```
import machine
import utime

sensor_temp = machine.ADC(machine.ADC.CORE_TEMP)

conversion_factor = 3.3 / (65535)
reading = sensor_temp.read_u16() * conversion_factor
temperature = 27 - (reading - 0.706)/0.001721
```

您可能认识这个代码：它与您在第8章中从Pico的机载温度传感器读取的代码相同。来自传感器的读数是要记录到文件系统的数据，所以您不希望像以前那样简单地将它们打印出来。打开一个要写的文件，在底部添加以下一行：

```
file = open("temps.txt", "w")
```

如果文件系统中不存在该文件，则创建该文件；如果它这样做了，它就会覆盖它——清空它的内容，以便您写入新数据



警告

打开一个文件，在微蛇中书写将删除任何你已经存储在它。如果你想保留它，请务必确保您已打开文件进行阅读，并将内

现在，你需要向文件写一些东西 – 从温度传感器读到的数值：

```
file.write(str(temperature))
```

不像以前那样将一个固定的字符串写入引号中，这次您将把变量temperature(一个浮点数，换句话说，其中有一个小数点)转换为一个字符串，然后将其写入文件。

和前面一样，为了确保数据被写入，你需要关闭文件:

```
file.close()
```

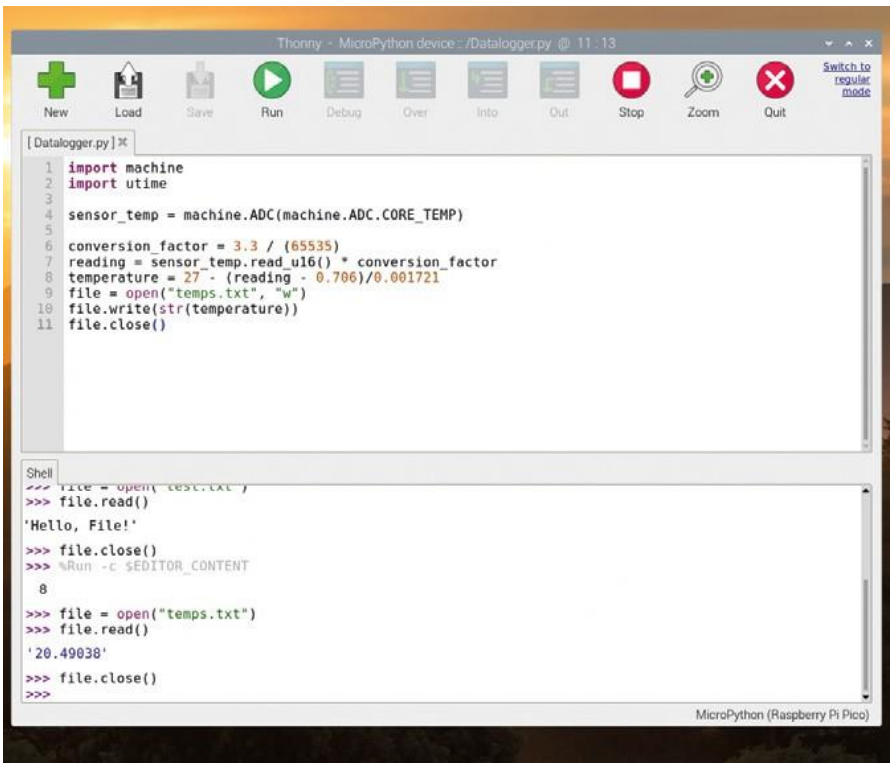
单击Run图标并将程序保存到MicroPython设备datallogger .py中。这个程序只需要几秒钟就可以运行;当>>>提示符重新出现在

Shell区域，点击进入它，并输入以下内容来打开和读取你的新文件:

```
file = open("temps.txt")
file.read()
file.close()
```

您将看到程序读取的温度显示在外壳中(图9-3)。

恭喜:你的数据记录器可以正常工作了!



▲图9-3： 你的文件记录了测量时的温度

但是，一个数据记录器只记录一次读取——一个数据——并没有那么有用。为了使你的数据记录器更强大，你需要修改它，使它需要大量的读数。再次单击Run，并再次读取文件：

```
file = open("temps.txt")
file.read()
file.close()
```

请注意，文件中仍然只有一个读取。当你的程序再次打开文件进行写入时，它会自动删除先前的内容——这意味着每次你的程序运行时，它都会删除文件并存储一次读取。

要解决这个问题，你需要修改你的程序。首先点击并拖动鼠标光标来突出显示这些行：

```
reading = sensor_temp.read_u16() * conversion_factor
temperature = 27 - (reading - 0.706)/0.001721
```

当你突出显示了两行-确保不遗漏任何部分-放开鼠标按钮和按CTRL+X键盘上的切线;你会看到他们从程序中消失。

接下来，到程序的底部，删除后面的所有内容：

```
file = open("temps.txt", "w")
```

现在键入：

```
while True:
```

在该行末尾按下ENTER后，按住CTRL键并按下V键粘贴前面剪切的两行。您将看到它们出现，这样您就不必输入它们了——但只有第一行将被适当缩进，因此它将嵌套为您刚刚打开的无限循环的一部分。通过单击并按下空格键四次，将光标放在下面一行的开头，以适当缩进该行，然后将光标移动到行尾并按ENTER。

输入下面的行，确保它是正确的缩进：

```
file.write(str(temperature)
```

但是现在，你需要做一些新的事情。如果您像以前那样关闭文件，您将无法再次写入它，除非重新打开它并删除它的内容。如果不关闭文件，数据将永远不会写入文件系统。

解决方式： 刷新文件，而不是关闭它。键入：

```
file.flush()
```

当您写入一个文件，但数据实际上并没有写入文件系统时，它被存储在所谓的缓冲区中——一个临时存储区域。当您关闭文件时，缓冲区将以称为刷新的进程写入文件。使用file.flush()等价于file.close()，它会将缓冲区的内容刷新到文件中——但与file.close()不同的是，文件保持打开状态，以便以后写入更多数据。

现在你只需要在阅读之间暂停你的程序：

```
utime.sleep(10)
```

你完成的程序将看起来像这样：

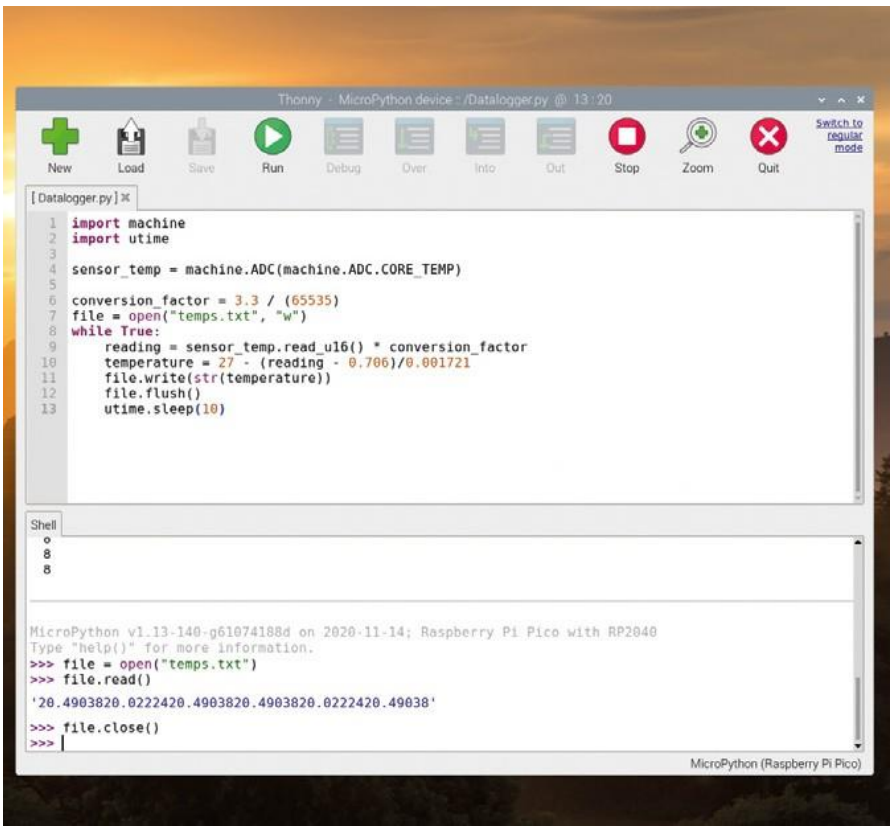
```
import machine
import utime

sensor_temp = machine.ADC(machine.ADC.CORE_TEMP)

conversion_factor = 3.3 / (65535)
file = open("temps.txt", "w")
while True:
    reading = sensor_temp.read_u16() * conversion_factor
    temperature = 27 - (reading - 0.706)/0.001721
    file.write(str(temperature))
    file.flush()
    utime.sleep(10)
```

点击运行图标并数到60，然后点击停止图标。现在在脚本区域打开并读取你的文件：

```
file = open("temps.txt")
file.read()
file.close()
```

好消息是您的程序已经工作了，并且您在一个文件中记录了多个读数——大约6个，根据您的计数速度有几个误差。坏消息是，它们都混在一起在一条线上(图9-4)——这使得阅读起来很困难。

要解决这个问题，您需要在数据写入文件时对其进行格式化。回到程序中的file.write()行，并修改它，使其看起来像：



▲图9-4：所有的测量值都在那里，但是格式使其难以阅读

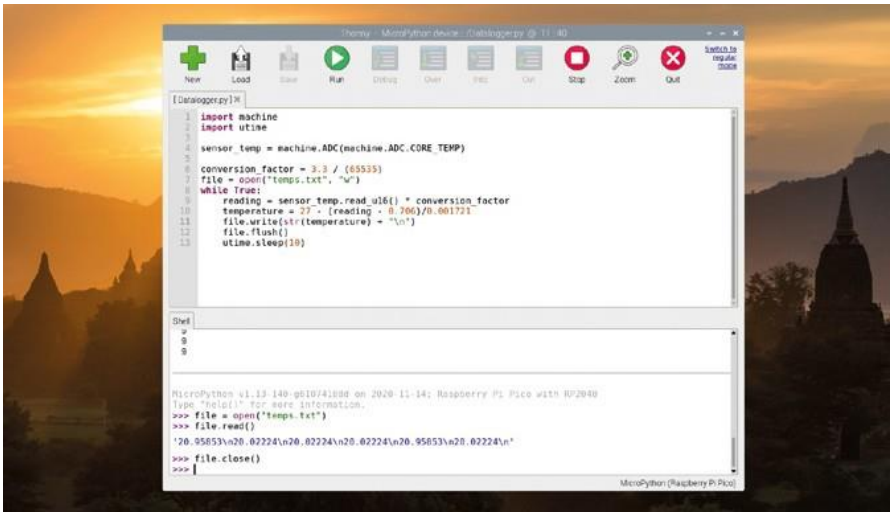
```
file.write(str(temperature) + "\n")
```

加号(+)告诉MicroPython您想要追加后面的内容，将两个字符串连接在一起;“\n”是一个被称为控制字符的特殊字符串——它的作用相当于按下ENTER键，这意味着数据日志中的每一行都应该是单独的一行。

单击运行图标，再次数到60，然后单击停止。打开并读取你的文件:

```
file = open("temps.txt")
file.read()
file.close()
```

你已经取得了进展，但它仍然不正确:\n控制字符的行为不像一个press输入，但打印为两个可见字符(图9-5，背页)。这是因为file.read()引入了文件的原始内容，而没有尝试为屏幕格式化它。

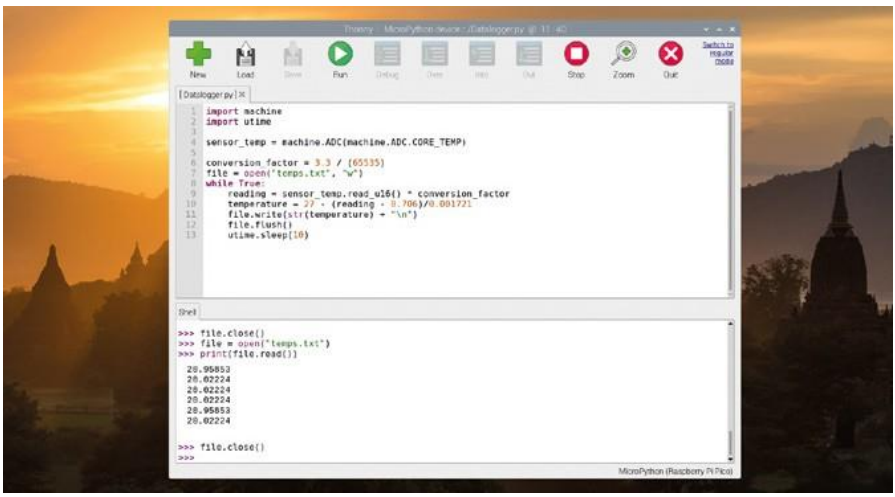


▲图 9-5： 你离一份易于阅读的打印文件越来越近了

要解决格式问题，你需要用print()函数包装读取的文件:

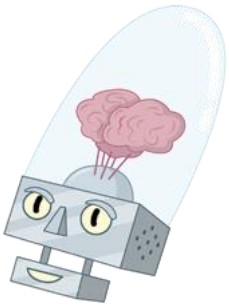
```
file = open("temps.txt")
print(file.read())
file.close()
```

这一次，您将看到每一份阅读材料都被打印出来，格式整齐，易于阅读(图9-6)。



▲图9-6： 现在你可以读取你数据记录器已经捕获的所有的温度

祝贺您:您已经构建了一个数据记录器，它可以进行多次读取并将它们存储在您的Pico的文件系统中!



文件存储

您的Pico的文件系统的大小是1.375MiB，这意味着它可以容纳1,441,792字节的数据。您保存在Pico上的每个文件，包括数据记录器的存储文件，都会占用空间。填满存储需要多长时间取决于您有多少其他文件以及您的数据记录器节省一次读取的频率:每10秒读取9个字节，您将在大约18.5天内填满1.375MiB;如果你每分钟读取一次数据，你的数据记录器可以运行111天左右;如果你每小时只读一次，你的数据记录器可以运行18年以上!

无论是否连接到树莓派或其他电脑，你的Pico文件系统都能正常工作。如果你有一个micro USB充电器或USB电池包USB线，你可以把你的数据记录器在你的房子里的任何房间，并让它自己运行-但你需要一种方法，让你的程序运行，而不必点击运行在Thonny图标。

在不连接计算机的情况下使用-称为无头操作-你可以将你的程序保存在一个特殊的文件名:main.py下。当MicroPython在它的文件系统中发现一个名为main.py的文件时，它会在每次上电或重置时自动运行这个文件，而不需要你点击Run。

在Thonny，停止程序后，如果运行，点击文件菜单，然后保存为。点击

在弹出的MicroPython设备中，然后键入' main。在单击Save之前，将文件名改为py '。一开始，似乎什么都不会发生:这是因为sonny将您的Pico设置为交互模式，这将阻止它自动运行您刚刚保存的程序。

要强制程序运行，单击Shell区域的底部，按住CTRL键，按D键，然后松开CTRL键。这会给你的Pico发送一个软重置命令，这将打破它的交互模式，并启动程序运行。找一些其他的事情做5分钟左右，然后按停止图标，打开你的数据日志:

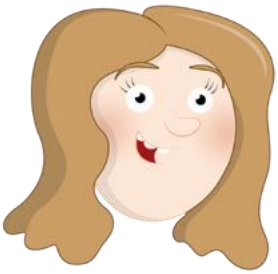
```
file = open("temps.txt")
print(file.read())
file.close()
```

你会看到一个温度读数的列表，即使你没有点击运行图标-因为你的程序在你的Pico重置时自动运行。

如果你有一个微型USB充电器或USB电池组，断开你的Pico和你的树莓派，把它带到另一个房间，把它连接到充电器或电池组上。把它放在那里十分钟，然后回来把插头拔掉。

把它放回你的树莓派，插上电源，再读一遍你的文件：你会看到来自另一个房间的读数，证明你的Pico可以没有你的覆盆子派的帮助也能运行得很好。

恭喜你：您的数据记录器现在是完全功能和完全便携，准备好了您可以随时随地记录您需要的数据！



警告

您的数据记录器程序将运行每次您的Pico是通电，而没有连接到索尼。如果你不希望发生这种情况，你可以简单地在thon中打开main.py并删除其中的所有代码，然后再次保存它。用一个空的main.py时，您的Pico将再次坐下并等待指令。

挑战

你能改变你的程序，从连接到你的Pico的ADC引脚的外部传感器记录数据吗?您是否可以让程序在文件的开头写一个标题，以便更容易地看到这些值的含义?你能写一个程序来记录一个按钮开关被按了多少次吗?你能想出一种方法，比如复制粘贴，把你的数据输入LibreOffice Calc或其他电子表格程序来创建图表吗?

第十章

数字通信协议：I2C 和 SPI

探索这两个流行的沟通协议，并使用它们在LCD上显示数据

在本书中，我们已经了解了如何使用一些常见的硬件，但随着你自己构建更多的项目，你可能想要扩展到使用各种不同的传感器、执行器和显示器。你将如何与这些人沟通?有时，您可能会发现有一个MicroPython库可以使用，其中有人已经将低级函数转换为易于使用的接口。然而，情况并不总是这样。

幸运的是，有一些连接低级数字设备的标准方法在MicroPython中实现的集成电路(I2C)和串行外设接口(SPI)。在许多方面，它们非常相似，因为它们都定义了一种将两个设备之间的双向接口连接起来的方式。事实上，许多部件都有这两种版本接口，这样您就可以选择一个适合您的项目

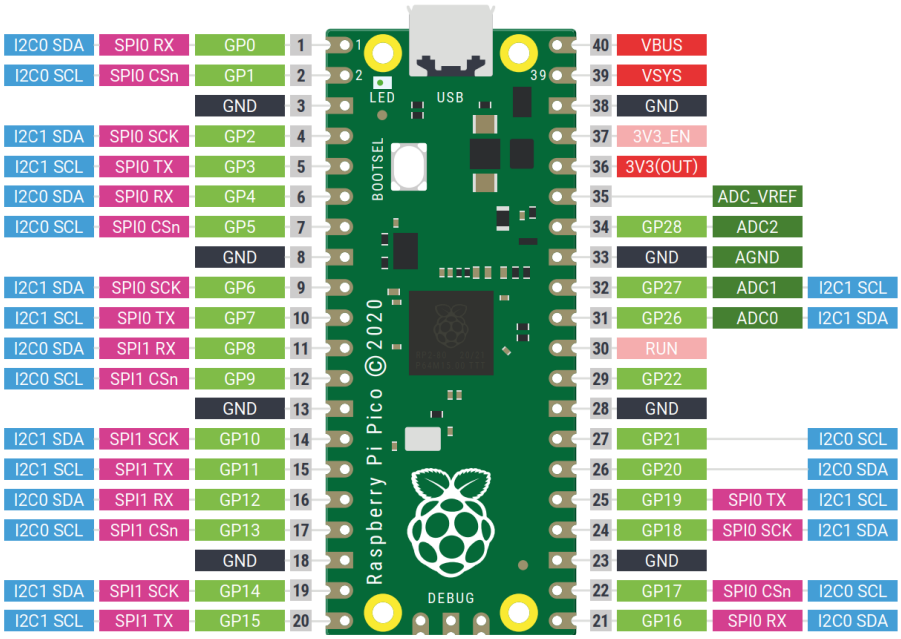
的接口。在这两种情况下，都有一个设备它控制通信(您的Pico)和一个(或多个)等待来自它的指令的主要设备。然而，也有一些不同之处。我们会时不时地看看这两种协议我们将帮助您为每个项目选择正确的。

I2C

电压等级

你的Pico的GPIO引脚工作在3.3伏特。对它们施加更高的电压可能会损坏它们。幸运的是，这是一个常见的工作电压，你遇到的大部分设备将工作在3.3伏特。但是，在将一些新硬件插入到您的Pico之前，始终要反复检查它是否是3.3 V设备，因为I2C和SPI设备有时都可以在5v下运行，这将损坏您的Pico。

I2C上的通信在两条线路上进行:一个时钟(通常标记为SCL)和一个数据通道(通常标记为SDA)。



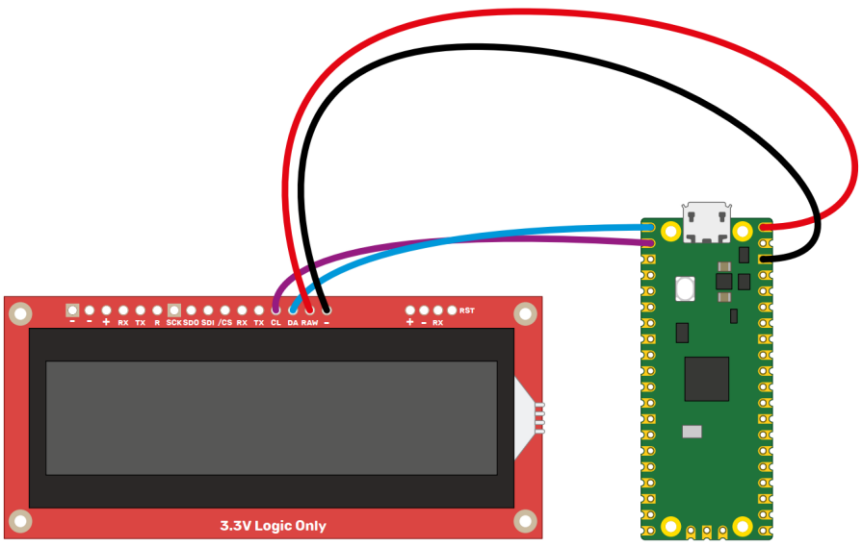
▲图10-1：树莓派的引脚图。浅蓝色为I2C功能;SPI功能为粉红色

这些必须连接到Pico上的特定引脚上。有一些选择;看看选项的插销图(图10-1)。有两种I2C总线(I2C0和I2C1)，您可以使用其中一种或两种。在我们的示例中，我们将使用I2C0—GP0用于SDA，GP1用于SCL。

为了演示这些协议，我们将使用SparkFun的SerLCD模块。这样做的优点是它同时拥有I2C和SPI接口，因此我们可以看到在相同的硬件下这两种方法之间的区别。

这个液晶显示器可以显示两行，每一行最多16个字符。它是一种有用的设备，可以输出关于我们系统的信息。让我们看看如何使用它。

接线I2C只是将Pico上的SDA引脚与LCD上的SDA引脚连接的情况，SCL也是如此。由于I2C处理通信的方式，还需要一个电阻连接SDA到3.3 V和SCL到3.3 V。通常是4.7 k Ω。然而，我们的设备已经包含了这些电阻，所以我们不需要添加任何额外的电阻。



▲图10-2：为I2C连接一个串行LCD模块

将其连接起来(如图10-2所示)，在屏幕上显示信息非常简单:

```
import machine
sda=machine.Pin(0)
scl=machine.Pin(1)
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000)
i2c.writeto(114, '\x7C')
i2c.writeto(114, '\x2D')
i2c.writeto(114, "hello world")
```

这段代码并没有做很多事情。它连接到I2C设备并发送一些数据。然而，也有一些看起来有点不寻常。

I2C .writeto()行中的114指的是I2C设备的地址。您可以将许多设备连接到I2C总线(稍后详细介绍)，每次要发送或接收数据时，都需要指定要与之通信的设备的地址。这个地址是硬连接到设备上的(尽管你可以通过在PCB上切割一个痕迹或焊接一个斑点来改变它——详细信息请参阅设备的文档)。

您应该在文档中找到设备的地址，但是您可以扫描I2C总线来查看当前使用的地址。设置好I2C总线后，可以运行scan方法输出当前使用的地址：

```
import machine
sda=machine.Pin(0)
scl=machine.Pin(1)
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000)
print(i2c.scan())
```

下一个看起来有点奇怪的位是写入的\x7C和\x2D命令。每一个I2C设备需要以特定格式发送数据。这方面没有标准，所以您必须参考文档了解您正在设置的任何I2C设备。每一个开头的\x告诉MicroPython，我们正在发送一个十六进制字符串(参见'十六进制'框)，这是一种常见的方式，以确保您正在发送您想要的确切数据。对于我们的液晶显示器，7C进入命令模式，2D空白液晶显示器，并将光标设置到开始。接下来，我们可以发送数据显示到屏幕上：

```
file.close()
```

十六进制

十六进制是一种以16为基数的编号系统。这意味着有16位数字:0-F。所以，十进制的10就是十六进制的A。十进制17是十六进制11。这样做的好处是每个数字包含两个字节的信息。这使它成为一种紧凑但仍可理解的书写数字信息的方式。在处理将指令作为数字值的设备(如我们的LCD)时，你会遇到很多这样的情况。

如果您感到困惑，您可以使用在线十六进制-十进制转换器在这两者之间切换。例如:hsmag.cc / hextodec。

当然，在屏幕上只显示Hello World并没有多大用处，所以让我们看看如何把它变成一个更有用的东西——温度计。在第8章中，您学习了如何使用Pico的内部温度传感器使用ADC读取温度。现在，我们可以在此代码的基础上构建一个独立的温度计，它不需要计算机读取输出。在你的LCD仍然像之前一样连接，运行以下代码:

```
import machine
import utime

sda=machine.Pin(0)
scl=machine.Pin(1)
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000)

adc = machine.ADC(4)
conversion_factor = 3.3 / (65535)
while True:
    reading = adc.read_u16() * conversion_factor
    temperature = 25 - (reading - 0.706)/0.001721
    i2c.writeto(114, '\x7C')
    i2c.writeto(114, '\x2D')
    out_string = "Temp: " + str(temperature)
    i2c.writeto(114, out_string)
    utime.sleep(2)
```

这看起来应该很熟悉。对之前的温度代码的唯一细微变化是，在输出计算结果(数字)之前，LCD需要显示字符，因此我们使用str函数将数字转换为字符串。然后，我们可以将它与"Temp: "组合起来，将其构建为一个信息更丰富的输出。

如您所见，I2C是一种将额外的硬件链接到Pico的简单方法。你需要确保你有任何你想要连接的设备的适当的文档，让你知道什么命令做什么，但只要你知道这一点，你可以很容易地添加各种各样的比特和鲍勃到你的Pico和创建令人印象深刻的构建。

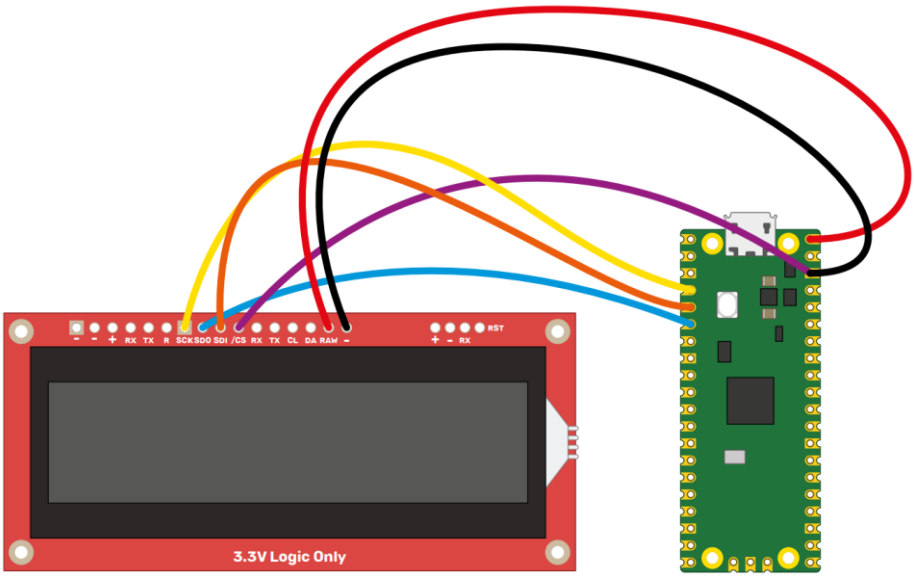
串行外围接口

我们已经了解了I2C的工作原理，现在让我们来看看SPI。我们将使用完全相同的LCD，因此命令和其他一切都是相同的，只是我们发送数据的协议不同

SPI有四个连接:SCLK，MOSI，MISO，和CS(有时标记为SS)。SCLK是时钟，MOSI是将数据从你的Pico到外围设备(参见“SPI术语”MISO将数据从外围设备发送到Pico。CS代表芯片选择用于将多个设备连接到单个SPI总线。你只需要在第十章数字通信协议:I2C和SPI CS线启用SPI外设，并拉低它来禁用它。为了让大家有点迷惑，这个特定的设备没有CS，但是/CS代表的不是CS，换句话

说，它是相反的CS，所以你把它低来启用LCD和高来禁用它。你可以把CS到GPIO引脚并切换这个开关来启用和禁用显示，但由于我们只有一个设备，我们可以简单地将它连接到地面，以保持它的启用(图10-3)。因此，SerLCD的电源线连接到VBUS和GND，我们只需要连接itsSDO到Pico的MISO (GP4 / SPIO RX)， SDI到MOSI (GP3 / SPIO TX)， SCK到SCLK (GP2 / SPIO TX)SCK)和/CS至GND。

因此，SerLCD的电源线连接到VBUS和GND，我们只需要连接它的SDO到Pico的MISO (GP4 / SPIO RX)， SDI到MOSI (GP3 / SPIO TX)， SCK到SCLK (GP2 / SPIO TX)SCK)和/CS至GND。



▲图10-3：通过SPI连接一个串行LCD屏模块

SPI术语

SPI需要四个连接:一个从主服务器获取数据设备到从设备，另一个在相反的数据方向，加上动力和地面。两根数据线意味着数据可以同时在两个方向上旅行。这些通常被称为Master Out Slave In (MOSI)和Master In Slave Out (MISO)然而,你会遇到不同的名字。如果你看一下Raspberry Pi Pico pinout(附录B)，它们被称为SPI TX(发送)和SPI RX(接收)。这是因为Pico可以是两者之一主设备或从设备，所以无论这些连接是MOSI还是味噌取决于Pico的当前功能。在我们使用的液晶显示器上，它们被标记为SDI(串行数据输入)和SDO(串行数据输出)

SPI中没有地址，所以我们可以直接写代码:

```
import machine

spi_sck=machine.Pin(2)
spi_tx=machine.Pin(3)
spi_rx=machine.Pin(4)

spi=machine.SPI(0,baudrate=100000,sck=spi_sck, mosi=spi_tx, miso=spi_rx)
spi.write('\x7C')
spi.write('\x2D')
spi.write("hello world")
```

在这种情况下，我们使用的是SPI0，一组可用的引脚是GP2，GP3和GP4。大多数类型的串行通信有一个速度或波特率，这基本上是多少每秒可以通过信道传输的数据位。很多事情都会影响这个，比如连接的两个设备的能力和它们之间的连接(多长时间)如果有其他设备的干扰)。如果你发现数据错误或丢失问题，那么您可能需要减少它。对于我们的小屏幕，我们只发送一个字节的数据字符，因此我们发送它的速度并不重要，但是对于其他一些SPI设备(例如(基于像素的显示)，微调波特率可能很重要。

让我们看看下面我们的使用的温度计代码:

```
import machine
import utime

spi_sck=machine.Pin(2)
spi_tx=machine.Pin(3)
spi_rx=machine.Pin(4)

spi=machine.SPI(0,baudrate=100000,sck=spi_sck, mosi=spi_tx, miso=spi_rx)

adc = machine.ADC(4)
conversion_factor = 3.3 / (65535)

while True:
    reading = adc.read_u16() * conversion_factor
    temperature = 25 - (reading - 0.706)/0.001721
    spi.write('\x7C')
    spi.write('\x2D')
    out_string = "Temp: " + str(temperature)
```

```
spi.write(out_string)
utime.sleep(2)
```

正如您所看到的，I2C和SPI之间的代码实际上差别很小。一旦你建立了一切,唯一的真正改变是与I2C你指定的地址发送数据时,同时与SPI你不(虽然记得如果你有多个设备连接,您需要切换CS GPIO选择适当的设备)。

那么，如果它们如此相似，那么在构建项目时，您应该选择哪种协议呢?有需要考虑的几个因素。第一个是您想要附加的东西的可用性。有时,一个传感器只能作为I2C或SPI，所以你必须使用它。但是，如果你有选择的话对于硬件来说，当你使用多个额外的设备时，影响最大。通过I2C,您可以在一个I2C总线上连接多达128个设备;然而，它们都需要有一个单独的地址。这些地址是硬连接的。有时可以用。来改变地址可焊接(或可切割)连接，但有时它不是。如果你想要这个的倍数相同类型的传感器(例如，如果您正在监测您的项目)，您可能会受到传感器的I2C地址数量的限制。在本例中，SPI也许是更好的选择。

另外，SPI可以有无限数量的设备连接;然而，每一个都必须有自己的CS线。在Pico上，有26个GPIO管脚。你需要他们中的三个SPI总线，所以这意味着有23可用的CS线。这是假设你不需要任何其他东西。如果可用的gpio非常紧缺，那么您可能需要考虑I2C。

实际上，对于许多项目，您都可以随意地使用其中任何一种协议，您可能会发现这一点选择使用哪一个更多的是与你在零件箱里找到的零件有关，而不是两者在技术上的区别。

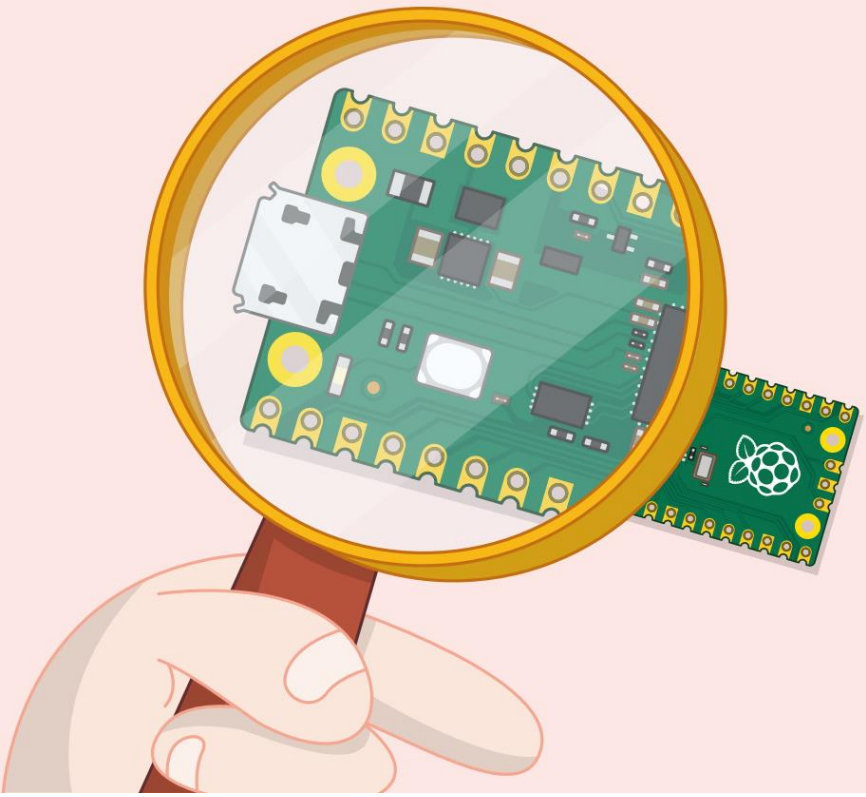
BIT BANGING

您的Pico有两个硬件I2C总线和两个硬件SPI总线。然而，如果你想要更多的话，你可以使用更多。I2C和SPI可以在软件而不是硬件中实现。这意味着主处理核心处理通信协议，而不是微控制器的专用部分。这就是所谓的“BIT BANGING”。虽然它很有用，但与使用专门的硬件相比，它会给处理器核心带来更多的压力，而且您可能会发现无法达到较高的波特率。

Pico有个妙招。我们将在本书后面(附录C)中更仔细地研究它，但它是微控制器中的一个额外的硬件，可以专用于输入/输出协议，如I2C和SPI。使用PIO，您可以创建额外的I2C或SPI总线无需占用主处理器核心。

附录 A

Raspberry Pi Pico 规格



个微控制器的各种部件和功能被称为它的规格，看一下规格给你你需要比较两个微控制器的信息。

这些说明一开始可能会让人感到困惑，它们是高度技术性的，您不需要知道它们就可以使用Raspberry Pi Pico，但是这里为好奇的读者提供了它们。

Raspberry Pi Pico的微控制器芯片是Raspberry Pi RP2040，如果你仔细观察，你会看到组件顶部蚀刻的标记。微控制器的名字可以分为几个部分，每个部分都有特定的含义：

- RP 意思是"Raspberry Pi（树莓派）"，很简单。
- 2 是微控制器的处理器内核数量。
- 0 是处理器核心的类型，在这种情况下，RP2040使用来自基于剑桥的Arm的称为Cortex-M0+的处理器核心。
- 4 是微控制器有多少随机存取存储器(RAM)，基于一个特殊的数学函数:地板(log2(RAM/16))。在这种情况下，' 4 '表示芯片有264千字节(kB) RAM。
- 0 是芯片有多少非易失性(NV)存储，并以与RAM相同的方式计算:地板($\log_2(NV/16)$)。在本例中，0仅仅意味着没有非易失性存储。

RP2040是树莓派首款微控制器;当未来的模型发布时，这些数字将被使用，这样你就可以快速地看到它们的功能对比情况。

您的Pico的两个Cortex-M0+处理器核心以48MHz(4800万周/秒)的速度运行，不过如果您的程序需要更高的性能，这在软件中最多可以更改为133MHz(1.33亿周/秒)。

微控制器的RAM与处理器的核心本身一样内置在同一块芯片中，并以总共264kB(264000字节)静态内存组的形式存在随机存取存储器(SRAM)。RAM是用来存储你的程序和它们需要的数据。

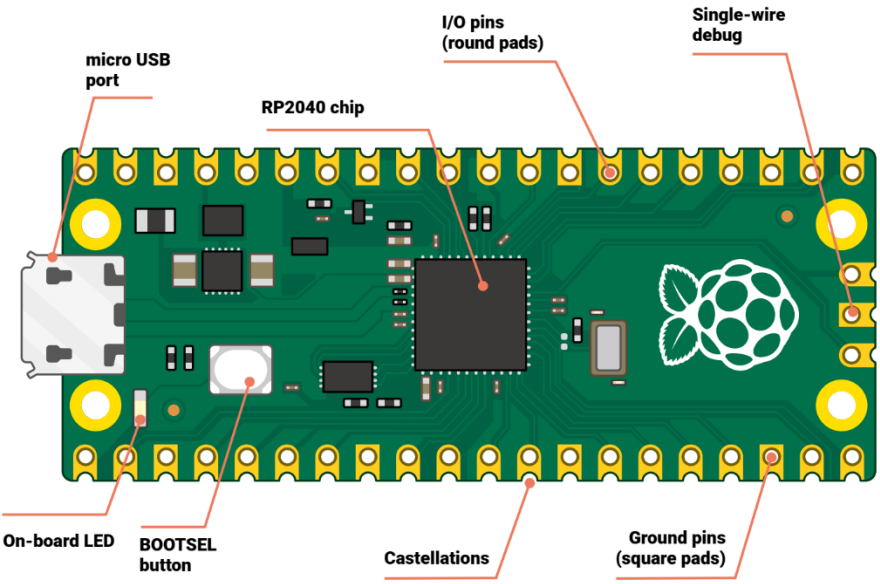
RP2040包括30个多功能通用输入/输出(GPIO)引脚，其中26个引脚带出到Pico上的物理引脚连接器上，其中一个连接到板载LED。其中三个GPIO引脚连接到一个模拟数字转换器上

RP2040包括两个通用异步收发器(UART)、两个串行外围接口(SPI)和两个集成电路(I2C)总线，用于连接到外部硬件设备，如传感器、显示器、数字-模拟转换器(dac)等。微控制器还包括可编程输入/输出(PIO)，它让程序员在软件中定义新的硬件功能和总线。

您的Pico包括一个micro USB连接器，它提供了一个到USB的uart串行连接RP2040微控制器用于编程和交互，并为芯片供电。插入线缆时按住BOOTSEL按钮，将微控制器切换到USB海量存储设备模式，允许您加载新的固件。

RP2040还包括一个精确的片上时钟和计时器，允许它跟踪时间和日期。时钟可以存储年、月、日、周、小时、分钟和秒，并自动跟踪所经过的时间，只要提供电力。

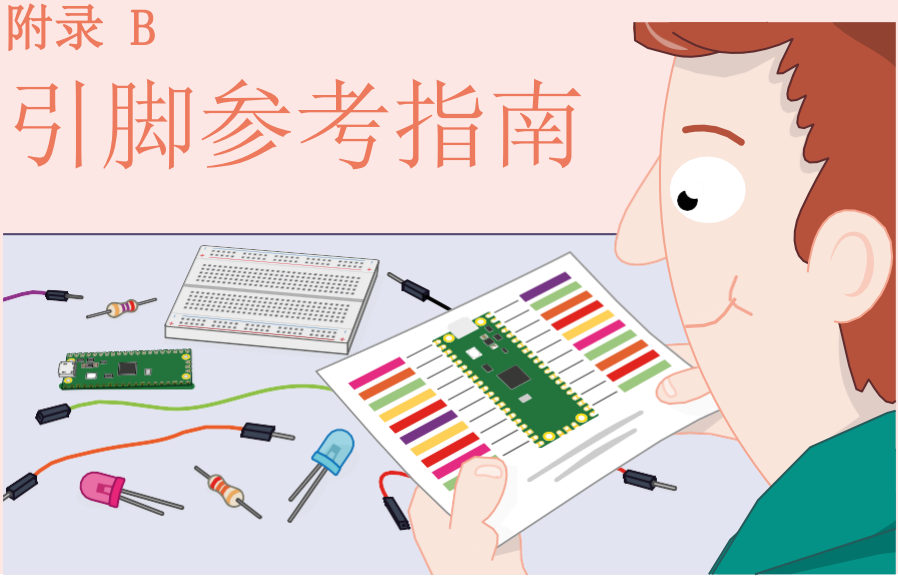
最后，在Pico底部有三个引脚，RP2040包含用于硬件调试的单线调试(SWD)。



- CPU：48MHz的32位双核ARM Cortex-M0+，可配置至133MHz
- RAM：通过六个独立可配置的池的 264kB SRAM
- 存储：2MB 外部闪存内存
- GPIO：26 引脚
- ADC：3 个×12位ADC引脚
- PWM：8片，每片共2个输出，一共16个输出
- 时钟：准确的 片上 时钟 和 计时器，包括 年、月、日、周、小时、秒和自动闰年计算
- 传感器：连接到 12 位 ADC 通道的片上温度传感器）
- LED：板载 LED
- **总线连接：** 2 × UART， 2 × SPI， 2 × I2C， 可编程输入/输出 （PIO）
- **硬件解试：** 单总线调试 （SWD）
- **安装选项：** 通孔和槽形销(未填充)有4个安装孔
- **电源**：5 V 通过micro USB 连接器，3.3 V 通过 3V3 引脚，或 2-5V 通过 VSYS 引脚

附录 B

引脚参考指南



Raspberry Pi Pico曝光了30个可能的RP2040 GPIO(通用输入/输出)引脚中的26个，直接将它们路由到Pico头引脚。GP0到GP22是数字专用的，GP 26-28既可以作为数字GPIO也可以作为ADC使用(模拟-数字转换器)输入，可在软件中选择。大多数GPIO引脚还为SPI、I2C或UART通信协议提供辅助功能。所有GPIO管脚也可以与PWM(脉宽调制)一起使用-参见第8章了解更多细节。（模拟到数字转换器）输入，可在软件中选择。大多数GPIO引脚还为 SPI、I2C或 UART 通信协议提供辅助功能。所有 GPIO引脚也可与PWM（脉冲宽度调制）一起使用 -有关详细信息，请参阅第 8章。

VBUS 是micro-USB输入电压，连接到micro-USB接口引脚1。这是5v(或0 V，如果USB未连接或未通电)。

VSYS是系统的主要输入电压，可以在1.8 V到5.5 V的允许范围内变化，它被板载SMPS(开关电源)用来产生3.3 V的RP2040及其GPIO。

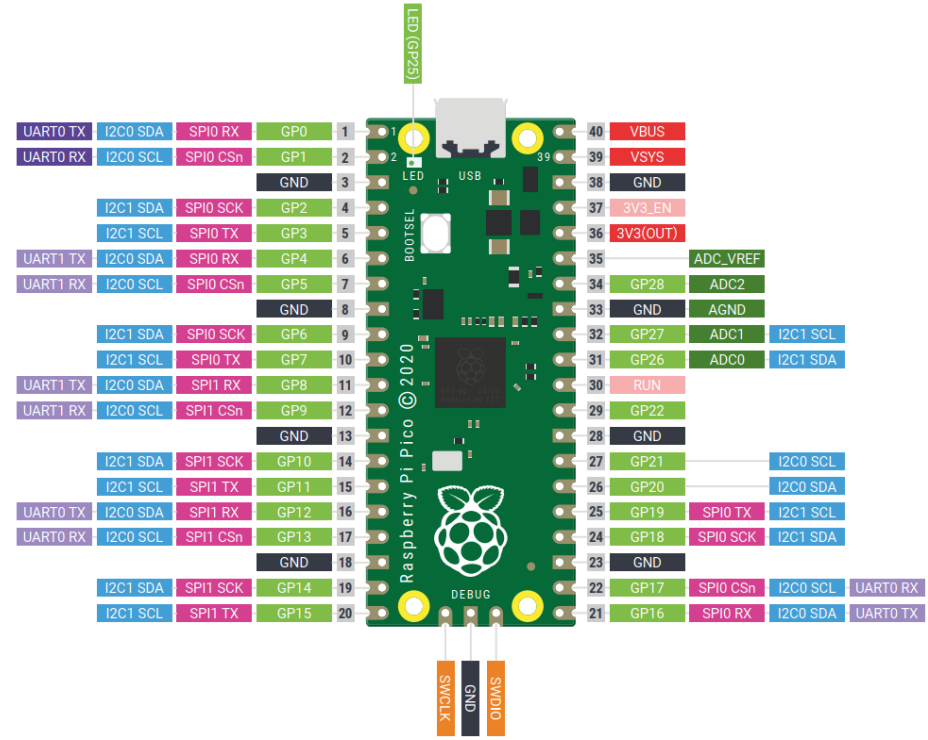
3V3_EN连接到板上SMPS使能引脚，并通过100 k Ω电阻拉高(到VSYS)。要禁用3.3 V(这也使RP2040失能)，短这个引脚低。

3V3是RP2040及其IO的主要3.3 V电源，由板载SMPS产生。该引脚可以用来为外部电路供电(最大输出电流将取决于RP2040负载和VSYS电压，建议将该引脚上的负载保持在300毫安以下)。

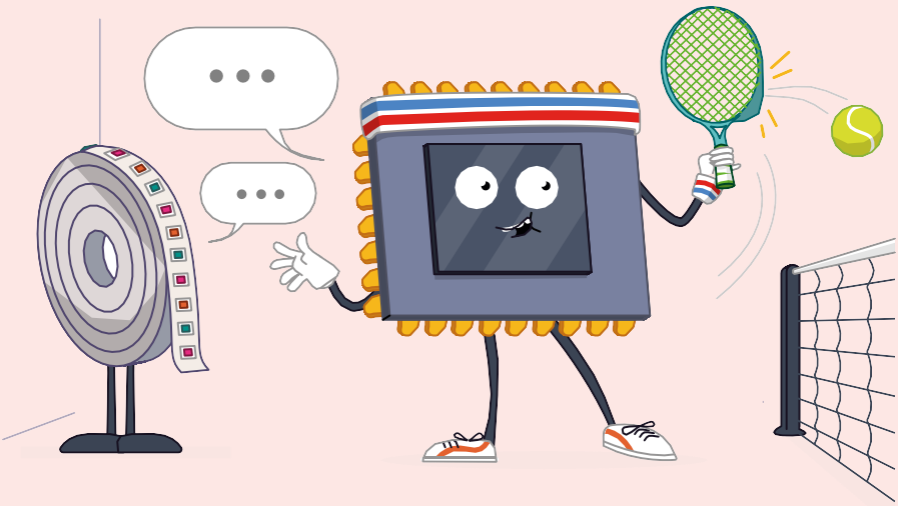
ADC_VREF是ADC电源(和参考)电压，通过3.3 V电源滤波在Pico上产生。如果需要更好的ADC性能，这个引脚可以与外部参考一起使用。

GPIO26-29的地参考;有一个单独的模拟接地面在这些信号下运行，并在这个引脚上终止。如果ADC未被使用或ADC性能不是关键，这个引脚可以连接到数字地。

RUN 是RP2040使能引脚，并有一个内部(片上)3.3 V左右的上拉电阻 ~ 50 kΩ。要重置RP2040，把这个引脚调低。


LED (GP25)
UART0 TX I2C0 SDA SPI0 RX GP0 1
UART0 RX I2C0 SCL SPI0 CSn GP1 2
GND 3
I2C1 SDA SPI0 SCK GP2 4
I2C1 SCL SPI0 TX GP3 5
UART1 TX I2C0 SDA SPI0 RX GP4 6
UART1 RX I2C0 SCL SPI0 CSn GP5 7
GND 8
I2C1 SDA SPI0 SCK GP6 9
I2C1 SCL SPI0 TX GP7 10
UART1 TX I2C0 SDA SPI1 RX GP8 11
UART1 RX I2C0 SCL SPI1 CSn GP9 12
GND 13
I2C1 SDA SPI1 SCK GP10 14
I2C1 SCL SPI1 TX GP11 15
UART0 TX I2C0 SDA SPI1 RX GP12 16
UART0 RX I2C0 SCL SPI1 CSn GP13 17
GND 18
I2C1 SDA SPI1 SCK GP14 19
I2C1 SCL SPI1 TX GP15 20
40 VBUS
39 VSYS
38 GND
37 3V3_EN
36 3V3(OUT)
35 ADC_VREF
34 GP28 ADC2
33 GND AGND
32 GP27 ADC1 I2C1 SCL
31 GP26 ADC0 I2C1 SDA
30 RUN
29 GP22
28 GND
27 GP21 I2C0 SCL
26 GP20 I2C0 SDA
25 GP19 SPI0 TX I2C1 SCL
24 GP18 SPI0 SCK I2C1 SDA
23 GND
22 GP17 SPI0 CSn I2C0 SCL UART0 RX
21 GP16 SPI0 RX I2C0 SDA UART0 TX
BOOTSEL
LED
USB
DEBUG
Raspberry Pi Pico © 2020
SWCLK
GND
SWDIO
Power
Ground
UART / UART (default)
GPIO and PIO
ADC
SPI
I2C
System Control
Debugging

附录 C
可编程 IO



在这个附录中，我们看到的代码看起来与我们在书的其余部分中处理过的代码非常不同。那是因为我们不得不在低层次上处理事情。大多数时候，MicroPython可以隐藏很多在微控制器上工作的复杂性。

当我们这样做的时候：

```
print("hello")
```

...我们不必担心微控制器存储字母的方式，或它们被发送到串行终端的格式，或串行终端所接受的时钟周期的数量。这些都是在后台处理的。然而，当我们进入可编程输入和输出时(PIO)，我们需要在更低的层次上处理事情。

我们将简要介绍PIO并介绍一些高级主题，以便您了解正在发生的事情，并希望了解Pico上的PIO相对于其他微控制器上的选项是如何提供一些真正的优势的。但是，理解创建PIO程序所需的所有低级数据操作需要花时间来完全理解，所以如果它看起来有点不透明也不用担心。如果您对处理这种低级编程感兴趣，那么我们将为您提供入门知识，并为您指明继续您的旅程的正确方向。：如果您更感兴趣的是在更高的层次上工作，而宁愿把低层次的争论留给其他人，我们将向您展示如何使用PIO程序。

数据输入和数据输出

在本书中，我们已经看到了控制您的Pico上的大头针使用的方法MicroPython。我们可以开关它们，接受输入，甚至使用专用SPI和I2C控制器发送数据。但是，如果我们想连接一个不使用SPI或I2C通信的设备，该怎么办?如果它有自己的特殊协议呢?

有几种方法可以做到这一点。在大多数MicroPython设备上，您需要执行一个称为“位碰撞”的过程，在这个过程中您使用MicroPython实现协议。使用此功能，您可以按正确的顺序打开或关闭pin以发送数据。

这有三个缺点。首先，它很慢。MicroPython在某些方面做得非常好，但它的运行速度不如本地编译的代码快。

第二，我们必须将其与运行在微控制器上的其余代码相协调。

第三，一些时间关键型代码很难可靠地实现。快速协议需要事情在非常精确的时间发生，而使用MicroPython我们可以做到相当精确，但如果你试图每秒传输几兆位，你需要每一毫秒或可能每几百纳秒发生事情。这在MicroPython中很难可靠地实现。

Pico有一个解决方案:可编程IO。有一些额外的，真正剥离的处理核心，可以运行简单的程序来控制IO引脚。你不能用MicroPython编程这些核心-你必须使用专门的语言-但你可以用MicroPython编程它们。让我们看一个例子：

```
from rp2 import PIO, StateMachine, asm_pio
from machine import Pin
import time

@asm_pio(set_init=PIO.OUT_LOW)
def led_quarter_brightness():
    set(pins, 0) [2]
    set(pins, 1)

@asm_pio(set_init=PIO.OUT_LOW)
def led_half_brightness():
    set(pins, 0)
    set(pins, 1)

@asm_pio(set_init=PIO.OUT_HIGH)
def led_full_brightness():
    set(pins, 1)

sm1 = StateMachine(1, led_quarter_brightness, freq=10000, set_base=Pin(25))
sm2 = StateMachine(2, led_half_brightness, freq=10000, set_base=Pin(25))
```

```
sm3 = StateMachine(3, led_full_brightness, freq=10000, set_base=Pin(25))


while(True):
    sm1.active(1)
    time.sleep(1)
    sm1.active(0)

    sm2.active(1)
    time.sleep(1)
    sm2.active(0)

    sm3.active(1)
    time.sleep(1)
    sm3.active(0)
```

这里有三种方法，看起来都有点奇怪，但设置板上导致四分之一，一半，和全亮度。它们看起来有点奇怪的原因是它们是用一种特殊的语言为Pico的PIO系统编写的。你大概可以猜到他们在做什么——用类似于我们使用PWM的方式快速地开关LED。指令集(引脚，0)关闭GPIO引脚，指令集(引脚，1)打开GPIO引脚。

这三个方法的上面都有一个描述符，告诉MicroPython将其视为PIO程序，而不是普通方法。这些描述符还可以接受影响程序行为的参数。在这些情况下，我们使用set_init参数来告诉GPIO管脚在开始时是低还是高。

这些方法——它们实际上是运行在PIO状态机上的小程序——都在不断循环。所以，例如，led_half_亮度会不断地打开和关闭LED，这样LED就会有一半的时间是关闭的，一半的时间是打开的。led_full_亮度将类似地循环，但由于惟一的指令是打开LED，这实际上并没有改变任何东西。

这里稍微有点不寻常的是led_quarter_亮度。每个PIO指令只需要运行一个时钟周期(可以通过设置频率来更改时钟周期的长度，稍后我们将看到)。但是，我们可以在一条指令之后用方括号添加一个1到31之间的数字，这告诉PIO状态机在运行下一条指令之前暂停这个时钟周期。然后，在led_quarter_亮度中，两个set指令每个使用一个时钟周期，延迟使用两个时钟周期，因此总循环使用四个时钟周期。在第一行中，set指令需要一个周期，延迟需要两个周期，所以GPIO管脚在这四个周期中的三个是关闭的。这使得LED的亮度达到了持续亮灯的四分之一。

一旦有了PIO程序，就需要将其加载到状态机中。因为我们有三个程序，所以需要将它们加载到三个状态机中(有8个状态机可以使用，编号为0-7)。这可以用一行来完成：

```
sm1 = StateMachine(1, led_quarter_brightness, freq=10000, set_base=Pin(25))
```

参数如下：

- 状态机器编号
- PIO 程序加载
- 频率（必须在 2000 到 125000000 之间）
- 状态机器操纵的GPIO 引脚

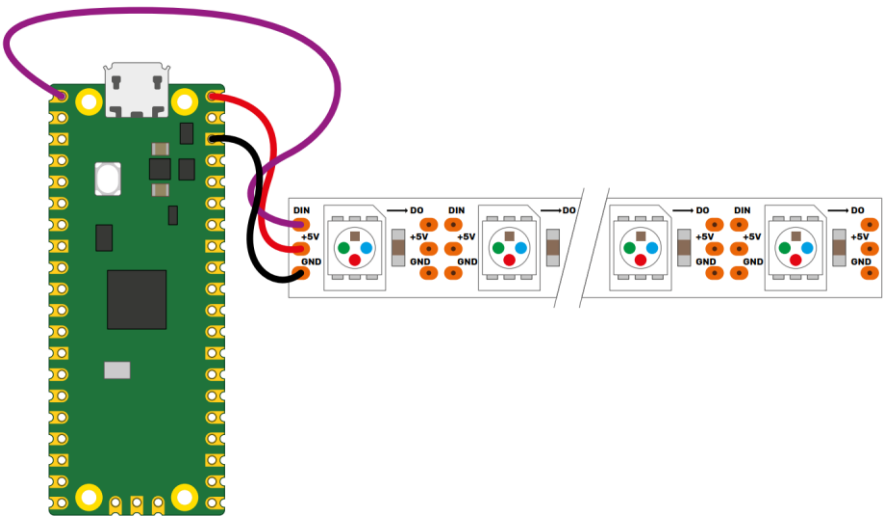
还有一些额外的参数，你会在其他程序中看到，我们这里不需要。

一旦创建了状态机，就可以使用active方法启动和停止状态机，1表示启动，0表示停止。在我们的循环中，我们循环三个不同的状态机。

一个真实的例子

前面的示例有点做作，所以让我们通过一个实际示例来看看使用PIO的方法。WS2812B led(有时称为新像素)是一种包含三个led(一个红、一个绿、一个蓝)和一个小型微控制器的光。它们由一根数据线控制，带有计时相关协议，很难被bit-bang.。

LED排的接线很简单，如图C-1所示。取决于你的LED条的制造商，你可能已经有电线连接，你可能有一个插座，你可以把头部电线推进去，或者你可能需要自己焊接它们。



▲图C-1： 连接 LED 灯条

你需要注意的一件事是潜在的电流吸引。虽然你可以向你的Pico中添加无穷无尽的新像素，但从Pico上的5v引脚获得的功率是有限的。

Pico上的5 V引脚。在这里，我们将使用8个led，这是非常安全的，但如果你想使用更多的led，你需要了解限制，可能需要添加一个单独的电源。你可以剪一个更长的条，led之间应该有切割线来告诉你该剪哪里。在hsmag.cc/neopixelpower上有一个很好的关于各种问题的讨论。

现在我们已经把led连接好了，让我们看看如何用PIO来控制它：

```
import array, time
from machine import Pin
import rp2
from rp2 import PIO, StateMachine, asm_pio

# Configure the number of WS2812 LEDs.
NUM_LEDS = 10


@asm_pio(sideset_init=PIO.OUT_LOW, out_shiftdir=PIO.SHIFT_LEFT,
autopull=True, pull_thresh=24)
def ws2812():
    T1 = 2
    T2 = 5
    T3 = 3
    label("bitloop")
    out(x, 1)               .side(0)    [T3 - 1]
    jmp(not_x, "do_zero")   .side(1)    [T1 - 1]
    jmp("bitloop")          .side(1)    [T2 - 1]
    label("do_zero")
    nop()                   .side(0)    [T2 - 1]


# Create the StateMachine with the ws2812 program, outputting on Pin(22).
sm = StateMachine(0, ws2812, freq=8000000, sideset_base=Pin(0))

# Start the StateMachine, it will wait for data on its FIFO.
sm.active(1)
```

它的基本工作方式是每秒发送800,000位数据(注意频率是8000000，程序的每个周期是10个时钟周期)。每一位数据都是一个脉冲——一个短脉冲表示0，一个长脉冲表示1。这个程序和我们之前的程序之间的一个很大的区别是，MicroPython需要能够向这个程序发送数据PIO的程序。

数据进入状态机有两个阶段。第一个是称为先进先出(FIFO)的内存。这是我们的主Python程序发送数据到的地方。第二个是输出移位寄存器(OSR)。这就是out()指令获取数据的地方。两者通过拉指令连接，拉指令从FIFO获取数据并将其放在OSR中。然而，由于我们的程序设置了启用autopull的阈值为24，所以每次我们从OSR读取24位时，它将从FIFO重新加载

指令out(x,1)从OSR中获取一位数据，并将其放入名为x的变量中(PIO中只有两个可用变量:x和y)。

jmp指令告诉代码直接移动到特定的标签，但是它可以有一个条件。指令jmp(not_x，“do_zero”)告诉代码，如果x的值为0(或者，在逻辑术语中，如果not_x为真，并且not_x是x的反面——在pio级别中，0为假，任何其他数字为真)，则移动到do_zero。

有一点jmp混乱，主要是为了确保计时一致，因为每次迭代循环必须使用完全相同的周期数，以保持协议的计时一致。

这里我们一直忽略的一个方面是.side()位。它们与set()类似，但它们与另一条指令同时发生。这意味着out(x,1)发生时，.side(0)设置侧集引脚的值为0。

哇，对于这么小的一个程序来说，这实在是太多了。现在我们已经激活了它，让我们看看如何使用它。下面的代码需要在程序中位于上述代码之下，以便将数据发送到PIO程序。

```
# Display a pattern on the LEDs via an array of LED RGB values.
ar = array.array("I", [0 for _ in range(NUM_LEDS)])

print("blue")
for j in range(0, 255):
    for i in range(NUM_LEDS):
        ar[i] = j
    sm.put(ar,8)
    time.sleep_ms(100)

print("red")

for j in range(0, 255):
    for i in range(NUM_LEDS):
        ar[i] = j<<8
    sm.put(ar,8)
    time.sleep_ms(100)
```

```
print("green")
for j in range(0, 255):
    for i in range(NUM_LEDS):
        ar[i] = j<<16
    sm.put(ar,8)
    time.sleep_ms(100)

print("white")
for j in range(0, 255):
    for i in range(NUM_LEDS):
        ar[i] = j<<16 + j<<8 + j
    sm.put(ar,8)
    time.sleep_ms(100)
```

在这里，我们跟踪一个名为ar的数组，它保存了我们希望led拥有的数据(稍后我们将了解为什么我们以这种方式创建数组)。数组中的每个数字都包含了一盏灯上所有三种颜色的数据。格式有点奇怪，因为它是二进制的。使用PIO的一个问题是，您经常需要处理单个数据位。每一位数据都是1或0，数字可以通过这种方式建立，所以以10为基数的2(我们称之为普通数)就是二进制的10。以10为基数的3在二进制中等于11。二进制数的8位中最大的数是11111111，或者以10为基数的255。我们不会在这里深入讨论二进制，但如果你想了解更多，你可以在这里尝试binary Hero项目:hsmag.cc/binaryhero。

让人更困惑的是，我们实际上把三个数字存储在一个数字中。这是因为在MicroPython中，整数存储在32位，但每个数字只需要8位。最后还有一点空闲空间，因为我们只需要24位，不过没关系。

前八位是蓝色，后八位是红色，最后八位是绿色。8位最多可以存储255个数字，所以每个LED都有255个亮度级别。我们可以使用移位运算符<<来实现这一点。这将在一个数字的末尾加上一定数量的0，所以如果我们想让LED的红色、绿色和蓝色亮度达到1级，我们将每个值都设为1，然后将它们移动到合适的位数。对于绿色，我们有:

1 <<16 = 10000000000000000

对于红色，我们有:

1 <<8 = 100000000

对于蓝色，我们根本不需要移位位，所以我们只有1。如果我们把所有这些加在一起，我们得到以下(如果我们把前面的位加起来，得到一个24位的数):

```
00000001000000010000001
```

最右边的八位是蓝色的，接下来的八位是红色的，最左边的八位是绿色的。

最后一点可能看起来有点令人困惑的是这句话：

```
ar = array.array("I", [0 for _ in range(NUM_LEDS)])
```

这创建了一个数组，第一个值是I，然后每个LED都是0。在开头有一个I的原因是它告诉MicroPython我们使用的是一系列32位的值。但是，对于每个值，我们只需要将24位发送给PIO，所以我们告诉put命令删除8位：

```
sm.put(ar,8)
```

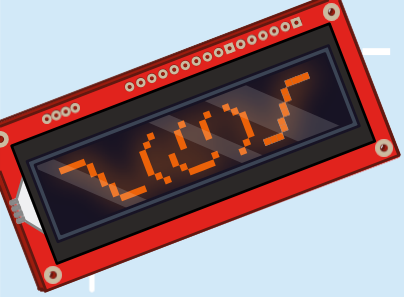
所有说明

PIO状态机使用的语言非常简洁，所以只有少量的指令。除了我们已经看过的，你还可以使用：

- **in() -**移动1到32位到状态机(与out()类似，但相反)。
- **push()——**将数据发送到连接状态机和主存的内存中MicroPython程序。
- **pull()——**从连接状态机和主存的内存块中获取数据MicroPython程序。这里我们没有使用它，因为通过在程序中包含autopull=True，当我们使用out()时，会自动发生这种情况。
- **mov() -**在两个位置之间移动数据(例如x和y变量)。
- **irq() -**控制中断。如果您需要触发一个特定的东西以在程序的MicroPython端运行，就可以使用这些。
- **wait() -**暂停直到发生一些事情(例如IO pin更改了一个设定值或中断发生)。

虽然只有少量的指令，但可以实现大量的通信协议。大多数指令都是用于以某种形式移动数据。如果您需要以任何特定的方式准备数据，例如操纵您希望led的颜色，这应该在您的主MicroPython程序中完成，而不是在PIO程序中。

您可以找到关于如何使用这些的更多信息，以及关于PIO的在MicroPython中的全部选项，在Raspberry Pi Pico 的MicroPython SDK文档-以及一个关于PIO在RP2040中如何工作的完整参考，这两者均可在**rptl.io/rp2040-get-started**找到。



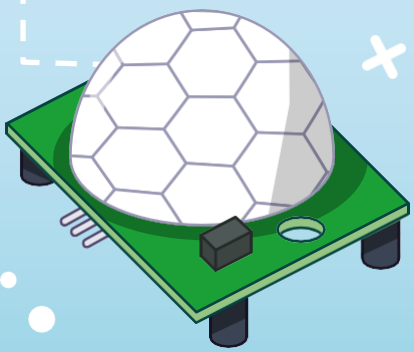
开始 MicroPython 在Raspberry Pi Pico上

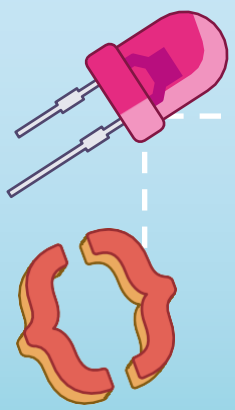
微控制器，如位于Raspberry Pi Pico中心的RP2040，是一款只保留基本功能的计算机。你不使用显示器或键盘，就可以从输入端接受信号，对其进行编程，并将其处理后的数据输出发送到输入/输出引脚。

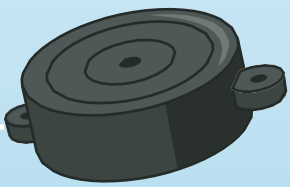
使用这些可编程连接，你可以亮自己想要的LED灯、发出想要的声音、发送文本到屏幕进行显示等。在Raspberry Pi Pico用MicroPython开始，你会学习如何使用初学者友好的语言MicroPython写程序和连接硬件，使你的Raspberry Pi Pico与周围的世界互动。

使用这些技能，无论是为了好玩还是为了让你的生活更轻松，你可以创建自己的机电工程项目。

机器人的未来就在眼前，我们会告诉你怎么做，你只要自己动手就可以了。







raspberrypi.org

价格：£10

9 781912 047864

Raspberry Pi PRESS