Machine Translated by Google

视差螺旋桨 2
Spin2 语言文档

2025年2月16日

v51

文档状态

版本日期		进步
	2020年2月6日	已开始文件。
v34t	2020_07_15 添加了 DEBUG,文档已更新。	
v34u	2020_07_19 DEBUG 改进,文档更新。	
第 35 版	2020_11_18 DEBUG 改进了整个过程的抗锯齿功能,并添加了 QSIN / QCOS。	
v35e	2021_01_06 添加了 DEBUG_BAUD 符号。修复了 Spin2 堆栈定位错误。	
v35f	2021_01_29 调试修复。之前在 63 个调试时出错,现在错误数量增加到 255 个。之前 DEBUG.log 文件并不总是需要重置。	
v35克	2021_02_13 调试修复。线剪辑例程导致浮点异常和内存访问冲突。	
v35h	2021年2月15日	· Spin2 解释器中的前 16 个 LUT 寄存器被释放,以允许流式传输“imm-->LUT”的使用。这是旨在通过中断支持 1/2/4 位视频,在解释器运行的同一 cog 中。为了进行补偿,内联 PASM 限制从 134 美元降至 124 美元。 · 添加了新的 DEBUG_WINDOWS_OFF 符号,以禁止在以下情况下打开任何 DEBUG 窗口:下载。现在可以设置 DEBUG_BAUD 来改变 DEBUG 与 PNut.exe 一起使用的波特率。
v35i	2021年2月20日	· 添加了命令行仅 DEBUG 模式,用于呈现闪存编程的 DEBUG 数据和显示。 · 修复了SCOPE_XY 中的浮点错误。
v35j	2021年3月16日	修复了 DEBUG_BAUD <> 2_000_000 在某些主板上不起作用的问题。
v35k	2021年3月19日	将 DOWNLOAD_BAUD 添加到现有的 DEBUG_BAUD,以覆盖默认的 2 Mbaud 下载和 DEBUG。
v35L	2021年3月23日	为 PNut.exe 添加了完整的命令行界面,并包含用于调用 PNut.exe 和返回的批处理文件错误状态输出到 STDOUT、STDERR 和 ERRORLEVEL。请参阅“PNut.exe 的命令行选项”。
v35m	2021年5月3日	· 改进了 PNut.exe 的命令行界面,支持使用/不使用 DEBUG 和使用/不使用 flash 加载器,无需下载即可保存.bin 文件。 · 在 DEBUG 中为 PLOT 显示添加了轴反转。
v35n	2021年5月23日	· 在 DEBUG PLOT 窗口中添加了精灵。 · REPEAT-var 已修复,以便 var = REPEAT 后的最终值（最终值 +/- 步骤）。
v35o,p	2021年9月22日	Spin2 新增浮点运算符,并遵循正常优先级规则。修复了 v35p 中的 FSQRT 错误。
v35q	2021年10月13日	主符号表从64KB增加到256KB,其他符号表从4KB增加到32KB。
v35r	2021年12月22日	添加了 PC_KEY 和 PC_MOUSE,用于将键盘和鼠标反馈从主机传输到 DEBUG 显示器。
v35s	2022年2月5日	· 负浮点常数前面可以加一个简单的“-”,因此只有变量才需要“-.”和表情。 · 修复了编译器和解释器中 FSQRT() 的错误。两者都在 FSQRT(-0.0) 和编译器上失败为 FSQRT(0.0) 生成了错误的结果。 · 改进了编译器和解释器中的浮点舍入操作,以便即使尾数分数为 0.500 的数在截断前不会像往常一样添加 0.500。这样就消除了四舍五入偏差。 · 添加了BYTEFIT,它与用于DAT 部分的BYTE 类似,但验证字节数据是-$80 到$FF。 · 添加了 WORDFIT,它类似于用于 DAT 部分的 WORD,但验证字数据是 -$8000 到 $FFFF。 · 添加了@ Text ,它是STRING() 的简写版本,只允许引号之间的文本。
v35t	2022年8月12日	· 添加了新的PASM 级调试器,用于单步执行和断点,由Spin2/PASM 中的“DEBUG”调用。 · DEBUG() 命令 PC_MOUSE 现在报告第 7 个长整型,其中包含 $00RRGGBB 像素颜色。
v35u	2022年8月26日	串行接口代码现在在单独的线程中运行,以便更好地与 GUI 并发。应该更加可靠。
v35v	2022年9月11日	· 如果 P62 上没有上拉电阻,则串行传输引脚（P62)现在在 DEBUG 之前保持高电平。这使得 PASM 级调试器能够在没有串行上拉电阻的早期 P2 Edge 模块上工作。 · 使用非 RCFAST 时钟模式的 PASM 专用程序现在会预先添加一个 16 位长的时钟设置器设置时钟模式的程序,将 PASM 程序移动到指定位置,然后执行该程序。这意味着在仅使用 PASM 的程序启动时不再需要 ASMCLK 指令。这与 PASM 级调试器的操作协调一致,其中时钟自动设置。
第 36 版	2022年9月18日	· DEBUG 现在可以适应运行时时钟频率的变化。这是通过使用串行接收引脚 (P63) 实现的。在长存储库模式下,存储调试中断之外的时钟频率。Spin2 CLKSET 指令现在支持此功能。

Machine Translated by Google

第 37 版	2022年11月19日	· 向子对象实例添加参数化。 · 每个子对象最多可传递 16 个参数。 · 参数覆盖子对象中同名的 CON 符号。 · 对于使用缓冲区大小、引脚号等对子对象进行硬编码很有用。 · ObjName：“ObjFile”｜参数A = 1,参数B = 2，… · Spin2 局部变量现在在方法进入时清零。 · 新的 ^@variable 返回任何集线器字节/字/长或寄存器的字段指针,包括任何位字段。 · 新的 FIELD[ptr] 变量别名使用 ^@variable 指针,使所有变量都可以作为参数传递。 · 新的“...”可用于忽略该行的其余部分并继续解析下一行。 · 新的 Spin2 GETCRC(dataptr,crcpoly,bytecount) 方法使用任何多项式计算字节的 CRC。 · 新的 Spin2 STRCOPY(destination,source,maxsize) 方法复制 z 字符串,包括零。 · DEBUG 显示 BITMAP 现在具有“稀疏颜色”,可以在背景颜色上绘制大圆形像素。 · 除了 GREY 之外,GRAY 现在也被识别为 DEBUG 显示中的一种颜色。 · 调试器的“Go/Stop/Break”按钮现在在单击时会暂时反转。
第 38 节	2023年2月3日	· 从 v37 开始修复不允许父对象 CON 块使用子对象的 CON 符号的错误。 · 修复了解释器中导致 ROTXY()/POLXY()/XYPOL() 不起作用的错误。 · REPEAT-var 在 REPEAT 之后恢复到原始行为,其中 var = (最终值 +/- 步长)。 · 所有 DEBUG 显示器现在都使用伽马校正的 alpha 混合来实现抗锯齿。
第 39 节	2023年3月5日	· 从 v37 开始修复导致同一文件的唯一参数化子对象全部相同的错误。 · 不再自动生成“.obj”文件,因为对象现在缓冲在 PC RAM 中以保持唯一性。 · 不再自动生成“.lst”列表文件,现在只能通过 Ctrl-L 或 Ctrl-I。 · 不再自动生成“.txt”文档文件,现在只能通过 Ctrl-D 生成。 · 不再自动生成“.bin”二进制文件,现在通过 Ctrl-R 切换。 · 修复了 v38 中导致 PASM 调试器的 REG/LUT/HUB 映射对比度低的错误。 · PASM 调试器现在对集线器 RAM 进行更直接的校验和,应该可以改善视觉变化响应。
v40	2023年9月21日	· 添加了新的更小/更快的 REPEAT 形式,用于从 0 到 n-1 迭代变量,其中 n > 0。 · REPEAT n WITH i · REPEAT 将变量从 0 迭代到 n - 1 的最佳方法 i 从 0 到 n - 1 一般等价,尽管 WITH 需要 n > 0
第 41 版	2023年9月24日	修复了浮点相等运算符（<.、>.、<>.、==.、<=.、>=.)中的一个错误。
v42	2023年11月11日	· 添加了BYTES()/WORDS()/LONGS() 方法来声明返回指针的大小值的字符串。 · 添加了 LSTRING() 方法,类似于 STRING(),但以长度字节开头并且可以包含零。
第 43 版	2023年12月13日	· 将BYTES()/WORDS()/LONGS() 方法重命名为BYTE()/WORD()/LONG() 以节省名称空间。 · 向 DEBUG SCOPE Display 添加了新的 AUTO 关键字,以自动缩放跟踪数据。 · 添加了新的 %“文本”,用于在长、小端、 以零填充。 · 实施 Spin2 关键字门控,以抑制将来添加新关键字时发生的命名空间冲突。 · 在任何 Spin2 代码之前寻找注释 {Spin2_v##},以启用新关键字。 · 例如,{Spin2_v43} 将启用新的 LSTRING 关键字（实际上是在 v42 中引入的）。 · 如果未找到 {Spin2_v##} 注释,则 {Spin2_v41} 为默认值。 · 当您启用较新的关键字时,您可能需要更改符号名称以解决冲突。 · 这样,现有代码就不会因为 Spin2 命名空间的增长而自动变得无法编译。
四十四	2024年3月13日	· 添加数据结构以帮助简化复杂的应用程序。 · 可以使用简单的语法在 CON 块内定义结构。 · 结构可以在 VAR 块和 PUB/PRI 标头中实例化。 · 结构和结构指针以相同的方式访问结构成员。 · 添加了 FILL/COPY/SWAP/COMP 方法来执行批量结构操作。 · 添加了BYTESWAP()/WORDSWAP()/LONGSWAP()方法来快速交换集线器内存范围。 · 添加了BYTECOMP()/WORDCOMP()/LONGCOMP()方法来快速比较集线器内存的范围。 · 向 DEBUG SCOPE 显示中添加了“TRIGGER channel AUTO {offset}”以实现自动触发。 · 向 DEBUG 输出命令添加 BOOL/BOOL_,如果非 0 则输出“TRUE”,如果为 0 则输出“FALSE”。 · 添加了 DEBUG 反引号模式输出命令：“?（布尔值）”和“。（浮点数）。 · 在没有时钟设置的 DEBUG 下载时,将采用 20 MHz 晶振模式来方便 DEBUG。 · 修复了导致 DAT 块 ORG 部分在 FVAR/FVARS 之后不将零填充到下一个长整数的错误。
v45	2024年11月13日	· 数据结构已得到改进,退出并取代了 v44 功能。 · 新关键字 STRUCT 用于开始 CON 块中的结构定义。 ■ CON STRUCT 点（x， y）,STRUCT 线（点 a,点 b） · 结构被打包,没有填充或对齐。 · 可以在 VAR 块中声明结构变量（示例使用上面的“行”结构）。 ■ VAR 线 a, b, c[8] · 结构变量可以在 PUB/PRI 块中声明为参数、返回值和局部变量。 ■ PUB 方法(行 a) : 行 b | 行 c[3] ■ 最多 15 个 long 结构可以作为参数和返回值传递 · 可以在 DAT 块中声明结构,然后在尾行中填充。 ■ DAT p point 下一行可以定义点 p 的内容 (LONG x,y) · v44 中的 FILL/COPY/SWAP/COMP 结构方法已被删除,现在由操作员处理。 ■ 结构~ 用 $FF 填充结 用 $00 填充结构 ■ 结构~~ 构 ■structureA:=structureB 复制结构的内容 ■structureA:=:structureB 交换结构 内容 ■structureA ==structureB ‘检查结构’相等性并返回TRUE/FALSE ■ 结构A <> 结构B 检查结构 不等式并返回 TRUE/FALSE ■ 结构 := 1,2,3 将 long 写入结构体 · 新的 SIZEOF(structure) 方法以字节为单位返回结构的大小。 ■ 接受STRUCT 名称、结构变量或结构指针变量。 · 为BYTE、WORD、LONG 和STRUCT 变量添加了指针变量。 · 每个指针都需要一个 LONG 并保存所指向变量的地址。 · 可以在 VAR 块中声明指针。 ■ VAR ^BYTE a、b、c ■ VAR ^WORD d, e, f ■ VAR ^LONG g, h, i ■ VAR ^structname j, k, l

Machine Translated by Google

		· 指针可以声明为 PUB/PRI 参数、返回值和局部变量： ■ PUB 方法（^BYTE a）:^WORD b | ^LONG c, ^structname d · 指针具有与其指向的变量相同的使用语法,但具有额外的功能。 指向的变量,后增 读取/修改/写入指向的变量 ■ ptrvar ■ ptrvar[++] 读取/修改/写入指针 ■ ptrvar[--] 读取/修改/写入指向的变量,后减指针 ■ [++]ptrvar 读取/修改/写入指向的变量,前增指针 ■ [--]ptrvar 读取/修改/写入指向的变量,前减指针 ■ [ptrvar] 读取/修改/写入指针变量本身 · 指针上的所有 ++/-- 操作将按 BYTE/WORD/LONG/STRUCT 大小（1/2/4/？）进行。 注意:v45 中存在一个已知错误,每当执行 FIELD 时都会导致解释器崩溃。此错误已在最新的 PNut_v46.zip 文件中修复。
v46	2024年11月20日	· 添加了 DEBUG 门控和禁用。 · 定义常量 DEBUG_MASK 来为文件/对象建立 32 个不同的权限位。· 使用 DEBUG[bitnumber]{(parameters…)} 通过 DEBUG_MASK 中的某个位来控制 DEBUG。· 将常量 DEBUG_DISABLE 定义为非 0 值,以禁用文件/对象中的所有 DEBUG。 · 现在可以禁止将时钟设置程序自动添加到仅限 PASM 的程序前面。· 定义常量 _AUTOCLK = 0 以停止将时钟设置程序添加到前面。· 如果需要,然后可以使用 ASMCLK 伪指令来设置时钟模式。 · VAR 块现在可以在每一行切换类型声明,而不是每行只允许一种类型。 · VAR BYTE a、b、c,WORD d、e、f,LONG g、h、i · 新增 DEBUG 命令 C_Z 将输出 C 和 Z 标志的状态,例如“C=0 Z=1”。 注意:PNut_v46.zip 文件已于 2024 年 11 月 24 日更新,以修复 Spin2 解释器 v45 中引入的一个错误。每当执行 FIELD 时,该错误都会导致解释器崩溃。
v47	2024年12月9日	· 增加了协作多任务处理功能,每个齿轮最多可执行 32 个任务。 · TASKSPIN(task,method({parameters}),stack_address) ■ 初始化 Spin2 任务,类似于 COGSPIN 初始化 Spin2 cog 的方式。■ Task = 0..31（对于固定任务)或 -1（对于第一个空闲任务）。■ 如果用作表达式项,则返回任务编号,如果没有空闲任务,则返回 -1。· TASKNEXT() ■ 切换到下一个未暂停的任务。■ 最终返回到当前任务中的下一条指令。■ 所有任务都必须定期执行 TASKNEXT() 以保持多任务处理。■ 如果 TASKNEXT() 在唯一剩余的任务中执行,则它不起作用。 · TASKSTOP(task) ■ 停止并释放任务。■ Task = 0..31（表示固定任务)或 -1（表示当前任务）。■ 任何剩余任务将继续运行。■ 如果没有剩余任务,则 cog 将停止并释放。■ 从方法和任务的顶层返回实际上执行的是 TASKSTOP(-1)。 · TASKHALT(task) ■ 暂停任务,直到 TASKCONT 允许其继续执行。■ Task = 0..31（固定任务)或 -1（当前任务）。■ 如果任务自行暂停,TASKNEXT() 将自动执行。■ 寄存器 TASKHLT 包含所有任务的暂停位（以相反的顺序排列）。 · PASM 中断例程可以影响 TASKHLT 位来暂停/取消暂停任务。· 如果所有任务都暂停,则切换器将等待中断来取消暂停一个任务。· TASKCONT(task) ■ 继续由 TASKHALT 暂停的任务（0..31）。· TASKCHK(task) ■ 检查任务（0..31)的状态。 ■ 如果任务空闲,则返回 0;如果任务正在运行,则返回 1;如果任务已暂停,则返回 2。 · TASKID() ■ 返回当前任务的 ID（0..31）。· 任务指针在最后 32 个空闲 cog 寄存器中向下构建,从 $11F..$100。 · 命令行界面添加二进制文件下载。 · 要编译并生成 .bin 文件： ■ PNut_v47 文件名 -c ■ PNut_v47 文件名 -cd ■ PNut_v47 文件名 -cf ■ PNut_v47 文件名 -cb · 要下载并运行 .bin 文件： ■ PNut_v47 文件名 -b ■ PNut_v47 文件名 -bd - 编译源文件 - 使用启用的 DEBUG 进行编译 - 使用附加的 Flash 加载程序进行编译 - 使用 DEBUG 和 Flash 加载程序进行编译 - 下载 .bin 文件并运行 - 下载 .bin 文件并使用 DEBUG 运行 · 在 Spin2 表达式中,#register 现在返回寄存器的地址。· #pr0 现在解析为 $1D8 · #inb 现在解析为 $1FF 注意:在 v47 中发现了导致 SEND() 和 RECV() 不起作用的错误,并在 v48 中修复。

Machine Translated by Google

v48	2025年1月6日	· 添加了预处理器,可以对源代码进行条件编译。 · 命令行语法可用于定义最多 16 个预处理器符号,这些符号可以通过编译中的所有源文件。■ PNut_v48 文件名 -D egg -D bee · 预处理命令可用于在源文件中检查、定义和取消定义预处理符号。每个文件都以命令行中定义的预处理符号开头。 ■ #DEFINE 符号 · 定义文件内前向引用的预处理器符号。 ■ #UNDEF 符号 · 取消定义文件内前向引用的预处理器符号。 ■ #IFDEF 符号 · 启动新的条件范围,如果符号已定义则为 true。 ■ #IFNDEF 符号 · 启动新的条件范围,如果符号未定义则为真。■ #ELSEIFDEF 符号 · 向当前范围添加一个替代条件,如果符号已定义,则为 true。 ■ #ELSEIFNDEF 符号 · 向当前范围添加一个替代条件,如果符号未定义,则为 true。 ■ #ELSE · 向当前范围添加默认条件,如果其他条件均不成立,则为成立。■ #ENDIF · 结束当前条件范围并恢复到任何外部范围。■ __DEBUG__ · 当启用 DEBUG 编译时定义此预处理器符号。 · 最多允许 8 层 #IFDEF/#IFNDEF 嵌套。 · 命令行界面添加了 Flash 图像输出。 · 闪存映像:■ 对于自定义闪存更新方案非常有用。■ 包含通常烧写到闪存中的加载程序和应用程序代码。■ 必须从 $000000 开始烧写到闪存中,以便在上电时启动。 · 编译并生成 Flash 映像: ■ PNut_v48 文件名 -ci - 编译并输出filename.flash
第 49 版	2025年2月2日	· CON STRUCT 声明现在被导出到父对象,就像 CON 整数和 CON 浮点数一样。 · CON STRUCT StructX(Object.StructA x[10]) StructX 是十个 StructA,已导出 · CON STRUCT StructY = Object.StructA · VAR Object.StructA StructJ · VAR ^Object.StructA StructK · PUB 名称(^Object.StructA StructL) · DAT StructM对象.StructureA StructY 是 StructA 的一个副本,导出 StructJ 是 StructA 的一个实例 StructK 是指向 StructA 的指针 StructL 是指向 StructA 的指针 StructM 是 StructA 的一个实例 · DEBUG LOGIC 显示现在可以使用 RANGE 关键字将多位组绘制为模拟波形。 · 修复了 DEBUG 显示线渲染错误,该错误导致斜率接近 1 时线条具有垂直和水平段。此错误始于 v44,原因是 DebugDisplayUnit.pas 中的 SmoothLine 程序优化不完整。 注意:在 v49 中发现了一个导致结构大小错误的错误,并在 v50 中修复。
v50	2025年2月16日	· 隐藏的位图层现在可以加载到 DEBUG PLOT 显示中,以进行整体或裁剪显示。 · 要加载一个图层(“layer_id”必须是 1 到 8):■ DEBUG(`plotname LAYER layer_id filename.bmp ) · 要将整个图层复制到显示屏,左上对齐(适用于相同大小的背景): ■ DEBUG(`plotname CROP layer_id) · 要将整个层复制到显示的某个位置:■ DEBUG(`plotname CROP layer_id display_left display_top) · 要将图层的一部分复制到显示屏,从相同的区域复制到相同的区域: ■ DEBUG(`plotname CROP layer_id left top width height) · 要将图层的一部分复制到显示中,从图层中的一个区域复制到显示中的另一个区域: ■ DEBUG(`plotname CROP layer_id layer_left layer_top width height plot_left 绘图顶部 · DAT 块和内联 PASM 部分现在支持迭代代码/数据生成,这特别有用对于参数化对象。 · DITTO count 用于启动生成块。· 块内的所有代码将生成 count 次。· Count 可以是正整数或零(不会生成任何代码)。· 该块可以包含任意数量的行。· 块内有一个特殊的索引变量 $$,它从 0 迭代到 count - 1。· 块内不允许使用任何符号,因为符号不能重新定义。· 要在块内分支,请使用 $(起点),即 TJZ reg,#$+5 。· DITTO END 终止生成块。 {Spin2_v50} 此代码… 符号1 同上 8 wypin pin_nco+ $$,#pin_base+$$ 此处允许使用符号 内部不允许使用符号,请使用 #$+n 此处允许使用符号 符号2 同上结束

Machine Translated by Google

```
生成…

符号1
        wypin pin_nco+0,#pin_base+0 wypin          $$ 从 0 迭代到 7
        pin_nco+1,#pin_base+1
        wypin pin_nco+2,#pin_base+2
        wypin pin_nco+3,#pin_base+3
        wypin pin_nco+4,#pin_base+4
        wypin pin_nco+5,#pin_base+5
        wypin pin_nco+6,#pin_base+6
        wypin pin_nco+7,#pin_base+7
符号2
```

- PUB/PRI 方法现在除了支持 ORG (cog) 内联 PASM 代码外,还支持 ORGH (hub) 内联 PASM 代码。
  - 与 ORG 类似,ORGH 将前 16 个本地长变量从集线器 RAM 加载到 cog 寄存器中,执行内联代码,然后将寄存器更新回集线器 RAM。
  - 与 ORG 内联代码不同,ORGH 内联代码不会将代码加载到 cog 寄存器 $000..$11F 中,但可以最长可达 $FFFF 条指令,因为它保留在集线器 RAM 中并在其中执行。
  - ORGH 允许内联 PASM 代码,而不会干扰 $000..$11F cog 寄存器空间,因此,那些 cog 寄存器可以完全用于驻留代码,例如中断服务程序或经常调用的快速 PASM 例程。

```
PUB go() | i

        奥格                        通过局部变量访问从集线器执行 PASM 代码
        sub i,#1 调试               SUB,1长
         (uhex (i) )长              DEBUG,1长
        0[$FFFB] 调试 (sdec         “大量 NOP,$FFFB 多头
         (i) )                       DEBUG,1 个长字符,后跟 RET,1 个长字符
        结尾                        PASM 集线器代码结束,长度限制为 $FFFF
```

- 新的@\ string\n 的作用类似于@ string ,但允许转义字符序列。
  - \a = 7,警钟
  - \b = 8,退格键
  - \t = 9,制表符
  - \n = 10,换行
  - \f = 12,换页
  - \r = 13,回车
  - \\ = 92,“\”
  - \x01 至 \xFF = $01 至 $FF
  - 未知序列只是逐字传递(即 \d = “\d”)。
- 现在允许在 CON 块表达式中使用预定义寄存器,如 PR0、IJMP1、DIRA、OUTA 和 INA。
- PASM DEBUG 指令现在可以以条件开头,而不仅仅是 _RET_。
  - 因为用于 DEBUG 的 BRK 指令在流水线早期处理,所以条件没有效果,但 _RET_ 将正常执行。
  - 为了使 BRK 指令有条件,放置了相反条件的 SKIP 指令使其在所需条件下执行 BRK。注意,这会增加 1 条指令。

```
此代码…

    IF_C 调试 (“你好”)                    仅在有条件的情况下执行 DEBUG

生成…

    IF_NC 跳过 #1                         在相反条件下,跳过下一条指令
            调试 (“你好”)                 用于 DEBUG 的 BRK 指令
```

v51	2025年4月2日	- 结构内的长变量现在可以用作方法指针。
		- 方法指针实例现在可以使用 CON STRUCT 名称来定义返回值计数。
		  - 构造 sABC(方法,时间)
		  - 增值转售 sABC ABC
		  - PUB/PRI… ABC := ABC.Method(ABC.Time) : sABC
		- 除 PUB 和 PRI 块外,SIZEOF(struct) 现在还可用于 DAT 和 VAR 块。
		- 添加了新的浮点对数和指数运算符。
		  - fpx POW fpy 返回 fpx 的 fpy 次方,3.0 POW 4.0 = 81.0
		  - LOG2 fp 返回 fp 的以 2 为底的对数,LOG2 257.0 = 8.005625
		  - EXP2 fp 返回 2 的 fp 次方,EXP2 8.005625 = 257.0
		  - LOG10 fp 返回 fp 的以 10 为底的对数,LOG10 150.0 = 2.176091
		  - EXP10 fp 返回 10 的 fp 次方,EXP10 2.176091 = 150.0
		  - LOG fp 返回 fp 的自然对数,LOG 0.0001 = -9.210340
		  - EXP fp 返回 e 的 fp 次方,EXP -9.210340 = 0.0001
		- 修复了忽略返回值“_(paramcount)”中的错误,并将下划线+括号语法更改为下划线+括号语法更清晰。由于这个bug,导致堆栈不平衡,没有人能够无论如何,已经成功使用了这个功能,因此我们抓住机会改进了它的语法。

Machine Translated by Google

		· _[4],a,b,c,d := 1,2,3,4,5,6,7,8 忽略 1,2,3,4 并将 5,6,7,8 写入 a,b,c,d · astruct, _[structdef] := method() 写入 astruct 并忽略其他结果 注意:在 v49-v51 版本中发现了一个错误,该错误导致“object.method()”调用的作用域列计算错误。该错误已修复,新的 v51a 版本已在 OBEX 中发布。请参阅下方“Spin2 概述”部分中的最后一个链接。

新版本引入的新关键字

版本	新关键字	类型	描述	最低启用
第 43 版	长字符串	方法	声明一个以长度字节开头的常量字符串。	{Spin2_v43}
v44	BYTESWAP 词交换 长交换 字节公司 词频计算 龙康普 布尔值,布尔值_ ~~充满~~ ~~复制~~ ~~交换~~ ~~补偿~~	方法 方法 方法 方法 方法 方法 调试 ~~方法~~ ~~方法~~ ~~方法~~ ~~方法~~	交换两个字节范围。 交换两个单词范围。 交换两个范围的多头。 比较两个范围的字节。 比较两个范围的单词。 比较两个范围的多头。 输出布尔值,如果非 0 则输出“TRUE”,如果为 0 则输出“FALSE”。 ~~用字节值填充结构。~~ ~~将一个结构复制到另一个结构。~~ ~~交换结构的内容。~~ ~~比较结构的内容。~~	{Spin2_v44}
v45	结构体 大小	关键词 方法	在 CON 块中,位于结构定义之前。 返回结构的大小（以字节为单位）。	{Spin2_v45}
v46	C_Z	DEBUG 输出 C 和 Z 标志状态		{Spin2_v46}
v47	任务旋转 任务下一步 任务停止 任务组 任务控制 任务检查 任务ID 新任务 这项任务 任务	方法 方法 方法 方法 方法 方法 方法 持续的 持续的 登记	初始化一个新任务。 切换到下一个未暂停的任务。 停止并释放任务。 停止一项任务。 继续一项任务。 检查任务的状态。未使用/正在运行/已停止 = 0/1/2。 获取当前任务的ID。 （-1)用于 TASKSPIN。 （-1)用于 TASKSTOP 和 TASKHALT。 保存 HALT 位的寄存器（按相反顺序）	{Spin2_v47}
v50	同上	指示	在 DAT 块中,开始/结束迭代生成部分。	{Spin2_v50}
v51	战俘 日志2 经验2 日志10 经验10 日志 经验值	操作员 操作员 操作员 操作员 操作员 操作员 操作员	浮点 x 的 y 次方函数 浮点型以 2 为底的对数函数 浮点数 2 的 x 次方函数 浮点型十进制对数函数 浮点数 10 的 x 次方函数 浮点自然对数函数 浮点 e 的 x 次方函数	{Spin2_v51}

Spin2 概述

Spin2 语言设计简洁,功能强大。Spin2 不隐藏计算机运行的底层二进制现象,而是允许您利用它进行高效的编程。Spin2 还支持 Propeller 2 汇编语言 (PASM),包括内联序列、可调用例程和独立程序。

有编程经验的人能够在很短的时间内扎实地理解 Spin2。学习 Spin2 的好处在于,它能让你专注于自己的想法,而无需费力地处理大量的类型转换和使用规则。你的大脑会乐于保持忙碌,编译+下载+执行的时间不到 1。

第二。

在 Spin2 中:

· 代码由可调用方法组成,最多可接受 127 个参数,最多返回 15 个值,并包含最多 64KB 的局部变量。· 有四种基本变量类型:BYTE（8 位）、WORD（16 位）、LONG（32 位)以及包含 BYTE、WORD、LONG 和其他嵌套 STRUCT 的结构。
每个都支持数组和位域。

· 有四种指针变量类型,提供动态字节、字、长整型和结构体访问。· 所有数学运算均以 32 位执行,并有有符号/无符号整数和 IEEE-754 浮点运算符。· 程序（称为对象)可以轻松合并其他作者编写的对象。· 对象编译为紧凑的硬件加速字节码块,这些块可调用驻留在 cog 中的短序列 PASM 代码。· 源代码不区分大小写

· 符号名称的长度最多可达32 个字符。

在本文档中,所有关键字均为大写,以便清晰起见,小写字母代表用户定义的符号名称。

Propeller 2 程序员还对另外两份核心文档感兴趣。

· Parallax Propeller 2 文档 v35 - 修订版 B/C 硅片
· Parallax Propeller 2 说明书 v35 - Rev B/C Silicon

这是包含 PNut_v51a.exe 和示例文件的最新 zip 文件:

· https://obex.parallax.com/obex/pnut-spin2-latest-version/

Machine Translated by Google

Spin2 程序结构

Spin2 程序由一个或多个对象构建而成。对象是包含至少一个公共方法的文件,以及可选的常量、数据结构、子对象,变量、附加方法和数据。对象被组装成一个顶级对象,其中包含内部的子对象层次结构。每个对象实例在运行时都会获得由对象定义的一组变量,用于维持其独特的操作状态。

对象的不同部分在块内声明,所有块都以 3 个字母的块标识符开头。

编译器实际上可以生成仅 PASM 的程序以及 Spin2+PASM 程序,具体取决于 .spin2 文件中存在哪些块。

注意:请确保文件保存为“.spin2”文件,否则示例程序将无法运行。如果您收到“expected unique parameter name”(预期唯一参数名称)的错误代码,则可能是你的问题。

堵塞标识符	区块内容	Spin2+PASM 程序	仅限 PASM 程序
反对	常量和数据结构声明(CON 是初始/默认块类型)	允许	允许
对象	子对象实例化	允许	不允许
风险自回归	变量声明 - 该对象的每个实例都有自己的 VAR 内存	允许	不允许
酒吧	供父对象和本对象使用的公共方法	必需的	不允许
优先等级	此对象中使用的私有方法	允许	不允许
数据	数据声明,包括 PASM 代码	允许	必需的

这里有一些极简的 Spin2 和 PASM 专用程序。如果你将它们复制粘贴到 PNut.exe 中,就可以按 F10 运行它们。

最小 Spin2 程序	PUB MinimalSpin2Program()		第一个 PUB 方法执行
	重复		
	PINWRITE(7..0, GETRND())		将随机模式写入 P7..P0
	等待时间(100)		等待十分之一秒,循环
最小 动脉平滑肌细胞 程序	数据	组织	在中心 $00000 处启动 PASM,用于齿轮 $000
	环形	DRVRND #0 附加项 7	将随机模式写入 P7..P0
		WAITX ##clkfreq_/10 #loop 跳转	等待十分之一秒,循环

这是一个包含每种块类型的 Spin2 程序。

全块 Spin2 程序	CON _clkfreq = 297_000_000	设置时钟频率
	OBJ vga:“VGA_640x480_文本_80x40”	实例化vga对象
	VAR 时间,i	声明对象范围的变量
	酒吧去()	第一个公共方法执行后,cog 停止
	VGA.启动(8)	在基座引脚 8 上启动 vga
	发送 := @vga.print	建立SEND指针
	发送(4,$004040,5,$00FFFF)	将浅青色设置为深青色
	时间 := GETCT()	捕获时间
	我:= @文本	将文件打印到 vga 屏幕
	重复@textend-i	
	发送(字节[i++])	
	时间 := GETCT() - 时间	以时钟周期为单位捕获时间增量
	时间:= MULDIV64(时间,1_000_000,时钟频率)	获取微秒的时间增量
	SEND(12, 打印耗时 , dec(time), 微秒。) 打印时间增量	
	PRI dec(值) | 标志,位置,数字	私有方法打印小数,三个局部变量
	标志~位	重置数字打印标志
	置 := 1_000_000_000	从十亿开始向下
	重复	
	如果标志 ||= (数字 := 值 / 位置 // 10) || 位置 == 1	打印一个数字?
	发送(“0”+数字)	是的
	如果 LOOKDOWN(地点:1_000_000_000,1_000_000,1_000)	还打印逗号吗?
	发送(”,”)	是的
	WHILE 位置 /= 10	“下一个地方,完成了吗?”
	数据	
	文本文件“VGA_640x480_text_80x40.txt”文本结束	包括用于打印的原始文件数据

Machine Translated by Google

以下是每种块类型的细分。

CON 块

CON 块用于声明可在整个对象中使用的符号常量和数据结构。

符号常量：

· 符号常量解析为 32 位值。· 符号常量可以使用“=”或通过在枚举列表
中表达其名称来赋值。· 符号常量可被文件中的每个块引用，包括 CON 块。· 符号常量可通过“对象名称.常量名称”语法被父对象的方法引用。· 如果十进制数字中
出现“.”或“e”，则该值将以 IEEE-754 单精度格式编码。

数据结构：

· 数据结构声明定义了一组压缩的字节、字、长整型和子结构。· 结构定义以 STRUCT 开头，然后是名称，接着是用括号括起来的成员列表：

· STRUCT structSTRUCT structname(BYTE|WORD|LONG|substructnamemembername{[arraysize]}, …)

· 结构的每个成员都是具有名称的 BYTE/WORD/LONG/STRUCT。如果仅给出名称，则默认为 LONG。· 可以通过在成员名称后添加 [arraysize] 将结构的每个成员声明为数组。· 结构声明可
以包含无限级别的嵌套。· 结构成员名称的作用域为结构本身，因此不存在命名空间冲突。· 数据结构限制为 $FFFF 字节，但可以实例化最多包含 $FFFF 结构的数组。
· 在将结构实例化为 VAR 块或 PUB/PRI 标头中的变量之前，不会分配存储空间。· 在 Spin2 中使用
以下语法访问结构变量和结构指针变量：

· structvar{[索引]}{.substructure_name{[索引]}...}{.byte_word_long_name{[索引]}}

· 也可以使用 STRUCT 名称和地址访问结构：· structname[address]{[index]}{.substructure_name{[index]}...}
{.byte_word_long_name{[index]}}

反对 直接的 持续的 作业	CON 启用流 = 8 单一任务 禁用流量 = 4 色彩突发频率 = 3_579_545 UpperNibs = $F0F0F0F0 PWM_base = 8 PWM_pins = PWM_base ADDPINS 7 x = 5, y = -5, z = 1 逗号分隔的赋值 HalfPi = 1.5707963268 QuarPi = HalfPi / 2.0 IEEE-754 单精度浮点值 NegG = -1e9 Micro = 1e-6 j = ROUND(4000.0 / QuarPi) 浮点数转为整数
反对 枚举 持续的 作业	缺点 #0,a,b,c,d #1,e,f,g,h a=0, b=1, c=2, d=3 e=1, f=2, g=3, h=4 （开始=0,步骤=1）（开始=1,步骤=1） #4[2],i,j,k,l #-1[-1],m,n,p i=4, j=6, k=8, l=10 (开始=4, 步长=2) m=-1, n=-2, p=-3 （开始=-1,步骤=-1） #16 start=16, step=1 q=16 r=17 qr[0] ([0] 是步长乘数) s=17 t=18 英 石 你[2] u=19 ([2] 是步长乘数) v=21 五 西 w=22 连接词 e0,e1,e2 e0=0, e1=1, e2=2 (start=0, step=1) ..每次 CON 时都会重置枚举
反对 数据 结构 定义	反对 STRUCT sPoint(x, y) sPoint 包含 long x 和 long y。如果将 sPoint 实例化为“VAR sPoint point”，则 会在内存中生成以下内容： +00: 长点.x +04: 长点.y STRUCT sLine(sPoint a, sPoint b, BYTE color) sLine 包含 sPoint a、sPoint b 和字节 颜色。如果实例化为“VAR sLine line”， sLine 会在内存中分配这部分空间： +00: 长 line.ax +04: 长 line.ay +08: 长 line.bx +0C: 长 line.by +10: 字节 line.color 如果实例化为“VAR sLine line[2]”，sLine 将在内存中分配此空间： +00: 长 line[0].ax +04: 长 line[0].ay +08: 长 line[0].bx +0C: 长 line[0].by +10: 字节 line[0].color

	+11: 长线[1].ax +15: 长线[1].ay +19: 长线[1].bx +1D: 长线[1].by +21: 字节行[1].颜色 结构 sCopyA = sLine sCopyA 是 sLine 结构的副本 STRUCT sCopyB = 对象.结构 sCopyB 是子对象结构的副本

OBJ块

OBJ 块用于将子对象实例化为当前（父)对象。

可以使用参数来实例化子对象,这些参数会覆盖子对象内同名的 CON 符号。

- 最多允许16 个参数。
- 适用于硬编码缓冲区大小、引脚等。

子对象的方法可以在运行时被执行,并且其常量可以被父对象引用。

- 一个父对象最多可以包含32 个不同的子对象。
- 子对象可以单独实例化,也可以以最多 255 个子对象的数组形式实例化。
- 每个父对象最多允许有1024 个子对象。

OBJ语法如下：

OBJ 对象名称{[实例]}： 对象文件名{.spin2} {| 参数 = 值{, ...}}

对象	OBJ VGA	：“VGA_驱动程序”	将“VGA_Driver.spin2”实例化为“vga”
子对象 实例化	老鼠	：“USB鼠标”	将“USB_Mouse.spin2”实例化为“鼠标”
	脉宽调制	：“PWM_Driver” | p = 8, w = 4	将“PWM_Driver.spin2”实例化为带有参数的“pwm”
	v[16]	：“人声合成器”	实例化一个由 16 个对象组成的数组,v[0] 到 v[15]

在父对象方法中,可以使用以下语法调用子对象方法：

对象名称.方法名称（{任何参数}）

在父对象方法中,可以使用以下语法引用子对象常量：

对象名称.常量名称

VAR 块

VAR 块用于声明符号变量,这些变量可被对象内的所有方法使用。对象的每个实例都有自己的一组变量。

- 变量可以是以下类型：
  - BYTE（8 位）,可以声明为单个或数组
  - WORD（16 位）,可以声明为单个或数组
  - LONG（32 位,默认类型）,可以声明为单个或数组
  - STRUCT（包含BYTE、WORD、LONG和嵌套STRUCT类型）,可以声明为单个或数组
  - ^BYTE 指针（32 位）,引用时可以 +/-1 为步长。
  - ^WORD 指针（32 位）,引用时可以 +/-2 为步长。
  - ^LONG 指针（32 位）,引用时可以 +/-4 为步长。
  - ^STRUCT 指针（32 位）,引用时可以按 +/-STRUCT 大小进行步进。

- 指针变量的使用语法与常规变量相同,包括大小覆盖、索引和位域,但具有一些附加功能。
  - 指针变量 读取/修改/写入指向的变量,使用语法与常规变量相同
  - ptrvar[++] 读取/修改/写入指向的变量,通过 BYTE/WORD/LONG/STRUCT (1/2/4/?) 后添加指针
  - ptrvar[--] · [+ 读取/修改/写入指向变量,通过 BYTE/WORD/LONG/STRUCT (1/2/4/?) 对指针进行后减
  +]ptrvar · [--]ptrvar 读取/修改/写入指向的变量,通过 BYTE/WORD/LONG/STRUCT（1/2/4/？)预先增加指针
  读取/修改/写入指向的变量,通过 BYTE/WORD/LONG/STRUCT（1/2/4/？)预先减去指针

  - [ptrvar] 读取/修改/写入指针本身
    - [ptrvar] := @regvar 将指针指向一个 BYTE/WORD/LONG/STRUCT
    - [ptrvar]++ 通过 BYTE/WORD/LONG/STRUCT (1/2/4/?) 后置指针

  - 指针,从外到内：
    - ptrvar 指向的变量,具有与常规变量相同的使用语法
    @ptrvar [ptrvar] 指向变量的地址,等于指针变量
    - @[ptrvar] 指针变量,等于指向变量的地址
    指针变量的地址

- 变量按照声明的顺序装入内存,从长对齐的地址开始。

Machine Translated by Google

· 为了提高效率,每个对象的前 15 个长整型变量内存通过特殊字节码访问。· 每个对象的实例都需要一个长整型变量,加上其声明的 VAR 空间,再加上 0 到 3 个字节,以便与下一个对象的 VAR 空间长对齐。· 变量在运行时初始化为零。

VAR语法如下:

VAR {{^}BYTE|{^}WORD|{^}LONG|{^}StructName} VarName{[ArraySize]} {, VarName{[ArraySize]} {, ...}}

风险自回归 多变的 声明		
向量自回归模型		默认变量大小为 LONG（32 位）。
光标模式		
多变的	位置X	前 15 个长整型具有特殊的字节码,用于更快/更小的代码。
声明	铭文	
	发送指针	因此,首先将最常见的变量声明为 long。
	字节字符串	字节变量（8 位） 字节变量数组（64 字
	字节字符串缓冲区[64]	节） 逗号分隔的声明
	字节 a,b,c[1000],d	
	WORD 当前周期	字变量（16 位） 字变量数组（200 个字）
	字循环[200]	逗号分隔的声明
	字 e,f[5],g,h[10]	
	LONG 值	长变量 长变量数组（15
	长整型值[15]	个长整型） 逗号分隔的声明
	长整型i[100],j,k,l	
	StructTypeA sRecord	StructTypeA的结构变量 StructTypeB的结构变量数组
	StructTypeB sRecord[20]	
	^BYTE 字节指针	字节指针变量（长整型） 字指针变量（长整型）
	^WORD 字指针	长整型指针变量（长整型） StructTypeC 的结构体
	^LONG longPtr	指针变量（长整型）
	^StructTypeC StructPtr	
	BYTE a,b,c, WORD d, LONG e	可以在同一行声明多种类型。
	对齐	字对齐到集线器内存,根据需要推进变量指针
	对齐	“长对齐到集线器内存,根据需要推进变量指针”……对于制作用于 FIFO 包装的长对齐缓冲区很有用
	BYTE 位图[640*480]	

PUB 和 PRI 块

PUB 和 PRI 块用于定义公共和私有可执行 Spin2 方法。

· PUB 方法可用于父对象以及定义它们的对象。· PRI 方法仅适用于定义它们的对象。· 对象中的第一个 PUB 方法是当该对象作为顶级对象运行时执行的方法。· 方法可以有 0 到 127 个输入参数长整型,由单个长整型和最多 15 个长整型的结构组成。

· ^BYTE、^WORD、^LONG 和 ^StructName 覆盖将导致参数变成指针,而不是长整型。

· 方法可以有0 到15 个输出结果长整型,由各个长整型和最多15 个长整型的结构组成。

· ^BYTE、^WORD、^LONG 和 ^StructName 覆盖将导致结果变成指针,而不是长整型。

· 方法最多可拥有64KB的局部变量。

· BYTE、WORD、^LONG 和 ^StructName 覆盖可以实例化单个或数组变量。· ^BYTE、^WORD、^LONG 和 ^StructName 覆盖将实例化指针变量。· 没有覆盖将导致长变量。· 覆盖仅适用于正在声明的变量,不适用于后续变量。· 按照声明的顺序,参数、结果和局部变量将打包到堆栈存储器中。· 内联 PASM 代码可以通过具有相同符号名称的寄存器访问前 16 个长参数/结果/局部变量。· 结果和局部变量在方法入口处初始化为零。

PUB/PRI语法如下:

PUB|PRI 方法名称 ({{^BYTE|^WORD|^LONG|^StructName} 参数{, ...}}) {: {^BYTE|^WORD|^LONG|^StructName} 结果{, ...}} {| {ALIGNW|ALIGNL} {{^}BYTE|{^}WORD|{^}LONG|{^}StructName} LocalVar{[ArraySize]}{, ...}}

PUB/PRI 声明 （方法代码将位于每个声明下方）	输入 参数 （长）	输出 结果 （长）	当地的 变量 （长整型、字、字节、结构体、结构体指针）
酒吧去（）	0	0	0
PUB 设置ADC（引脚）	1	0	0
PUB StartTx（引脚,波特率）：好的	2	1	0
PRI 旋转XY（X、Y、角度）：NewX、NewY | p,q,r	3	2	3 个长整型
PRI 洗牌() | i, j	0	0	2 个长整型
PRI FFT1024(^LONG DataPtr) | a,b,x[1024],y[1024]	1	0	1+1+1024+1024 个长整型 20000
PRI ReMix() : 长度,采样率 | WORD Buff[20000],k	0	2	个字 + 1 个长整型 1 个长整型 + 64
PRI StrCheck(StrPtrA, StrPtrB) :通过 |我,字节Str[64]	2	1	字节 sizeof(StructTypeX) x
PRI 分析(^StructTypeX pX) | StructTypeX sX[10]	1	0	10

DAT块

Machine Translated by Google

DAT块用于表达数据和PASM代码。

- 数据按照声明的顺序打包到内存中,从长对齐的地址开始。
- 数据使用以下语法表示:{symbolname} BYTE/WORD/LONG data{[count]} {,data...}
- 数据和 PASM 指令之前的符号解析为地址
  - 在 Spin2+PASM 程序中,集线器地址相对于对象的起始位置,可以按如下方式引用:
    - SymbolName 将根据符号的大小（字节/字/长)返回符号处的数据。
    - @SymbolName 将返回数据的地址。
    - @@SymbolName 将数据中的 @Symbol 转换为绝对地址（参见“DAT 数据指针”）
  - 在仅限 PASM 的程序中,集线器地址是绝对的。

数据 符号和数据			
数据			没有数据的符号占用前一个声明的大小
HexChrs 符号 0	字节	“0123456789ABCDEF”	HexChrs 是指向“0”的字节符号 symbol0 是一个指向“F”后面的字节符号
图案 符号1	单词	$CCCC,$3333,$AAAA,$5555 模式是指向 $CCCC 的单词符号	symbol1 是一个指向 $5555 之后的单词符号
十亿符号2	长的	1_000_000_000	Billion 是一个指向 1_000_000_000 的长符号 symbol2 是一个指向 1_000_000_000 之后的长符号
DoNothing 符号3	NOP		DoNothing 是一个指向 NOP 指令的长符号 symbol3 是一个指向 NOP 指令后的长符号
符号4 符号5 符 号6	字节 单词 长的		symbol4 是一个指向 $78 的字节符号 symbol5 是一个指向 $5678 的单词符号 symbol6 是一个指向 $12345678 的长符号
	长的	12345678美元	长值$12345678
	长的	1.0	IEEE-754 1.0 是长值 $3F800000
	字节	100[64]	64 个字节,值为 100
	字节 字节	10,WORD 500,LONG $FC000 允许单个值覆盖 BYTE/WORD/LONG FVARS 99、FVARS -99	允许 FVAR/FVARS 覆盖,可通过 RFVAR/RFVARS 读取
	BYTEFIT -$80,$FF WORDFIT-$8000,$FFFF		尺寸检查数据,允许单个值覆盖 尺寸检查数据,允许单个值覆盖
基线	线 长的	0,0,1919,1079	BaseLine 是标记“线”结构开始的符号 定义 line 结构的内容
文件数据	文件	“文件名”	包含二进制文件,FileDat 是指向文件的字节符号
	对齐 对齐		必要时通过发出零字节对字进行对齐 如有必要,通过发出 1 到 3 个零字节与集线器进行长对齐

数据 数据指针			
数据			
Str0 Str1 Str2 Str3	字节 字节 字节 字节	Monkeys ,0 带有符号的字符串 “大猩猩”,0 “黑猩猩”,0 “人猿”,0	
字符串列表	单词 单词 单词 单词	@Str0 @Str1 @Str2 @Str3	在 Spin2 中,这些是字符串相对于对象起始位置的偏移量 在 Spin2 中,@@StrList[i] 将返回 Str0..Str3 的地址（i = 0..3） 在仅 PASM 程序中,这些是字符串的绝对地址 （使用 WORD 假设偏移量/地址低于 64KB）

数据 执行官			
数据	组织		开始 cog-exec 程序（ORG 之前不允许有符号） COGINIT(16, @IncPins, 0) 将在免费 cog 中启动此程序
IncPins 环形	移动 添加 和 跳转	DIRA,#$FF 输出,#$01 OUTA,#$FF #环形	对于 Spin2 代码,IncPins 是 MOV 指令（长） 对于 Spin2 代码,@IncPins 是 MOV 指令的集线器地址 对于 Spin2 代码,#IncPins 是 MOV 指令的 cog 地址 对于 PASM 代码,#Loop 是 ADD 指令的 cog 地址 ($001)
	跳转	#$	$ 是当前原点,每执行一次 cog-exec 指令,原点就会增加 1
	组织 组织 组织 组织 组织	 100美元 100美元,120美元 200美元 300美元,380美元	设置 cog-exec 模式,cog 地址 = $000,cog 限制 = $1F8 (reg,均为默认值) 设置 cog-exec 模式,cog 地址 = $100,cog 限制 = $1F8 (reg,默认限制) 设置 cog-exec 模式,cog 地址 = $100,cog 限制 = $120 (reg) 设置 cog-exec 模式,cog 地址 = $200,cog 限制 = $400（LUT,默认限制） 设置 cog-exec 模式,cog 地址 = $300,cog 限制 = $380 (LUT)
	添加	register,#1 在 cog-exec 模式下,指令强制对齐 cog/LUT 寄存器	
	有机生长因子	$040	用零填充 cog 地址 $040（ORGF 之前不允许有符号）
	合身	$020	测试以确保 cog 地址不超过 $020
x y z 增益	可再生能源 可再生能源 可再生能源 可再生能源	1 1 1 16	保留 1 个寄存器,将 cog 地址提前 1,不提前 hub 地址 保留 1 个寄存器,将 cog 地址提前 1,不提前 hub 地址 保留 1 个寄存器,将 cog 地址提前 1,不提前 hub 地址 保留 16 个寄存器,将 cog 地址提前 16,不提前 hub 地址

Machine Translated by Google

数据 Hub-exec			
数据	ORGH 400 美元		以 $400 开始一个 hub-exec 程序（ORGH 之前不允许出现任何符号） COGINIT(32+16, @IncPins, 0) 将在免费 cog 中启动此程序
IncPins	移动	DIRA,#$FF	在 Spin2 中,IncPins 是 MOV 指令（长）
环形	添加	OUTA,#1	在 Spin2 中,@IncPins 是 MOV 指令的集线器地址
	跳转	#环形	在 PASM 中,Loop 是“ADD”指令的集线器地址 ($00404)
	跳转	#$	$ 是当前原点,每执行一次 hub-exec 指令,其步长为 4
	奥格		设置 hub-exec 模式,hub 原点 = $00400,原点限制 = $100000（均为默认值）
	ORGH 1000 美元		设置 hub-exec 模式,hub 原点 = $01000,原点限制 = $100000（默认限制）
	奥格	$FC000,$FC800	设置 hub-exec 模式,hub 原点 = $FC000,原点限制 = $FC800
	合身	2000美元	测试以确保中心地址不超过 2000 美元

当涉及 hub-exec 代码时,Spin2+PASM 程序和仅 PASM 程序之间存在一些差异：

Spin2+PASM 程序	· Hub-exec 代码必须使用相对寻址,因为它不位于其原点。 · LOC 指令可用于获取相关 hub-exec 代码内数据资产的地址。 · ORGH 必须指定至少$400,以便汇编纯 hub-exec 代码。 · 默认的 ORGH 地址 $400 始终是合适的,除非您正在编写将 在运行时移动到其实际的 ORGH 地址,以便它可以使用绝对寻址。		
	数据	奥格	设置 hub-exec 模式并将原点设置为 $400
		奥格 $FC000	设置 hub-exec 模式并将原点设置为 $FC000
仅限 PASM 程序	· Hub-exec 代码可以使用绝对寻址和相对寻址,因为原点始终与集线器地址匹配。 · ORGH 用零填充集线器内存,直到指定地址。		
	数据	奥格	在当前集线器地址设置集线器执行模式
		奥格 400美元	设置 hub-exec 模式并用零填充 hub 内存至 $400

Spin2 语言

评论

注释可以出现在 Spin2 或 PASM 代码中的任何位置,并有多种形式：

评论	示例	描述
到行尾	a := 0 在此处评论	· 以撇号开头,行的其余部分被忽略
到行尾 （文档）	b := 1 在此处评论	· 由两个撇号开头,行的其余部分被忽略 · 注释文本放入文档文件中
线内 或多行	x := 4, {在此评论} y := 5 {在此评论 在这里评论}	· 括号内的所有内容都会被忽略,包括行尾
线内 或多行 （文档）	x := 4, {{此处评论}} y := 5 {{此处评论 在这里发表评论}}	· 双括号内的所有内容都会被忽略,包括行尾 · 注释文本放入文档文件中
继续代码 在下一行	z := 100 * x ...在此处发表评论 ...在此处发表评论 -w	· 以三个句点开头,忽略行的其余部分 · 解析继续到下一行,就像没有遇到行尾一样

常量

常量解析为 32 位值,可以表示如下：

常量	示例	描述
十进制	1 -150 3_000_000	· 十进制值使用数字“0”到“9” · 第一个数字后允许使用下划线“_”作为占位符
十六进制	10亿美元 AA55美元 $FFFF_FFFF	· 十六进制值以“$”开头,使用数字“0”到“9”和“A”到“F” · 第一个数字后允许使用下划线“_”作为占位符
双二进制	%%21 %%01_23 %%3333_2222_1111_0000	· 双二进制值以“%%”开头,使用数字“0”到“3” · 第一个数字后允许使用下划线“_”作为占位符
二进制	%0110 %1_1111_1000 %0001_0010_0011_0100	· 二进制值以“%”开头,使用数字“0”和“1” · 第一个数字后允许使用下划线“_”作为占位符

Machine Translated by Google

漂浮	-1.0 1_250_000.0 1e9 5e+6 -1.23456e-7	· 浮点值使用数字“0”到“9”,并且其中包含“.”和/或“e”· 浮点数采用 IEEE-754 单精度 32 位格式编码· 第一个数字后允许使用下划线“_”作为占位符· 特殊浮点运算符 (+. -. *. /.) 将长整型值视为浮点数
特点	“H”	· 引号中的单个字符解析为 8 位 ASCII 值 · A → $41
细绳	“你好”	· 引号中的多个字符解析为以逗号分隔的 8 位 ASCII 值 · “Hello”→ $48、$65、$6C、$6C、$6F
压缩字符% ABCD % 123		· 最多将四个 8 位 ASCII 值打包成一个长整型、小端序、以零填充的值 · % ABCD → $44_43_42_41 · % 123 → $00_33_32_31

变量

Spin2 中,变量分为用户定义变量和永久变量。用户定义变量来源如下表所示,永久变量来源如下表所示。

· VAR 变量（集线器） · PUB/PRI
参数、返回值和局部变量（集线器） · DAT 符号（集线器） · Cog 寄存器

变量（全部为 LONG）	多变的姓名	地址或偏移量	描述	有用 Spin2	有用 Spin2-PASM	有用 仅限 PASM
枢纽位置	时钟模式 时钟频率	$00040 $00044	时钟模式值 时钟频率值	是的 是的	是的 是的	不 不
枢纽价值再投资公司	变量基础	+0	对象基指针,@VARBASE 是 VAR 基,由方法指针调用使用	或许	不	不
齿轮寄存器	PR0 PR1 PR2 PR3 PR4 PR5 PR6 PR7	$1D8 $1D9 $1DA $1DB $1DC $1DD $1DE $1DF	Spin2 <-> PASM 通信	是的 是的 是的 是的 是的 是的 是的 是的	是的 是的 是的 是的 是的 是的 是的 是的	不 不 不 不 不 不 不 不
	IJMP3 IRET3 IJMP2 雷特2 IJMP1 IRET1	$1F0 $1F1 $1F2 $1F3 $1F4 $1F5	中断 JMP 和 RET	不 不 不 不 不 不	是的 是的 是的 是的 是的 是的	是的 是的 是的 是的 是的 是的
	英字法尼图州 铅 局部放射治疗 PTRB	$1F6 $1F7 $1F8 $1F9	指针寄存器 从 COGINIT 传递的数据指针 从 COGINIT 传递的代码指针	不 不 不 不	是的 是的 是的 是的	是的 是的 是的 是的
	迪拉 DIRB 奥塔 输出 伊纳 异烟肼	$1FA 1FB美元 $1FC 1FD 美元 $1FE $1FF	P31..P0 的输出使能 P63..P32 的输出使能 P31..P0 的输出状态 P63..P32 的输出状态 P31..P0 的输入状态 P63..P32 的输入状态	是的 是的 是的 是的 是的 是的	是的 是的 是的 是的 是的 是的	是的 是的 是的 是的 是的 是的

在 Spin2 中,所有变量都可以作为位域进行索引和访问。此外,符号集线器变量可以具有 BYTE/WORD/LONG 大小覆盖：

变量使用	例子	描述
清楚的	AnyVar HubVar.WORD BYTE[地址] REG[寄存器]	集线器或永久寄存器变量 具有 BYTE/WORD/LONG 大小覆盖的集线器变量 根据地址选择集线器 BYTE/WORD/LONG 寄存器，“寄存器”可能是在 ORG 部分中声明的符号
带索引	AnyVar[索引] HubVar.BYTE[索引] LONG[地址][索引] REG[寄存器][索引]	带索引的集线器或永久寄存器变量 具有尺寸覆盖和索引的轮毂变量 通过带索引的地址来访问 Hub BYTE/WORD/LONG 使用索引注册
使用位字段	AnyVar.[位字段] HubVar.LONG.[位字段] WORD[地址].[位域] REG[寄存器].[位域]	具有位域的集线器或永久寄存器变量 具有大小覆盖和位字段的集线器变量 通过位域地址来控制集线器 BYTE/WORD/LONG 使用位字段进行注册
使用索引和位字段	AnyVar[index].[bitfield] HubVar.BYTE[索引].[位域] LONG[地址][索引].[位域] REG[寄存器][索引].[位域]	具有索引和位域的集线器或永久寄存器变量 具有大小覆盖、索引和位字段的集线器变量 通过带有索引和位域的地址来访问集线器 BYTE/WORD/LONG 带有索引和位字段的寄存器

位域是一个 10 位的值,其中包含第 4 到第 0 位中的基数和第 9 到第 5 位中的附加位数。位域可以通过几种不同的方式定义：

位域	位范围	细节

Machine Translated by Google

.[%00000_00000]	0	基位 0 以上的 0 个附加位,即单位位域
.[%00000_11111]	31	0 基位上方的附加位 31,单位位域
.[%00010_01111]	17..15	基位 15 以上的 2 个附加位,即三位位域
.[%11110_00000]	30..0	基位 0 之上还有 30 个附加位,即 31 位位域
.[%11111_10000]	15..0, 31..16	在基位 16 之上附加 31 位,环绕 32 位位域
.[%00001_11111]	0,31	基位 31 上方有 1 个附加位,环绕 2 位位域
[23]	23	仅基本位,不添加任何额外位
.[31..20]	31..20	.[] 中允许使用 Top..Bottom 语法,如果 Top < Bottom,则换行
.[5 添加 7]	12..5	ADDBITS 可用于计算位域
.[BitfieldCon]	13..9	CON BitfieldCon = 9 ADDBITS 4 BitfieldCon 在 PASM 中也很有用
.[位字段变量]	?	BitfieldVar := BaseBit ADDBITS ExtraBits 如果 BaseBit + ExtraBits > 31 则换行

除了位域之外,还有引脚域,用于选择同一 32 引脚模块中的一系列 I/O 引脚（P63..P32 或 P31..P0）。引脚域是 11 位值,
包含位 5..0 中的基极引脚号和位 10..6 中的附加引脚号。引脚字段由与引脚接口的指令使用。

平菲尔德	引脚范围	细节
PINLOW(%00000_000000)	0	0 个基极引脚上方的附加引脚,即单引脚引脚区
PINLOW(%00000_111111)	63	基座引脚 63 上方有 0 个附加引脚,即单引脚引脚区
PINLOW(%00011_100000)	35..32	32 号基座引脚上方有 3 个附加引脚,即四引脚引脚区
PINLOW(%11111_001000)	7..0, 31..8	基座引脚 8 上方有 31 个附加引脚,环绕形成 32 引脚引脚区
平洛(19)	19	仅基座引脚,不添加额外引脚
平洛(49..40)	49..40	.[] 中允许使用 Top..Bottom 语法,如果 Top < Bottom,则换行
PINLOW(11 附加 4)	15..11	ADDPINS 可用于计算 pinfield
PINLOW（PinfieldCon）	53..50	CON PinfieldCon = 50 ADDPINS 3 PinfieldCon 在 PASM 中也很有用
PINLOW(PinfieldVar)	?	PinfieldVar := BasePin ADDPINS ExtraPins 如果 BasePin + ExtraPins > 31 则换行

表达式

- 运行时表达式可以包含常量、变量和方法的返回值
- 编译时表达式只能使用常量。
- 所有表达式都可以使用运算符。

以下是一些表达式的示例：

表达	细节
字节[i++]	‘i’ 指向的字节,后增 ‘i’
(数字 := 值 / 位 // 10) 或 位 == 1	带有隐藏“数字”赋值的布尔值
位置 /= 10	将“地点”除以 10
“0”+数字	获取“数字”字符
PIN 码读取(17..12)	读取引脚 17..12

运算符

下表列出了 Spin2 中所有可用的运算符。编译时表达式可以使用一元运算符、二元运算符、三元运算符和浮点运算符。

变量前缀 运算符	学期 （仅限 PUB/PRI）	学期 优先事项	分配 （仅限 PUB/PRI）	分配 优先事项	描述	
++（前）	++变量	1	++变量	1	预增 var,返回 var	
--（前）	--var	1	--var	1	预先减少 var,返回 var	
??（前）	??变量	1	??变量	1	根据 XORO32 迭代 long var,返回伪随机值	
Var-后缀 运算符	**学期 （仅限 PUB/PRI）**	**学期 优先事项**	**分配 （仅限 PUB/PRI）**	**分配 优先事项**	**描述**	
（帖子)++	变量++	1	变量++	1	返回var,后自增var	
（邮政） -	变量--	1	变量--	1	返回 var,后减 var	
（邮政）！！	变量！！	1	变量！！	1	返回 var,后逻辑非 var（0 → -1,非 0 → 0）	
（邮政）！	变量！	1	变量！	1	返回 var,按位后非 var	
（邮政）\	var\x	1	var\x	1	返回 var,将 x 后赋值给 var	

Machine Translated by Google

（帖子）～	变量~	1	变量~	1	返回 var,清除 var 中的所有位	
（帖子）~~	变量~~	1	变量~~	1	返回 var,对 var 中的所有位进行后置	
地址 运算符	学期 （仅限 PUB/PRI）	学期 优先事项			描述	
^@	^@anyvar	1			指向任何集线器或寄存器变量的字段指针,包括位字段	
@	@hubvar	1			VAR/PUB/PRI/DAT变量的集线器地址	
@	@方法	1			指向方法的指针,可能是@object{[i]}.method	
@@	@@x	1			该对象的集线器地址 + x,　DAT x long @dat_symbol	
#	#reg_symbol	1			cog/LUT符号的寄存器地址	
一元 运算符	学期 （所有区块）	学期 优先事项	分配 （仅限 PUB/PRI）	分配 优先事项	描述	浮点 操作员
！！， 不是	!!x	12	!!= 变量	1	逻辑非（0 → -1,非 0 → 0）	
！	!x	2	!= 变量	1	按位非（1 的补码）	
-	-x	2	-= 变量	1	取反（2 的补码）	仅限 CON *
-.	-.x	2			浮点取反（切换 MSB）	所有区块
ABS	ABS x	2	ABS=变量	1	绝对值	仅限 CON *
晶圆厂	脂肪酸 x	2			浮点绝对值（清除 MSB）	所有区块
编码	编码 x	2	ENCOD=var	1	编码 MSB,0..31	
解码	解码 x	2	解码=var	1	解码,1 << (x & $1F)	
屏蔽	BMASK x	2	BMASK=var	1	位掩码,(2 << (x & $1F)) - 1	
一	一个 x	2	一个= var	1	将所有 ‘1’ 位相加,0..32	
平方根	平方根 x	2	SQRT=变量	1	无符号值的平方根	
函数求平方根	FSQRT x	2			浮点平方根	
质量日志	QLOG x	2	QLOG=变量	1	无符号值转对数 {5　整数, 27　分数}	
速递	QEXP x	2	QEXP=var	1	对数转为无符号值	
二进制 运算符	学期 （所有区块）	学期 优先事项	分配 （仅限 PUB/PRI）	分配 优先事项	描述	浮点 操作员
>>	x >> y	3	var >>= y	17	将 x 右移 y 位,插入 0	
<<	x << y	3	变量 <<= y	17	将 x 左移 y 位,插入 0	
特区	x SAR y	3	变量 SAR= y	17	将 x 右移 y 位,插入 MSB	
回流率	x ROR y	3	var ROR= y	17	将 x 右移 y 位	
角色扮演	x 方向	3	变量 ROL= y	17	将 x 左移 y 位	
修订版	x 转 y	3	var REV= y	17	x 的 0..y 位的逆序和零扩展	
零度	x 零 y	3	var ZEROX= y	17	位 y 之上的零扩展	
SIGNX	x 符号X y	3	var SIGNX= y	17	从位 y 进行符号扩展	
&	x 和 y	4	变量 &= y	17	按位与	
^	x ^ y	5	变量 ^= y	17	按位异或	
|	x | y	6	变量 |= y	17	按位或	
*	x * y	7	变量 *= y	17	有符号乘法	仅限 CON *
*.e	x *. y	7			浮点乘法	所有区块
/	x / y	7	变量 /= y	17	有符号除法,返回商	仅限 CON *
/。	x / y	7			浮点除法	所有区块
+/	x + / y	7	变量 +/= y	17	无符号除法,返回商	
//	x // 7	7	var //= y	17	有符号除法,返回余数	
+//	x +// y	7	变量 +//= y	17	无符号除法,返回余数	
声门控	x SCA y	7	变量 SCA= y	17	无符号比例,(x * y) >> 32	
南卡罗来纳大学	x SCAS y	7	变量 SCAS=y	17	有符号比例,(x * y) >> 30	
压裂	x 分数 y	7	var FRAC=y	17	无符号分数,(x << 32) / y	
+	x + y	8	向量自回归 += y	17	添加	仅限 CON *
+。	x +. y	8			浮点加法	所有区块
-	x - y	8	变量 -= y	17	减去	仅限 CON *
-.	x - y	8			浮点减法	所有区块
#>	x #> y	9	var #>= y	17	强制 x => y,有符号	仅限 CON *
<#	x <# y	9	变量 <#= y	17	强制 x <= y,有符号	仅限 CON *
添加比特	x 添加位 y	10	var ADDBITS=y	17	制作位域，（x & $1F）｜（y & $1F)<< 5	
附加信息	x 附加元件 y	10	var ADDPINS=y	17	制作 pinfield,(x & $3F) | (y & $1F) << 6	
<	x < y	11			有符号小于（返回 0 或 -1）	仅限 CON **
+<	x +< y	11			无符号小于（返回 0 或 -1）	
<.	x <. y	11			浮点小于（返回 0 或 -1）	所有区块
<=	x <= y	11			有符号小于或等于（返回 0 或 -1）	仅限 CON **
+<=	x +<= y	11			无符号小于或等于（返回 0 或 -1）	
<=。	x <=. y	11			浮点小于或等于（返回 0 或 -1）	所有区块
==	x == y	11			相等（返回 0 或 -1）	仅限 CON **
==。	x ==. y	11			浮点相等（返回 0 或 -1）	所有区块
<>	x <> y	11			不等于（返回 0 或 -1）	仅限 CON **
<>。	x <>. y	11			浮点不等于（返回 0 或 -1）	所有区块
>=	x >= y	11			有符号大于或等于（返回 0 或 -1）	仅限 CON **

+>=	x +>= y	11			无符号大于或等于（返回 0 或 -1）	
>=。	x >=. y	11			浮点大于或等于（返回 0 或 -1）	所有区块
>	x > y	11			有符号大于（返回 0 或 -1）	** 仅限 CON
+>	x +> y	11			无符号大于（返回 0 或 -1）	
>.	x > . y	11			浮点大于（返回 0 或 -1）	所有区块
<=>	x <=> y	11			有符号比较（<,=,> 返回 -1,0,1）	*** 仅限 CON
&&， 和	x && y	+≡	变量 &&= y	17	逻辑与（x <> 0 与 y <> 0,返回 0 或 -1）	
^^,异或	x ^^ y	14	变量^^= y	17	逻辑异或（x <> 0 XOR y <> 0,返回 0 或 -1）	
||,或	x || y	15	变量 ||= y	17	逻辑或（x <> 0 或 y <> 0,返回 0 或 -1）	
三元 操作员	学期 （所有区块）	优先事项 （学期）			描述	
？：	x?y:z	16			如果 x <> 0 则返回 y,否则返回 z	
分配 操作员			分配 （仅限 PUB/PRI）	优先事项	描述	
:=			变量 := x v1,v2 := x,y	17	将 var 设置为 x 将 v1 设置为 x,将 v2 设置为 y,等等。（左侧的 _ = 忽略）	
等同 操作员			分配 （仅限 CON）	优先事项	描述	
=			符号 = x	17	在 CON 块中将符号设置为 x	
漂浮 转换	学期 （所有区块）				描述	浮点 操作员
漂浮（）	浮点（x）				将整数 x 转换为浮点数	所有区块
圆形的（）	圆整（x）				将浮点数 x 转换为四舍五入的整数	所有区块
截断()	截断(x)				将浮点数 x 转换为截断整数	所有区块

*,**,***在 CON 块中,当应用于浮点常量和符号时,此运算符将具有浮点功能。

**在 CON 块中,当应用于浮点常量和符号时,关系运算符（<、<=、==、<>、>=、>)将返回 1.0 或 0.0,而不是整数 -1 或 0。

***在 CON 块中,当应用于浮点常量和符号时,<=> 运算符将返回 -1.0、0.0 或 1.0,而不是整数 -1、0 或 1。

Spin2 版本选择

为了避免未来 Spin2 关键字添加与用户符号之间的命名空间冲突,从 v43 开始实现了一种控制新关键字的方法。

编译器在 .spin2 文件中表达任何代码之前搜索“{Spin2_v##}”注释。## 是一个两位数,用于选择其对应的 Spin2 版本。
并且所有后续版本的关键字都将启用。如果未找到 {Spin2_v##},编译器将默认启用 v41 中使用的所有关键字。

例如,要选择 v43（这将启用 LSTRING() 方法）,您可以将此注释放在文件顶部：

{Spin2_v43}

低于 43 的版本号将被忽略,从而使用 v41。如果代码中发现的版本号超出了当前编译器的版本,则会产生错误。并非所有
Spin2 的未来版本将构成一个有意义的版本号,用于版本选择,因为它可能不包含任何需要门控的新关键字,但它可能会有所帮助
对于使用代码的人来说,了解作者对编译器其他方面的期望是什么。

重复 “永远循环...

插销(56) 切换 I/O 56（P2 EVAL 板上的 LED）

waitms(250)

内置方法

中心方法	细节
HUBSET(值)	使用 Value 执行 HUBSET 指令。
CLKSET（新CLKMODE,新CLKFREQ）	安全地建立新的时钟设置并更新 CLKMODE 和 CLKFREQ。
COGSPIN(CogNum, 方法 ({Pars}), StkAddr)	在齿轮中启动 Spin2 方法,如果用作表达式元素则返回齿轮的 ID，-1 = 没有空闲齿轮。
COGINIT（CogNum、PASMaddr、PTRAvalue）	在齿轮中启动 PASM 代码,如果用作表达式元素则返回齿轮的 ID，-1 = 没有空闲齿轮。
COGSTOP(CogNum)	停止齿轮 CogNum。
COGID() : 认知编号	获取此齿轮的 ID。
COGCHK(CogNum) : 正在运行	检查 cog CogNum 是否正在运行,如果正在运行则返回 -1,否则返回 0。
LOCKNEW() : 锁号	从库存中检出一个新的 LOCK,如果成功则 LockNum = 0..15,如果没有可用的 LOCK 则为 < 0。
LOCKRET(锁定号)	将某个锁放回库存。
LOCKTRY（LockNum）:LockState	尝试捕获某个 LOCK,如果成功则 LockState = -1,如果另一个齿轮捕获了该 LOCK,则 LockState = 0。
LOCKREL(锁定号)	释放某个LOCK。
LOCKCHK(锁定编号) : 锁定状态	检查某个 LOCK 的状态,LockState[31] = 已捕获,LockState[3:0] = 当前或最后一个所有者。
COGATN（CogMask）	根据 16 位 CogMask 选通齿轮的 ATN 输入。

Machine Translated by Google

POLLATN() : AtnFlag	检查此齿轮是否已收到 ATN 频闪,如果 ATN 频闪则为 AtnFlag = -1,如果没有频闪则为 0。
等待()	等待该齿轮接收 ATN 频闪。

引脚方法	细节
PINW | PINWRITE（PinField,数据）	使用数据驱动 PinField 引脚。
PINL | PINLOW(PinField)	将 PinField 引脚驱动至低电平。
PINH | PINHIGH(PinField)	将 PinField 引脚驱动至高位。
品脱 | PINTOGGLE（PinField）	驱动并切换 PinField 引脚。
PINF | PINFLOAT(PinField)	浮动 PinField 引脚。
PINR | PINREAD(PinField) : PinStates	读取 PinField 引脚。
PINSTART(PinField,模式,Xval,Yval)	启动 PinField 智能引脚:DIR=0,然后 WRPIN=Mode、WXPIN=Xval、WYPIN=Yval,然后 DIR=1。
PINCLEAR（PinField）	清除 PinField 智能引脚:DIR=0,然后 WRPIN=0。
WRPIN(PinField,数据)	将数据写入 PinField 智能引脚的‘模式’寄存器。
WXPIN(PinField,数据)	将数据写入 PinField 智能引脚的‘X’寄存器。
WYPIN(PinField,数据)	将数据写入 PinField 智能引脚的‘Y’寄存器。
AKPIN（PinField）	确认 PinField 智能密码。
RDPIN(引脚) : Zval	读取 Pin 智能引脚并确认,Zval[31] = 来自 RDPIN 的 C 标志,其他位是 RDPIN 数据。
RQPIN(引脚) : Zval	读取 Pin 智能引脚无需确认,Zval[31] = 来自 RQPIN 的 C 标志,其他位是 RQPIN 数据。

计时方法	细节
GETCT() :计数	获取32位系统计数器。
POLLCT（勾选）:过去	检查系统计数器是否已经超过“Tick”,如果超过则返回 -1,如果没有超过则返回 0。
等待CT（勾选）	等待系统计数器超过“Tick”。
WAITUS（微秒）	等待微秒,使用 CLKFREQ,持续时间不得超过 $8000_0000 个时钟。
WAITMS（毫秒）	等待毫秒,使用 CLKFREQ,持续时间不得超过 $8000_0000 个时钟。
GETSEC() :秒	获取自启动以来的秒数,使用 64 位系统计数器和 CLKFREQ,每 136 年滚动一次。
GETMS() :毫秒	获取自启动以来的毫秒数,使用 64 位系统计数器和 CLKFREQ,每 49.7 天翻转一次。

PASM接口	细节
CALL(寄存器或集线器地址)	在 Addr 处调用 PASM 代码,PASM 代码应避开寄存器 $120..$1D7 和 LUT $010..$1FF。
REGEXEC(HubAddr)	将 HubAddr 中自定义的 PASM 代码块加载到寄存器中并调用它。请参阅 REGEXEC 说明。
REGLOAD（集线器地址）	将 HubAddr 中自定义的 PASM 代码或数据块加载到寄存器中。请参阅 REGLOAD 的说明。

数学方法	细节
ROTXY（x,y,angle32bit）:rotx,roty	将(x,y)旋转32位角度并返回旋转后的(x,y)。
POLXY(长度,角度32位) : x,y	将（长度,角度32位)转换为（x,y）。
XYPOL（x,y）:长度,角度32位	将（x,y)转换为（长度,角度32位）。
QSIN(长度,步长,圆周步数) : y	将 (length,0) 旋转 (step / stepsInCircle) * 2Pi 并返回 y。stepsInCircle = $1_0000_0000 取值为 0。 stepsInCircle 未签名。
QCOS(长度,步长,圆内步数) : x	旋转 (length,0) (step / stepsInCircle) * stepsInCircle 为无符 2Pi 并返回 x。stepsInCircle = $1_0000_0000 取 0。 号。
MULDIV64(mult1,mult2,divisor) :商将“mult1”和“mult2”的 64 位乘积除以“divisor”,返回商（无符号运算）。	
GETRND() : rnd	获取随机长（来自 xoroshiro128** PRNG,在启动时使用来自 ADC 的热噪声播种）。
NAN（浮点数）:非数字	确定浮点值是否不是数字,返回 true (-1) 或 false (0)。

记忆方法	细节
GETREGS（HubAddr、CogAddr、计数）	将 CogAddr 处的计数寄存器移动到 HubAddr 处的长整型。
SETREGS（HubAddr、 CogAddr、计数）	将 HubAddr 处的计数长整型移动到 CogAddr 处的寄存器。
BYTEFILL(目的地,值,计数)	用值填充从目标开始的计数字节。
WORDFILL(目的地,值,计数)	用值填充从目的地开始的单词数。
LONGFILL(目的地,价值,数量)	从目的地开始用值填充计数长。
BYTEMOVE(目标,源,计数)	将计数字节从源移动到目标。
WORDMOVE(目标,源,计数)	将计数单词从源移动到目标。
LONGMOVE(目的地,来源,计数)	将 Count long 从源移动到目标。

BYTESWAP(地址A,地址B,计数)	从 AddrA 和 AddrB 开始交换 Count 字节数据。
WORDSWAP(地址A,地址B,计数)	交换从 AddrA 和 AddrB 开始的 Count 个字的数据。
LONGSWAP(地址A,地址B,计数)	从 AddrA 和 AddrB 开始交换 Count 长的数据。
BYTECOMP(AddrA,AddrB,Count) :匹配	比较从 AddrA 和 AddrB 开始的 Count 字节数据,如果匹配则返回 -1,如果不匹配则返回 0。
WORDCOMP(AddrA,AddrB,Count) :匹配	比较从 AddrA 和 AddrB 开始的数据 Count 个字,如果匹配则返回 -1,如果不匹配则返回 0。
LONGCOMP(AddrA,AddrB,Count) :匹配	比较从 AddrA 和 AddrB 开始的 Count 长度的数据,如果匹配则返回 -1,如果不匹配则返回 0。
SIZEOF（结构）:字节数	获取结构的大小（以字节为单位）。结构可以是结构变量、结构指针变量或 结构名称。

字符串方法	细节
STRSIZE(地址) :大小	计算 Addr 处以零结尾的字符串的字节数并返回字符串大小（不包括零）。
STRCOMP(AddrA,AddrB) :匹配	比较 AddrA 和 AddrB 处的零终止字符串,如果匹配则返回 -1,如果不匹配则返回 0。
STRCOPY(目标,源,最大值)	将最多包含 Max 个字符的以零结尾的字符串从源复制到目标。复制的字符串将 最多占用 Max+1 个字节,包括零终止符。
@“文本”:字符串地址	从引号内的文本组成一个以零结尾的字符串,返回字符串的地址。
STRING(“文本”,13) : 字符串地址	组成一个以零结尾的字符串（引用的字符和值 1..255）,返回字符串的地址。
LSTRING(你好 ,0, Terve ,0)	组成一个长度为头的字符串（引用的字符和值 0..255）,返回字符串的地址。
字节（$80，$09，$77,字$1234,长-1）	编写一个字节字符串,返回字符串的地址。允许 WORD/LONG 大小覆盖。
WORD(1_000,10_000,50_000,LONG $12345678)	组成一个字符串,返回字符串的地址。允许 BYTE/LONG 大小覆盖。
LONG(1e-6,1e-3,1.0,1e3,1e6,-50,字节$FF)	编写一个长整型字符串,返回字符串的地址。允许字节/字大小覆盖。
GETCRC(BytePtr,Poly,Count) :CRC	使用最多 32 位的自定义多项式计算从 BytePtr 开始的 Count 字节的 CRC。

指数值方法	细节
LOOKUP（索引:v1,v2..v3等）:值	使用基于 1 的索引查找值（允许的值和范围）,返回值（如果索引超出范围则返回 0）。
LOOKUPZ（索引:v1,v2..v3等）:值	使用基于 0 的索引查找值（允许的值和范围）,返回值（如果索引超出范围则返回 0）。
LOOKDOWN（值:v1、v2..v3 等）:索引	确定匹配值的基于 1 的索引（允许的值和范围）,返回索引（如果不匹配则为 0）。
LOOKDOWNZ（值:v1、v2..v3 等）:索引确定匹配值的基于 0 的索引（允许的值和范围）,返回索引（如果不匹配则为 0）。	

使用方法

返回单个结果的方法可以用作表达式中的术语：

```
x := GETRND() +// 100                    获取 0 到 99 之间的随机数

BYTEMOVE(ToStr,FromStr,STRSIZE(FromStr) + 1)
```

返回多个结果的方法（如 POLXY)可用于向其他方法提供多个参数：

```
x,y := SumPoints（POLXY（rho1,theta1）,POLXY（rho2,theta2））
```

…在哪里…

```
PRI 点数和（x1,y1,x2,y2）:x,y
    返回 x1+x2, y1+y2
```

可以将多个方法的结果分配给变量,或者通过使用下划线代替变量名来忽略它们：：

```
x,y := ROTXY（xin,yin,theta） _,y := ROTXY          同时使用 x 和 y 结果
  （xin,yin,theta） x,_ := ROTXY（xin,yin,theta）    仅使用 y 结果
                                                    仅使用 x 结果
```

赋值操作非常灵活。假设以下结构体中每个都有 5 个 long 值：

```
    数据结构1,数据结构2 := 5,4,1,7,3,8,2,0,6,9                    加载 DataStruct1 和 DataStruct2
_(数据结构1),数据结构2 := 5,4,1,7,3,8,2,0,6,9                    仅加载 DataStruct2
                    _(5),数据结构2 := 5,4,1,7,3,8,2,0,6,9         仅加载 DataStruct2
```

要忽略赋值语句右侧的多个值,可以在左侧使用“_(?)”语法,其中“?”是常量、STRUCT 名称或结构
变量/指针。

返回一个或多个结果的用户定义方法也可以用作指令,其中返回值将被忽略。但是,诸如 STRSIZE 之类的内置方法
返回结果,只能作为表达用语。

中止

Machine Translated by Google

Spin2 有一个“中止”机制,可以从任何深度的嵌套方法调用中立即返回到在方法名称前使用“\”的基调用者。单次返回值可以从中止点传送回基调用者:

PRI Sub1():错误	Sub1 调用 Sub2 并带有 ABORT 陷阱
错误 := \Sub2()	\ 表示调用方法并捕获任何 ABORT
\Sub2()	在这种情况下,ABORT 值被忽略
PRI Sub2()	Sub2 调用 Sub3
Sub3()	由于 ABORT,Sub3 永远不会返回这里
PINHIGH(0)	PINHIGH 永远不会执行
PRI Sub3()	Sub3 ABORT,返回 Sub1 并显示 ErrorCode
ABORT错误代码	ABORT 并返回 ErrorCode
平洛(0)	PINLOW 永远不会执行

无论特定方法有多少个返回值,当使用前面的“\”调用该方法时,都只有一个返回值,该返回值可能是忽略。

如果 ABORT 后未指定值,则返回零。

如果调用方法时前面带有“\”,但没有发生 ABORT,则返回零。

如果在调用链中的某处执行 ABORT 而没有“\”陷阱,则 cog 将返回顶层方法并执行 COGSTOP(COGID),从而自行关闭。

中止机制旨在作为一种从出现某些错误情况的深度嵌套子程序中返回的手段,但它可以用于任何目的。基本上,这是一种无需在调用链的每一层检查条件即可返回到基类调用者的方法。它一路返回到调用者,并使用“\”中止陷阱,携带 ABORT 值。您可以组成多级“\”中止陷阱和 ABORT 点。

方法指针

方法指针是指向方法的 LONG 值,然后用于间接调用该方法。

要建立方法指针,可以在方法名前使用“@”赋值一个长整型变量。注意,方法名后没有括号:

LongVar := @SomeMethod	当前对象中的方法
LongVar := @SomeObject.SomeMethod LongVar :=	子对象中的方法
@SomeObject[index].SomeMethod	索引子对象中的方法

方法指针可以动态生成并作为参数传递:

设置IO（@InMethod,@OutMethod）

然后通过以下方式使用方法指针来调用方法:

LongVar()	没有参数,没有返回值
长变量（第 1 项,第 2 项）	两个参数,没有返回值
Var := LongVar():1	没有参数,只有一个返回值
Var1,Var2 := LongVar（Par1）:2	一个参数和两个返回值
Var1,Var2 := POLXY(LongVar(Par1,Par2,Par3):2)	三个参数和两个返回值

编译时无法感知方法实际指向的参数数量。你需要编写方法指针的用法,以便提供正确的参数的数量,并在冒号“:”后指定适当数量的返回值,以便与指向的方法一致。

方法指针可以通过对象层次结构传递,以便从任何地方直接调用任何方法。它们还可以用来动态指向不同的具有相同数量参数和返回值的方法。

方法指针如何工作

@method 表达式生成一个 32 位值,该值具有两个位字段:

[31..20] = 方法的索引,相对于方法的对象基数。第一个方法的索引将是实例化对象数量的两倍

[19..0] = 方法 VAR 基址的地址。反过来,该方法的 VAR 基址包含该方法对象基址的地址。

通过将方法的索引和 VAR 基地址放在一起放入 32 位值中,并让 VAR 基地址包含方法的对象基地址,一个完整的方法指针以单个 long 形式建立,可以像任何其他变量一样处理。

为了容纳方法指针,每个对象实例都会保留其 VAR 空间的第一个长整型作为对象基址。当 @method 表达式执行时,第一个 long 写入的是对象的基地址。

发送

SEND 是一个特殊的方法指针,它继承自调用方法,并传递给所有被调用方法。其目的是提供高效的输出数据机制。

SEND 可以像方法指针一样被分配,但它必须指向一个接受一个参数并且没有返回值的方法:

发送:=@OutMethod

当用作方法时,SEND 将传递所有参数（包括被调用方法的任何返回值)到 SEND 指向的方法:

SEND(你好! ,GetDigit()+ 0 ,13)

Machine Translated by Google

在 SEND 参数中调用的任何方法都将继承 SEND 指针,以便它们也可以执行 SEND 方法:

```
酒吧围棋（）
    发送 := @SetLED
    重复
        发送(Flash(),$01,$02,$04,$08,$10,$20,$40,$80)

PRI 闪存() : x
    重复2
        发送（$00，$FF,$00）
    返回 $AA

PRI 设置 LED(x)
    PINWRITE(56 ADDPINS 7, !x)
    等待MS(125)
```

在上面的例子中,按重复顺序输出以下值:$00、$FF、$00、$00、$FF、$00、$AA、$01、$02、$04、$08、$10、$20、$40、$80（但对于 LED 则反转）

虽然被调用的方法继承了当前的 SEND 指针,但它可能会出于自身目的更改该指针。从该方法返回时,SEND 指针将恢复到调用该方法之前的状态。因此,SEND 指针的值会在方法调用时传播,而不会在方法返回时传播。

接收

RECV 与 SEND 类似,是一个特殊的方法指针,它继承自调用方法,并传递给所有被调用方法。其目的是提供高效的输入数据机制。

RECV 可以像方法指针一样被分配,但它必须指向一个不带参数并返回单个值的方法:

接收:=@InMethod

使用 RECV 的示例:

向量自回归

```
酒吧围棋（）
    接收:=@GetPattern
    重复
        PINWRITE(56 ADDPINS 7, !RECV())
        等待MS(125)

PRI GetPattern() : 模式
    返回解码（i++ & 7）
```

在上面的例子中,按重复顺序输出以下值:$01、$02、$04、$08、$10、$20、$40、$80（但对于 LED 则反转）

虽然被调用的方法继承了当前的 RECV 指针,但它可能会出于自身目的更改该指针。从该方法返回时,RECV 指针将恢复到调用该方法之前的状态。因此,RECV 指针的值会在方法调用时传播,而不会在方法返回时传播。

流量控制

Spin2 有三种基本的流控制结构:

如果/如果非 + 否则如果/否则如果非 + 否则	- 具有随机决策逻辑的条件执行
案例/案例_快速	- 使用单目标和多匹配测试进行条件执行
重复	- 循环执行各种模式

所有这些构造都使用相对缩进来确定哪些代码在其控制之下:

```
中频齿轮
    齿轮停止（齿轮-1）
    PINCLEAR(av_base_pin_ADDPINS 4)
    如果 cog <> 0    ..
则停止 cog    ..则清除 pin 模
式
```

流控制结构可以按任意顺序嵌套:

```
CASE 标志
    0:CASE_FAST字符
        0:      BYTEFILL(@screen,      , 屏幕尺寸)
                列:=行:= 0
        1:      列:=行:= 0
        2..7:标志:= chr
                返回
        8:      如果col
                    科尔--
        9:      重复
                    出去（””）
                当 col & 7
        10:     返回
```

Machine Translated by Google

```
        11:     颜色 := $00
        12:     颜色 := 80 美元
        13:     换行符（）
        其他:输出（chr）

    2:      col := chr // 列
    3:      row := chr // 行数
    4..7:background0_[flag-$04] := chr << 8
标志 := 0
```

el;SIF /IFNOT+ELSEIF/ELSEIFNOT+ELSE

IF 结构以 IF 或 IFNOT 开头,并可选地使用 ELSEIF、ELSEIFNOT 和 ELSE。为了成为同一棵决策树的一部分,这些关键字必须具有相同级别的缩进。

如果 <condition> 不为零,则执行 IF 或 ELSEIF 下的缩进代码。如果 <condition> 为零,则执行 IFNOT 或 ELSEIFNOT 下的代码。如果 <condition> 为零,则执行 ELSE 下的代码如果没有执行其他缩进代码则执行:

```
如果/如果非 <条件>                          - 初始 IF 或 IFNOT
    <缩进代码>
ELSEIF / ELSEIFNOT <条件>                   - 可选的 ELSEIF 或 ELSEIFNOT
    <缩进代码>
别的                                        - 可选的最终 ELSE
    <缩进代码>
```

案例/案例_快速

CASE 语句按顺序将目标值与可能的匹配列表进行比较。当找到匹配项时,将执行相关代码。

匹配值/范围必须缩进 CASE 关键字之后。多个匹配值/范围可以用逗号分隔符表示。任何与匹配值/范围必须缩进匹配值/范围之后:

```
CASE目标                                    - 具有目标值的 CASE
    <匹配> : <代码>                         - 匹配值和代码
        <缩进代码>
    <匹配..匹配> : <代码>                   - 匹配范围和代码
        <缩进代码>
    <匹配>,<匹配..匹配> : <代码>            - 匹配值、范围和代码
        <缩进代码>
    其他:<code>                             - 可选的其他情况,以防未找到匹配项
        <缩进代码>
```

CASE_FAST 与 CASE 类似,但它不是按顺序将目标与可能的匹配列表进行比较,而是使用最多 256 个条目的索引跳转表来立即分支到适当的代码,节省时间,但可能会增加编译代码量。如果只有连续的匹配值,而没有匹配范围,则生成的代码将实际上比具有多个匹配值的正常 CASE 构造要小。

为了使 CASE_FAST 编译,匹配值/范围必须是唯一的常量,并且彼此之间的差值都在 255 以内。

参见上面“流量控制”下的 CASE_FAST 示例。¶

¶

重复

所有循环都是通过 REPEAT 结构实现的,它有几种形式:

```
重复                                        - 永远重复（如果您不想让齿轮停止并停止驱动其 I/O,则将其放在程序末尾很有用）
    <缩进代码>

重复<次数>                                  - 重复 <count> 次,如果 <count> 为零,则跳过 <indented code>
    <缩进代码>

重复 <positive_count> 与 <variable>         - 重复 <positive_count> 次,同时迭代 <variable> 从 0 到 <positive_count> - 1
    <缩进代码>                              - 完成后,<variable> = <positive_count>

重复 <变量> 从 <第一个> 到 <最后一个>       - 重复迭代 <variable> 从 <first> 到 <last>,步长为 +/-1
    <缩进代码>                              - 完成后,<variable> = <last> +/- 1
```

Machine Translated by Google

```
重复 <variable> 从 <first> 到 <last> 步骤 <delta>          - 重复迭代 <variable> 从 <first> 到 <last>,步长为 +/-<delta>
    <缩进代码>                                              - 完成后,<variable> = <last> +/- <delta>

重复执行<条件>                                              - 当 <condition> 不为零时重复,<condition> 在 <indented code> 执行之前进行评估
    <缩进代码>

重复直到<条件>                                              - 重复直到 <condition> 不为零,<condition> 在 <indented code> 执行之前进行评估
    <缩进代码>

重复                                                        - 当 <condition> 不为零时重复,<condition> 在 <indented code> 执行后进行评估
    <缩进代码>
WHILE <条件>                                                - WHILE 必须与 REPEAT 具有相同的缩进

重复                                                        - 重复直到 <condition> 不为零,<condition> 在 <indented code> 执行后进行评估
    <缩进代码>
直到<条件>                                                  - UNTIL 必须与 REPEAT 具有相同的缩进
```

在 REPEAT 结构中,有两个特殊指令可用于改变执行过程:NEXT 和 QUIT。NEXT 指令将立即跳转到

在 REPEAT 结构中,决定再次循环,而 QUIT 指令将退出 REPEAT 结构并继续执行。这些指令通常用于

有条件地:

```
重复
    <缩进代码>
    如果 <条件>                                             - 可选地强制执行 REPEAT 的下一次迭代
        下一个
    <缩进代码>
    如果 <条件>                                             - 可选择退出重复
        辞职
    <缩进代码>
```

内嵌 PASM 代码

Spin2 方法可以通过在 PASM 代码前添加“ORG {start{, limit}”并以END 结尾来执行内联 PASM 代码。 “Start”是 PASM 代码将被汇编到的第一个寄存器，“limit”是不得占用的上限寄存器。“start”和“limit”的默认值分别为 $000 和 $120。

```
PUB go() | x

    重复

        组织
            GETRND WC                   将随机位旋转到 x
            皇家加勒比      x,#1
        结尾

        PINWRITE(56 ADDPINS 7, x)       将 x 输出到 P2 评估板的 LED
        等待时间(100)
```

您的 PASM 代码将在末尾添加一条 RET 指令,以确保它返回到 Spin2,以防没有早期的 _RET_ 或 RET 执行。

以下是执行内联 PASM 代码的内部 Spin2 过程:

· 保存当前流送器字节码地址,以便在 PASM 代码执行后恢复。· 将方法的前 16 个长整型变量（包括任何参数、返回值和局部变量)从集线器 RAM 复制到 cog 寄存器 $1E0..$1EF。· 将内联 PASM 代码长整型从集线器 RAM 复制到 cog 寄存器,从 ORG 后指定的寄存器地址开始（默认为 $000）。· 调用 PASM 代码。当执行中间的 _RET_ 或 RET,或执行附加的 RET 时,PASM 代码返回。· 将 cog 寄存器 $1E0..$1EF 中的 16 个长整型恢复回集线器 RAM,以更新任何修改过的方法变量。· 恢复流送器地址并恢复 Spin2 字节码执行。

在您的内联 PASM 代码中,您可以执行以下所有操作:

· 读写以下寄存器区域:· $000..$11F,用于加载 PASM 代码。您甚至可以在此范围内的不同地址加载不同的 PASM 程序,并从旋转2。· $1D8..$1DF,是通用寄存器,名为 PR0..PR7,可用于 PASM 和 Spin2 代码。· $1E0..$1EF,包含该方法的前 16 个长集线器 RAM 变量,并分配了相同的符号名称,供您在 PASM 代码中使用。· $1F0..$1FF,包括 IJMP3、IRET3、IJMP2、IRET2、IJMP1、IRET1、PA、PB、PTRA、PTRB、DIRA、DIRB、OUTA、OUTB、INA 和 INB。· LUT $000..$00F,可用于任何用途,非常适合使用 LUT 的流媒体模式。· 避免写入 $120..$1D7 和 LUT RAM $010..$1FF,因为 Spin2 解释器占用了这些区域。您可以在“Spin2_interpreter.spin2”中查看解释器代码。

· 通过执行 RDFAST/WRFAST 和 RFxxxx/WFxxxx 指令临时使用 FIFO。· 使用流器,包括利用 LUT $000..$00F 的 LUT 模式。· 使用最多 5 级硬件堆栈进行嵌套调用,包括对集线器 RAM 的调用。· 声明并引用常规符号和本地符号。这些符号在 PASM 代码之外无法访问。· 声明 BYTE、WORD 和 LONG 数据。BYTEFIT 和 WORDFIT 也是允许的。· 使用 RES、ORGF 和 FIT 命令。内联 PASM 代码中不允许使用命令 ORG、ORGH、ALIGNW、ALIGNL 和 FILE。· 建立一个中断,执行保留在 cog 寄存器 $000..$11F 中的代码。Spin2 可以适应中断,但只会短暂地将其停止。

Machine Translated by Google

- 随时通过执行_RET_ 或 RET 指令返回到 Spin2。

从 SPIN2 调用 PASM

您可以在 Spin2 中执行CALL(address)以在 cog 寄存器空间或 hub RAM 中执行 PASM 代码。

```
酒吧去（）

  重复
    调用（@随机）
    PINWRITE（56 ADDPINS 7,pr0）
    等待时间(100)


数据          奥格          hub PASM 程序将随机位旋转到 x

随机 GETRND WC
 _RET_ RCL pr0,#1
```

以下是执行 CALL 的内部 Spin2 过程：

- 保存当前流媒体字节码地址,以便在PASM 代码执行后恢复。
- 调用PASM代码。
- 恢复流媒体地址并恢复Spin2字节码执行。

在您调用的代码中,您可以执行以下所有操作：

- 读取和写入以下cog寄存器和LUT区域：
  - $000..$11F,可能包含您之前加载的 PASM 代码和/或数据。
  - $1D8..$1DF,是通用寄存器,名为 PR0..PR7,可用于 PASM 和 Spin2 代码。
  - $1E0..$1EF,可供暂存器使用,但在 Spin2 恢复时可能会被重写。
  - $1F0..$1FF,包括 IJMP3、IRET3、IJMP2、IRET2、IJMP1、IRET1、PA、PB、PTRA、PTRB、DIRA、DIRB、OUTA、OUTB、INA 和 INB。
  - LUT $000..$00F,可用于任何用途,非常适合使用 LUT 的流媒体模式。
  - 避免写入寄存器 $120..$1D7 和 LUT RAM $010..$1FF,因为 Spin2 解释器占用了这些区域。您可以查看“Spin2_interpreter.spin2”
    查看解释器代码。
- 通过执行RDFAST/WRFAST 和RFxxxx/WFxxxx 指令暂时使用FIFO。
- 使用流媒体,包括利用 LUT $000..$00F 的 LUT 模式。
- 使用最多5 级硬件堆栈进行嵌套CALL,包括对集线器RAM 的CALL。
- 建立一个中断,执行 cog 寄存器 $000..$11F 中剩余的代码。Spin2 可以适应中断,但只会短暂地停止它们。
- 随时通过执行_RET_ 或 RET 指令返回到 Spin2。

REGLOAD 和 REGEXEC

Spin2 指令REGLOAD(HubAddress)和REGEXEC(HubAddress)用于将 PASM 代码和/或数据块从集线器 RAM 加载到 cog 中,或者加载并执行 PASM 代码和/或数据块。
寄存器。

PASM 代码块和/或数据前面必须有两个字,这两个字提供起始寄存器和要加载的寄存器数量（长整型)减 1。

```
酒吧去（）

  REGLOAD(@chunk)          将自定义块从集线器加载到寄存器中

  重复
    调用（#开始）          在寄存器地址处调用块内的程序
    等待时间(100)

数据

词块                start,finish-start-1    定义块开始和大小-1

          组织      100美元                 org 可以是 $000..$120 大小

启动 DRVRND #0 ADDPINS 7                    一些代码
 _RET_ DRVNOT #8 完成                       更多代码+返回
```

REGEXEC 的工作方式与 REGLOAD 类似,但它在加载块之后还会调用块的起始寄存器。

在下面的例子中,REGEXEC 在上层寄存器内存中启动了一段代码,该代码设置了一个定时器中断,然后返回到 Spin2。与此同时,由于 Spin2
方法每 100ms 重复随机化引脚 60..63,加载到上部寄存器内存的代码块会持续定时器中断,并每隔 100ms 切换引脚 56..59
500毫秒。请注意,寄存器$000..$117仍可供其他代码块使用,并且中断2和3仍未使用。

```
酒吧去（）

  REGEXEC(@chunk)                    加载自定义块并执行
                                     chunk启动定时器中断并返回
  重复
    PINWRITE(60 ADDPINS 3, GETRND())  随机化引脚 60..63
    等待时间(100)                     引脚 56..59 通过中断切换

数据

词块                开始,结束-开始-1 $118   定义块的起始和大小-1
          组织                             org 可以是 $000..$120 大小


开始 MOV            IJMP1,#isr             设置 int1 向量
          设置INT1 #1                      将 int1 设置为 ct-passed-ct1 事件
```

Machine Translated by Google

```
                GETCT PR0 _ret_                                  获取 ct
ADDCT1 PR0,bigwait                                               设置初始 ct1 目标,返回 Spin2

                DRVNOT #56 附加项 3                              中断服务程序,切换 56..59    设置下一个 ct1 目标    从中断返回
                ADDCT1 PR0,大等待
                雷蒂1

bigwait 长完成          20_000_000 / 2                           RCFAST 上 500 毫秒
```

数据结构

数据结构通过封装可以轻松地组织变量。可以声明一组相关变量,并将其作为单个参数传递,可以是值传递,也可以是指针传递。

在下面的例子中,drawLines 函数传入了一个“@Lines”,它是线结构体数组的基地址。该地址被 drawLines 函数以结构体指针“pLine”的形式接收,并在该指针处使用。

```
{Spin2_v46}

CON STRUCT sPoint（字节 x,字节 y）
        STRUCT sLine(sPoint a, sPoint b, 字节颜色)

        行数 = 100

VAR sLine 行[行数]                                   Line 是 sLine 结构的数组


PUB go() | i

    调试（`绘制 myplot 大小 256 256 hsv8x 更新）

    重复 重复
        LineCount 与 i                                 设置随机线
            Line[i].ax := getrnd()
            Line[i].ay := getrnd()
            Line[i].bx := getrnd()
            Line[i].by := getrnd()
            Line[i].color := getrnd()

        绘制线（@Line,LineCount）                      通过传递 Line 基结构地址来绘制它们


PRI drawLines(^sLine pLine, count) | i    pLine 是 sLine 类型的结构指针

    调试（`myplot 清除线条大小 2）

    用 i 重复计数
        调试（`myplot 颜色`（pLine[i].color））调试（`myplot 设置`
         （pLine[i].ax,pLine[i].ay））调试（`myplot 线`（pLine[i].bx,pLine[i].by））


    调试（`myplot 更新）
```

小结构可以通过值传递,也可以通过地址传递:

· 长度不超过15米的结构…

· 可以通过值作为多长参数和返回值传递 · 将在最后一个长整型中用零填充任何未使用的高位字节 · 可以用于多长赋值（结构:= 1,2,3）

· 长度不超过 1 的结构…

· 可以作为单长参数和返回值按值传递 · 将在长整型中用零填充任何未使用的高位字节

除了针对小型结构的一般任意分配之外,还有四种特殊的结构分配操作可对任意大小的结构进行操作:

· 结构~ 用 $00 填充结构 用 $FF 填充结构 · structure~~
· structureA :=:structureB 交换结构的内容 · structureA :=structureB 复制结构的内容

· 结构 := 1,2,3 将任意长整型写入结构体（最多 15 个长整型）

有两种结构比较操作可解析为单个表达式项:

· 结构A == 结构B 检查结构 相等性并返回 TRUE/FALSE · 结构A <> 结构B 检查结构 不等性并返回 TRUE/FALSE

现场指针

字段指针允许您指向任何集线器字节/字/长位置或 cog 寄存器,而无需在传递和使用字段指针时进行区分。

任何集线器或寄存器变量都可以获取字段指针。通过在字段指针声明中指定可选的位范围,即可使用该字段指针索引非标准位宽的子变量数组。

Machine Translated by Google

^@variable 运算符将返回一个 32 位值,该值将完全定义变量的位置以及组成它的位范围。

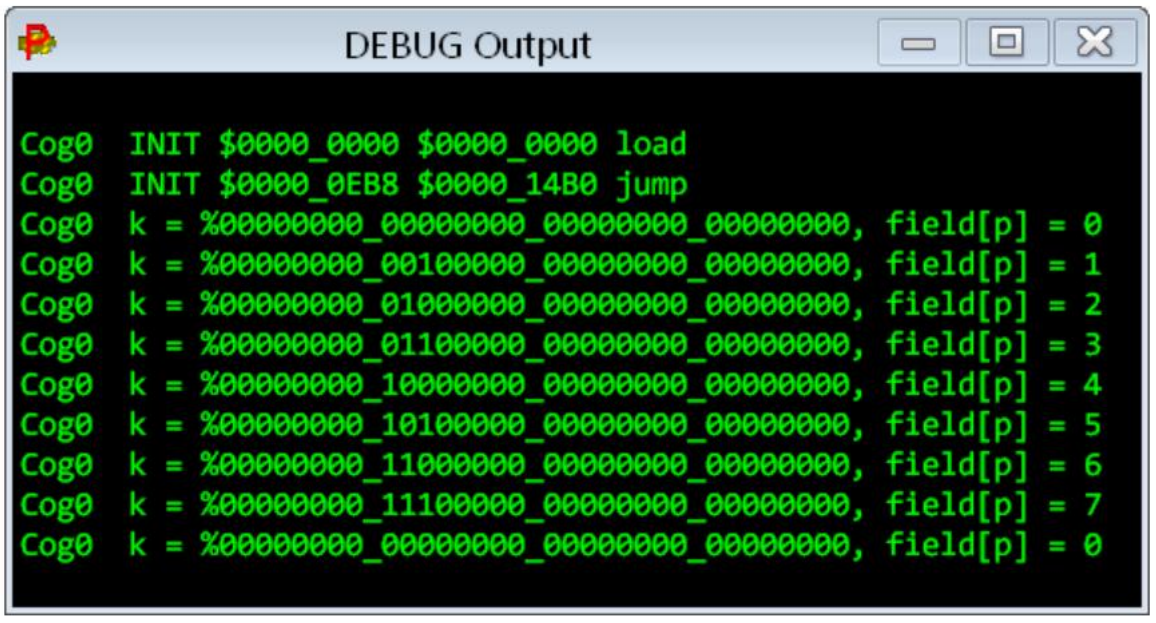
一旦获得该字段指针,就可以在方法之间传递它,并使用 FIELD[fieldpointer] 来访问它指向的变量。

索引也支持通过 FIELD[fieldpointer][index] 进行。如果指向的变量长度为两位,则索引将以两位为单位进行递增。非 2 的幂次方的位域大小也适用,但必须指向集线器存储器中的 WORD 或 LONG 类型,以便基址读/写地址可以以字节为增量移动,从而允许在 WORD 或 LONG 的高位字节中读取或写入高位。

在计划索引 n 位字段数组时,请确保为该数组选择足够大的变量大小（BYTE/WORD/LONG）,以便索引访问始终在 BYTE/WORD/LONG 边界内。例如,单位字段始终在 BYTE 数组中工作,但三位字段可以跨越两个字节,因此它们需要一个 WORD 数组。任何十位或更大的字段都需要一个 LONG 数组,因为它们可能跨越三个字节。

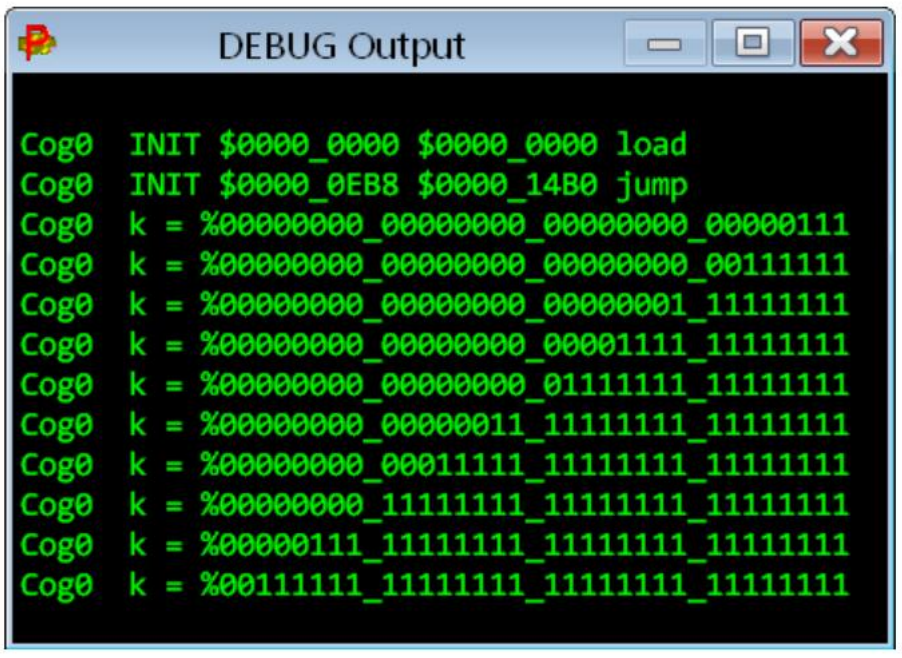
下面是一个示例程序,它使用字段指针访问 long 变量中的三位。请注意,指针“p”可以在代码中传递,然后与 FIELD 一起使用来读取、写入或修改其指向的数据。

```
CON _clkfreq = 10_000_000

PUB go() | p, k

    p := ^@k.[23..21]     获取指向 k 内三位的指针

    重复 9
        debug(ubin_long(k), udec(field[p]++))     通过 p 显示 k 和 3 位
```



下面是使用索引来影响连续位域的示例。

```
CON _clkfreq = 10_000_000

PUB go() | p,k,i

    p := ^@k.[2..0]                     获取指向 k 的最低三位的指针

    重复 10 字段[p]
        [i++]~~                         每次设置三位,向上
        debug(ubin_long(k))
```



除了支持可选的位字段外,字段指针还区分集线器内存和寄存器。因此,字段指针无需任何特殊语法即可引用这两种类型的内存。

字段指针编码为 32 位值的方式如下:

变量语法	字段指针声明	字段指针编码

注册名称 REG[地址]	^@register ^@register.[bbbbb addbits sssss] ^@register.[msbit..lsbit] ^@register.[bit]	00_11111_00000_00000000000rrrrrrrrr 00_sssss_bbbbb_00000000000rrrrrrrrr
字节名 BYTE[地址]	^@byte ^@byte.[bbbbb addbits sssss] ^@byte.[msbit..lsbit] ^@byte.[bit]	01_00111_00000_aaaaaaaaaaaaaaaaaaaa 01_sssss_bbbbb_aaaaaaaaaaaaaaaaaaaa
单词名称 WORD[地址]	^@word ^@word.[bbbbb addbits sssss] ^@word. [msbit..lsbit] ^@word.[位]	10_01111_00000_aaaaaaaaaaaaaaaaaaaa 10_sssss_bbbbb_aaaaaaaaaaaaaaaaaaaa
长名称 LONG[地址]	^@long ^@long.[bbbbb addbits sssss] ^@long.[msbit..lsbit] ^@long.[bit]	11_11111_00000_aaaaaaaaaaaaaaaaaaaa 11_sssss_bbbbb_aaaaaaaaaaaaaaaaaaaa

请注意,由于字段指针的底部 20 位是基地址,因此在使用时可以方便地添加或减去它们的值:

FIELD[字段指针 + @record].
FIELD[字段指针 + SectorBase(x)].
FIELD[字段指针 - 4].

调试

Spin2 编译器包含一个隐秘的调试程序,可以随应用程序自动下载。它使用最后 16 KB 的 RAM,以及每个 Spin2 DEBUG 语句,每个 PASM DEBUG 语句对应一条指令。您可以在应用程序中放置 DEBUG() 语句,其中包含输出命令,将在应用程序运行时串行传输变量和方程的状态。每次在执行过程中遇到 DEBUG 语句时,调试程序都会调用并输出该语句的消息。此外,还有一个单步 PASM 调试器,可以通过不包含括号内的任何参数。在 PNut 中,通过在常用的 F10 键“运行”或 F11 键“编程”中添加 Ctrl 键来启动调试;在 PropellerTool 中,通过启用调试使用 Ctrl+D 进入调试模式,然后像平常一样使用 F10 或 F11。这将使用所有 DEBUG 语句编译您的应用程序,并将调试程序添加到下载中,然后打开 DEBUG 输出窗口,该窗口在应用程序启动时开始接收消息。

关于 DEBUG 系统你需要知道的事情

- 要使用调试器,必须配置至少 10 MHz 的时钟,该时钟来自晶振或外部输入。不能使用 RCFAST 或 RCSLOW。
- 调试程序占用了集线器 RAM 的前 16 KB,重新映射到 $FC000..$FFFFF 并处于写保护状态。位于 $7C000..$7FFFF 的集线器 RAM 将不再可用的。
- 定义每个 DEBUG() 语句的数据都存储在 RAM 顶部 16 KB 的调试器映像中,从而最大限度地减少对应用程序代码的影响。
- 在 Spin2 中,每个 DEBUG 语句都会添加三个字节,加上引用变量和解析 DEBUG() 语句中使用的运行时表达式所需的任何代码。
- 在PASM 中,每个DEBUG 语句都会添加一条指令（长指令）。
- 当未使用 DEBUG 模式进行编译时,DEBUG 语句会被编译器忽略,因此在未使用调试时无需将其注释掉。
- 如果您的应用程序中不存在 DEBUG 语句,并且您正在运行调试程序,则在启动 cogs 时您仍会收到通知消息。
- 通过在常用的 F9..F11 键之前按住 CTRL（在 PNut 中)或使用 CTRL+D（在 Propeller Tool 中)启用调试来调用调试,以编译、下载和程序到闪存。
- 在执行过程中,当遇到 DEBUG() 语句时,文本消息以 8-N-1 格式在 P62 上以 2 Mbaud 串行发送。
- DEBUG() 消息始终以“CogN ”开头,其中 N 是齿轮编号,后跟两个空格,并且始终以 CR+LF（新行)结尾。
- 您的应用程序中最多可以存在 255 个 DEBUG() 语句,因为 BRK 指令用于中断和选择特定的 DEBUG() 语句定义。
- 您可以定义几个符号来修改调试器行为:DEBUG_COGS、DEBUG_DELAY、DEBUG_BAUD、DEBUG_PIN、DEBUG_TIMESTAMP 等。请参阅表格。
- 每次启动启用调试的 cog 时,都会输出一条调试消息,以指示 cog 编号、代码地址 (PTRB)、参数 (PTRA) 和“加载”或“跳转”模式。
- 对于 Spin2,DEBUG() 语句可以输出表达式和变量值、集线器字节/字/长数组以及寄存器数组。
- 对于 PASM,DEBUG() 语句可以输出寄存器值/数组、集线器字节/字/长数组、C/Z 标志和常量。使用 PASM 语法:隐含寄存器或#即时。
- DEBUG() 输出数据可显示为浮点数、十进制、十六进制或二进制,并可调整大小为字节、字、长整型或自动。此外,还支持集线器字符串。
- DEBUG() 输出命令同时显示源和值:“DEBUG(UHEX(x))”可能输出“x = $ABC”。
- 输出数据的 DEBUG() 命令可以有多组参数,以逗号分隔:SDEC(x,y,z) 和 LSTR(ptr1,size1,ptr2,size2)
- 数据之间自动输出逗号:“DEBUG(UHEX_BYTE(d,e,f), SDEC(g))”可能输出“d = $45, e = $67, f = $89, g = -1_024”。
- 所有 DEBUG() 输出命令都有替代版本,以“_”结尾,仅输出值:DEBUG(UHEX_BYTE_(d,e,f)) 可能输出“$45、$67、$89”。
- DEBUG() 语句除了命令之外,还可以包含逗号分隔的字符串和字符:DEBUG(We got here! Oh, Nooooo... , 13, 13)
- DEBUG() 语句可能包含 IF() 和 IFNOT() 命令,用于控制语句中的后续输出。初始 IF/IFNOT 将控制整个消息。
- DEBUG() 语句可能包含一个最终的 DLY（毫秒)命令来减慢 cog 的消息传递速度,因为消息可能以每秒约 10,000 条的速度流动。第二。
- DEBUG() 语句可能包含 PC_KEY() 和 PC_MOUSE() 命令,以将主机键盘和鼠标的状态放入 DEBUG() 显示中。
- DEBUG() 串行输出可以重定向到不同的引脚,以不同的波特率,以便在其他地方显示/记录。
- 不带括号的 DEBUG 将从 Spin2 或 PASM 调用该 cog 的 PASM 级调试器。普通 DEBUG 命令的数量没有限制。
- 通过定义 DEBUG_COGINIT 或 DEBUG_MAIN 符号,PASM 级调试器将在每个 cog 的 COGINIT 上自动启动。
- LOCK[15] 由调试器分配,并在调试中断期间在所有 cog 中使用,以分时共享 DEBUG 串行 TX 和 RX 引脚以及一些 RAM。
- P63 配置为长存储库模式,用于保存调试中断之间的时钟频率值。时钟频率改变时,必须更新该寄存器。
- 命令行支持仅 DEBUG 模式:PNut -debug {CommPort if not 1} {BaudRate if not 2_000_000}

Machine Translated by Google

DEBUG() 语句中使用的命令

条件语句	细节
IF(条件)	如果条件 <> 0,则继续执行 DEBUG() 语句中的下一个命令,否则跳过所有剩余的命令并输出 CR+LF。如果用作 DEBUG() 语句中的第一个命令,IF 将控制该语句的所有输出,包括“CogN +CR+LF。这样,DEBUG() 消息就可以被完全抑制,以便您可以过滤掉重要的内容。
IFNOT(条件)	如果条件 = 0,则继续执行 DEBUG() 语句中的下一个命令,否则跳过所有剩余的命令并输出 CR+LF。如果用作 DEBUG() 语句中的第一个命令,IFNOT 将控制该语句的所有输出,包括“CogN +CR+LF。这样,DEBUG() 消息就可以被完全抑制,以便您可以过滤掉重要的内容。

布尔输出*	细节	输出
BOOL(值)	如果值不为 0,则输出“TRUE”,如果为 0,则输出“FALSE”。	正确 / 错误

字符串输出*	细节	输出
ZSTR(hub_pointer)	在 hub_pointer 处输出以零结尾的字符串。	“你好！”
LSTR(hub_pointer,大小)	在 hub_pointer 处输出‘size’个字符的字符串。	“再见。”

浮点输出*	细节	最小输出	最大输出
FDEC(值)	输出浮点值。	-3.4e+38	3.4e+38
FDEC_REG_ARRAY(reg_pointer,大小)	将寄存器数组输出为浮点值。	-3.4e+38	3.4e+38
FDEC_ARRAY(hub_pointer,大小)	将集线器长数组输出为浮点值。	-3.4e+38	3.4e+38

十进制输出,无符号 *	细节	最小输出	最大输出
UDEC(值)	输出无符号十进制值。	0	4_294_967_295
UDEC_BYTE(值)	输出字节大小的无符号十进制值。	0	255
UDEC_WORD(值)	输出字大小的无符号十进制值。	0	65_535
UDEC_LONG(值)	输出长整型无符号十进制值。	0	4_294_967_295
UDEC_REG_ARRAY(reg_pointer,大小)	将寄存器数组输出为无符号十进制值。	0	4_294_967_295
UDEC_BYTE_ARRAY(hub_pointer,size)	将集线器字节数组输出为无符号十进制值。	0	255
UDEC_WORD_ARRAY(hub_pointer,size)	将集线器字数组输出为无符号十进制值。	0	65_535
UDEC_LONG_ARRAY(hub_pointer,size)	将集线器长数组输出为无符号十进制值。	0	4_294_967_295
十进制输出,有符号 *	细节	最小输出	最大输出
SDEC(值)	输出有符号的十进制值。	-2_147_483_648	2_147_483_647
SDEC_BYTE(值)	输出字节大小的有符号十进制值。	-128	127
SDEC_WORD(值)	输出字大小的有符号十进制值。	-32_768	32_767
SDEC_LONG(值)	输出长整型有符号十进制值。	-2_147_483_648	2_147_483_647
SDEC_REG_ARRAY(reg_pointer,大小)	输出寄存器数组作为有符号的十进制值。	-2_147_483_648	2_147_483_647
SDEC_BYTE_ARRAY(hub_pointer,size)	将集线器字节数组输出为有符号的十进制值。	符号-128	127
SDEC_WORD_ARRAY(hub_pointer,size)	将集线器字数组输出为有符号的十进制值。	-32_768	32_767
SDEC_LONG_ARRAY(hub_pointer,size)	将集线器长数组输出为有符号的十进制值。	-2_147_483_648	2_147_483_647
十六进制输出,无符号 *	细节	最小输出	最大输出
UHEX(值)	输出自动调整大小的无符号十六进制值。	0美元	$FFFF_FFFF
UHEX_BYTE(值)	输出字节大小的无符号十六进制值。	$00	$FF
UHEX_WORD(值)	输出字大小的无符号十六进制值。	$0000	$FFFF
UHEX_LONG(值)	输出长整型无符号十六进制值。	$0000_0000	$FFFF_FFFF
UHEX_REG_ARRAY(reg_pointer,大小)	输出寄存器数组作为无符号十六进制值。	$0000_0000	$FFFF_FFFF
UHEX_BYTE_ARRAY(hub_pointer,size)	将集线器字节数组输出为无符号十六进制值。	$00	$FF
UHEX_WORD_ARRAY(hub_pointer,size)	将集线器字数组输出为无符号十六进制值。	$0000	$FFFF
UHEX_LONG_ARRAY(hub_pointer,size)	将集线器长数组输出为无符号十六进制值。	$0000_0000	$FFFF_FFFF
十六进制输出,有符号 *	细节	最小输出	最大输出
SHEX(值)	输出自动大小的有符号十六进制值。	-$8000_0000	$7FFF_FFFF

SHEX_BYTE(值)	输出字节大小的有符号十六进制值。	-80美元	7F
SHEX_WORD(值)	输出字大小的有符号十六进制值。	-8000美元	$7FFF
SHEX_LONG(值)	输出长尺寸有符号十六进制值。	-$8000_0000	$7FFF_FFFF
SHEX_REG_ARRAY(reg_pointer,大小)	输出寄存器数组作为有符号的十六进制值。	-$8000_0000	$7FFF_FFFF
SHEX_BYTE_ARRAY(hub_pointer,size) 将集线器字节数组输出为有符号的十六进制值。		-80美元	7F
SHEX_WORD_ARRAY(hub_pointer,size) 以有符号的十六进制值形式输出集线器字数组。		-8000美元	$7FFF
SHEX_LONG_ARRAY(hub_pointer,size) 输出集线器长数组作为有符号的十六进制值。		-$8000_0000	$7FFF_FFFF
二进制输出,无符号 *	细节	最小输出	最大输出
UBIN(值)	输出自动大小的无符号二进制值。	%0	%11111111_11111111_11111111_11111111
UBIN_BYTE(值)	输出字节大小的无符号二进制值。	%00000000	%11111111
UBIN_WORD(值)	输出字大小的无符号二进制值。	%00000000_00000000	%11111111_11111111
UBIN_LONG(值)	输出长整型无符号二进制值。	%00000000_00000000_00000000_00000000	%11111111_11111111_11111111_11111111
UBIN_REG_ARRAY(reg_pointer,大小)	输出寄存器数组作为无符号二进制值。	%00000000_00000000_00000000_00000000	%11111111_11111111_11111111_11111111
UBIN_BYTE_ARRAY(hub_pointer,size) 将集线器字节数组输出为无符号二进制值。		%00000000	%11111111
UBIN_WORD_ARRAY(hub_pointer,size) 将集线器字数组输出为无符号二进制值。		%00000000_00000000	%11111111_11111111
UBIN_LONG_ARRAY(hub_pointer,size) 将集线器长数组输出为无符号二进制值。		%00000000_00000000_00000000_00000000	%11111111_11111111_11111111_11111111
二进制输出,有符号 *	细节	最小输出	最大输出
SBIN(值)	输出自动大小的有符号二进制值。	-%10000000_00000000_00000000_00000000	%01111111_11111111_11111111_11111111
SBIN_BYTE(值)	输出字节大小的有符号二进制值。	-%10000000	%01111111
SBIN_WORD(值)	输出字大小的有符号二进制值。	-%10000000_00000000	%01111111_11111111
SBIN_LONG(值)	输出长尺寸有符号二进制值。	-%10000000_00000000_00000000_00000000	%01111111_11111111_11111111_11111111
SBIN_REG_ARRAY(reg_pointer,大小)	输出寄存器数组作为有符号的二进制值。	-%10000000_00000000_00000000_00000000	%01111111_11111111_11111111_11111111
SBIN_BYTE_ARRAY(hub_pointer,size) 将集线器字节数组输出为有符号二进制值。		-%10000000	%01111111
SBIN_WORD_ARRAY(hub_pointer,size) 将集线器字数组输出为有符号二进制值。		-%10000000_00000000	%01111111_11111111
SBIN_LONG_ARRAY(hub_pointer,size) 将集线器长数组输出为有符号二进制值。		-%10000000_00000000_00000000_00000000	%01111111_11111111_11111111_11111111

* 这些命令接受多个参数或多组参数。其他名称相同但以“_”结尾的命令也可用于仅输入值

输出（即 BOOL_、ZSTR_、LSTR_、UDEC_）。

各种各样的	细节
DLY（毫秒）	延迟几毫秒,以减慢此齿轮的连续消息输出速度。DLY 仅允许作为最后一个命令 在 DEBUG() 语句中,因为它在延迟之前释放了 LOCK[15],从而允许其他 cog 捕获 LOCK[15],以便它们 可以控制 DEBUG() 串行传输引脚并输出自己的 DEBUG() 消息。
PC_KEY（指向长指针）	用于图形 DEBUG() 显示 - 必须是 DEBUG() 语句中的最后一个命令。 将过去 100 毫秒内发生的任何新的主机 PC 按键事件返回到芯片内部的一个 long 中。DEBUG() 显示 必须集中注意力才能注意到按键动作。 长键 接收按键的键长（如果没有按键则为 0） 0 = <无按键> 1 = 左箭头 2 = 右箭头 3 = 向上箭头 4 = 向下箭头 5 = 主页 6 = 结束 7 = 删除 8 = 退格键 9 = Tab 10 = 插入 11 = 向上翻页 12 = 向下翻页 13 = 输入 27 = Esc 32..126 = 空格至“~”,包括所有符号、数字和字母 如果在 Spin2 代码中使用,则 long 必须在集线器中（使用 @key 作为指针）。 如果在 PASM 代码中使用,则 long 必须是 cog 寄存器（使用 #key 作为指针）。
PC_MOUSE（指针指向_7_长整型）	用于图形 DEBUG() 显示 - 必须是 DEBUG() 语句中的最后一个命令。 将当前主机PC鼠标状态返回到芯片内部的7位长的结构中,排列如下： 长曝光 DEBUG 显示内的 X 位置（如果鼠标在外面,则 xpos<0 且 ypos<0） 长句 调试显示中的 Y 位置 长轮三角洲 滚轮增量,如果改变则为 0 或 +/-1（DEBUG 显示必须具有焦点） 长按钮 左键状态,按下时为 0 或 -1 长按钮 中键状态,按下时为 0 或 -1 长按钮 右键状态,按下时为 0 或 -1 长像素 鼠标位置的像素颜色,如果超出 DEBUG 显示范围,则为 $00_RR_GG_BB 或 -1 如果在 Spin2 代码中使用,则七个 long 必须位于集线器中（使用 @xpos 作为指针）。 如果在 PASM 代码中使用,这七个 long 必须是 cog 寄存器（使用 #xpos 作为指针）。

Machine Translated by Google

C_Z	将 C 和 Z 标志输出为“C=? Z=?”。在 PASM 代码中很有用。

可以定义修改 DEBUG 行为的符号

CON 符号	默认	目的
下载波特率	2_000_000	设置下载波特率。
调试_COGS	%11111111	选择哪些 cogs 启用了调试中断。位 7..0 启用 cogs 7..0 中的调试中断。
调试认知	不明确的	通过声明此符号,当发生 COGINIT 时,每个 cog 的 PASM 级调试器将首先被调用。
调试_主	不明确的	通过声明此符号,每个 cog 的 PASM 级调试器将在 COGINIT 发生时首次被调用,并且它将准备好 单步执行主程序（非中断)代码。在这种情况下,DEBUG 命令将被忽略,直到您选择“DEBUG” 调试器中的敏感度。
调试延迟	0	在您的应用程序运行并开始传输 DEBUG 消息之前设置一个以毫秒为单位的延迟。
调试_PIN_TX	62	设置 DEBUG 串行输出引脚。要打开 DEBUG 窗口,DEBUG_PIN 必须为 62。
调试_PIN_RX	63	设置 DEBUG 串行输入引脚以便与主机 PC 进行交互。
调试波特率	DOWNLOAD_BAUD 设置	DEBUG 波特率。如果 DEBUG_BAUD 小于 DOWNLOAD_BAUD,可能需要添加 DEBUG_DELAY。
调试时间戳	不明确的	通过声明此符号,每个 DEBUG 消息都将带有 64 位 CT 值的时间戳。
调试日志大小	0	设置用于收集 DEBUG 消息的“DEBUG.log”文件的最大大小（以字节为单位）。值为 0 时将禁止日志文件 一代。
调试_左	（动态的）	设置 DEBUG 消息窗口出现的左屏幕坐标。
调试顶部	（动态的）	设置 DEBUG 消息窗口出现的顶部屏幕坐标。
调试宽度	（动态的）	设置 DEBUG 消息窗口的宽度。
调试高度	（动态的）	设置 DEBUG 消息窗口的高度。
调试_显示_左	0	设置任何 DEBUG 显示将出现的整体左屏幕偏移（添加到每个 DEBUG 显示中的“POS”x 坐标）。
调试显示顶部	0	设置任何 DEBUG 显示将出现的整体顶部屏幕偏移（添加到每个 DEBUG 显示中的“POS”y 坐标）。
调试_WINDOWS_OFF	0	如果设置为非零值,则禁止在下载后打开任何 DEBUG 窗口。
调试掩码	不明确的	为该符号分配一个 32 位掩码值,允许根据 特定的掩码位。具体做法是,在 DEBUG 关键字后方括号中放置一个掩码位号 (0..31),然后 在任何参数之前:DEBUG[MaskBitNumber]{(parameters…)}。如果特定的掩码位为高,则 DEBUG 将 编译,否则将被忽略。
调试_禁用	不明确的	为此符号分配非 0 值将禁用文件/对象中的所有 DEBUG 语句。

Spin2 中的简单 DEBUG 示例

```
CON _clkfreq = 10_000_000                 设置 10 MHz 时钟（假设 20 MHz 晶振）

PUB go() | i
    重复 i 从 0 到 9                       从 0 数到 9
        调试（UDEC（i））                  调试,输出 i
```

使用 Ctrl-F10 运行时,将打开“调试”窗口并显示以下内容：

```
Cog0 INIT $0000_0000 $0000_0000 加载
Cog0 INIT $0000_0D6C $0000_10BC 跳转
Cog0 i = 0
Cog0 i = 1
Cog0 i = 2
Cog0 i = 3
Cog0 i = 4
Cog0 i = 5
Cog0 i = 6
Cog0 i = 7
Cog0 i = 8
Cog0 i = 9
```

在报告的第一行,你可以看到 Cog0 从 $00000 加载 Spin2 的设置代码。在第二行,Spin2 解释器从 $00D6C 启动,其堆栈空间从 $010BC 开始。之后,Spin2 程序运行,你会看到“i”从 0 迭代到 9。

如果将 REPEAT 中的“9”更改为“99”,数据将滚动得太快而无法读取,但通过在 DEBUG 语句末尾添加 DLY 命令,可以减慢输出速度：

```
    调试（udec（i）,dly（250））          调试,每次报告后延迟 250ms 输出 i
```

假设你想限制输出的消息,以便只显示“i”的奇数值。你可以在 DEBUG 语句的开头使用 IF 来检查 i 的最低有效位。当 IF 为假时,不会输出任何消息,只显示 i 的奇数值：

```
    调试（如果（i & 1）,udec（i）,dly（250））   调试,每次报告后仅输出奇数 i 值,延迟 250ms
```

Machine Translated by Google

PASM 中的简单 DEBUG 示例

```
CON _clkfreq = 10_000_000                    设置 10 MHz 时钟（假设 20 MHz 晶振）

数据        组织

            MOV i,#9    将 i 设置为 9
环形        DEBUG (UHEX_LONG(i))    调试,以十六进制输出 i
            DJNF i,#loop    若 i 不为 -1 则递减并循环
            跳转        #$    别乱跑,留在这里

我          可再生能源      1                    保留一个寄存器作为  i
```

使用 Ctrl-F10 运行时,将打开“调试”窗口并显示以下内容：

```
Cog0 INIT $0000_0000 $0000_0000 加载
Cog0 i = $0000_0009
Cog0 i = $0000_0008
Cog0 i = $0000_0007
Cog0 i = $0000_0006
Cog0 i = $0000_0005
Cog0 i = $0000_0004
Cog0 i = $0000_0003
Cog0 i = $0000_0002
Cog0 i = $0000_0001
Cog0 i = $0000_0000
```

在报告的第一行,您会看到 Cog0 从 $00000 加载我们的 PASM 程序。之后,程序运行,您会看到“i”从 9 向下迭代到 0。

如果您将 MOV 指令中的“9”更改为“99”,并且想要减慢速度,请在 DEBUG 语句中添加 DLY 命令,并确保以毫秒为单位作为#250,因为简单的 250 将被理解为寄存器 250：

```
调试（uhex_long（i）,dly（#250））        调试,以十六进制输出 i 并在每次报告后延迟 250 毫秒
```

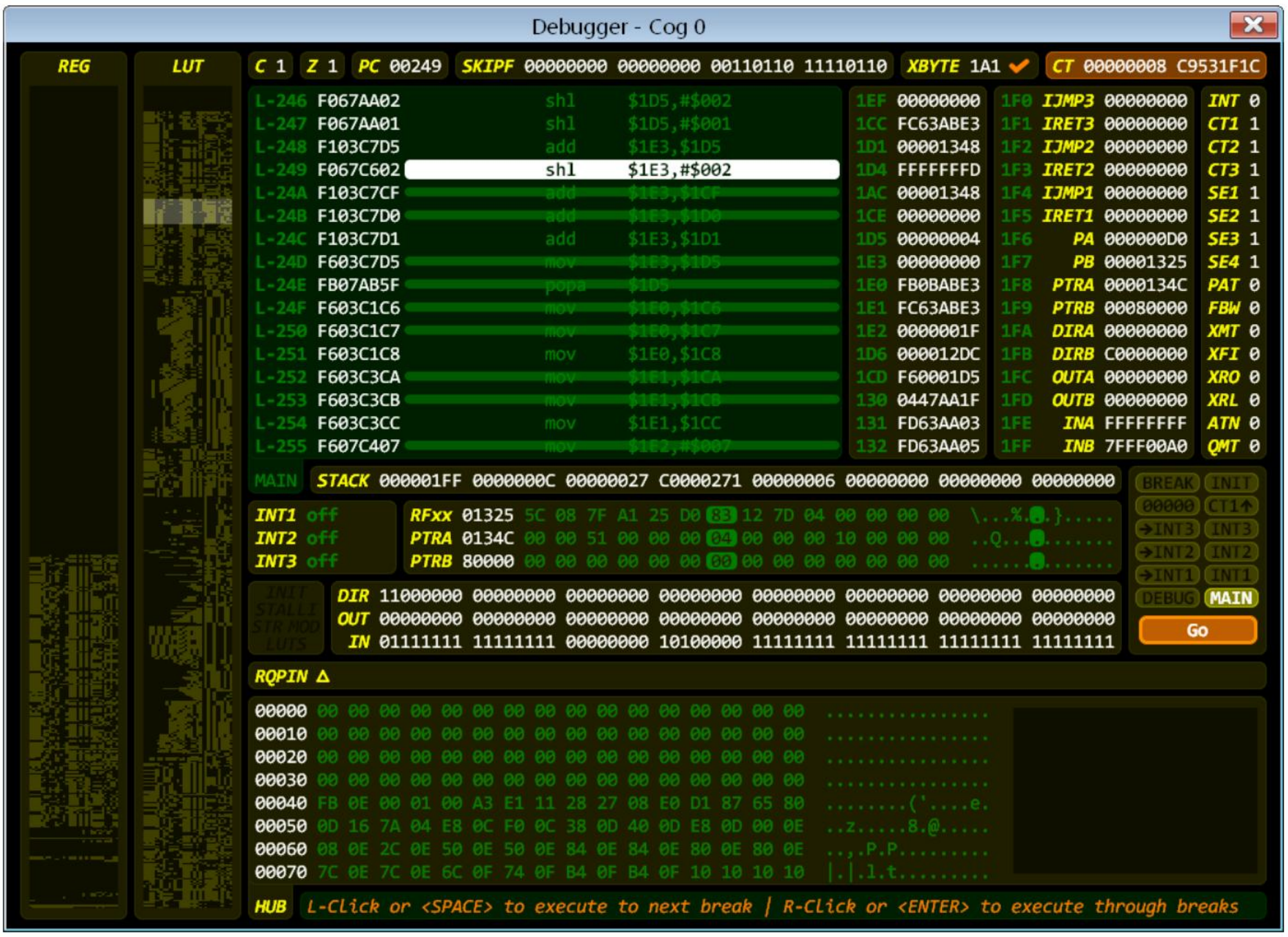
PASM 级调试器

```
CON _clkfreq = 200_000_000
        调试主程序                          运行所有主代码的调试器

PUB go() | i

    coginit（newcog,@pasm,0）               使用 pasm 程序启动另一个 cog

    重复 i++                                增加 i

数据        组织

帕斯姆      添加        $100,#1             增加一些寄存器
            添加        $101,#1
            添加        102美元,#1
            添加        103美元,#1
            跳转        #pasm               环形

            长的        0[11]               为了清晰起见,请清除代码后的空间
```

在上面的例子中,DEBUG_MAIN 符号会导致每个 cog 在通过 COGINIT 首次启动时打开一个调试器窗口。上面的示例将启动两个 cog 和调试器。Cog 0 将运行 Spin2 程序,该程序在 REPEAT 循环中递增变量“i”。Cog 1 将运行 PASM 程序,该程序将寄存器 $100 至 $103 反复加一。

进入调试器后,您必须确认所需的断点条件,然后点击“Go”按钮执行代码到下一个断点。当您移动鼠标时在调试器窗口内,底部会给出提示,提醒您可用的选项。调试器的设计非常直观。

请注意,“DEBUG”中断敏感度与“INIT”（COGINIT)敏感度无关。这是因为普通的 DEBUG 命令只能与 DEBUG() 区分。如果没有启用其他调试中断源,则命令。异步“BREAK”实际上始终处于启用状态,其视觉指示是所有其他中断源均未启用。除“INIT”外,其他所有敏感度均相同。由于 COGINIT 总是可以在调试中断中检测到,因此“INIT”敏感度与其他所有敏感度无关。要使用异步中断功能,您必须有另一个 cog 频繁更新其自己的调试器,以便它可以作为信使为 cog 生成异步中断感兴趣。



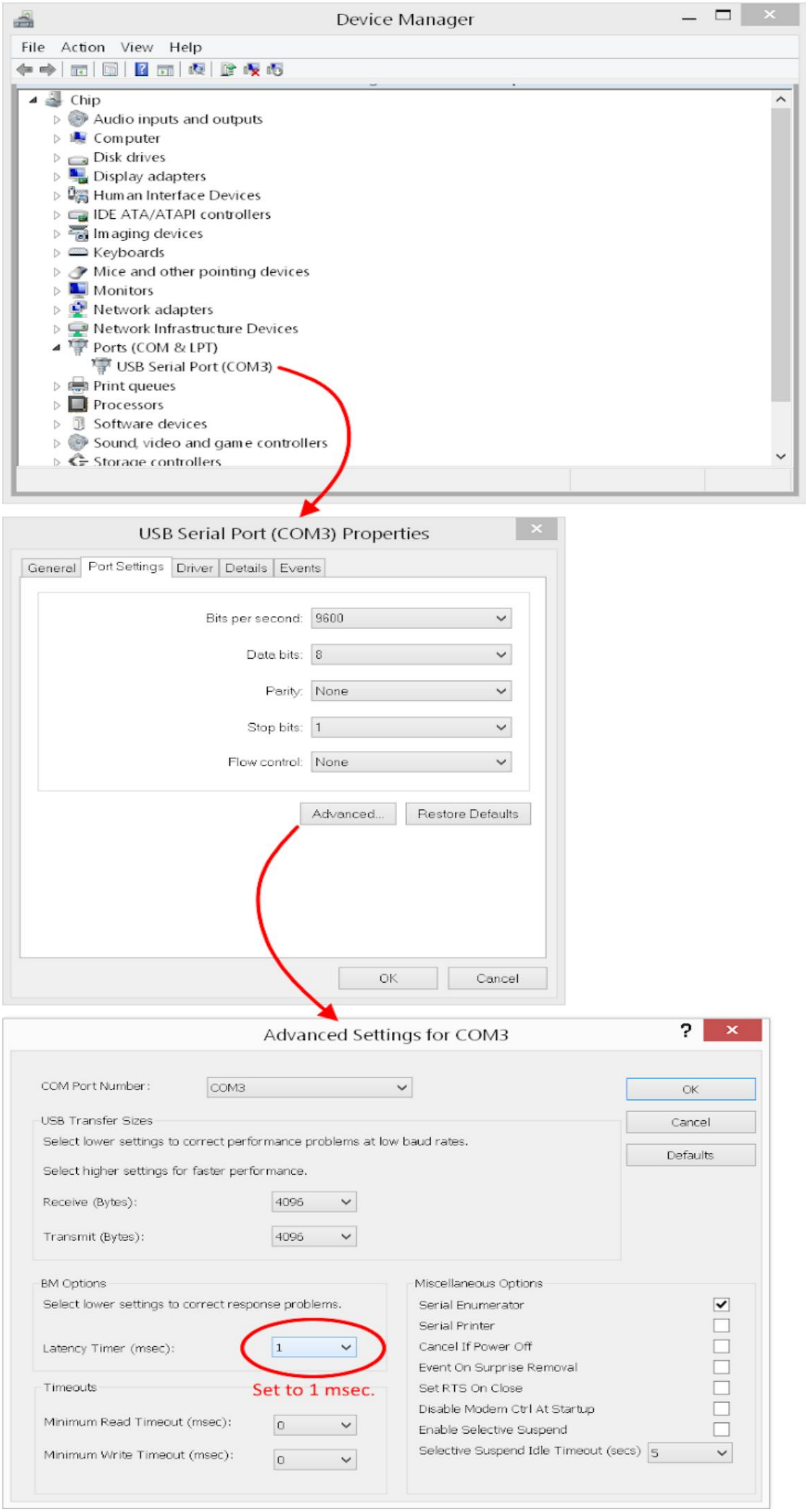
要启动调试器或强制更新已打开的调试器,您可以将一个简单的 DEBUG 命令插入到 Spin2 或 PASM 代码中,在您想要的位置更新发生。您可以在整个应用程序中放置任意数量的普通 DEBUG 命令,因为它们都解析为“BRK #0”指令,而 DEBUG() 命令解析为唯一的“BRK #1..255”指令。对于按下“Go”按钮后调试器随后注册的普通 DEBUG 命令,必须设置“DEBUG”灵敏度按钮。这将是默认灵敏度,除非定义了 DEBUG_COGINIT 或 DEBUG_MAIN 符号,从而设置初始

Machine Translated by Google

对‘INIT’或‘MAIN’的敏感度。

为了获得良好的调试器性能,需要进入 Windows 设备管理器并将 USB 串行端口的延迟计时器设置为 1 毫秒,而不是默认的 16 毫秒。以下是更改延迟计时器设置所需的窗口。同时请确保“USB 传输大小”均设置为 4096：

Machine Translated by Google



Machine Translated by Google

DEBUG动态时钟频率适配

当启用 DEBUG 时,串行接收引脚 (P63) 配置为长存储库,用于保存时钟频率值,以便调试器可以计算正确的波特率

调试中断期间的速率。每当时钟频率发生变化时,都必须更新此长存储库值,以确保调试器正常通信。

下面是一段代码片段,演示了如何执行此操作。

数据	组织	
时钟变化	代表 #99,#1	使用 REP 来停止所有中断（包括调试）
	andn old_mode,#%11 hubset old_mode	切换到 20 MHz,同时保持旧的 pll/xtal 设置
	移动 old_mode,new_mode andn old_mode,#%11 hubset old_mode	建立新的 pll/xtal 设置,同时保持在 20 MHz
	waitx ##20_000_000/100	等待 10ms 让新设置稳定下来
	移动 old_mode,new_mode hubset old_mode	切换到新设置
	dirh #63 wxpin new_freq,#63 _ret_ dirl #63	必须启用智能密码才能更新长存储库 将 new_freq 写入 rx 引脚长存储库 让智能引脚重新进入睡眠状态,REP 在 _ret_ 时取消
旧模式 新模式 新频率	水库 1 水库 1 水库 1	

DEBUG() 内存利用率

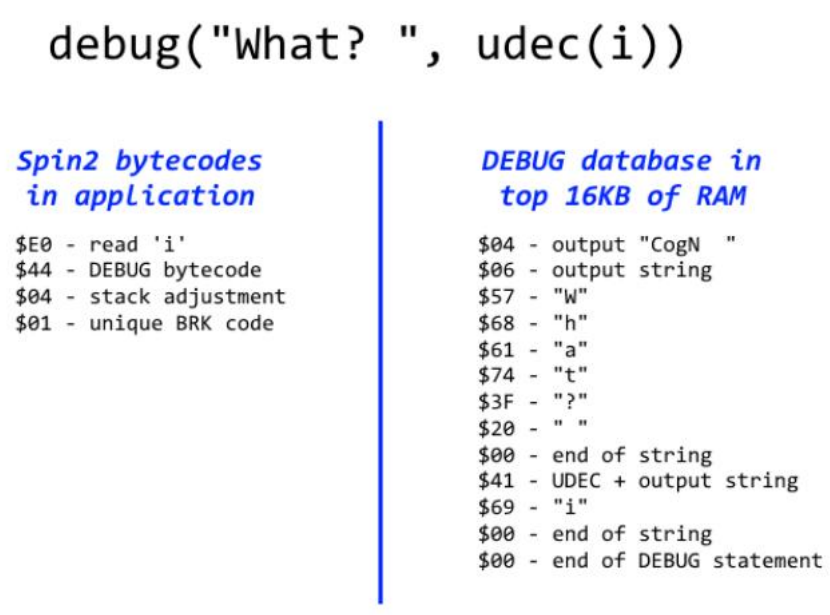
以下是 Spin2 DEBUG() 命令的内存利用率。可以看到,在 Spin2 端,需要执行一段字节码来读取变量“i”,然后执行三段

强制字节码构成实际的 DEBUG() 命令。

“堆栈调整”字节告诉解释器要将堆栈降低到什么程度,才能有效地“弹出”所有为 DEBUG() 事件做准备而压入的表达式。在这种情况下

仅在“i”的情况下,堆栈需要减少四个字节（一个长字节）。当调试程序被调用时,它所需的值将按当前 Spin2 的正上方排序。

堆栈指针。



“唯一 BRK 代码”字节（1..255)用作索引,在内存顶部的 DEBUG() 数据库中查找特定记录,调试程序从中读取

它的命令。

在调试处于活动状态,但 cog 已通过 DEBUG_COGS 符号禁用其调试中断的情况下,Spin2 DEBUG 命令将不会触发调试

中断,但它们仍然从堆栈中弹出任何 DEBUG 预期值,因此这些都是无害事件。

对于 PASM DEBUG 命令,会在 DEBUG 命令所在位置插入一条“BRK #code”指令,所有相关数据均存储在 DEBUG 数据库中。如果 cog 的

调试中断被禁用时,“BRK #code”指令不执行任何操作,需要两个时钟。

调试和中断

在 DEBUG 命令期间收到的中断请求将在 DEBUG 完成后执行,但响应时间可能会出现偏差,导致

中断将无法正常发生。高频周期性智能引脚中断尤其容易出现此问题。假设您在正常的 ISR 中执行了 AKPIN 指令

（中断服务程序)将 INA/INB 信号拉低,以便智能引脚能够使其再次升高,从而触发新的中断。同时,在 AKPIN 之后和

RETIx,智能引脚触发,将 INA/INB 拉高。这种情况只会因为您的周期帧时序因 DEBUG 命令而出现偏差。此中断

Machine Translated by Google

由于它发生在 ISR 繁忙时,因此不会被发现。这将导致中断停止循环。然而,CT 中断不容易出现这个问题,因为它们 $8000_0000 个时钟周期内需要进行识别。为了解决智能引脚重新触发问题,您可以在 INA/INB-high 时触发,而不是 INA/INB-rise 时,但这可能导致智能引脚配置出现性能问题。

解决此 DEBUG/中断困境的一个安全方法是,只在后台不执行 ISR 的 cog 上执行 DEBUG 命令。如果 ISR 可以容忍时间偏差,并且没有挂起中断循环的风险,您可以在了解一些中断时间退化的情况下执行 DEBUG 命令。

图形调试显示

DEBUG() 命令可以调用工具内置的特殊图形 DEBUG 显示。这些图形显示各自以独特的窗口形式呈现。一旦实例化后,显示器可以连续输入数据,以生成动画可视化效果。这些显示器对于开发和调试非常方便,因为各种数据类型可以在适当的上下文中查看。最多可同时运行 32 个图形显示。

当 DEBUG 消息包含反引号 (`) 字符 (ASCII $60) 时,会将包含从反引号到消息结尾的所有内容的字符串发送到图形 DEBUG 显示解析器。解析器会查找几种不同的元素类型,并将任何逗号视为空格:

元素类型	例子	描述
显示类型	逻辑、范围、情节、位图	这是您希望实例化的图形 DEBUG 显示类型的正式名称。
未知符号	MyLogicDisplay	每个图形 DEBUG 显示实例必须被赋予一个唯一的符号名称。
实例名称	MyLogicDisplay	一旦实例化,图形 DEBUG 显示实例将通过其符号名称进行引用。
关键词	标题、POS、尺寸、样品	关键字用于配置显示。关键字后面可能跟有数字、字符串或其他关键字。
数字	1024，$FF,%1010	数字可以用十进制、十六进制（$)和二进制（%)表示。
细绳	“这是一个字符串”	字符串在单引号内表示。

在深入探讨所有这些是如何结合在一起的之前,我们需要先了解一些可用于显示的特殊 DEBUG()-display 语法。当第一个 DEBUG() 命令中的字符是反引号。这会导致 DEBUG() 命令中的所有内容都被视为字符串,除非后续的反引号充当“转义”字符允许使用正常或简写 DEBUG() 命令。

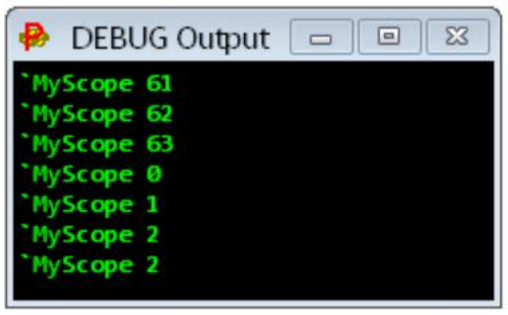
DEBUG语句 （v = 100,w = 1.0,字节[a] = 1,2,3,4,5）	DEBUG 消息输出	笔记
DEBUG（“`LOGIC MyLog SAMPLES”,SDEC_（v）)Cog0`LOGIC MyLog SAMPLES 100		常规 DEBUG() 语法可以驱动 DEBUG() 显示,但它很冗长。
调试（`LOGIC MyLog SAMPLES 100）	`LOGIC MyLog 样本 100	DEBUG()-显示语法更简单,并且在输出中省略了“CogN”。
DEBUG（`LOGIC MyLog SAMPLES`?（v））	`LOGIC MyLog 样本为真	布尔值使用 `?(值) 符号输出。BOOL_ 的缩写。
DEBUG（`LOGIC MyLog SAMPLES`。（w））	`LOGIC MyLog SAMPLES 1.000000e+00 浮点值使用 `.(值) 符号输出。FDEC_ 的缩写。	
调试（`LOGIC MyLog SAMPLES`（v））	`LOGIC MyLog 样本 100	十进制数使用 `(值) 表示法输出。SDEC_ 的缩写。
调试（`LOGIC MyLog SAMPLES`$（v））	`LOGIC MyLog 样本 64 美元	十六进制数使用 `$(value) 符号输出。UHEX_ 的缩写。
调试（`LOGIC MyLog SAMPLES`%（v））	`LOGIC MyLog 样本 %1100100	二进制数使用“%(value)”格式输出。UBIN_ 的缩写。
DEBUG（`LOGIC MyLog TITLE　`#(v)　）	`LOGIC MyLog TITLE　d	字符使用 `#(value) 符号输出。
DEBUG（`MyLog`UDEC_BYTE_ARRAY_（@a,5））	`MyLog 1、2、3、4、5	常规 DEBUG() 命令也可以跟在反引号后面。

使用图形 DEBUG() 显示有两个步骤。首先,必须实例化它们;其次,必须为它们提供数据:

使用显示器:	第一	第二	第三	第四	笔记
首先,实例化它。	`	显示类型	未知符号	关键字、数字、字符串 Unknown_symbol 变为 instance_name。	
然后,喂它。	`	实例名称	关键字、数字、字符串		多个显示器可以输入相同的数据。

为了将所有这些结合起来,让我们在 SCOPE 显示屏上显示锯齿波:

```
CON _clkfreq = 10_000_000

PUB go() | i

    调试（`SCOPE MyScope SIZE 254 84 SAMPLES 128）
    调试（`MyScope　锯齿　0 63 64 10%1111）

    重复
        调试（`MyScope`（i&63））
        我++
        waitms(50)
```



Machine Translated by Google

在上面的例子中,实例化了一个名为 MyScope 的 SCOPE,其尺寸为 254 x 84 像素,显示 128 个样本。由于样本编号为 0 到 127,因此宽度选择为 254。

我希望它们以恒定的两像素间距（127 * 2 = 254)间隔排列。高度选择为 84,这样波形上方和下方各有 10 个像素,

高度为 64 像素。定义了一个名为“锯齿”的通道,为了便于显示,其底部值为 0,顶部值为 63,高度为 64 像素。

范围,并且比示波器窗口底部高出 10 个像素。%1111 启用顶部和底部图例值以及顶部和底部线条。在 REPEAT 块中,

SCOPE 接收一个 0 到 63 的重复模式,形成锯齿波。SCOPE 每次收到一个值时都会更新其显示。如果定义了八个通道,

它不只是一个,而是在收到第八个值时更新显示,绘制所有八个通道。

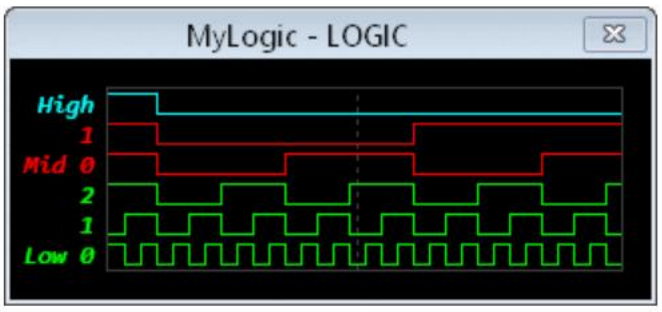
目前,已实现以下图形 DEBUG() 显示,但将来会添加更多内容:

显示类型	描述
逻辑	具有单位和多位标签的逻辑分析仪,1..32 个通道,可触发模式
范围	具有 1 至 8 个通道的示波器,可以触发滞后电平
范围_XY	XY 示波器,具有 1..8 个通道、0..512 个样本的持久性、极坐标模式、对数刻度模式
快速傅里叶变换	具有 1..8 个通道、4..2048 个点、窗口结果、对数刻度模式的快速傅里叶变换
斯派克	具有 4 至 2048 点 FFT、窗口结果、相位着色和对数刻度模式的光谱仪
阴谋	具有笛卡尔和极坐标模式的通用绘图仪
学期	文本终端最多可容纳 300 x 200 个字符、6..200 点字体大小、4 种同时配色方案
位图	位图,1..2048 x 1..2048 像素,1/2/4/8/16/32 位像素,具有 19 种颜色系统,15 种方向/自动滚动模式,独立 X 和 Y 像素大小为 1..256
MIDI	带有 1 至 128 个键的钢琴键盘,速度描述,可变屏幕比例

以下是每个 DEBUG() 显示类型的详细说明。

逻辑显示　　具有单位和多位标签的逻辑分析仪,1..32 个通道,可触发模式

```
CON _clkfreq = 10_000_000

PUB go() | i

    调试（`LOGIC MyLogic SAMPLES 32　低　3　中　2　高　）
    调试（`MyLogic TRIGGER $07 $04 HOLDOFF 2）

    重复
        调试（`MyLogic`（i&63））
        我++
        waitms(25)
```



逻辑实例	描述	默认
TITLE ‘字符串’	将窗口标题设置为“字符串”。	<无>
POS左上角	设置窗口位置。	0,0
样本 4_to_2048	设置要跟踪和显示的样本数量。	≡+≡
间距 2_to_32	设置样本间距。显示宽度为“样本数”*“间距”。	8
比率 1_to_2048	设置每次显示更新之前的样本数（或触发器数,如果启用）。	1
线尺寸 1_到_7	设置线条大小。	1
文本大小 6_到_200	设置图例文本大小。文本高度决定逻辑层级的高度。	编辑器文本大小
颜色背景颜色{网格颜色}	设置背景和网格颜色*。	黑色,灰色 4
名称　{1_to_32 {颜色}}	设置下一个通道或通道组名称、可选组位数、可选颜色*。如果 如果未指定名称,则将建立一个包含 32 个通道的组。	1、默认颜色
名称　2_to_32 范围 {颜色}	设置下一个通道组名称,以波形形式绘制,并带有可选颜色*。	默认颜色
打包数据模式	启用打包数据模式。请参阅本节末尾的说明。	<无>
希德西	隐藏鼠标指针上显示的 X、Y 鼠标坐标。	没有隐藏
逻辑喂养	描述	默认
触发掩码匹配sample_offset	触发开启 (数据 & 掩码) = 匹配。如果掩码 = 0,则禁用触发。	0、1、样本/2
延迟 2_to_2048	设置每次触发所需的最小样本数。	样品
数据	数值数据以 LSB 优先的方式应用于通道。	
清除	清除样本缓冲区和显示屏,等待新数据。	
保存 {WINDOW}‘文件名’	保存整个窗口或仅显示区域的位图文件 (.bmp)。	

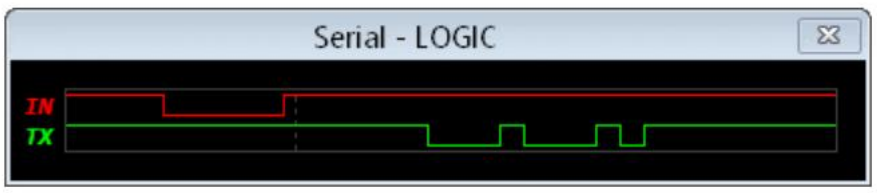
Machine Translated by Google

关闭	关闭窗口。	

* 颜色为 rgb24 值,否则为黑色/白色或橙色/蓝色/绿色/青色/红色/洋红色/黄色/灰色,后接可选的 0..15 亮度（默认为 8)。

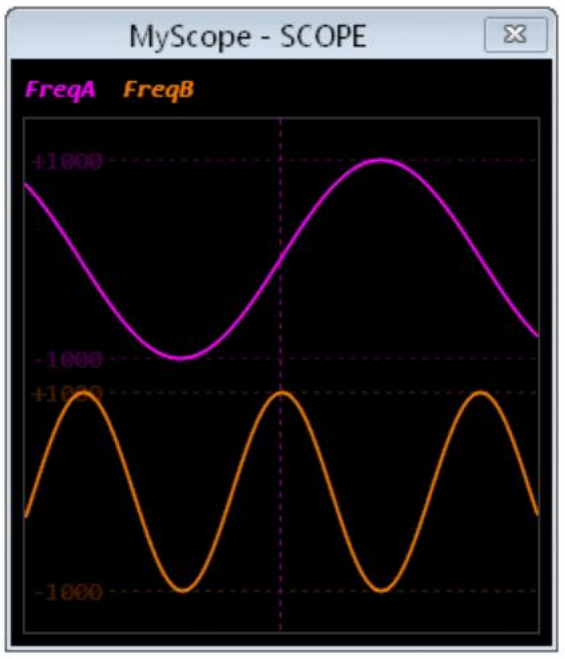
LOGIC 显示器可用于显示高速捕获的数据。在下面的示例中,P2 使用智能引脚以 333 Mbaud 生成 8-N-1 串行数据。这比特流可以被流处理器捕获。在每个时钟周期,流处理器都会记录智能引脚的输入信号及其输出状态,并从相邻引脚读取。每次获得四个两位样本集后,它会执行 RFBYTE 操作,将它们保存到集线器 RAM,形成连续的字节、字和长整型。通过调用 LONGS_2BIT 打包数据模式,我们可以让 LOGIC 显示器从 long 中解压出两位样本集,每个 long 产生 16 个集合。

```
CON _clkfreq = 333_333_333    速度非常快,3ns 时钟周期
        rxpin          = 24                          偶数针
        txpin          = rxpin+1                     奇怪的别针
        samps          = 32                          16 个样本的倍数
        bufflongs = samps / 16    每个 long 保存 16 个 2 位样本
        x模式          = $D0800000 + rxpin << 17 + samps    流光模式

VAR buff[bufflongs]

PUB go() | i,buffaddr

    调试（`逻辑串行样本`（samps)间距12   TX      IN   longs_2bit）
    调试（`串行触发器%10%10 22）
    buffaddr := @buff

    重复
        组织
        沃尔平          ##+1<<28,#rxpin                    rxpin 在 rxpin+1 处输入 txpin

        wrpin           #%01_11110_0,#txpin    设置 txpin 异步 tx 模式
        wxpin           ##1<<16+8-1,#txpin    设置波特率=sysclock/1 和 size=8
        dirh            #txpin   启用智能密码

        沃法斯特        #0,buffaddr                        将写入速度设置为 buff
        xinit           ##xmode,#0                         开始捕获 2 位样本

        怀平            我,#txpin                          发送串行字节

        waitxfi结束                                        等待流光捕获完成


        调试（`串行`uhex_long_array_（@buff,bufflongs））
        我++
        waitms(20)
```



示波器显示示波器具有 1..8 个通道,可以触发滞后电平

```
CON _clkfreq = 100_000_000

PUB go() | a,af,b,bf

    调试（`SCOPE MyScope）
    调试（`MyScope   FreqA   -1000 1000 100 136 15 MAGENTA）
    调试（`MyScope   FreqB   -1000 1000 100 20 15 ORANGE）
    调试（`MyScope TRIGGER 0 HOLDOFF 2）

    重复
        a := qsin(1000, af++, 200)
        b := qsin(1000, bf++, 99)
        调试（`MyScope`（a,b））
        韦特斯(200)
```



范围实例化	描述	默认
TITLE ‘字符串’	将窗口标题设置为“字符串”。	<无>
POS左上角	设置窗口位置。	0,0
尺寸 宽度 高度	设置显示尺寸（32..2048 x 32..2048）	255、256
样本 16_to_2048	设置要跟踪和显示的样本数量。	256
比率 1_to_2048	设置每次显示更新之前的样本数（或触发器数,如果启用）。	1
点大小 0_到_32	设置点的大小（以像素为单位)以显示精确的样本点。	0
线尺寸 0_到_32	设置连接采样点的线条大小（以半像素为单位）。	3
文本大小 6_到_200	设置图例文字大小。	编辑器文本大小
颜色背景颜色{网格颜色}	设置背景和网格颜色*。	黑色的,

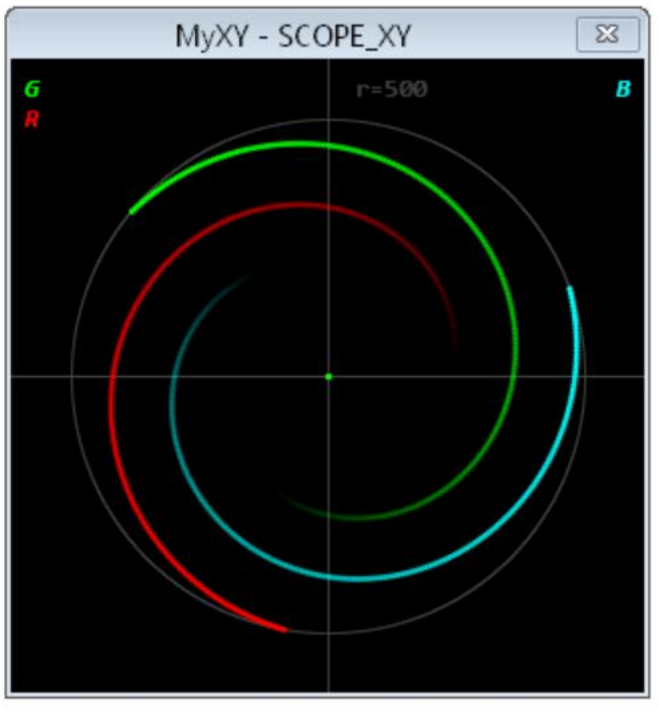
Machine Translated by Google

		黑光4
打包数据模式	启用打包数据模式。请参阅本节末尾的说明。	<无>
希德西	隐藏鼠标指针上显示的 X、Y 鼠标坐标。	没有隐藏
范围喂养	描述	默认
name {min {max {y_size {y_base {legend} {color}}}}设置第一个/下一个通道名称、最小值、最大值、y 尺寸、y 基准、图例和颜色*。	图例为 %abcd,其中 %a 到 %d 分别表示最大图例、最小图例、最大行、最小行。	完整,无图例, 默认颜色
名称 AUTO {y_size {y_base {图例} {颜色}}}	设置第一个/下一个通道名称、自动缩放、y 轴大小、y 轴基座、图例和颜色*。图例为 %abcd,其中 %a 到 %d 启用最大图例、最小图例、最大线、最小线。	自动,无图例, 默认颜色
触发通道 {arm_level {trigger_level {offset}}}	设置触发通道、启动电平、触发电平和右偏移。如果 channel=-1,则禁用。	-1，-1,0,宽度/2
触发通道自动{偏移}	设置触发通道、33% 启动电平、50% 触发电平和右偏移。如果 channel=-1, 已禁用。	-1,宽度/2
延迟 2_to_2048	设置每次触发所需的最小样本数。	样品
数据	数值数据按升序应用于通道。	
清除	清除样本缓冲区和显示屏,等待新数据。	
保存 {WINDOW}‘文件名’	保存整个窗口或仅显示区域的位图文件 (.bmp)。	
关闭	关闭窗口。	

* 颜色为 rgb24 值,否则为黑色/白色或橙色/蓝色/绿色/青色/红色/洋红色/黄色/灰色,后接可选的 0..15 亮度（默认为 8)。

SCOPE_XY显示　　XY 示波器,具有 1..8 个通道、1..512 个样本的持久性、极坐标模式、对数刻度模式

```
CON _clkfreq = 100_000_000

PUB go() | i

    调试（`SCOPE_XY MyXY RANGE 500 POLAR 360    G      R      B   ）

    重复
        重复 i 从 0 到 500
            调试（`MyXY`（i,i,i,i + 120,i,i + 240））
            waitms(5)
```



SCOPE_XY 实例	描述	默认
TITLE‘字符串’	将窗口标题设置为“字符串”。	<无>
POS左上角	设置窗口位置。	0,0
尺寸半径	以像素为单位设置显示半径。	128
范围 1 到 7FFFFFFF	设置输入数据的单位圆半径	$7FFFFFFF
样本 0_to_512	设置要跟踪并显示余辉的样本数量。0 表示无限余辉。	256
比率 1_to_512	设置每次显示更新之前的样本数。	1
点尺寸 2_到_20	以半像素为单位设置点的大小来显示样本点。	6
文本大小 6_到_200	设置图例文字大小。	编辑器文本大小
颜色背景颜色{网格颜色}	设置背景和网格颜色*。	黑色,灰色 4
极地 {twopi {offset}}	设置极坐标模式、二圆周率值和偏移量。如果二圆周率值为 $100000000 或 -$100000000,则使用 0 或 -1。	1亿美元,0
对数尺度	设置对数尺度模式来放大单位圆内的点。	<关闭>
‘名称’{颜色}	设置第一个/下一个通道名称并可选择为其分配颜色*。	默认颜色
打包数据模式	启用打包数据模式。请参阅本节末尾的说明。	<无>
希德西	隐藏鼠标指针上显示的 X、Y 鼠标坐标。	没有隐藏
SCOPE_XY 喂食	描述	默认

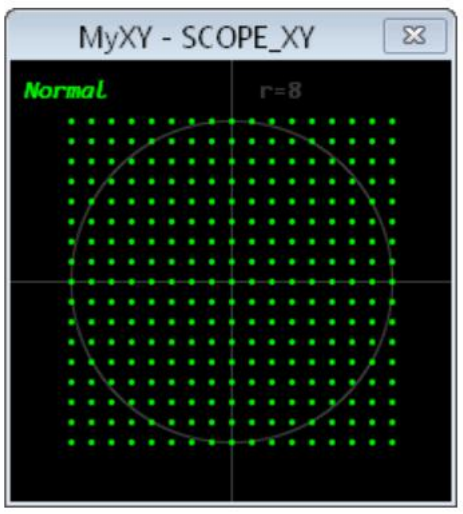
Machine Translated by Google

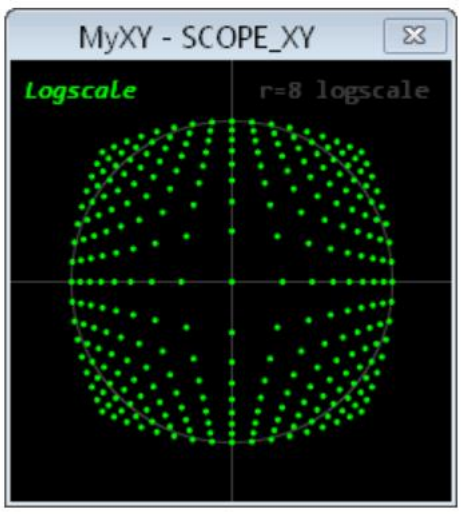
坐标	XY 数据对按升序应用于通道。在极坐标模式下,x=长度,y=角度。	
清除	清除样本缓冲区和显示屏,等待新数据。	
保存 {WINDOW}‘文件名’	保存整个窗口或仅显示区域的位图文件 (.bmp)。	
关闭	关闭窗口。	

* 颜色为 rgb24 值,否则为黑色/白色或橙色/蓝色/绿色/青色/红色/洋红色/黄色/灰色,后接可选的 0..15 亮度（默认为 8)。

```
CON _clkfreq = 10_000_000                    正常模式

PUB go() | x, y
    调试（`SCOPE_XY MyXY SIZE 80 RANGE 8 SAMPLES 0    正常    ）
    重复 x 从 -8 到 8
        重复 y 从 -8 到 8
            调试（`MyXY`（x,y））
```

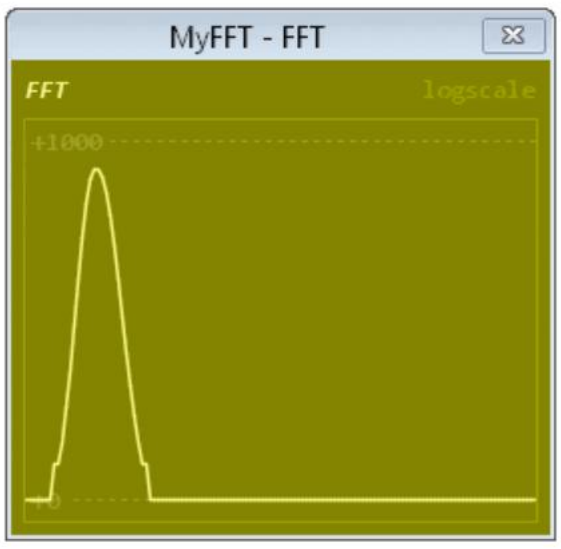
```
CON _clkfreq = 10_000_000                    LOGSCALE 模式放大低级细节

PUB go() | x, y
    调试（`SCOPE_XY MyXY SIZE 80 RANGE 8 SAMPLES 0 LOGSCALE    对数尺度    ）
    重复 x 从 -8 到 8
        重复 y 从 -8 到 8
            调试（`MyXY`（x,y））
```

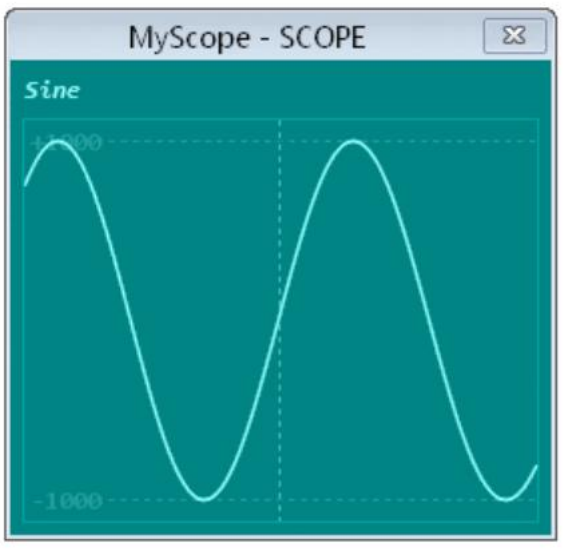




FFT显示　　具有 1..8 个通道、4..2048 个点、窗口结果、对数刻度模式的快速傅里叶变换

```
CON _clkfreq = 100_000_000

PUB go() | i,j,k

        设置 FFT
    调试（`FFT MyFFT 大小 250 200 样本 2048 0 127 速率 256 对数尺度颜色黄色 4 黄色 5）
    调试（`MyFFT    FFT    0 1000 180 10 15黄色12）

        设置范围
    调试（`scope MyScope POS 300 0 SIZE 255 200 COLOR CYAN 4 CYAN 5）
    调试（`MyScope    正弦    -1000 1000 180 10 15 CYAN 12）
    调试（`MyScope TRIGGER 0）

    重复
        j += 1550 + qsin(1300, i++, 31_000)
        k := qsin(1000, j, 50_000)
        调试（`MyFFT MyScope`（k））
        韦特斯(100)
```





FFT实例	描述	默认
TITLE‘字符串’	将窗口标题设置为“字符串”。	<无>
POS左上角	设置窗口位置。	0,0
尺寸 宽度 高度	设置显示尺寸（32..2048 x 32..2048）	256,256
SAMPLES 4_to_2048 {first {last}}设置 2　个 FFT 输入点,加上要显示的第一个和最后一个结果值。		512, 0, 255

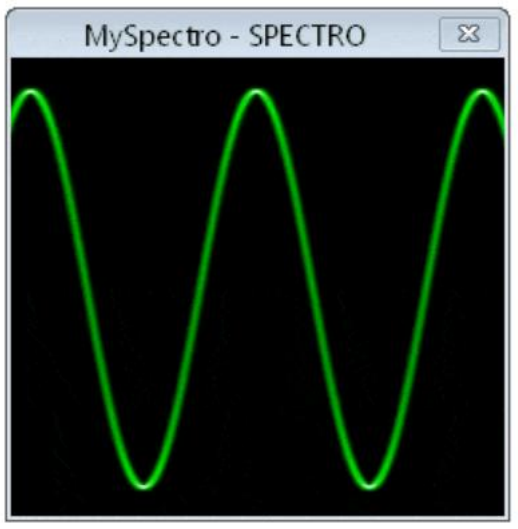
Machine Translated by Google

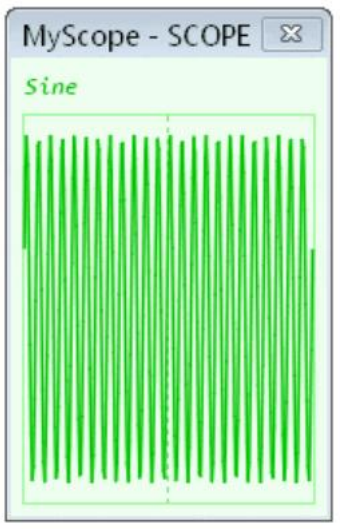
比率 1_to_2048	设置每次显示更新之前的样本数。	样品
点大小 0_到_32	设置点的大小（以像素为单位)以显示精确的样本点。	0
LINESIZE neg32_to_32	设置用于连接采样点的线条大小（以半像素为单位）。负值的线条大小将产生孤立的垂直线条。	3
文本大小 6_到_200	设置图例文字大小。	编辑器文本大小
颜色背景颜色{网格颜色}	设置背景和网格颜色*。	黑色,灰色 4
对数尺度	设置对数尺度模式来放大低级结果。	<关闭>
打包数据模式	启用打包数据模式。请参阅本节末尾的说明。	<无>
希德西	隐藏鼠标指针上显示的 X、Y 鼠标坐标。	没有隐藏
FFT进料	描述	默认
名称 {mag {max {y_size {y_base {图例 {颜色}}}}}	设置第一个/下一个通道名称、放大倍数（2 ,n = 0..11）、最大振幅、y 尺寸、y 基数、图例和颜色*。图例为 %abcd,其中 %a 到 %d 启用最大图例、最小图例、最大线、最小线。	完整,无图例,默认颜色
数据	数值数据被输入到通道的滑动汉宁窗口中,FFT 从中计算功率级别。	
清除	清除样本缓冲区和显示屏,等待新数据。	
保存 {WINDOW}‘文件名’	保存整个窗口或仅显示区域的位图文件 (.bmp)。	
关闭	关闭窗口。	

* 颜色为 rgb24 值,否则为黑色/白色或橙色/蓝色/绿色/青色/红色/洋红色/黄色/灰色,后接可选的 0..15 亮度（默认为 8)。

光谱显示　　具有 4 至 2048 点 FFT、相位着色和对数刻度模式的光谱仪

```
CON _clkfreq = 100_000_000

PUB go() | i,j,k

    设置 SPECTRO
  调试（`SPECTRO MySpectro SAMPLES 2048 0 236 RANGE 1000 LUMA8X GREEN）

    设置范围
  调试（`SCOPE MyScope POS 280 尺寸 150 200 颜色绿色 15 绿色 12）
  调试（`MyScope 正弦 -1000 1000 180 10 0 GREEN 6）
  调试（`MyScope TRIGGER 0）

  重复
    j += 2850 + qsin(2500, i++, 30_000)
    k := qsin(1000, j, 50_000)
    调试（`MySpectro MyScope`（k））
    韦特斯(100)
```



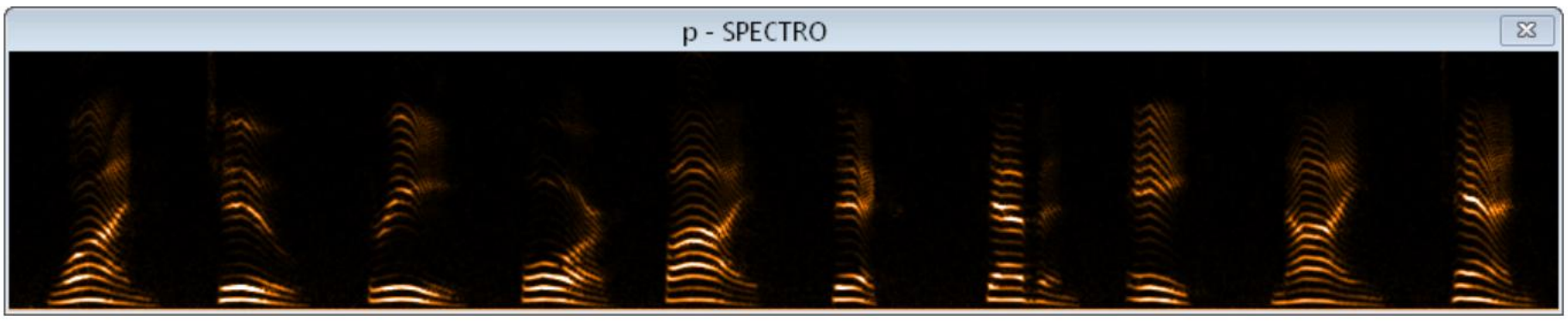


SPECTRO 实例	描述	默认
TITLE‘字符串’	将窗口标题设置为“字符串”。	<无>
POS左上角	设置窗口位置。	0,0
样本 4_to_2048 {first {last}}	设置 2 个 FFT 输入点,加上要显示的第一个和最后一个结果值（定义显示高度）。	512, 0, 255
深度 1_到_2048	设置要显示的垂直线 FFT 结果的数量（定义显示宽度）。	256
MAG 0_to_11	设置放大倍数（2 ,n = 0..11）。	0
范围饱和度功率	设置像素亮度饱和的功率级别。	$7FFFFFFF
比率 1_to_2048	设置每次显示更新之前的样本数。	样品 / 8
跟踪 0_到_15	设置跟踪模式（参见位图显示中的跟踪动画）。	15（右、上、滚动）
DOTSIZE width_and_height {height}将光谱仪像素宽度和像素高度（1..16)一起设置,或者单独设置它们。		1,1
luma_or_hsv {颜色或相位}	将配色方案设置为带有颜色*的 LUMA8(W)(X),或带有 0..255 相位着色偏移的 HSV16(W)(X)。	LUMA8X 橙色
对数尺度	设置对数尺度模式来放大低级结果。	<关闭>
打包数据模式	启用打包数据模式。请参阅本节末尾的说明。	<无>
希德西	隐藏鼠标指针上显示的 X、Y 鼠标坐标。	没有隐藏
SPECTRO 喂料	描述	默认
数据	数值数据被输入到滑动汉宁窗中,FFT 从中计算功率和相位。	
清除	清除样本缓冲区和显示屏,等待新数据。	
保存 {WINDOW}‘文件名’	保存整个窗口或仅显示区域的位图文件 (.bmp)。	
关闭	关闭窗口。	

Machine Translated by Google

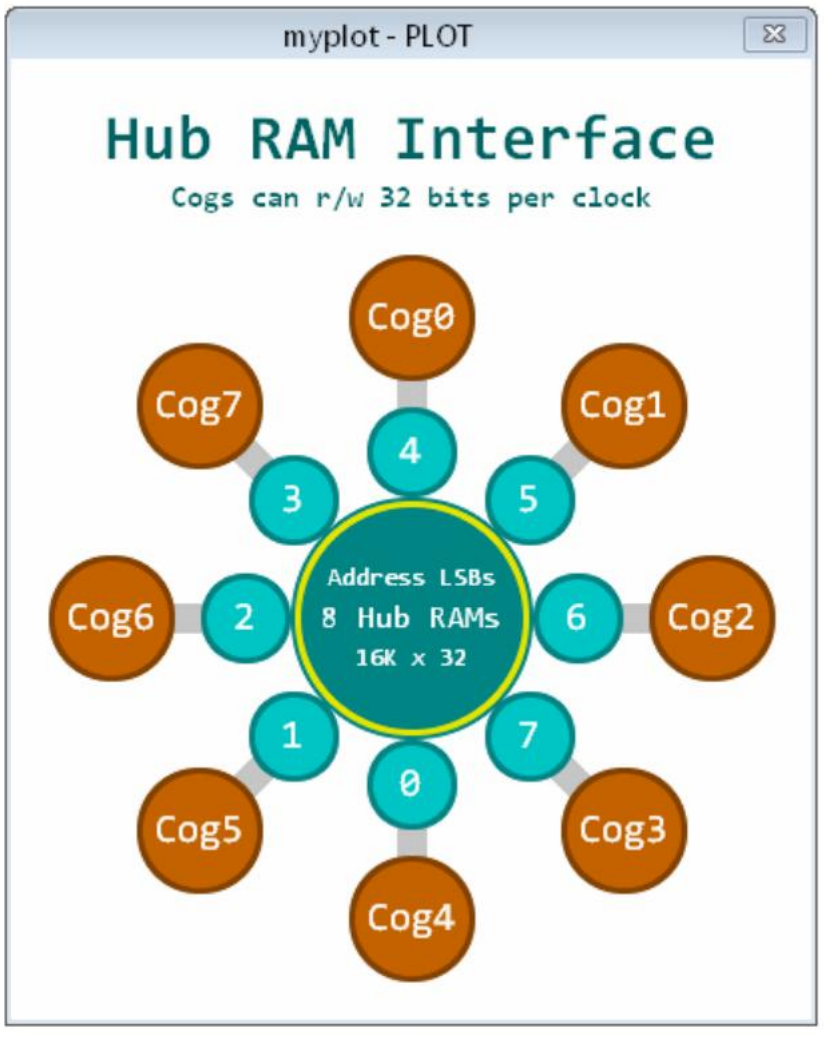
* 颜色为橙色 / 蓝色 / 绿色 / 青色 / 红色 / 洋红色 / 黄色 / 灰色。

下图中,一个连接到麦克风的针脚将 ADC 样本输入到 SPECTRO 显示器。这就是从“1”到“10”的口头计数在光谱上的样子。“1”在左边是“10”,右边是“10”。水平趋势线之间的垂直距离表示声门音高。较大的亮度趋势表示声音共振峰。这多少能说明我们的耳朵感知言语:



绘图显示 具有笛卡尔和极坐标模式的通用绘图仪

```
CON _clkfreq = 10_000_000

PUB go() :i,j,k

    调试（`绘制 myplot 大小 400 480 背景颜色白色更新）
    调试（`myplot 原点 200 200 极坐标 -64 -16）
    嗯~
    重复
        调试（`myplot 清除）
        调试（`myplot 设置 240 0 青色 3 文本 24 3    Hub RAM 接口    ）
        调试（`myplot 设置 210 0 文本 11 3    齿轮每时钟可以读取/写入 32 位    ）

        如果 k & 8 j++            “移动 RAM 还是拉动辐条？”

        别的
            重复 i 从 0 到 7
                调试（`myplot 灰色 12 设置 83`（i * 8)第 150 行`（i * 8)15）

        调试（`myplot 设置 0 0 青色 4 圆圈 121 黄色 7 圆圈 117 3）
        调试（`myplot 设置 20 0 白色文本 9    地址 LSB    ）
        调试（`myplot 设置 0 0 文本 11 1    8 个 Hub RAM    ）
        调试（`myplot 设置 20 32 文本 9    16K x 32    ）

        重复 i 从 0 到 7                                画出公羊和齿轮
            调试（`myplot 青色 6 设置 83`（i * 8-j)圆圈 43 文本 14    `（i)    ）
            调试（`myplot 青色 4 设置 83`（i*8-j)圆圈 45 3）
            调试（`myplot 橙色 6 设置 150`（i*8)圆圈 61 文本 13    Cog`（i)    ）
            调试（`myplot 橙色 4 设置 150`（i*8)圆圈 63 3）

        调试（`myplot 更新`dly（30））
        千
```



PLOT实例	描述	默认
TITLE ‘字符串’	将窗口标题设置为“字符串”。	<无>
POS左上角	设置窗口位置。	0,0
尺寸 宽度 高度	设置显示宽度（32..2048)和高度（32..2048）。	256,256
DOTSIZE 宽度和高度 {高度}	一起设置显示像素宽度和像素高度（1..256）,或者单独设置它们。	1,1
lut1_to_rgb24	设置颜色模式。	RGB24
LUTCOLORS rgb24 rgb24 ...	对于 LUT1..LUT8 颜色模式,使用 rgb24 颜色加载 LUT。使用 HEX_LONG_ARRAY_ 加载颜色。	默认颜色 0..7
背景颜色	根据当前颜色模式设置背景颜色。 *	黑色的
更新	设置更新模式。仅当输入“更新”命令时,显示才会更新。	自动更新
希德西	隐藏鼠标指针上显示的 X、Y 鼠标坐标。	没有隐藏
小区饲养	描述	默认
lut1_to_rgb24	设置颜色模式。	RGB24
LUTCOLORS rgb24 rgb24 ...	对于 LUT1..LUT8 颜色模式,使用 rgb24 颜色加载 LUT。使用 HEX_LONG_ARRAY_ 加载值。	默认颜色 0..7
背景颜色	根据当前颜色模式设置背景颜色。 *	黑色的
颜色 颜色	根据当前颜色模式设置绘图颜色。在 TEXT 之前使用可更改文本颜色。 *	青色

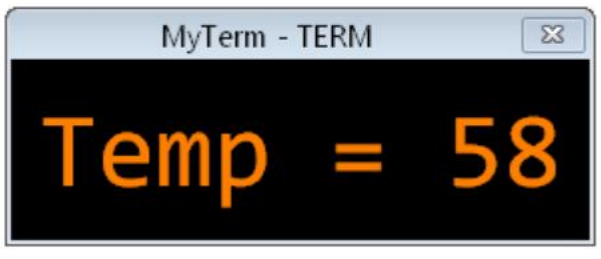
Machine Translated by Google

黑色/白色或橙色/蓝色/绿色/青色/ 红色/洋红色/黄色/灰色{亮度}	设置绘图颜色和可选的 0..15 亮度（橙色..灰色）（默认为 8）。	青色
不透明度	设置点、线、圆、椭圆、盒子和 OBOX 绘图的不透明度级别。0..255 = 清晰..不透明。	255
精确的	切换精确模式,其中 DOT 和 LINE 的线条大小和 (x,y) 以 256 个像素表示。	已禁用
LINESIZE 尺寸	设置 DOT 和 LINE 绘图的线条大小（以像素为单位）。	1
原点 {x_pos y_pos}	将原点设置为笛卡尔 (x_pos, y_pos),如果未指定值,则设置为当前 (x, y)。	0,0
设置 xy	将绘图位置设置为 (x, y)。LINE 之后,端点将成为新的绘图位置。	
点 {线条大小 {不透明度}}	使用可选的 LINESIZE 和 OPACITY 覆盖在当前位置绘制一个点。	
线 xy {线宽 {不透明度}}	从当前位置到 (x,y) 绘制一条线,并可选用 LINESIZE 和 OPACITY 覆盖。	
圆圈直径 {线条大小 {不透明度}}	围绕当前位置绘制一个圆圈,线条大小可选（无/0 = 实线)并且不透明度覆盖。	
OVAL 宽度高度 {linesize {opacity}}在当前位置周围绘制一个椭圆,可选线条大小（无/0 = 实线)和 OPACITY 覆盖。		
BOX 宽度 高度 {线条大小 {不透明度}}	在当前位置周围绘制一个框,具有可选的线条大小（无/0 = 实线)和不透明度覆盖。	
OBOX 宽度 高度 x_radius y_radius {线条大小 {不透明度}}	在当前位置周围绘制一个圆角框,具有宽度、高度、x 和 y 半径以及可选的线条大小 （无/0 = 实心)和 OPACITY 覆盖。	
TEXTSIZE 尺寸	设置文本大小（6..200）。	10
TEXTSTYLE 样式_YYXXUIWW	将文本样式设置为%YYXXUIWW： %YY 是垂直对齐:%00 = 中间,%10 = 底部,%11 = 顶部。 %XX 是水平对齐:%00 = 中间,%10 = 右,%11 = 左。 %U 是下划线:%1 = 下划线。 %I 为斜体:%1 = 斜体。 %WW 表示粗细:%00 = 细,%01 = 正常,%10 = 粗体,%11 = 重。	%00000001
TEXTANGLE 角度	设置文本角度。在笛卡尔模式下,角度以度为单位。	0
文本 {大小 {样式 {角度}}}　文本	绘制文本,并覆盖其大小、样式和角度。要更改文本颜色,请在“TEXT”前声明颜色。	
LAYER 图层 ‘filename.bmp’	将位图图像文件加载到图层（1..8)中,以便稍后通过 CROP 复制到绘图中。	
CROP 层 {left_layer top_layer 宽度 高度 {left_plot top_plot}}	将图层图像复制到绘图中。如果没有给出坐标,则整个图层图像将被复制到 图的左上角（用于背景）。如果指定了前四个坐标,则该区域 图层图像将被复制到绘图的同一区域（对于静态覆盖很有用）。如果最后两个 坐标也指定了,它们将改变图层图像区域在图中被复制到的位置（有用 用于动态叠加）。此命令的坐标始终为（从左到右,从上到下）。	
裁剪图层AUTO左图上图	将整个图层图像复制到图中指定坐标处（从左到右,从上到下）。	
SPRITEDEF id x_dim y_dim 像素… 颜色…	定义一个精灵。唯一 ID 必须是 0 到 255。每个精灵的尺寸必须是 1 到 32。像素是字节,用于选择 调色板颜色,按从左到右、从上到下的顺序排列。颜色是定义调色板的长整数 由像素字节引用;$AARRGGBB 值指定 alpha-blend、红色、绿色和蓝色。	
SPRITE id {方向 {比例 {不透明度}}}	在当前位置渲染精灵,并设置方向、比例和不透明度覆盖。方向范围为 0 到 7。 前八种 TRACE 模式。比例尺为 1 至 64。请参阅 DEBUG_PLOT_Sprites.spin2 文件。	<id>, 0, 1, 255
极地 {twopi {offset}}	设置极坐标模式、二元极化值和偏移量。例如,POLAR -12 -3 表示钟面。 对于 twopi 值为 $100000000 或 -$100000000 的情况,请使用 0 或 -1。 在极坐标模式下,（x,y)坐标被解释为（长度,角度）。	1亿美元,0
笛卡尔 {ydir {xdir}}	设置笛卡尔模式,并可选地设置 Y 轴和 X 轴极性。默认为笛卡尔模式。 如果 ydir 为 0,则 Y 轴向上。如果 ydir 非 0,则 Y 轴向下。 如果 xdir 为 0,则 X 轴指向右。如果 xdir 非 0,则 X 轴指向左。	0,0
清除	将绘图清除为背景颜色。	
更新	使用当前绘图更新窗口。用于 UPDATE 模式。	
保存 {WINDOW}‘文件名’	保存整个窗口或仅显示区域的位图文件 (.bmp)。	
关闭	关闭窗口。	

* 颜色是模态值,否则为黑色/白色或橙色/蓝色/绿色/青色/红色/洋红色/黄色/灰色,后接可选的 0..15 亮度
（默认值为 8）。

术语显示　　显示文本的终端

```
CON _clkfreq = 10_000_000

PUB go() | i

    调试（`TERM MyTerm SIZE 9 1 TEXTSIZE 40）
    重复
        重复 i 从 50 到 60
            调试（`MyTerm 1    Temp =`（i）    ）
            waitms(500)
```



TERM 实例	描述	默认
TITLE ‘字符串’	将窗口标题设置为“字符串”。	<无>

Machine Translated by Google

POS左上角	设置窗口位置。	0,0
大小 列 行	设置终端列数（1..256)和终端行数（1..256）。	40,20
TEXTSIZE 尺寸	设置终端文本大小（6..200）。	编辑器文本大小
颜色 文本颜色 背景颜色 ...	设置文本颜色和背景颜色组合 #0..#3。*	默认颜色
背景颜色	* 设置显示背景颜色。	黑色的
更新	设置更新模式。仅当输入“更新”命令时,显示才会更新。	自动更新
希德西	隐藏鼠标指针上显示的 X、Y 鼠标坐标。	没有隐藏
足月喂养	描述	默认
特点	0 = 清除终端显示和主光标。 1 = 主光标。 2 = 将列设置为下一个字符值。 3 = 将行设置为下一个字符值。 4 = 选择颜色组合#0。 5 = 选择颜色组合#1。 6 = 选择颜色组合#2。 7 = 选择颜色组合#3。 8 = 退格键。 9 = 跳至下一个第 8 列。 13+10 或 13 或 10 = 新行。 32..255 = 可打印字符。	
细绳	打印字符串。	
清除	清除显示内容,恢复背景颜色。	
更新	使用当前文本屏幕更新窗口。用于 UPDATE 模式。	
保存 {WINDOW}‘文件名’	保存整个窗口或仅显示区域的位图文件 (.bmp)。	
关闭	关闭窗口。	

* 颜色是模态值,否则为黑色/白色或橙色/蓝色/绿色/青色/红色/洋红色/黄色/灰色,后接可选的 0..15 亮度

（默认值为 8）。

位图显示　　像素驱动的位图

```
CON _clkfreq = 10_000_000

PUB go() | i

    调试（`位图 MyBitmap SIZE 32 16 DOTSIZE 8 LUT2 LONGS_2BIT）
    调试（`MyBitmap TRACE 14 LUTCOLORS 白色 红色 蓝色 黄色 6）
    重复
        调试（`MyBitmap`uhex_（flag[i++ & $1F])`dly（100））

数据

标志长%%3333333333333330
            长整型 %%0010101022222220
            长整型 %%0010101020202020
            长整型 %%0010101022222220
            长整型 %%0010101022020220
            长整型 %%0010101022222220
            长整型 %%0010101020202020
            长整型 %%0010101022222220
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型 %%0010101010101010
            长整型%%0000000000000000
            长整型%%0000000000000000
            长整型%%0000000000000000
            长整型%%0000000000000000
            长整型%%0000000000000000
```

BITMAP实例	描述	默认
TITLE‘字符串’	将窗口标题设置为“字符串”。	<无>

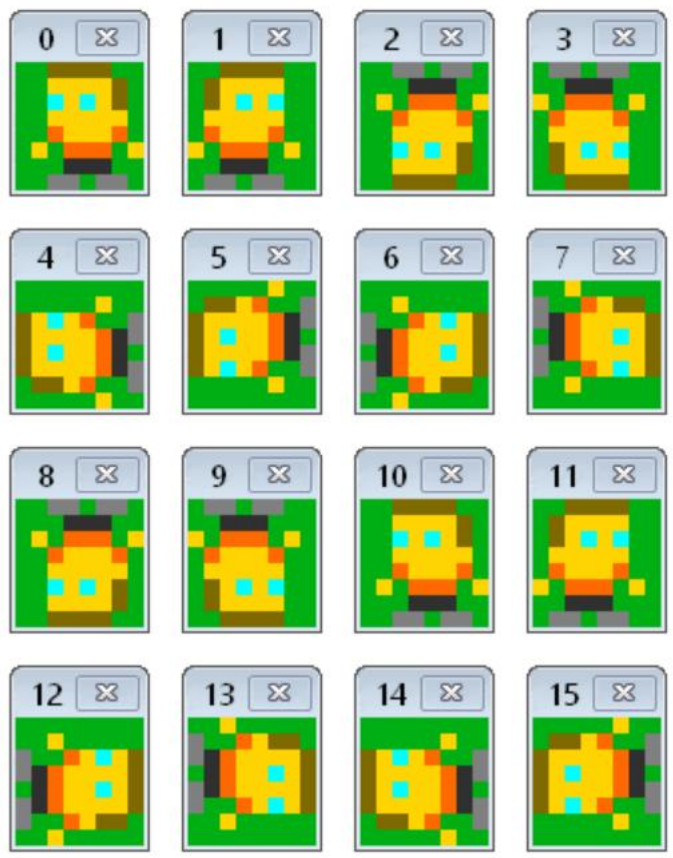
Machine Translated by Google

POS左上角	设置窗口位置。	0,0
大小 x_像素 y_像素	设置位图中的像素数（x 和 y 均为 1..2048）。	256,256
DOTSIZE 宽度和高度 {高度}	将位图像素宽度和像素高度（1..256)一起设置,或者单独设置它们。	1,1
稀疏颜色	在彩色背景上显示较大的圆形像素。点大小 (DOTSIZE) 必须至少为 4。 *	<关闭>
lut1_to_rgb24	设置颜色模式。参见下图。	RGB24
LUTCOLORS rgb24 rgb24 ...	对于 LUT1..LUT8 颜色模式,使用 RGB24 种颜色加载 LUT。使用 HEX_LONG_ARRAY_ 进行加载。	默认颜色 0..7
跟踪 0_到_15	设置像素加载方向以及每行填充完成后是否滚动。参见下方动画。	0
每次更新像素率	设置每次显示更新前的像素数。“RATE -1”将速率设置为位图大小。	线尺寸
打包数据模式	启用打包数据模式。请参阅本节末尾的说明。	<无>
更新	设置更新模式。仅当输入“更新”命令时,显示才会更新。	自动更新
希德西	隐藏鼠标指针上显示的 X、Y 鼠标坐标。	没有隐藏
位图馈送	描述	默认
像素	输入位图的数字像素数据。	
lut1_to_rgb24	更改颜色模式。	RGB24
LUTCOLORS rgb24 rgb24 ...	对于 LUT1..LUT8 颜色模式,使用 rgb24 颜色加载 LUT。使用 HEX_LONG_ARRAY_ 加载颜色。	默认颜色 0..7
跟踪 0_到_15	更改像素加载到位图的方向。将速率设置为线条大小。	0
每次更新像素率	设置每次显示更新前的像素数。“RATE -1”将速率设置为位图大小。	
设置 x_position {y_position}	设置当前像素加载位置。清除TRACE寄存器的第3位可取消滚动模式。	
滚动 x_scroll y_scroll	将位图滚动一定数量的像素。正/负值决定方向,0 = 无。	
清除	将位图清除为零值像素。	
更新	使用当前位图更新窗口。用于 UPDATE 模式。	
保存 {WINDOW}‘文件名’	保存整个窗口的位图文件 (.bmp) 或仅保存 1 倍比例的位图。	
关闭	关闭窗口。	

* 颜色为橙色 / 蓝色 / 绿色 / 青色 / 红色 / 洋红色 / 黄色 / 灰色。

TRACE 模式

速率设置为 1,以便每个像素在加载时都可以看到。

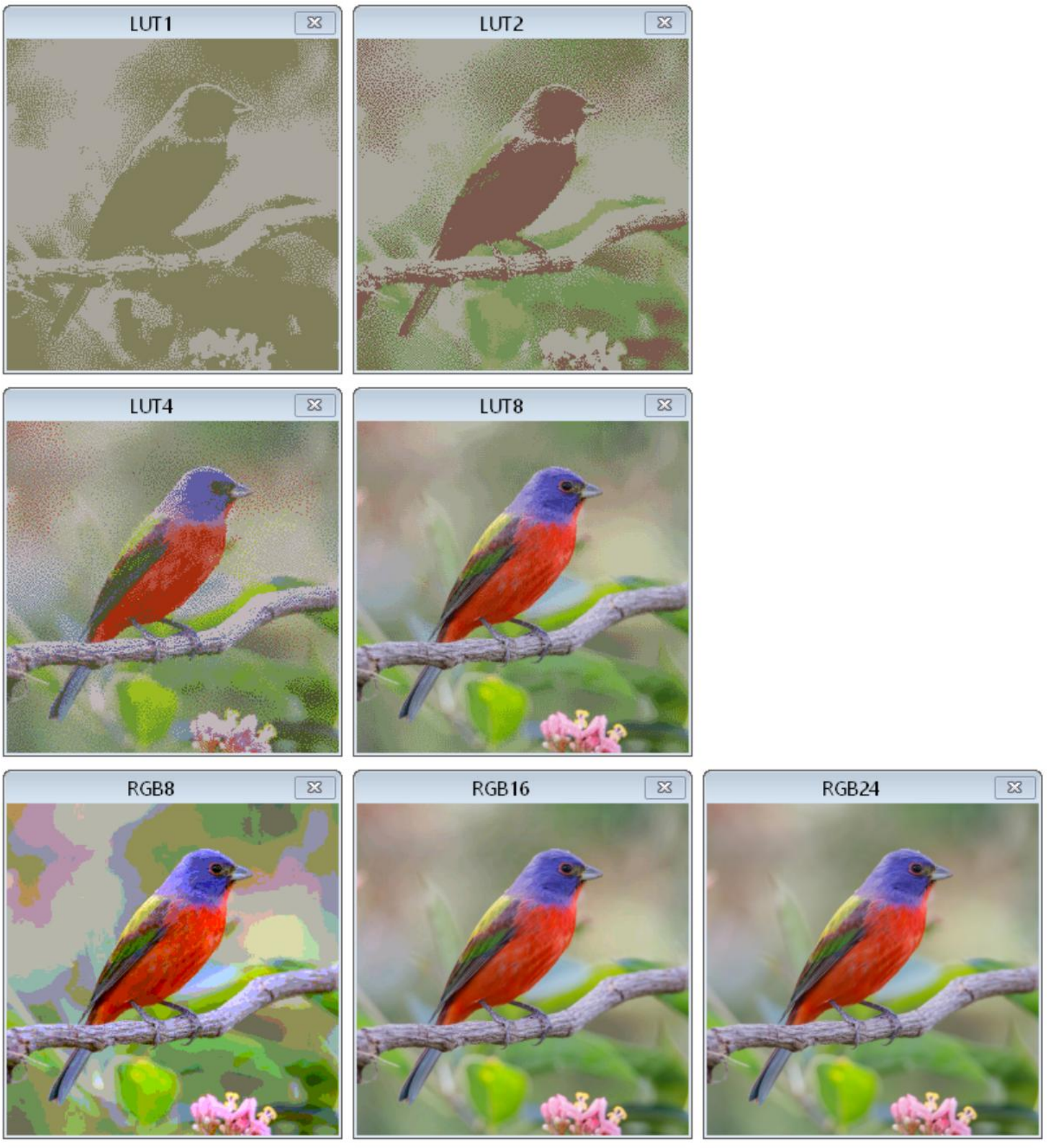


颜色模式	位/像素	描述	意图
查找表1	1	像素索引 LUT 颜色 0..1	节省内存的双色调色板图形
查找表2	2	像素索引 LUT 颜色 0..3	节省内存的 4 色调色板图形
LUT4	4	像素索引 LUT 颜色 0..15	节省内存的 16 色调色板图形
LUT8	8	像素索引 LUT 颜色 0..255	节省内存的 256 色调色板图形。

Machine Translated by Google

LUMA8	8	从黑色到彩色 *	亮度指示液位的仪器
LUMA8W	8	从白色到彩色 *	饱和度指示水平的仪器
LUMA8X	8	从黑色到彩色 * 变白	亮度指示水平的仪器,白色峰值
单纯疱疹病毒8	8	从黑色到彩色:%HHHHSSSS	16 种色调,16 种亮度级别
HSV8W	8	从白色到彩色:%HHHHSSSS	16 种色调,16 种饱和度,来自白色
HSV8X	8	从黑色到彩色到白色:%HHHHSSSS	16 种色调,16 种亮度等级,白色达到峰值
RGBI8	8	从黑色到彩色:%RGBIIII	8 种基本颜色,32 个亮度等级
RGBI8W	8	从白色到彩色:%RGBIIII	8 种基本颜色,32 种饱和度等级,来自白色
RGBI8X	8	从黑色到彩色到白色:%RGBIIII	8 种基本颜色,32 种亮度等级,白色达到峰值
RGB8	8	%RRRGGGBB	字节级 RGB,具有 8 个红色、8 个绿色和 4 个蓝色级别
单纯疱疹病毒16	16	从黑色到彩色:%HHHHHHHH_SSSSSSSS	256 种色调,256 种亮度级别
HSV16W	16	从白色到彩色:%HHHHHHHH_SSSSSSSS	256 种色调,256 种饱和度,来自白色
HSV16X	16	从黑色到彩色再到白色:%HHHHHHHH_SSSSSSSS	256 种色调,256 种亮度级别,白色达到峰值
RGB16	16	%RRRRRGGG_GGGBBBBB	具有 32 个红色级别、64 个绿色级别和 32 个蓝色级别的字级 RGB
RGB24	24	%RRRRRRRR_GGGGGGGG_BBBBBBBB	全 RGB,红、绿、蓝 256 级

* 颜色为橙色 / 蓝色 / 绿色 / 青色 / 红色 / 洋红色 / 黄色 / 灰色。

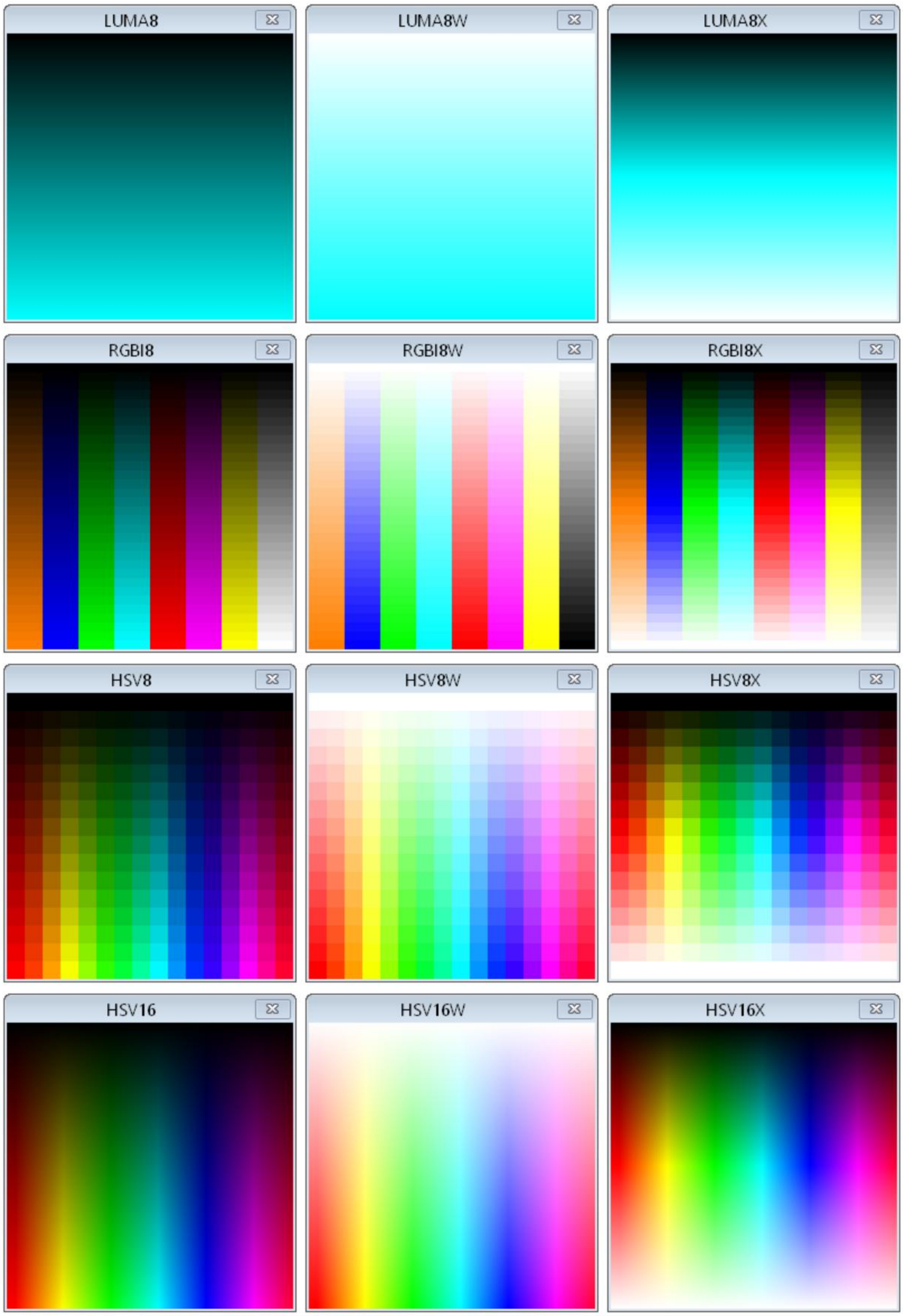



```
CON _clkfreq = 100_000_000

PUB go() | i
    调试（`位图 a 标题   LUT1    pos 100 100 跟踪 2 lut1 longs_1bit alt）
    调试（`位图 b 标题   LUT2    pos 370 100 跟踪 2 lut2 longs_2bit alt）
    调试（`位图 c 标题   LUT4    pos 100 395 跟踪 2 lut4 longs_4bit alt）
```

Machine Translated by Google

```
    调试（`位图 d 标题    LUT8    位置 370 395 跟踪 2 lut8 longs_8bit)调试（`位图 e 标题    RGB8    位置
    100 690 跟踪 2 rgb8)调试（`位图 f 标题    RGB16    位置 370 690 跟踪 2 rgb16)调试（`
    位图 g 标题    RGB24    位置 640 690 跟踪 2 rgb24)等待毫秒（1000）



    showbmp(    a    , @image1, $8A, 2, $800) showbmp(    b    ,          发送 LUT1 图像    发
    @image2, $36, 4, $1000) showbmp(    c    , @image3, $8A,          送 LUT2 图像    发送
    16, $2000) showbmp(    d    , @image4, $36, 256, $4000)          LUT4 图像    发送 LUT8
                                                                      图像

    i := @image5 + $36    从同一个 24-bpp 文件发送 RGB8/RGB16/RGB24 图像 repeat $10000 debug(`e `uhex_(byte[i+0] >> 6 + byte[i+1] >> 5 << 2 +
    byte[i+2] >> 5 << 5 ))
        debug(`f `uhex_(byte[i+0] >> 3 + byte[i+1] >> 2 << 5 + byte[i+2] >> 3 << 11)) debug(`g `uhex_(byte[i+0] + byte[i+1] << 8 + byte[i+2] << 16 )) i += 3



PRI showbmp(字母,图像地址,lut偏移量,lut大小,图像长度) | i
    image_address += lut_offset
    debug(``#(letter) lutcolors `uhex_long_array_(image_address, lut_size)) image_address += lut_size << 2
    - 4 重复 image_longs

        调试（``#（字母)`uhex_（long[image_address += 4]））

DAT
图像 1 文件“bird_lut1.bmp” 图像 2 文件“bird_lut2.bmp”
图像 3 文件“bird_lut4.bmp” 图像 4 文件“bird_lut8.bmp”
图像 5 文件“bird_rgb24.bmp”
```

Machine Translated by Google

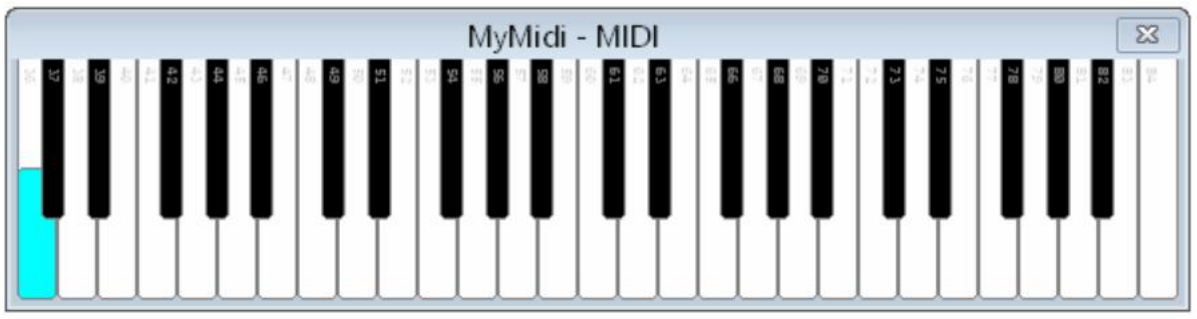



```
CON _clkfreq = 100_000_000

酒吧去（）| i 调试（`
    位图 a 标题    LUMA8    位置 100 100 大小 1 256 点大小 256 1 luma8 青色)调试（`位图 b 标题    LUMA8W    位置 370 100 大小 1
    256 点大小 256 1 luma8w 青色)调试（`位图 c 标题    LUMA8X    位置 640 100 大小 1 256 点大小 256 1 luma8x 青色)调试（`位图 d
    标题    RGBI8    位置 100 395 大小 8 32 点大小 32 8 跟踪 4 rgbi8)调试（`位图 e 标题    RGBI8W    位置 370 395 大小 8 32 点大小 32
    8 跟踪 4 rgbi8w)调试（`位图 f 标题    RGBI8X    位置640 395 大小 8 32 点大小 32 8 跟踪 4 rgbi8x) 调试（`位图 g 标题    HSV8    位
    置 100 690 大小 16 16 跟踪 4 点大小 16 hsv8) 调试（`位图 h 标题    HSV8W    位置 370 690 大小 16 16 跟踪 4 点大小 16 hsv8w) 调试
    （`位图 i 标题    HSV8X    位置 640 690 大小 16 16 跟踪 4 点大小 16 hsv8x) 调试（`位图 j 标题    HSV16    位置 100 985 大小 256 256
    跟踪 4 hsv16) 调试（`位图 k 标题    HSV16W    位置 370 985 大小 256 256 跟踪 4 hsv16w) debug(`bitmap l title    HSV16X
    pos 640 985 size 256 256 trace 4 hsv16x) waitms(1000) 重复 i 从 0 到 255 debug(`abcdefghi `uhex_(i)) 重复 i 从 0 到 65535
    debug(`jkl `uhex_(i))

                                    馈送 8 位显示器

                                    馈送 16 位显示器
```

Machine Translated by Google

MIDI显示　　MIDI 键盘,用于通过力度查看音符开启/关闭状态

```
CON _clkfreq = 10_000_000

PUB go() | i

    调试（`midi MyMidi 大小 3 范围 36 84）
    重复
        重复 i 从 36 到 84
            调试（`MyMidi $90`（i,getrnd（)&$7F））
            waitms(150)
            调试（`MyMidi $80`（i,0））
```

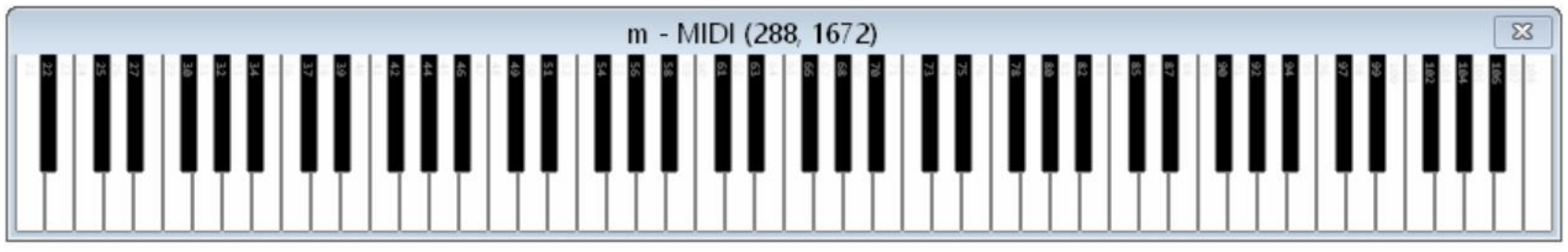


MIDI 实例	描述	默认
TITLE ‘字符串’	将窗口标题设置为“字符串”。	<无>
POS左上角	设置窗口位置。	0,0
尺寸 键盘尺寸	设置 MIDI 键盘显示的大小 (1..50)。	4
范围 第一个键 最后一个键	设置第一个和最后一个 MIDI 键编号 (0..127)。	21,108（88键）
频道 频道号	设置要观察的 MIDI 通道号（0..15）。	0
颜色 白键 黑键	将白键和黑键设置为“ON”颜色。 *	青色、品红色
MIDI 喂食	**描述**	**默认**
字节	如果 ($90 + 通道) 则为 NOTE_ON 模式,否则如果 ($80 + 通道) 则为 NOTE_OFF 模式。 如果是 NOTE_ON 模式,则接收一个键（$00..$7F)及其速度（$00..$7F）,更新显示。 如果是 NOTE_OFF 模式,则接收一个键（$00..$7F)及其速度（$00..$7F）,更新显示。	
清除	清除所有注释。	
保存 {WINDOW}‘文件名’	保存整个窗口或仅显示区域的位图文件 (.bmp)。	
关闭	关闭窗口。	

* 颜色为黑色/白色或橙色/蓝色/绿色/青色/红色/洋红色/黄色/灰色,亮度可选 0..15（默认为 8）。

这是一个 PASM 程序,它在 P16 上接收 MIDI 串行并将其发送到 MIDI 显示器：

```
反对        _clkfreq 接        = 10_000_000
            收引脚             = 16

数据        组织

            调试（`midi m 大小 2）

            wrpin #%11111_0,#rxpin
            wxpin ##(clkfreq_/31250) << 16 + 8-1, #rxpin
            drvl #rxpin

.等待 testp #rxpin wc
  if_nc jmp #.等待

            rdpin x,#rxpin
            shr x,#32-8

            调试（“`m”,uhex_byte_（x））

            跳转       #。等待

x           水库       1
```



打包数据模式

打包数据模式用于高效传输子字节数据类型,主机端会将其从接收到的字节、字或长整型数据中解包出来。此外,字节也可以发送

在字和长整型内,并且可以在长整型内发送字,以提高效率。

Machine Translated by Google

要建立打包数据操作,您必须指定下面列出的模式之一,后跟可选的“ALT”和“SIGNED”关键字:

packed_data_mode {ALT} {SIGNED}

ALT关键字将导致发送的每个字节中的位、双位或半字节在主机端按首尾顺序重新排序。这简化了以下情况:
你发送的原始数据的位字段相对于你正在使用的 DEBUG 显示器而言是乱序的。这对于在
标准格式。

SIGNED关键字将导致所有解压的数据值在主机端进行符号扩展。

打包数据模式	描述	最终值	最终值如果已签名
LONGS_1BIT	接收到的每个值都被转换成 32 个独立的 1 位值,从接收到的值的 LSB 开始。	0..1	-1..0
LONGS_2BIT	接收到的每个值都被转换成 16 个独立的 2 位值,从接收到的值的 LSB 开始。	0..3	-2..1
LONGS_4BIT	接收到的每个值都被转换成 8 个独立的 4 位值,从接收到的值的 LSB 开始。	0..15	-8..7
LONGS_8BIT	接收到的每个值都被转换成 4 个独立的 8 位值,从接收到的值的 LSB 开始。	0..255	-128..127
LONGS_16BIT	接收到的每个值都被转换成 2 个独立的 16 位值,从接收到的值的 LSB 开始。	0..65,535	-32,768...32,767
WORDS_1BIT	接收到的每个值都被转换成 16 个独立的 1 位值,从接收到的值的 LSB 开始。	0..1	-1..0
WORDS_2BIT	接收到的每个值都被转换成 8 个单独的 2 位值,从接收到的值的 LSB 开始。	0..3	-2..1
WORDS_4BIT	接收到的每个值都被转换成 4 个独立的 4 位值,从接收到的值的 LSB 开始。	0..15	-8..7
字_8位	接收到的每个值都被转换成 2 个独立的 8 位值,从接收到的值的 LSB 开始。	0..255	-128..127
BYTES_1BIT	接收到的每个值都被转换成 8 个单独的 1 位值,从接收到的值的 LSB 开始。	0..1	-1..0
BYTES_2BIT	接收到的每个值都被转换成 4 个独立的 2 位值,从接收到的值的 LSB 开始。	0..3	-2..1
字节_4位	接收到的每个值都被转换成 2 个独立的 4 位值,从接收到的值的 LSB 开始。	0..15	-8..7

智能引脚配置的内置符号

智能密码符号值	符号名称	细节
输入极性	(选一个)	
%0000_0000_000_00000000000000_00_00000_0	P_TRUE_A(默认)	真 A 输入
%1000_0000_000_00000000000000_00_00000_0	P_INVERT_A	反转 A 输入
输入选择	(选一个)	
%0000_0000_000_00000000000000_00_00000_0	P_LOCAL_A(默认)	选择 A 输入的本地引脚
%0001_0000_000_0000000000000_00_00000_0	P_PLUS1_A	选择引脚+1作为A输入
%0010_0000_000_00000000000000_00_00000_0	P_PLUS2_A	选择引脚+2 作为 A 输入
%0011_0000_000_0000000000000_00_00000_0	P_PLUS3_A	选择引脚+3 作为 A 输入
%0100_0000_000_0000000000000_00_00000_0	P_OUTBIT_A	选择 A 输入的 OUT 位
%0101_0000_000_0000000000000_00_00000_0	P_MINUS3_A	选择引脚 3 作为 A 输入
%0110_0000_000_00000000000000_00_00000_0	P_MINUS2_A	选择引脚 2 作为 A 输入
%0111_0000_000_0000000000000_00_00000_0	P_MINUS1_A	选择引脚 1 作为 A 输入
B 输入极性	(选一个)	
%0000_0000_000_00000000000000_00_00000_0	P_TRUE_B(默认)	真 B 输入
%0000_1000_000_0000000000000_00_00000_0	P_INVERT_B	反转 B 输入
B 输入选择	(选一个)	
%0000_0000_000_00000000000000_00_00000_0	P_LOCAL_B(默认)	选择 B 输入的本地引脚
%0000_0001_000_0000000000000_00_00000_0	P_PLUS1_B	选择引脚+1作为B输入
%0000_0010_000_0000000000000_00_00000_0	P_PLUS2_B	选择引脚+2 作为 B 输入
%0000_0011_000_0000000000000_00_00000_0	P_PLUS3_B	选择引脚+3 作为 B 输入
%0000_0100_000_0000000000000_00_00000_0	P_OUTBIT_B	选择 B 输入的 OUT 位
%0000_0101_000_0000000000000_00_00000_0	P_MINUS3_B	选择引脚 3 作为 B 输入
%0000_0110_000_0000000000000_00_00000_0	P_MINUS2_B	选择引脚 2 作为 B 输入
%0000_0111_000_0000000000000_00_00000_0	P_MINUS1_B	选择引脚 1 作为 B 输入
A、B输入逻辑	(选一个)	

Machine Translated by Google

%0000_0000_000_00000000000000_00_00000_0	P_PASS_AB（默认）	选择 A、B
%0000_0000_001_0000000000000_00_00000_0	P_AND_AB	选择 A 和 B、B
%0000_0000_010_0000000000000_00_00000_0	P_OR_AB	选择 A | B,B
%0000_0000_011_0000000000000_00_00000_0	P_XOR_AB	选择 A ^ B,B
%0000_0000_100_0000000000000_00_00000_0	P_FILT0_AB	选择 A、B 的 FILT0 设置
%0000_0000_101_0000000000000_00_00000_0	P_FILT1_AB	选择 A、B 的 FILT1 设置
%0000_0000_110_0000000000000_00_00000_0	P_FILT2_AB	选择 A、B 的 FILT2 设置
%0000_0000_111_0000000000000_00_00000_0	P_FILT3_AB	选择 A、B 的 FILT3 设置
低级引脚模式	（选一个）	
逻辑/施密特/比较器输入模式		
%0000_0000_000_00000000000000_00_00000_0	P_LOGIC_A（默认）	逻辑电平A→IN,输出OUT
%0000_0000_000_0001000000000_00_00000_0	P_LOGIC_A_FB	逻辑电平A→IN,输出反馈
%0000_0000_000_0010000000000_00_00000_0	P_逻辑_B_FB	逻辑电平B→IN,输出反馈
%0000_0000_000_0011000000000_00_00000_0	P_SCHMITT_A	施密特触发器A→IN,输出OUT
%0000_0000_000_0100000000000_00_00000_0	P_SCHMITT_A_FB	施密特触发器A→IN,输出反馈
%0000_0000_000_0101000000000_00_00000_0	P_SCHMITT_B_FB	施密特触发器B→IN,输出反馈
%0000_0000_000_0110000000000_00_00000_0	P_COMPARE_AB	A > B → IN,输出OUT
%0000_0000_000_0111000000000_00_00000_0	比较器比较AB_FB	A > B → IN,输出反馈
%xxxx_xxxx_xxx_xxxxSIOHHHLLL_xx_xxxxx_x		同步模式、输入/输出极性、高/低驱动
ADC输入模式		
%0000_0000_000_1000000000000_00_00000_0	ADC_GIO	ADC GIO → IN,输出OUT
%0000_0000_000_1000010000000_00_00000_0	电压输入	ADC VIO → IN,输出OUT
%0000_0000_000_1000100000000_00_00000_0	ADC浮点数	ADC FLOAT → IN,输出OUT
%0000_0000_000_1000110000000_00_00000_0	P_ADC_1X	ADC 1x → 输入,输出 OUT
%0000_0000_000_1001000000000_00_00000_0	电压表	ADC 3.16x → 输入,输出 OUT
%0000_0000_000_1001010000000_00_00000_0	电压模数转换器 (ADC) 10X	ADC 10x → 输入,输出 OUT
%0000_0000_000_1001100000000_00_00000_0	P_ADC_30X	ADC 31.6x → 输入,输出 OUT
%0000_0000_000_1001110000000_00_00000_0	电压模数转换器 (ADC) 100X	ADC 100x → 输入,输出 OUT
%xxxx_xxxx_xxx_xxxxxxOHHHLLL_xx_xxxxx_x		O = 输出极性,HHH/LLL = 高/低驱动
DAC输出模式		DIR 启用输出,OUT 启用 ADC
%0000_0000_000_1010000000000_00_00000_0	P_DAC_990R_3V	DAC 990Ω,3.3V 峰值,ADC 1x → IN
%0000_0000_000_1010100000000_00_00000_0	P_DAC_600R_2V	DAC 600Ω,2.0V 峰值,ADC 1x → IN
%0000_0000_000_1011000000000_00_00000_0	P_DAC_124R_3V	DAC 123.75Ω,3.3V 峰值,ADC 1x → IN
%0000_0000_000_1011100000000_00_00000_0	P_DAC_75R_2V	DAC 75Ω,2.0V 峰值,ADC 1x → IN
%xxxx_xxxx_xxx_xxxxxDDDDDDDD_xx_xxxxx_x		DDDDDDDD = 8 位 DAC 值
级别比较模式		DIR 使能输出（1.5kΩ 驱动）
%0000_0000_000_1100000000000_00_00000_0	P_LEVEL_A	A > 电平 → IN,输出 OUT
%0000_0000_000_1101000000000_00_00000_0	P_LEVEL_A_FBN	A>电平→IN,输出负反馈
%0000_0000_000_1110000000000_00_00000_0	P_LEVEL_B_FBP	B>电平→IN,输出正反馈
%0000_0000_000_1111000000000_00_00000_0	P_LEVEL_B_FBN	B>电平→IN,输出负反馈
%xxxx_xxxx_xxx_xxxxSLLLLLLLL_xx_xxxxx_x		S = 同步,LLLLLLLL = 8 位级别
低级引脚子模式		
同步模式	（选一个）	（适用于逻辑/施密特/比较器/电平模式）
%xxxx_xxxx_xxx_xxxxSxxxxxxxx_xx_xxxxx_x		同步模式位
%0000_0000_000_00000000000000_00_00000_0	P_ASYNC_IO（默认）	选择异步 I/O
%0000_0000_000_0000100000000_00_00000_0	同步输入输出	选择同步 I/O
输入极性	（选一个）	（适用于逻辑/施密特/比较器模式）
%xxxx_xxxx_xxx_xxxxxIxxxxxxx_xx_xxxxx_x		IN极性位
%0000_0000_000_00000000000000_00_00000_0	P_TRUE_IN（默认）	真 IN 位
%0000_0000_000_0000010000000_00_00000_0	P_INVERT_IN	反转 IN 位
输出极性	（选一个）	（适用于逻辑/施密特/比较器/ADC 模式）

Machine Translated by Google

%xxxx_xxxx_xxx_xxxxxxOxxxxxx_xx_xxxxx_x		输出极性位
%0000_0000_000_00000000000000_00_00000_0	P_TRUE_OUTPUT（默认） P_TRUE_OUT（简洁起见）	选择真实输出
%0000_0000_000_0000001000000_00_00000_0	P_INVERT_OUTPUT P_INVERT_OUT（简洁起见）	选择反相输出
高强度驱动	（选一个）	（适用于逻辑/施密特/比较器/ADC 模式）
%xxxx_xxxx_xxx_xxxxxxxHHHxxx_xx_xxxxx_x		驱动高选择器位
%0000_0000_000_00000000000000_00_00000_0	P_HIGH_FAST（默认）	快速驱动高电平（30mA）
%0000_0000_000_0000000001000_00_00000_0	P_HIGH_1K5	驱动高1.5kΩ
%0000_0000_000_0000000010000_00_00000_0	P_HIGH_15K	驱动高15kΩ
%0000_0000_000_0000000011000_00_00000_0	P_HIGH_150K	驱动高150kΩ
%0000_0000_000_0000000100000_00_00000_0	P_HIGH_1MA	驱动高1mA
%0000_0000_000_0000000101000_00_00000_0	P_HIGH_100UA	驱动高100μA
%0000_0000_000_0000000110000_00_00000_0	P_HIGH_10UA	驱动高10μA
%0000_0000_000_0000000111000_00_00000_0	P_HIGH_FLOAT	漂浮高
低强度驱动	（选一个）	（适用于逻辑/施密特/比较器/ADC 模式）
%xxxx_xxxx_xxx_xxxxxxxxxxLLL_xx_xxxxx_x		驱动低选择器位
%0000_0000_000_00000000000000_00_00000_0	P_LOW_FAST（默认）	快速驱动至低电平（30mA）
%0000_0000_000_00000000000001_00_00000_0	P_LOW_1K5	驱动低1.5kΩ
%0000_0000_000_0000000000010_00_00000_0	低15K	驱动低15kΩ
%0000_0000_000_00000000000011_00_00000_0	低电平 150K	驱动低150kΩ
%0000_0000_000_0000000000100_00_00000_0	低电平1MA	驱动低1mA
%0000_0000_000_00000000000101_00_00000_0	P_LOW_100UA	驱动低 100μA
%0000_0000_000_0000000000110_00_00000_0	低电平_10UA	驱动低10μA
%0000_0000_000_00000000000111_00_00000_0	P_LOW_FLOAT	低浮
DIR/OUT 控制	（选一个）	
%0000_0000_000_00000000000000_00_00000_0	P_TT_00（默认）	TT = %00
%0000_0000_000_00000000000000_01_00000_0	P_TT_01	TT = %01
%0000_0000_000_00000000000000_10_00000_0	P_TT_10	TT = %10
%0000_0000_000_00000000000000_11_00000_0	P_TT_11	TT = %11
%0000_0000_000_00000000000000_01_00000_0	电源	在智能引脚模式下启用输出,无论 DIR 如何
%0000_0000_000_00000000000000_01_00000_0	P_通道	在非智能引脚DAC模式下启用DAC通道
%0000_0000_000_00000000000000_10_00000_0	P_BITDAC	启用 BITDAC 以实现非智能引脚 DAC 模式
智能密码模式	（选一个）	
%0000_0000_000_00000000000000_00_00000_0	P_NORMAL（默认）	正常模式（非智能密码模式）
%0000_0000_000_00000000000000_00_00001_0	P_REPOSITORY	长存储库（非 DAC 模式）
%0000_0000_000_00000000000000_00_00001_0	P_DAC_噪声	DAC噪声（DAC模式）
%0000_0000_000_00000000000000_00_00010_0	P_DAC_DITHER_RND	DAC 16 位随机抖动（DAC 模式）
%0000_0000_000_00000000000000_00_00011_0	P_DAC_DITHER_PWM	DAC 16 位 PWM 抖动（DAC 模式）
%0000_0000_000_00000000000000_00_00100_0	脉冲	脉冲/周期输出
%0000_0000_000_00000000000000_00_00101_0	P_过渡	过渡输出
%0000_0000_000_00000000000000_00_00110_0	P_NCO_频率	NCO频率输出
%0000_0000_000_00000000000000_00_00111_0	士官职责	NCO 任务输出
%0000_0000_000_00000000000000_00_01000_0	P_PWM_三角形	PWM三角波输出
%0000_0000_000_00000000000000_00_01001_0	P_PWM_锯齿波	PWM锯齿波输出
%0000_0000_000_00000000000000_00_01010_0	电源PWM开关电源	PWM开关电源I/O
%0000_0000_000_00000000000000_00_01011_0	P_QUADRATURE	AB正交编码器输入
%0000_0000_000_00000000000000_00_01100_0	P_REG_UP	B 为高时 A 为上升 公司
%0000_0000_000_00000000000000_00_01101_0	P_REG_UP_DOWN	当 B 高时,A 上升时增加,当 B 低时,A 上升时减少
%0000_0000_000_00000000000000_00_01110_0	P_COUNT_RISES	A 点上升时上涨,B 点上升时可选下跌
%0000_0000_000_00000000000000_00_01111_0	P_COUNT_HIGHS	在 A 高时增加,在 B 高时可选减少
%0000_0000_000_00000000000000_00_10000_0	P_STATE_TICKS	对于 A-low 和 A-high 状态,计数 ticks

Machine Translated by Google

%0000_0000_000_00000000000000_00_10001_0	P_HIGH_TICKS	对于 A-high 状态,计数 ticks
%0000_0000_000_00000000000000_00_10010_0	~~CP_EVENTS_TICKS~~	对于 X A-高点/上升/边缘,计数刻度/ X 个无 A 高/上升/边沿的刻度超时
%0000_0000_000_00000000000000_00_10011_0	P_PERIODS_TICKS	对于 A 的 X 个周期,计数刻度
%0000_0000_000_00000000000000_00_10100_0	P_PERIODS_HIGHS	对于 A 的 X 个周期,计算最高点
%0000_0000_000_00000000000000_00_10101_0	P_COUNTER_TICKS	对于 X+ 个刻度内的 A 周期,计数刻度
%0000_0000_000_00000000000000_00_10110_0	最高点	对于 X+ 个刻度内的 A 周期,计算最高点
%0000_0000_000_00000000000000_00_10111_0	P_COUNTER_PERIODS	对于 X+ 个刻度内的 A 周期,计数周期
%0000_0000_000_00000000000000_00_11000_0	压控ADC	ADC 采样/滤波/捕获,内部时钟
%0000_0000_000_00000000000000_00_11001_0	电压源ADC扩展	ADC 采样/滤波/捕获,外部时钟
%0000_0000_000_00000000000000_00_11010_0	P_ADC_范围	带触发器的 ADC 示波器
%0000_0000_000_00000000000000_00_11011_0	USB 对	USB 引脚对
%0000_0000_000_00000000000000_00_11100_0	同步发送	同步串行传输
%0000_0000_000_00000000000000_00_11101_0	同步接收	同步串行接收
%0000_0000_000_00000000000000_00_11110_0	异步发送	异步串行传输
%0000_0000_000_00000000000000_00_11111_0	异步接收	异步串行接收

内置 Streamer 模式符号

飘带符号值	符号名称
立即数 → LUT → 引脚/DAC	
%0000_0000_0000_0000 << 16 %0000_DDDD_EPPP_BBBB << 16	X_IMM_32X1_LUT
%0001_0000_0000_0000 << 16 %0001_DDDD_EPPP_BBBB << 16	X_IMM_16X2_LUT
%0010_0000_0000_0000 << 16 %0010_DDDD_EPPP_BBBB << 16	X_IMM_8X4_LUT
%0011_0000_0000_0000 << 16 %0011_DDDD_EPPP_BBBB << 16	X_IMM_4X8_LUT
立即→引脚/DAC	
%0100_0000_0000_0000 << 16 %0100_DDDD_EPPP_PPPA << 16	X_IMM_32X1_1DAC1
%0101_0000_0000_0000 << 16 %0101_DDDD_EPPP_PP0A << 16	X_IMM_16X2_2DAC1
%0101_0000_0000_0010 << 16 %0101_DDDD_EPPP_PP1A << 16	X_IMM_16X2_1DAC2
%0110_0000_0000_0000 << 16 %0110_DDDD_EPPP_P00A << 16	X_IMM_8X4_4DAC1
%0110_0000_0000_0010 << 16 %0110_DDDD_EPPP_P01A << 16	X_IMM_8X4_2DAC2
%0110_0000_0000_0100 << 16 %0110_DDDD_EPPP_P10A << 16	X_IMM_8X4_1DAC4
%0110_0000_0000_0110 << 16 %0110_DDDD_EPPP_0110 << 16	X_IMM_4X8_4DAC2
%0110_0000_0000_0111 << 16 %0110_DDDD_EPPP_0111 << 16	X_IMM_4X8_2DAC4
%0110_0000_0000_1110 << 16 %0110_DDDD_EPPP_1110 << 16	X_IMM_4X8_1DAC8
%0110_0000_0000_1111 << 16 %0110_DDDD_EPPP_1111 << 16	X_IMM_2X16_4DAC4
%0111_0000_0000_0000 << 16 %0111_DDDD_EPPP_0000 << 16	X_IMM_2X16_2DAC8
%0111_0000_0000_0001 << 16 %0111_DDDD_EPPP_0001 << 16	X_IMM_1X32_4DAC8
RDFAST → LUT → 引脚/DAC	
%0111_0000_0000_0010 << 16 %0111_DDDD_EPPP_001A << 16	X_RFLONG_32X1_LUT
%0111_0000_0000_0100 << 16 %0111_DDDD_EPPP_010A << 16	X_RFLONG_16X2_LUT
%0111_0000_0000_0110 << 16 %0111_DDDD_EPPP_011A << 16	X_RFLONG_8X4_LUT

Machine Translated by Google

%0111_0000_0000_1000 << 16 %0111_DDDD_EPPP_1000 << 16	X_RFLONG_4X8_LUT
RDFAST → 引脚 / DAC	
%1000_0000_0000_0000 << 16 %1000_DDDD_EPPP_PPPA << 16	X_RFBYTE_1P_1DAC1
%1001_0000_0000_0000 << 16 %1001_DDDD_EPPP_PP0A << 16	X_RFBYTE_2P_2DAC1
%1001_0000_0000_0010 << 16 %1001_DDDD_EPPP_PP1A << 16	X_RFBYTE_2P_1DAC2
%1010_0000_0000_0000 << 16 %1010_DDDD_EPPP_P00A << 16	X_RFBYTE_4P_4DAC1
%1010_0000_0000_0010 << 16 %1010_DDDD_EPPP_P01A << 16	X_RFBYTE_4P_2DAC2
%1010_0000_0000_0100 << 16 %1010_DDDD_EPPP_P10A << 16	X_RFBYTE_4P_1DAC4
%1010_0000_0000_0110 << 16 %1010_DDDD_EPPP_0110 << 16	X_RFBYTE_8P_4DAC2
%1010_0000_0000_0111 << 16 %1010_DDDD_EPPP_0111 << 16	X_RFBYTE_8P_2DAC4
%1010_0000_0000_1110 << 16 %1010_DDDD_EPPP_1110 << 16	X_RFBYTE_8P_1DAC8
%1010_0000_0000_1111 << 16 %1010_DDDD_EPPP_1111 << 16	X_RFWORD_16P_4DAC4
%1011_0000_0000_0000 << 16 %1011_DDDD_EPPP_0000 << 16	X_RFWORD_16P_2DAC8
%1011_0000_0000_0001 << 16 %1011_DDDD_EPPP_0001 << 16	X_RFLONG_32P_4DAC8
RDFAST → RGB → 引脚/DAC	
%1011_0000_0000_0010 << 16 %1011_DDDD_EPPP_0010 << 16	X_RFBYTE_LUMA8
%1011_0000_0000_0011 << 16 %1011_DDDD_EPPP_0011 << 16	X_RFBYTE_RGBI8
%1011_0000_0000_0100 << 16 %1011_DDDD_EPPP_0100 << 16	X_RFBYTE_RGB8
%1011_0000_0000_0101 << 16 %1011_DDDD_EPPP_0101 << 16	X_RFWORD_RGB16
%1011_0000_0000_0110 << 16 %1011_DDDD_EPPP_0110 << 16	X_RFLONG_RGB24
引脚 → DAC / WRFAST	
%1100_0000_0000_0000 << 16 %1100_DDDD_WPPP_PPPA << 16	X_1P_1DAC1_WFBYTE
%1101_0000_0000_0000 << 16 %1101_DDDD_WPPP_PP0A << 16	X_2P_2DAC1_WFBYTE
%1101_0000_0000_0010 << 16 %1101_DDDD_WPPP_PP1A << 16	X_2P_1DAC2_WFBYTE
%1110_0000_0000_0000 << 16 %1110_DDDD_WPPP_P00A << 16	X_4P_4DAC1_WFBYTE
%1110_0000_0000_0010 << 16 %1110_DDDD_WPPP_P01A << 16	X_4P_2DAC2_WFBYTE
%1110_0000_0000_0100 << 16 %1110_DDDD_WPPP_P10A << 16	X_4P_1DAC4_WFBYTE
%1110_0000_0000_0110 << 16 %1110_DDDD_WPPP_0110 << 16	X_8P_4DAC2_WFBYTE
%1110_0000_0000_0111 << 16 %1110_DDDD_WPPP_0111 << 16	X_8P_2DAC4_WFBYTE
%1110_0000_0000_1110 << 16 %1110_DDDD_WPPP_1110 << 16	X_8P_1DAC8_WFBYTE
%1110_0000_0000_1111 << 16 %1110_DDDD_WPPP_1111 << 16	X_16P_4DAC4_WFWORD
%1111_0000_0000_0000 << 16 %1111_DDDD_WPPP_0000 << 16	X_16P_2DAC8_WFWORD
%1111_0000_0000_0001 << 16 %1111_DDDD_WPPP_0001 << 16	X_32P_4DAC8_WFLONG
ADC / 引脚 → DAC / WRFAST	
%1111_0000_0000_0010 << 16 %1111_DDDD_W000_0010 << 16	X_1ADC8_0P_1DAC8_WFBYTE
%1111_0000_0000_0011 << 16	X_1ADC8_8P_2DAC8_WFWORD

Machine Translated by Google

%1111_DDDD_WPPP_0011 << 16	
%1111_0000_0000_0100 << 16 %1111_DDDD_W000_0100 << 16	X_2ADC8_0P_2DAC8_WFWORD
%1111_0000_0000_0101 << 16 %1111_DDDD_WPPP_0101 << 16	X_2ADC8_16P_4DAC8_WFLONG
%1111_0000_0000_0110 << 16 %1111_DDDD_W000_0110 << 16	X_4ADC8_0P_4DAC8_WFLONG
DDS/Goertzel	
%1111_0000_0000_0111 << 16 %1111_DDDD_0PPP_P111 << 16	X_DDS_GOERTZEL_SINC1
%1111_0000_1000_0111 << 16 %1111_DDDD_1PPP_P111 << 16	X_DDS_GOERTZEL_SINC2
子字段	
DAC通道输出	
%xxxx_DDDD_xxxx_xxxx << 16 %0000_0000_0000_0000 << 16 %0000_0001_0000_0000 << 16 %0000_0010_0000_0000 << 16 %0000_0011_0000_0000 << 16 %0000_0100_0000_0000 << 16 %0000_0101_0000_0000 << 16 %0000_0110_0000_0000 << 16 %0000_0111_0000_0000 << 16 %0000_1000_0000_0000 << 16 %0000_1001_0000_0000 << 16 %0000_1010_0000_0000 << 16 %0000_1011_0000_0000 << 16 %0000_1100_0000_0000 << 16 %0000_1101_0000_0000 << 16 %0000_1110_0000_0000 << 16 %0000_1111_0000_0000 << 16	 X_DACS_OFF（默认） X_DACS_0_0_0_0 X_DACS_X_X_0_0 X_DACS_0_0_X_X X_DACS_X_X_X_0 X_DACS_X_X_0_X X_DACS_X_0_X_X X_DACS_0_X_X_X X_DACS_0N0_0N0 X_DACS_X_X_0N0 X_DACS_0N0_X_X X_DACS_1_0_1_0 X_DACS_X_X_1_0 X_DACS_1_0_X_X X_DACS_1N1_0N0 X_DACS_3_2_1_0
引脚输出控制	
%xxxx_xxxx_Exxx_xxxx << 16 %0000_0000_0000_0000 << 16 %0000_0000_1000_0000 << 16	 X_PINS_OFF（默认） X_PINS_ON
写入控制	
%xxxx_xxxx_Wxxx_xxxx << 16 %0000_0000_0000_0000 << 16 %0000_0000_1000_0000 << 16	 X_WRITE_OFF（默认） X_WRITE_ON
1/2/4 位交替顺序	
%xxxx_xxxx_xxxx_xxxA << 16 %0000_0000_0000_0000 << 16 %0000_0000_0000_0001 << 16	 X_ALT_OFF（默认） X_ALT_ON

事件和中断源的内置符号（仅限 PASM,请参阅硅片文档）

符号值	符号名称	细节
0	事件_INT / INT_OFF	发生中断事件或中断关闭
1	事件_CT1	CT通过CT1事件
2	事件_CT2	CT通过CT2事件
3	事件_CT3	CT通过CT3事件
4	事件_SE1	可选事件 1
5	事件_SE2	可选事件 2
6	事件_SE3	可选事件 3
7	事件_SE4	可选事件 4
8	事件模式	INA/INB 模式匹配/不匹配事件
9	事件_FBW	集线器 FIFO 块包装事件
10	事件发送	流光命令空事件
11	事件_XFI	流光命令完成事件
12	事件_XRO	Streamer NCO 翻转事件
十三	事件_XRL	流读取最后查找表位置事件
14	事件_ATN	需要关注的事件
15	事件_QMT	GETQX/GETQY-on-empty 事件

Machine Translated by Google

COGINIT() 用法的内置符号

COGINIT 符号值	符号名称	细节
%00_0000	COGEXEC（默认）	使用“COGEXEC + CogNumber”在 cogexec 模式下启动 cog
%10_0000	HUBEXEC	使用“HUBEXEC + CogNumber”在 hubexec 模式下启动 cog
%01_0000	COGEXEC_NEW	在 cogexec 模式下启动可用的 cog
%11_0000	HUBEXEC_NEW	在 hubexec 模式下启动可用的 cog
%01_0001	COGEXEC_NEW_PAIR	在 cogexec 模式下启动可用的 eve/odd cogs 对,这对于 LUT 共享很有用
%11_0001	HUBEXEC_NEW_PAIR	在 hubexec 模式下启动可用的 eve/odd 齿轮对,有助于 LUT 共享

用于 COGSPIN() 用法的内置符号

COGSPIN 符号值	符号名称	细节
%01_0000	新联合公司	启动可用的齿轮

TASKSPIN() 用法的内置符号

TASKSPIN 符号值	符号名称	细节
-1	新任务	启动可用任务

TASKSTOP() 和 TASKHALT() 用法的内置符号

TASKSPIN 符号值	符号名称	细节
-1	这项任务	停止或暂停此任务

内置数字符号

符号值	符号名称	细节
$0000_0000	错误的	与 0 相同
$FFFF_FFFF	真的	与 -1 相同
$8000_0000	NEGX	负极端整数，-2_147_483_648 ($8000_0000)
$7FFF_FFFF	POSX	正极端整数,+2_147_483_647 ($7FFF_FFFF)
$4049_0FDB	PI	π 的单精度浮点值,3.14159265

PNut.exe 的命令行选项

命令	编译 和 调试	编译 和 闪光	编译 并保存 OBJ 和 BIN	下载	开始 调试	行动	之后出现 ERROR.TXT 文件 （文件将包含以下其中一行）
普努特						启动 PNut.exe。	好的
pnut 文件名						加载源文件名（.spin2 扩展名 是假定的,但不强制执行。	好的
pnut文件名-c			✔			加载源文件名并编译， 然后退出。	好的 <文件名路径>:<行号>:错误:<错误消息>
pnut 文件名-cd	✔		✔			加载源文件名并进行编译 DEBUG,然后退出。	好的 <文件名路径>:<行号>:错误:<错误消息>
pnut 文件名-cf		✔	✔			加载源文件名并进行编译 闪存加载程序,然后退出。	好的 <文件名路径>:<行号>:错误:<错误消息>
pnut 文件名 -cb	✔	✔	✔			加载源文件名并进行编译 DEBUG 和闪存加载程序,然后 出口。	好的 <文件名路径>:<行号>:错误:<错误消息>
pnut 文件名-ci			✔			加载源文件名、编译并 保存适合的原始 flash 图像文件 写入闪存芯片,然后退出。	好的 <文件名路径>:<行号>:错误:<错误消息>

Machine Translated by Google

pnut文件名-r			✔	✔		加载源文件名,编译, 下载,然后退出。	好的 <文件名路径>:<行号>:错误:<错误消息> 串行错误
pnut 文件名 -rd	✔		✔	✔	✔	加载源文件名,编译 DEBUG,下载,启动DEBUG,然后 当 DEBUG 窗口 关闭。	好的 <文件名路径>:<行号>:错误:<错误消息> 串行错误
pnut文件名-f		✔	✔	✔		加载源文件名,编译 flash loader,下载,然后退出。	好的 <文件名路径>:<行号>:错误:<错误消息> 串行错误
pnut 文件名 -fd	✔	✔	✔	✔	✔	加载源文件名,编译 DEBUG 和闪存加载程序, 下载,启动DEBUG,然后退出 当 DEBUG 窗口关闭时。	好的 <文件名路径>:<行号>:错误:<错误消息> 串行错误
pnut文件名-b				✔		加载二进制文件名.bin并 下载。	好的 串行错误
pnut 文件名 -bd				✔	✔	加载二进制文件名.bin,下载, 启动 DEBUG,然后退出 DEBUG 窗口已关闭。	好的 串行错误
pnut-debug {通信端口} {波特率}					✔	打开 CommPort（默认值 = 1） 波特率（默认值 = 2_000_000）,开始 DEBUG,然后退出时 DEBUG 窗口已关闭。	好的 串行错误

包含批处理文件以调用 PNut.exe 并将状态返回到 STDOUT、STDERR 和 ERRORLEVEL

PNUT_SHELL.BAT文件	批处理文件行描述
@echo 关闭	取消回显到控制台。
设置 ERROR_FILE=error.txt	设置 ERROR.TXT 文件名。
如果存在 %ERROR_FILE% del /q /f %ERROR_FILE%	如果存在 ERROR.TXT,请将其删除。
如果存在%1设置GOOD_SRC=1	检查第一个参数是否为有效的源文件。
如果存在%1.spin2设置GOOD_SRC=1	检查第一个参数是否为有效的 .spin2 源文件。
如果定义了 GOOD_SRC（	如果源文件存在
pnut_v48 %1 %2 %3	...使用传递的参数调用 PNut。例如:pnut_shell filename -r
设置 pnuterror = %ERRORLEVEL%	...从 PNut 捕获错误级别（0 = 正常,1 = 错误）。
对于/f “tokens=*”%%i在（%ERROR_FILE%)中执行echo %%i 1>&2	...将 ERROR.TXT 文件复制到 STDOUT 和 STDERR。
） 别的 （	别的
设置 pnuterror=-1	...设置文件未找到错误。
echo 错误:未找到文件 - %1 1>&2	...将文件未找到错误消息返回到 STDOUT 和 STDERR。
)	
退出 %pnuterror%	返回 ERRORLEVEL。更改为“exit /b %pnuterror%”以维护控制台窗口。

时钟设置

要为程序建立初始时钟设置,您可以声明某些符号,编译器将查找这些符号来确定您的设置。这些符号必须在以下组合之一:

CON 符号声明 （例如数字可能会有所不同）	影响	集线器 %CC_SS**
CON _clkfreq = 250_000_000 _errfreq = 0	选择 XI/XO-crystal-plus-PLL 模式,假设晶体为 20 MHz。 将计算最佳 PLL 设置以实现 _clkfreq。 * 如果无法实现 _clkfreq ± _errfreq,则编译失败。	10_11
CON _xtlfreq = 12_000_000 _clkfreq = 148_500_000 _errfreq = 150_000	选择 XI/XO-crystal-plus-PLL 模式以及频率。 将计算最佳 PLL 设置以实现 _clkfreq。 * 如果无法实现 _clkfreq ± _errfreq,则编译失败。	1x_11
CON _xinfreq = 32_000_000 _clkfreq = 297_500_000 _errfreq = 100_000	选择 XI-input-plus-PLL 模式以及频率。 将计算最佳 PLL 设置以实现 _clkfreq。 * 如果无法实现 _clkfreq ± _errfreq,则编译失败。	01_11
CON _xtlfreq = 16_000_000	选择 XI/XO 晶体模式和频率。	1x_10
CON _xinfreq = 100_000_000	选择 XI 输入模式和频率。	01_10
CON _rcslow	选择以~20 KHz 运行的内部 RCSLOW 振荡器。	00_01
CON _rcfast	选择以 20 MHz+ 运行的内部 RCFAST 振荡器。	00_00
无符号且非 DEBUG 模式	选择以 20 MHz+ 运行的内部 RCFAST 振荡器。	00_00
无符号和 DEBUG 模式	选择 XI/XO 晶体模式和 20 MHz 以方便调试。	10_10

* _errfreq 声明是可选的,因为 _errfreq 默认为 1_000_000。

** 如果 _xtlfreq >= 16_000_000,则 x=0 表示每个 XI/XO 为 15pF,否则 x=1 表示每个 XI/XO 为 30pF。

在编译期间,编译器定义了两个常量符号,其值反映了编译后的时钟设置:

象征	描述
时钟模式_	编译的时钟模式,可通过 HUBSET 设置。 · 对于 Spin2 程序,在程序启动之前,HUBSET 将使用“clkmode_”调用,

Machine Translated by Google

	用于设置编译时钟模式。“clkmode_”值也将存储在集线器中 变量‘clkmode’。 · 对于纯 PASM 程序,可以使用“clkmode_”将时钟模式设置为不同于其初始 RCFAST 设置到任何晶振/PLL 编译设置,如下所示: HUBSET ##clkmode_ & !3 启动晶振/PLL,保持在 RCFAST WAITX ##20_000_000/100 等待 10 毫秒 HUBSET ##clkmode_ 切换到晶振/PLL · 应用程序层次结构中每个文件的“clkmode_”值可能不同。顶层以下的文件 文件不会继承顶级文件的值。
时钟频率	编译的时钟频率。 · 对于 Spin2 程序,“clkfreq_”值将存储在集线器变量“clkfreq”中。 · 对于纯PASM 程序,“clkfreq_”只能作为常量引用。 · 应用程序层次结构中每个文件的“clkfreq_”值可能不同。顶层以下的文件 文件不会继承顶级文件的值。

对于 Spin2 程序,需要维护两个反映当前时钟设置的集线器变量:

Spin2 变量	描述
时钟模式	当前时钟模式,位于 LONG[$40]。使用 clkmode_ 值初始化。
时钟频率	当前时钟频率,位于 LONG[$44]。使用 clkfreq_ 值初始化。
	· 对于 Spin2 方法,这些变量可以作为“clkmode”和“clkfreq”读取和写入。 与直接写入这些变量相比,使用以下方法更安全: CLKSET（新时钟模式,新时钟频率） 这样,所有其他代码都会看到“clkmode”和“clkfreq”的快速、并行更新,并且 时钟模式转换是安全地完成的,采用先前的值,以避免潜在的 时钟故障。 · 对于在Spin2 下运行的PASM 代码,可以按如下方式读写这些变量: RDLONG x,#@clkmode 将时钟模式读入 x WRLONG x,#@clkmode 将 x 写入 clkmode RDLONG x,#@clkfreq 将 clkfreq 读入 x WRLONG x,#@clkfreq 将 x 写入 clkfreq SETQ #2-1 将 clkmode 和 clkfreq 读入 x 和 x+1 RDLONG x,#@clkmode SETQ #2-1 将 x 和 x+1 写入 clkmode 和 clkfreq WRLONG x,#@clkmode

对于仅支持 PASM 的程序,有一条名为 ASMCLK 的特殊指令,用于设置时钟设置符号指定的时钟模式。ASMCLK 没有操作数,但是可以与条件前缀一起使用。ASMCLK 将汇编为一条或六条 PASM 指令,具体取决于时钟模式。

从 v35v 开始,在启动仅 PASM 程序时不再需要 ASMCLK,因为一个 16 位长的时钟设置程序会自动添加到仅 PASM 程序的前面,使用任何非 RCFAST（默认)时钟模式。此时钟设置程序会设置时钟模式,将 PASM 程序向下移动 16 个字节,然后通过 COGINIT 执行它。#0,#0,实现正常启动。

如果你不想将时钟设置程序添加到你的 PASM 专用程序中,你可以通过声明常量 _AUTOCLK = 0 来禁止它。然后,你的代码将开始使用默认的 RCFAST 模式执行。如果您想切换到其他时钟模式,则需要在代码中手动配置时钟,例如选择使用ASMCLK指令。

CON 声明 （例如数字可能会有所不同）	集线器 %CC_SS	ASMCLK 汇编为:	
CON _clkfreq = 250_000_000 _errfreq = 0	10_11		
CON _xtlfreq = 12_000_000 _clkfreq = 148_500_000 _errfreq = 150_000	1x_11	集线器 ##clkmode_ & !%11 WAITX ##20_000_000/100 集线器 ##clkmode_	启动外部时钟,保持在 RCFAST 模式 等待 10ms 让外部时钟稳定下来 切换到外部时钟模式
CON _xinfreq = 32_000_000 _clkfreq = 297_500_000 _errfreq = 100_000	01_11		
CON _xtlfreq = 16_000_000	1x_10		
CON _xinfreq = 100_000_000	01_10		
CON _rcslow	00_01	轮毂 #1	切换到 RCSLOW 模式
CON _rcfast	00_00	轮毂 #0	保持 RCFAST 模式