AN11538

SCTimer/PWM 操作手册

版本 5.0 — 2016 年 11 月 应用笔记

文档信息

信息	内容
关键字	LPC81x、LPC82x、LPC11U6x、LPC11E6x、LPC15xx、LPC54xxx、LPC5411x、LPC18S/43Sxx、LPC18/43xx 状态可配置定时器、SCT、SCTimer/PWM
摘要	此应用笔记汇集了针对恩智浦微控制器中所用的 SCTimer/PWM 数据块的示例和使用说明。

修订记录

版本	日期	说明
5.0	20160303	增加了对LPC5411x的支持。
4.0	20150707	增加了三个关于LPC81x的全新示例（WS2812 LED驱动器、带0-100%占空比的PWM和带0-100%占空比的双通道PWM）。
3.1	20150218	增加了对 LPC18S/43Sxx 的支持。
3.0	20141104	增加了对 LPC54xxx 的支持。
2.0	20140903	更新了 LPCOpen，增加了对 LPC82x 的支持。
1.0	20140821	初始版本。

联系信息

有关详细信息，请访问：http://www.nxp.com

欲咨询销售办事处地址，请发送电子邮件至：salesaddresses@nxp.com

1. 简介

1.1 概述

状态可配置定时器 (SCTimer/PWM) 是恩智浦半导体特有的一种外设。它可像大多数传统定时器一样工作，但也会添加状态机，从而提供更高水平的可配置性和控制度。这允许将 SCT 配置为多个 PWM、带死区控制的 PWM、带重置功能的 PWM 以及传统计时器无法复制的多种其他配置。除非使能了需要内核来服务的 SCTimer/PWM 中断，否则配置好 SCTimer/PWM 后，它就可以自主从微控制器内核运行。

下面的表 1 概述了包含 SCTimer/PWM 数据块（1 个或多个）的控制器系列，以及它们的合成方式（显示主要资源的可用数量，如输入、输出、状态等）。

表1. 各个系列的 SCTimer/PWM 资源

恩智浦芯片	输入	输出	状态	事件	匹配/捕获	SCTIPU	去抖	SCTPLL
LPC81x	4	4	2	6	5	☒	☒	☒
LPC82x	4	6	8	8	8	☒	☒	☒
LPC11U6x/E6x – SCT0/1	4	4	8	6	5	☒	☒	☒
LPC15xx – SCT0/1	8	10	16	16	16	☑	☑	☑
LPC15xx – SCT2/3	3	6	10	10	8	☒	☒	☒
LPC18/43xx（不带 flash）	8	16	32	16	16	☒	☒	☒
LPC18/43xx (flash)	8	16	32	16	16	☒	☑	☒
LPC18S/43Sxx（不带 flash）	8	16	32	16	16	☒	☒	☒
LPC18S/43Sxx（闪存）	8	16	32	16	16	☒	☑	☒
LPC5410x	8	8	13	13	13	☒	☒	☒
LPC5411x	8	8	10	10	10	☒	☒	☒

SCTimer/PWM 数据块的其他功能包括：

- 输入和输出可路由至外部引脚，并在内部路由至其他外设。
- 如果有更多 SCT 可用（如 LPC15xx），则 SCTimer/PWM 输出在内部连接至其他 SCTimer/PWM 输入。
- 每个 SCTimer/PWM 可用作 1 个 32 位计数器或拆分成 2 个 16 位计数器。
- 由总线时钟、选定输入或单独的 SCTPLL（在 LPC15xx SCT0/1 上）计时。
- 上行计数器或可逆计数器。
- 状态变量可以对多个计数器周期进行排序。
- 输入预处理器单元（LPC15xx 上）用于处理 SCTimer/PWM 输入和处理 SCTimer/PWM 中止。
- 以下条件定义了一次事件：计数器匹配条件、输入（或输出）条件，以及指定状态中匹配与/或输入/输出条件组合和计数方向。

- 事件控制输出、中断、DMA 请求和 SCTimer/PWM 状态。另外：
  - 匹配寄存器 0 可用作自动限值。
  - 在双向模式中，事件可以根据计数方向使能。
  - 可以保持匹配事件，直至发生另一个符合条件的事件。
  - 所选事件可以限制、终止、启动或停止计数器操作。

这本“操作手册”将介绍 SCT 的各种不同使用方式，但不包含这种特别外设的所有潜在应用。

每个示例显示 SCTimer/PWM 的已用资源、配置代码，除此之外，如果可用，还会配置任意使用开关矩阵的 SCTimer/PWM 输入或输出、SCTimer/PWM 输入多路输入和 SCTimer/PWM 输入处理单元。

1.2 术语

当您第一次看到 SCTimer/PWM 时，它可能像非常复杂的外设，但您会发现，它实际上也没那么难用，即便不使用 LPCXpresso IDE 中的 Red State 等设计工具也一样。先了解一下在本文中以及恩智浦用户手册中会看到的一些术语可能很有用，在讨论其他定时器时可能不会遇到这些术语。

限制 — 限制是导致计数器清零（单向模式）或者计数方向更改（双向模式）的条件或事件的别称。例如，如果发生定时器匹配，这会（但并非必须）限制计数器。您可以将限制条件想象成是某种定时器重置。SCTimer/PWM 限制寄存器可定义导致限制条件的事件。

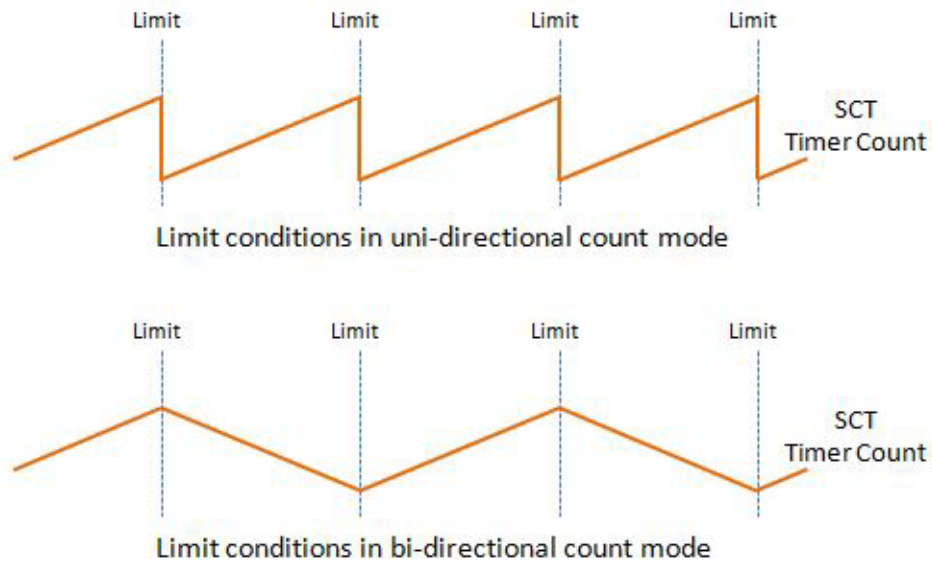


图1. 单向和双向计数模式中的定时器限制

事件 — 理解事件对于理解 SCTimer/PWM 非常关键。以下条件定义了潜在事件：计数器匹配条件、输入（或输出）条件，以及指定状态中匹配与/或输入/输出条件组合和计数方向。事件可以控制输出、中断、DMA 请求和 SCTimer/PWM 状态。它们还可导致发生定时器限制、暂停、启动或停止条件。

STOP — 当 SCTimer/PWM 定时器已停止时，计数器不会运行，但仍然会发生与计数器相关的 I/O 事件。如果发生在 START 寄存器中使能的事件，计数器将恢复运行。STOP 条件由 SCTimer/PWM 控制寄存器中的 STOP_L 和 STOP_H 位控制。STOP 位可以由事件或软件清除。

START — 如果 SCTimer/PWM 已停止，它可以通过事件重新启动。START 寄存器可确定启动定时器的事件。

HALT — HALT 与 STOP 相似，但是，不能通过事件重新启动定时器。因此，只能通过软件取消暂停定时器。如果您查看本文所含的示例代码，就会看到用户软件需要清除控制寄存器中的暂停条件以便开始计数。

统一定时器 — SCTimer/PWM 有 1 个 32 位计数器。此计数器可以配置为 1 个 32 位计数器（也称作“统一”计数器），或者可以用作 2 个 16 位计数器。

状态 — 状态变量是区分 SCTimer/PWM 与其他计数器/定时器/PWM 数据块的主要功能。仅在特定状态下才产生事件。而事件可以执行下列操作：

- 置位和清除输出
- 限制、停止和启动计数器
- 导致中断
- 修改状态变量

状态变量值完全在应用程序的控制之下。如果应用程序不使用状态，则状态变量值将保留为零，这是系统默认值。使用状态变量可以跟踪和控制任何所需操作序列中的相关计数器的多个周期。状态变量在逻辑上与代表 SCTimer/PWM 配置的状态机图相关。

1.3 目标硬件

大多数示例将使用 LPCXpresso V2 或 LPCXpresso MAX 板作为目标/测试硬件。关于这些板的原理图，请参考：

http://www.lpcware.com/LPCXpressoBoards

2. 重复中断

2.1 目的

SCT 可以执行大多数微控制器中采用的典型定时器所执行的相同简单功能。可以配置 SCTimer/PWM 中的定时器，以作为 2 个 16 位定时器或作为 1 个“统一”32 位定时器操作。此示例使用统一的 32 位定时器模式，每 10 毫秒生成一次 SCTimer/PWM 中断。SCTimer/PWM 中断处理（使用用户代码）将计算调用的次数，并在每隔 20 个中断周期或 200 毫秒切换一次通用 IO (LED)。

2.2 配置

此示例 (*SCTx_repetitive_irq*) 使用匹配寄存器 MATCH[0].U 来触发自动限制（重置）计数器并生成中断 (SCT_IRQ) 的 event0。

此示例仅使用 1 个匹配和 1 个事件（没有状态、没有输入、没有输出）。

```
void SCT_Init(void)
{
    LPC_SCT->CONFIG            = (1 << 0) | (1 << 17);     // unified 32-bit timer, auto limit

    LPC_SCT->MATCHREL[0].U     = SystemCoreClock/100;      // match 0 @ 100 Hz = 10 msec

    LPC_SCT->EVENT[0].STATE    = 0xFFFFFFFF;               // event 0 happens in all states
    LPC_SCT->EVENT[0].CTRL     = (1 << 12);                // match 0 condition only

    LPC_SCT->EVEN              = (1 << 0);                 // event 0 generates an interrupt

    NVIC_EnableIRQ(SCT_IRQn);                              // enable SCTimer/PWM interrupt

    LPC_SCT->CTRL_U           &= ~(1 << 2);                // unhalt by clearing bit 2 of the CTRL
}
```

图2. 代码 SCT_repetitive_irq

3. Blinky 匹配

3.1 目的

在此示例 (*SCTx_blinky_match*) 中，我们使用统一的 32 位定时器切换连接至控制器输出端口（例如连接至 LED）的 SCTx_OUT0。定时器状态将会每 100 毫秒更改一次。尽管不需要使用多个状态在 SCTx_OUT0 上创建切换输出，但该实例列举了使用两种状态的情况。

3.2 配置

- 已用的匹配：MATCH[0].U，100 毫秒。
- 已用的输出：SCTx_OUT0 连接至输出电平较低时（状态 0 期间）亮起的 LED。
- 已用的事件：事件 0 和 1。
- 已用的状态：状态 0 和 1。

下面的图 3 列举了我们使用 SCTimer/PWM 可获得的功能。红线显示 SCTimer/PWM 向上计数，直到达到匹配值，这时重新限制为 0。每个限制过后，应切换 SCTimer/PWM 输出。输出低电平时，状态应该为 0；输出高电平时，状态应该为 1。

您可以看到定时器 MATCH0 上发生 event0 和 event1。Event0 仅发生在 state0，并更改为 state1。Event1 仅发生在 state1 并将状态变回 0。统一定时器遇这两种事件都会限制（重置为 0），并且将置位或重置输出 SCTx_OUT0（SCTx 用于以下示例）。

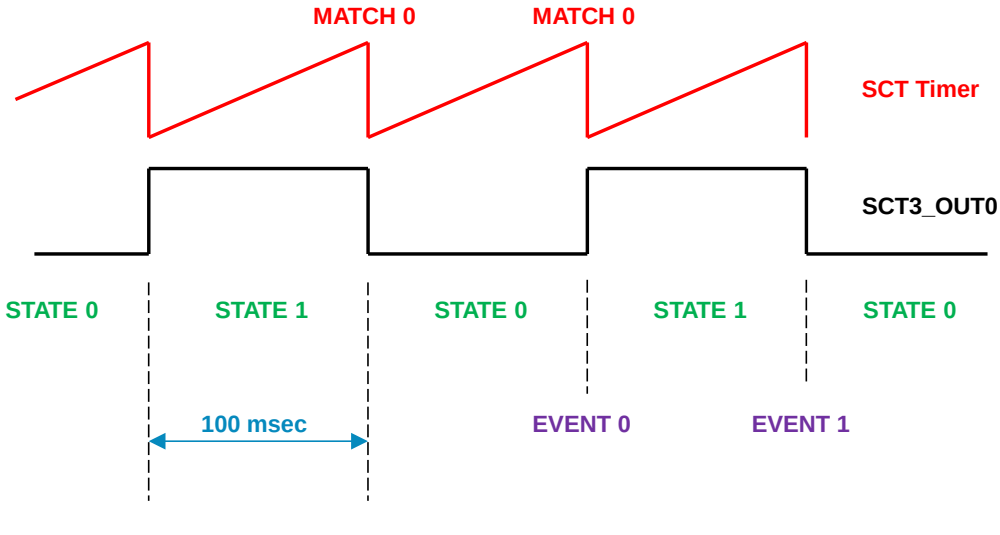


图3. Blinky 匹配说明

3.3 初始化代码

```
void SCT_Init(void)
{
    LPC_SCT->CONFIG           |= 1;                          // unified timer

    LPC_SCT->MATCHREL[0].U     = (SystemCoreClock/10)-1;     // match 0 @ 10 Hz = 100 msec

    LPC_SCT->EVENT[0].STATE    = (1 << 0);                   // event 0 only happens in state 0
    LPC_SCT->EVENT[0].CTRL     = (0 << 0)  |                 // related to match 0
                                 (1 << 12) |                 // COMBMODE[13:12] = match condition only
                                 (1 << 14) |                 // STATELD[14] = STATEV is loaded into state
                                 (1 << 15);                  // STATEV[15] = 1 (new state is 1)

    LPC_SCT->EVENT[1].STATE    = (1 << 1);                   // event 1 only happens in state 1
    LPC_SCT->EVENT[1].CTRL     = (0 << 0)  |                 // related to match 0
                                 (1 << 12) |                 // COMBMODE[13:12] = match condition only
                                 (1 << 14) |                 // STATELD[14] = STATEV is loaded into state
                                 (0 << 15);                  // STATEV[15] = 0 (new state is 0)

    LPC_SCT->OUT[0].SET        = (1 << 0);                   // event 0 will set   SCT_OUT0
    LPC_SCT->OUT[0].CLR        = (1 << 1);                   // event 1 will clear SCT_OUT0
    LPC_SCT->LIMIT_L           = 0x0003;                     // events 0 and 1 are used as counter limit

    LPC_SCT->CTRL_L           &= ~(1 << 2);                  // unhalt by clearing bit 2 of CTRL register
}
```

图4. 代码 SCT_blinky_match

备注：对于 LPC54xxx，使用 SCT_OUT[5]，因为 SCT_OUT[0] 未连接至 LPC54xxx LPCXPresso V2 板上的 LED。

4. 匹配切换

4.1 目的

SCTimer/PWM 可以使用 SET 和 CLR 寄存器直接置位或清除输出，但无法直接切换输出。我们将了解如何使用冲突解决寄存器来切换输出，并演示如何仅使用 1 个没有使用任何状态的事件来构建先前的示例。此外，本示例 (*SCTx_match_toggle*) 将仅使用较低的 16 位计数器，而非使用先前示例中所用的 32 位统一计数器。输出应每 100 毫秒切换 SCTx_OUT0 一次。

4.2 配置

- 已用的匹配：MATCH[0].L，100 毫秒。
- 已用的输出：每次发生事件 0，SCTx_OUT0 切换一次。
- 已用的事件：事件 0（仅在匹配 0 条件下触发）。
- 已用的状态：无。

MATCH[0].L 寄存器用于每 100 毫秒进行一次匹配。发生匹配时，定时器**自动**限制（重置）并生成事件 0。事件 0 使用冲突解决寄存器切换 SCTx_Out0。

使用 12 MHz 时钟和 1 个 16 位计数器的唯一问题就是最长延迟时间约有 5.5 毫秒。因此，我们需要使用 SCTimer/PWM 预分频器。每个 16 位定时器都有一个单独的 8 位预分频器。

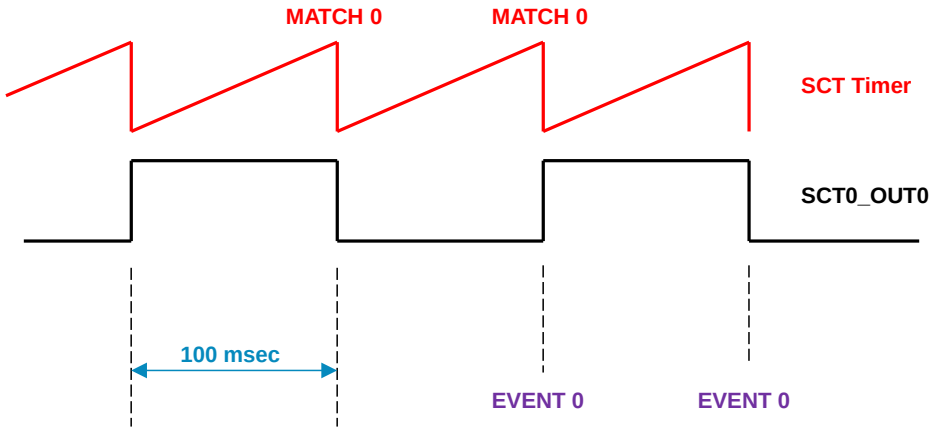


图5. 匹配切换图示（已使用 SCT0）

4.3 置位 SCTimer/PWM 预分频器

SCTimer/PWM 预分频器用于允许 16 位低位计数器实现 100 毫秒匹配间隔。为简化操作，我们通过将 12 MHz 主时钟作 120 分频，将 SCTimer/PWM 输入时钟置位为 100 kHz。

```
LPC_SCT->CTRL_L |= ((120 - 1) << 5); // set prescaler, SCTimer/PWM clock = 100 kHz
```

4.4 初始化代码

```
void SCT_Init(void)
{
    LPC_SCT->CONFIG        |= (1 << 17);     // two 16 bit timers, auto limit
    LPC_SCT->CTRL_L        |= (119 << 5);    // PRE_L[12:5] = 120-1 (SCTimer/PWM clock = 12MHz/120 = 100 kHz)

    LPC_SCT->MATCHREL[0].L  = (100000/10)-1; // match 0 @ 10 Hz = 100 msec

    LPC_SCT->EVENT[0].STATE = 0xFFFF;        // event 0 happens in all state
    LPC_SCT->EVENT[0].CTRL  = (1 << 12);     // match 0 condition only

    LPC_SCT->OUT[0].SET     = (1 << 0);      // event 0 will set   SCTx_OUT0
    LPC_SCT->OUT[0].CLR     = (1 << 0);      // event 0 will clear SCTx_OUT0
    LPC_SCT->RES            = (3 << 0);      // output 0 toggles on conflict

    LPC_SCT->CTRL_L        &= ~(1 << 2);     // start timer
```

图6. 代码 SCT_match_toggle

备注：对于 LPC54xxx，使用 SCT_OUT[5]，因为 SCT_OUT[0] 未连接至 LPC54xxx LPCXPresso V2 板上的 LED。

4.5 使用冲突解决寄存器

如图 6 所示，输出引脚 0 可通过事件 0 置位和清除。当事件置位并清除输出时，冲突解决寄存器可用于决定这种冲突将如何解决。

Bit	Symbol	Value	Description
1:0	O0RES		Effect of simultaneous set and clear on output 0.
		0x0	No change.
		0x1	Set output (or clear based on the SETCLR0 field).
		0x2	Clear output (or set based on the SETCLR0 field).
		0x3	Toggle output.

图7. 冲突解决寄存器

在 *SCTx_match_toggle* 示例中，冲突解决寄存器使用值 0x03 来指示 SCTimer/PWM 切换输出。

5. 使用 SCTPLL

5.1 目的

和一些其他器件相同，LPC15xx 拥有专用的内置 PLL，可创建 SCT0 与/或 SCT1 时钟。此示例 (*SCT1_use_PLL*) 使用 SCTPLL 针对 SCT1 生成 72 MHz 输入时钟，而系统时钟则为来自 IRC 的 12 MHz。SCTPLL 输入时钟拥有与 SCT1 输入 7 的固定连接。

5.2 配置

此代码基于先前的示例。其使用 SCT1，并使用以 12 MHz 运行的 LPC1549 在 LPCXpresso 板上进行测试。其使用 SCT1_IN7 接收来自 SCTPLL、统一定时器和 MATCH[0].U 寄存器的 72 MHz 时钟，以实现每 100 毫秒匹配一次。发生匹配时，定时器会自动限制（重置）并生成事件 0。事件 0 切换 SCT1 输出 0（连接至 P0_24 绿色 LED）。

5.3 置位 SCTPLL

上电并使能 SCTimer/PWM PLL 以 72 MHz 运行：

```
LPC_SYSCON->PDRUNCFG    &= ~PDEN_SCT_PLL;      // power-up SCTimer/PWM PLL
LPC_SYSCON->SCTPLLCLKSEL = 0;                  // select SCTimer/PWM PLL input = IRC
LPC_SYSCON->SCTPLLCTRL   = (5 << 0) |          // MSEL = 5 -> M = MSEL + 1 = 6
                           (0 << 6);           // PSEL = 0 -> P = 1
while (!(LPC_SYSCON->SCTPLLSTAT & 1));         // wait until SCTimer/PWM PLL locked
```

使用全局 CONFIG 寄存器以在输入 7 使用 SCTimer/PWM PLL：

```
LPC_SCT1->CONFIG |= (0x3 << 1) |     // CLKMODE = SCTimer/PWM clock is input selected by CLKSEL
                    (0xF << 3);      // CLKSEL  = falling edge of input 7 (SCTimer/PWM PLL)
```

5.4 初始化代码

```
void SCT1_Init(void)
{
    LPC_SYSCON->SYSAHBCLKCTRL1 |= EN1_SCT1;                // enable the SCT1 clock

    LPC_SCT1->CONFIG           |= (1   << 0)  |            // unified timer
                                  (0x3 << 1)  |            // SCTimer/PWM clock is input selected by CLKSEL
                                  (0xF << 3)  |            // falling edge of input 7 (SCTimer/PWM PLL)
                                  (1   << 17);             // auto limit

    LPC_SCT1->MATCH[0].U        = (72000000/10) -1;        // match 0 @ 10 Hz = 100 msec
    LPC_SCT1->MATCHREL[0].U     = (72000000/10) -1;

    LPC_SCT1->EVENT[0].STATE    = 0xFFFFFFFF;              // event 0 happens in all states
    LPC_SCT1->EVENT[0].CTRL     = (1 << 12);               // match 0 condition only

    LPC_SCT1->OUT[0].SET        = (1 << 0);                // event 0 will set SCT1_OUT0
    LPC_SCT1->OUT[0].CLR        = (1 << 0);                // event 0 will clear SCT1_OUT0
    LPC_SCT1->RES               = (3 << 0);                // output 0 toggles on conflict

    LPC_SCT1->CTRL_U           &= ~(1 << 2);               // start timer
}
```

图8. PWM（使用 PLL）初始化代码

6. 简单的 PWM

6.1 目的

本示例 (*SCTx_pwm*) 使用低电平 16 位 SCTimer/PWM 定时器在 SCTx_OUT0 时生成 100 kHz PWM 信号。两个按钮（LPCXpresso 电路板上的 SW1 和 SW2/SW3）用于通过更新 MATCHRELOAD 寄存器来减少和增加 PWM 信号的占空比。由于 LPC812 LPCXpresso 电路板上没有按钮，因此修整器 R38 用于减少（逆时针）和增加（顺时针）PWM 信号的占空比。通过将 SCTx_OUT0 连接至 LED，这可调整 LED 的亮度。

6.2 配置

- 已用的匹配：PWM 周期的匹配 0 和 PWM 占空比的匹配 1。
- 已用的输出：SCTx_OUT0 用于 PWM 输出信号。
- 已用的事件：事件 0 和 1。
- 已用的状态：无。

SCTimer/PWM 输入时钟已经预分频至 1 MHz。它使用 MATCH[0].L = 10 (1 MHz / 100 kHz) 生成 100 kHz 定时器匹配；这会自动限制（重置）计数器并生成事件 0。事件 0 随后会将 SCTimer/PWM 输出 0 置位为逻辑高电平。MATCH[0].L 寄存器用于定义 PWM 信号的周期长度。第二个匹配寄存器 MATCH[1].L 用于定义信号的占空比。发生匹配事件 1 时，它会清除 SCTimer/PWM 输出 0。图 9 显示此示例的波形。

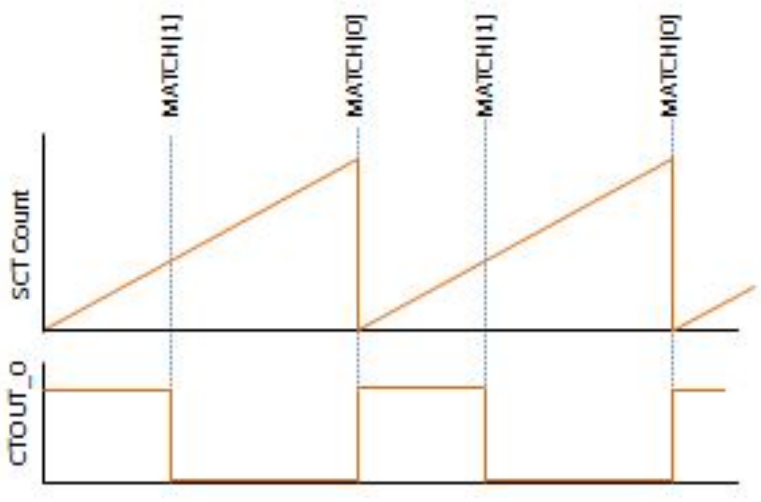


图9. PWM 计数器操作和输出

此应用代码使用（一个连接至 LPC812 LPCXpresso 上的修剪器 R38 的 GPIO 输入或两个连接至其他 LPCXpresso 电路板上的 SW1 和 SW2/SW3 的 GPIO 输入）来控制 PWM 输出信号的占空比。每当 SW1 从高电平变成低电平时（下降沿），都会增加占空比（10 步）

并降低 LED 亮度。每当 SW2/SW3 从高电平变成低电平时，都会减少占空比（10 步）并增加 LED 亮度。

注意： 在 LPC82x LPCXpresso 电路板上，开关 SW2 和红色 LED 连接至相同的端口引脚，因此当占空比减少或 LED 亮度增加时可能会看到多余的 LED 红灯闪烁。

6.3 配置代码

```
void SCT_Init(void)
{
    LPC_SCT->CONFIG        |= (1 << 17);              // two 16-bit timers, auto limit
    LPC_SCT->CTRL_L        |= (12-1) << 5;            // set prescaler, SCTimer/PWM clock = 1 MHz

    LPC_SCT->MATCHREL[0].L  = 10-1;                   // match 0 @ 10/1MHz = 10 usec (100 kHz PWM freq)
    LPC_SCT->MATCHREL[1].L  = 5;                      // match 1 used for duty cycle (in 10 steps)

    LPC_SCT->EVENT[0].STATE = 0xFFFFFFFF;             // event 0 happens in all states
    LPC_SCT->EVENT[0].CTRL  = (1 << 12);              // match 0 condition only

    LPC_SCT->EVENT[1].STATE = 0xFFFFFFFF;             // event 1 happens in all states
    LPC_SCT->EVENT[1].CTRL  = (1 << 0) | (1 << 12);   // match 1 condition only

    LPC_SCT->OUT[0].SET     = (1 << 0);               // event 0 will set   SCTx_OUT0
    LPC_SCT->OUT[0].CLR     = (1 << 1);               // event 1 will clear SCTx_OUT0

    LPC_SCT->CTRL_L        &= ~(1 << 2);              // unhalt it by clearing bit 2 of CTRL reg
}
```

图10. 简单的 PWM 配置代码

备注：对于 LPC54xxx，使用 SCT_OUT[5]，因为 SCT_OUT[0] 未连接至 LPC54xxx LPCXPresso V2 板上的 LED。

7. 居中对齐的 PWM

7.1 目的

SCTimer/PWM 能够向上计数至一个限值，然后向下计数至零。在这种情况下，您可以使用输出方向控制寄存器（为每个输出）指定计数方向如何影响置位和清除操作对该输出的含义。

此实例 (*SCTx_pwm_center_aligned*) 使用该功能生成一个居中对齐的 PWM 输出。这可演示如何仅使用一个事件来构建之前的示例。它会重新使用低 16 位 SCTimer/PWM 定时器在 SCTx_OUT0 时生成 100 kHz 的 PWM 信号。两个按钮（LPCXpresso 电路板上的 SW1 和 SW2/SW3）用于通过更新 MATCHRELOAD 寄存器来减少和增加 PWM 信号的占空比。由于 LPC812 LPCXpresso 电路板上没有按钮，因此修整器 R38 用于减少（逆时针）和增加（顺时针）PWM 信号的占空比。通过将 SCTx_OUT0 连接至 LED，这可调整 LED 的亮度。

7.2 配置

SCTimer/PWM 输入时钟已经预分频至 2 MHz。它使用 MATCH[0].L = 10 生成可更改自上向下计数的计数方向的定时器。因此，总的 PWM 周期为 20 个时钟，10 usec (100 kHz)。第二个匹配寄存器 MATCH[1].L 用于定义信号的占空比。发生事件 1 匹配时，它会在向上计数时置位 SCTimer/PWM 输出 0，并在向下计数时清除（反向）输出。图 11 显示此示例的波形。

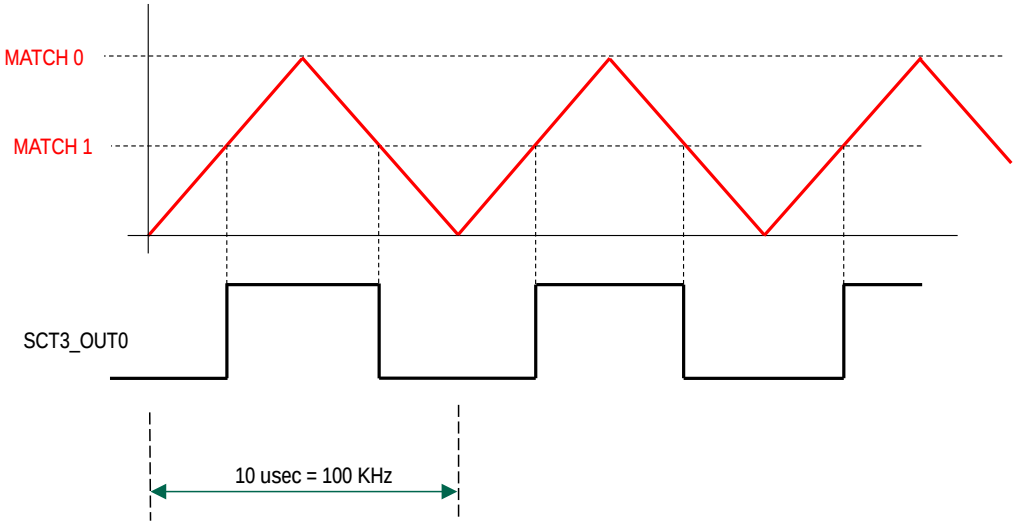


图11. 居中对齐的 PWM 计数器操作和输出

此应用代码使用（一个连接至 LPC812 LPCXpresso 上的修剪器 R38 的 GPIO 输入或两个连接至其他 LPCXpresso 电路板上的 SW1 和 SW2/SW3 的 GPIO 输入）来控制 PWM 输出信号的占空比。每当 SW1 从高电平变成低电平时（下降沿），都会增加占空比（10 步）

并降低 LED 亮度。每当 SW2/SW3 从高电平变成低电平时，都会减少占空比（10 步）并增加 LED 亮度。

7.3 配置代码

```
void SCT_Init(void)
{
    LPC_SCT->CONFIG        |= (1 << 17);                // two 16-bit timers, auto limit at match 0
    LPC_SCT->CTRL_L        |= (1 << 4) | (6-1) << 5;    // BIDIR mode, prescaler = 6, SCTimer/PWM clock = 2
MHz

    LPC_SCT->MATCHREL[0].L  = 10-1;                     // match 0 @ 10/2MHz = 5 usec (100 kHz PWM freq)
    LPC_SCT->MATCHREL[1].L  = 5;                        // match 1 used for duty cycle (in 10 steps)

    LPC_SCT->EVENT[0].STATE = 0xFFFFFFFF;               // event 0 happens in all states
    LPC_SCT->EVENT[0].CTRL  = (1 << 0) | (1 << 12);     // match 1 condition only

    LPC_SCT->OUT[0].SET     = (1 << 0);                 // event 0 will set SCTx_OUT0
    LPC_SCT->OUTPUTDIRCTRL  = (0x1 << 0);               // reverse output 0 set when down counting

    LPC_SCT->CTRL_L        &= ~(1 << 2);                // unhalt it by clearing bit 2 of CTRL reg
}
```

图12. 居中对齐的 PWM 初始代码

备注：对于 LPC54xxx，使用 SCT_OUT[4]，因为 SCT_OUT[0] 未连接至 LPC54xxx LPCXPresso V2 板上的 LED。

7.4 使用双向输出控制

在图 11 中，您将看到该定时器向上和向下计数。向上计数时，必须在 MATCH1 上置位输出引脚 0，但在向下计数时，必须在相同匹配和事件中重置输出引脚 0。这可使用双向输出控制寄存器来完成。

Table 187. SCT bidirectional output control register (OUTPUTDIRCTRL, address 0x1C01 8054) bit description

Bit	Symbol	Value	Description	Reset value
1:0	SETCLR0		Set/clear operation on output 0. Value 0x3 is reserved. Do not program this value.	0
		0x0	Set and clear do not depend on any counter.	
		0x1	Set and clear are reversed when counter L or the unified counter is counting down.	
		0x2	Set and clear are reversed when counter H is counting down. Do not use if UNIFY = 1.	
3:2	SETCLR1		Set/clear operation on output 1. Value 0x3 is reserved. Do not program this value.	0
		0x0	Set and clear do not depend on any counter.	

图13. 冲突解决寄存器

在此示例中，OUTPUTDIRCTRL 寄存器使用值 0x1 指示 SCTimer/PWM 在向下计数时反转输出 0 置位和清除。

8. 双通道 PWM

8.1 目的

此示例 (*SCT_pwm_2ch*) 显示使用不同占空比生成两个 PWM 信号。它使用统一的 32 位计数器模式。分配至 SCTimer/PWM 输入 0 (SCT_IN0) 的 GPIO 输入可选择有效哪个输出信号。LPC812 LPCXpresso 电路板上的修剪器 (R38) 或其他 LPCXpresso 电路板上的开关 SW1 用于在绿色和红色/蓝色闪烁 LED 之间进行选择。在一些硬件电路板（如 LPC11U6x LPCXpresso）中，GPIO (SW1) 输入不能用于 SCT_IN0，输出信号有效（红色/蓝色和绿色 LED 闪烁为时间多路复用）。最初红色/蓝色 LED 闪烁几秒，然后变成绿色。

此示例最初是使用图形 Red State 工具（参见图 14）构建的。它使用 ALWAYS 状态（U_ALWAYS 用于统一计数器，L_ALWAYS、H_ALWAYS 用于 16 位置位）。ALWAYS 状态不作为 SCTimer/PWM 的其中一个状态，因此您仍然可以将所有状态用于每个拆分定时器。ALWAYS 是一个可能发生在任意状态的条件。

SCTimer/PWM 配置寄存器中选定的自动限值允许匹配寄存器 0 引发限制条件。绿色 LED 以较短占空比闪烁，而红色/蓝色 LED 则以较长占空比闪烁。

8.2 配置

- 已用的输入：SCT_IN0（SW1 或 R38）。
- 已用的输出：SCTx_OUT0（绿色 LED）和 SCTx_OUT1 个（红色/蓝色 LED）。
- 已用的匹配：匹配 0 至 4。
- 已用的事件：事件 0 至 5。
- 已用的状态：状态 0 和 1。

8.3 Red State 图

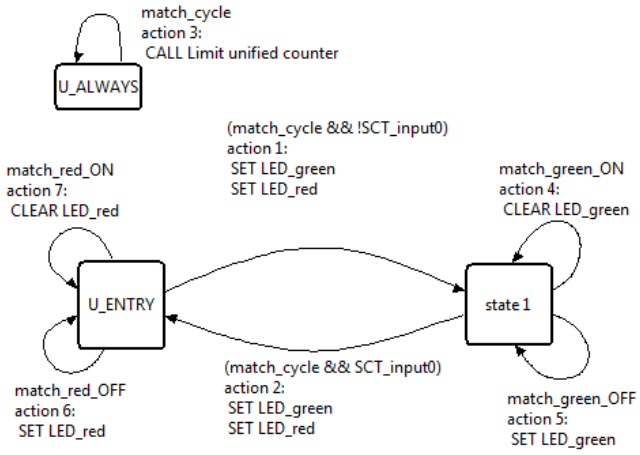


图14. 针对 SCT_pwm_2ch 的 Red State 图

8.4 初始化代码

参见图 15。此代码最初由 Red State 工具生成，随后被清理并重建。

```
void SCT_Init(void)
{
    LPC_SCT->CONFIG           |= (1 << 0) | (1 << 17);     // unified, auto limit

    LPC_SCT->MATCHREL[0].U  = delay;                      // match_cycle
    LPC_SCT->MATCHREL[1].U  = match_green_OFF;            // match_green_OFF
    LPC_SCT->MATCHREL[2].U  = match_green_ON;             // match_green_ON
    LPC_SCT->MATCHREL[3].U  = match_red_OFF;              // match_red_OFF
    LPC_SCT->MATCHREL[4].U  = match_red_ON;               // match_red_ON

    LPC_SCT->EVENT[0].STATE = (1 << 0);                   // event 0 happens in state 0 (U_ENTRY)
    LPC_SCT->EVENT[0].CTRL  = (0 << 0)  |                 // related to match_cycle
                              (0 << 10) |                 // IN_0 low
                              (3 << 12) |                 // match AND IO condition
                              (1 << 14) |                 // STATEV is loaded into state
                              (1 << 15);                  // new state is 1

    LPC_SCT->EVENT[1].STATE = (1 << 0);                   // event 1 happens in state 0 (U_ENTRY)
    LPC_SCT->EVENT[1].CTRL  = (3 << 0) | (1 << 12);       // match_red_OFF only condition

    LPC_SCT->EVENT[2].STATE = (1 << 0);                   // event 2 happens in state 0 (U_ENTRY)
    LPC_SCT->EVENT[2].CTRL  = (4 << 0) | (1 << 12);       // match_red_ON only condition

    LPC_SCT->EVENT[3].STATE = (1 << 1);                   // event 3 happens in state 1
    LPC_SCT->EVENT[3].CTRL  = (0 << 0)  |                 // related to match_cycle
                              (3 << 10) |                 // IN_0 high
                              (3 << 12) |                 // match AND IO condition
                              (1 << 14) |                 // STATEV is loaded into state
                              (0 << 15);                  // new state is 0

    LPC_SCT->EVENT[4].STATE = (1 << 1);                   // event 4 happens in state 1
    LPC_SCT->EVENT[4].CTRL  = (2 << 0) | (1 << 12);       // match_green_ON only condition

    LPC_SCT->EVENT[5].STATE = (1 << 1);                   // event 5 happens in state 1
    LPC_SCT->EVENT[5].CTRL  = (1 << 0) | (1 << 12);       // match_green_OFF only condition

    LPC_SCT->OUT[0].SET = (1 << 0) | (1 << 3) | (1 << 5); // event 0, 3 and 5 set OUT0 (green LED)
    LPC_SCT->OUT[0].CLR = (1 << 4);                       // event 4 clear OUT0 (green LED)
    LPC_SCT->OUT[1].SET = (1 << 0) | (1 << 1) | (1 << 3); // event 0, 1 and 3 set OUT1 (red LED)
    LPC_SCT->OUT[1].CLR = (1 << 2);                       // event 2 clear OUT1 (red LED)
    LPC_SCT->OUTPUT    |= 3;                              // default set OUT0 and OUT1

    LPC_SCT->CTRL_U       &= ~(1 << 2);                   // start timer
}
```

图15. Red State 工具生成的代码的清理版本

备注：对于 LPC54xxx，使用 SCT_OUT[4] 和 SCT_OUT[5]，因为 SCT_OUT[0] 和 SCT_OUT[1] 未连接至 LPC54xxx LPCXPresso V2 板上的 LED。

9. 带有死区的 PWM

9.1 目的

此示例 (*SCTx_pwm_deadtime*) 显示两条通道双边控制的 PWM 生成，旨在用作带死区控制的互补 PWM 对。它使用分离 16 位定时器模式（低电平计数器）。高电平计数器可用于生成其他带死区控制的互补 PWM 对，可能是与第一对相对的相移或者其他目的。中止输入也已经在 SCT_IN0 上实现（LPC15xx 示例使用 SCTimer/PWM 输入处理单元）。中止输入可驱动输出进入其关断状态（Out0 = HIGH，Out1 = LOW）。

当 SCTimer/PWM 检测到 ABORT 引脚 (SCT_IN0) 上的下降沿时，会调用 SCTimer/PWM 中断来重置计数器（参见 main.c 中的 SCT_IRQHandler），并清除 STOP 条件。ABORT 由连接至一些 LPCXpresso 板 (LPC15xx) 上的 SW1 的 GPIO 引脚生成或者在电路板硬件不支持时在内部生成。

9.2 配置

- 已用的输入：SCT_IN0 用作 ABORT（LPC15xx 的情况下来自 SCTIPU）。
- 已用的输出：SCT_OUT0（PWM1 蓝色 LED）和 SCT_OUT1（PWM2 红色/蓝色 LED）。
- 已用的匹配：匹配 0 至 2。
- 已用的事件：事件 0 至 3。
- 已用的状态：无。

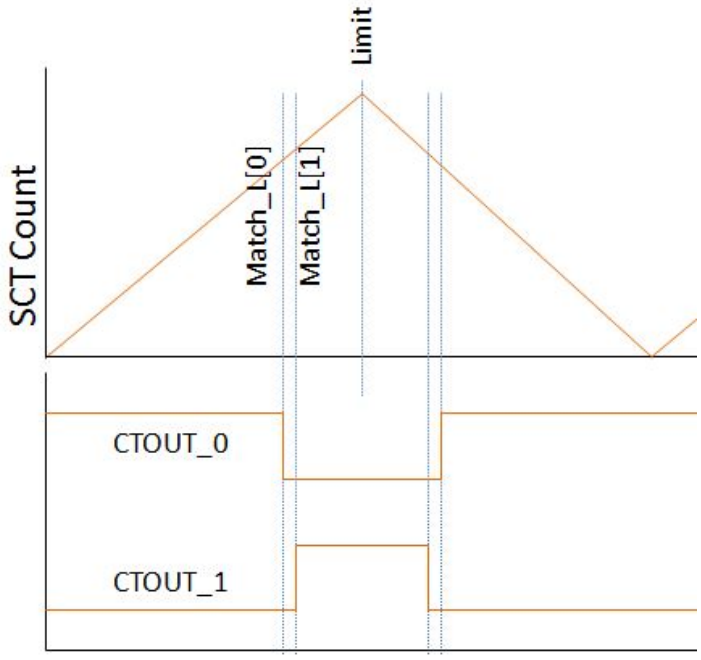


图16. 使用双向计数器且带死区的 PWM

9.3 LPC15xx 输入处理单元

要将 P1_9 配置为指向 SCT1_IN0 的 ABORT 输入，我们使用以下代码。注意：我们可以使用开关矩阵将任意 GPIO 端口引脚分配为 ABORT 引脚。

```
LPC_SWM->PINASSIGN10        |= 0x0000FF00;       // ASSIGN10(15:8) = FF
LPC_SWM->PINASSIGN10        &= 0xFFFF29FF;       // P1.9 (SW2) = SCT_ABORT0
LPC_SCT_IPU->ABORT[1].ENABLE = 1;                // enable SCT_ABORT0 from SWM
LPC_PMUX->SCT1_P_MUX0        = 17;               // SCT1_IN0 = SCTIPU_ABORT = P1.9 (SW2)
```

9.4 初始化代码

图 17 显示不使用状态、使用四个事件和三个匹配/匹配重新加载寄存器的 SCTimer/PWM 初始化代码。

备注：对于 LPC54xxx，使用 SCT_OUT[4] 和 SCT_OUT[5]，因为 SCT_OUT[0] 和 SCT_OUT[1] 未连接至 LPC54xxx LPCXPresso V2 板上的 LED。

```
#define DC1        (130)                                  // duty cycle 1
#define DC2        (135)                                  // duty cycle 2
#define hperiod    (180)

void SCT_Init(void)
{
    LPC_SCT->CONFIG           |= (1 << 17);               // split timers, auto limit
    LPC_SCT->CTRL_L           |= (1 << 4);                // configure SCT1 as BIDIR

    LPC_SCT->MATCH[0].L        = hperiod;                 // match on (half) PWM period
    LPC_SCT->MATCHREL[0].L     = hperiod;
    LPC_SCT->MATCH[1].L        = DC1;                     // match on duty cycle 1
    LPC_SCT->MATCHREL[1].L     = DC1;
    LPC_SCT->MATCH[2].L        = DC2;                     // match on duty cycle 2
    LPC_SCT->MATCHREL[2].L     = DC2;

    LPC_SCT->EVENT[0].STATE    = 0xFFFFFFFF;              // event 0 happens in all states
    LPC_SCT->EVENT[0].CTRL     = (2 << 10) | (2 << 12);   // IN_0 falling edge only condition

    LPC_SCT->EVENT[1].STATE    = 0xFFFFFFFF;              // event 1 happens in all states
    LPC_SCT->EVENT[1].CTRL     = (1 << 10) | (2 << 12);   // IN_0 rising edge only condition

    LPC_SCT->EVENT[2].STATE    = 0xFFFFFFFF;              // event 2 happens in all states
    LPC_SCT->EVENT[2].CTRL     = (1 << 0) | (1 << 12);    // match 1 (DC1) only condition

    LPC_SCT->EVENT[3].STATE    = 0xFFFFFFFF;              // event 3 happens in all states
    LPC_SCT->EVENT[3].CTRL     = (2 << 0) | (1 << 12);    // match 2 (DC) only condition

    LPC_SCT->OUT[0].SET        = (1 << 0) | (1 << 2);     // event 0 and 2 set OUT0 (blue LED)
    LPC_SCT->OUT[0].CLR        = (1 << 2);                // event 2 clears OUT0 (blue LED)
    LPC_SCT->OUT[1].SET        = (1 << 3);                // event 3 sets OUT1 (red LED)
    LPC_SCT->OUT[1].CLR        = (1 << 0) | (1 << 3);     // event 0 and 3 clear OUT1 (red LED)
    LPC_SCT->RES              |= 0x0000000F;              // toggle OUT0 and OUT1 on conflict
    LPC_SCT->OUTPUT           |= 1;                       // default set OUT0 and clear OUT1

    LPC_SCT->STOP_L            = (1 << 0);                // event 0 will stop the timer
    LPC_SCT->EVEN              = (1 << 1);                // event 1 will generate an irq

    NVIC_EnableIRQ(SCT_IRQn);                             // enable SCTx interrupt

    LPC_SCT->CTRL_L           &= ~(1 << 2);               // start timer
}
```

图17. 带死区初始化代码的 PWM

9.5 调整占空比

可以通过两个稍有不同的占空比置位死区。可通过以下步骤更新占空比：

临时禁用更新低电平计数器的匹配寄存器（寄存器 CONFIG 中的置位位 NORELOAD_L）。

加载匹配寄存器的新值

重新启用更新低电平计数器的匹配寄存器（寄存器 CONFIG 中的清除位 NORELOAD_L）。

9.6 结果

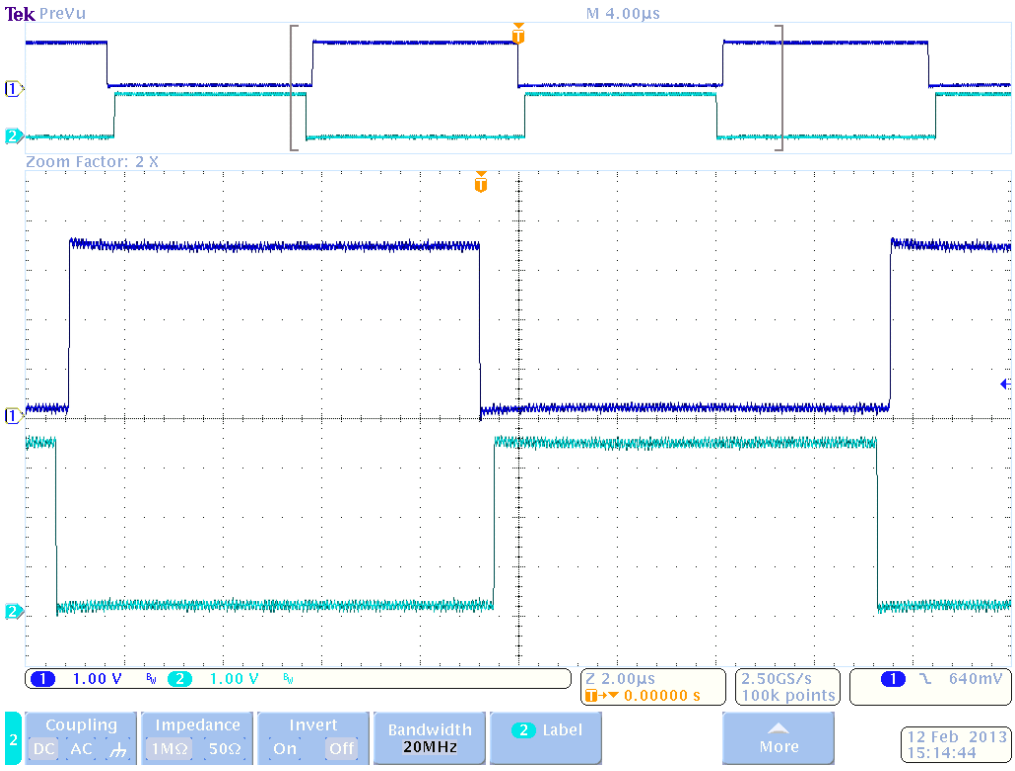


图18. 带死区控制的双通道 PWM：一条通道变成低电平和另一条通道变为高电平之间存在延迟

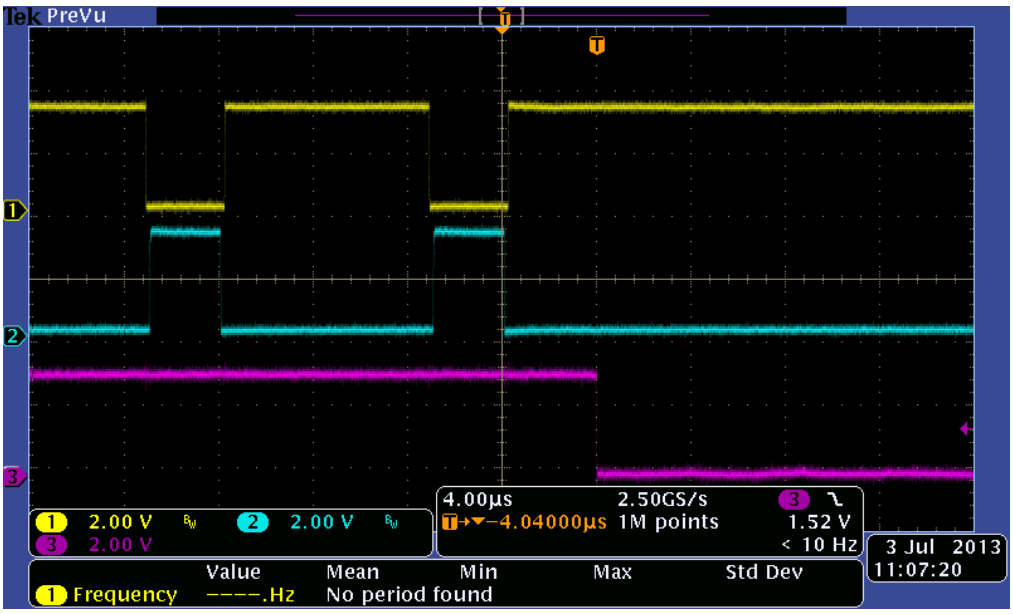


图19. ABORT 输入变成低电平并停止 PWM 输出

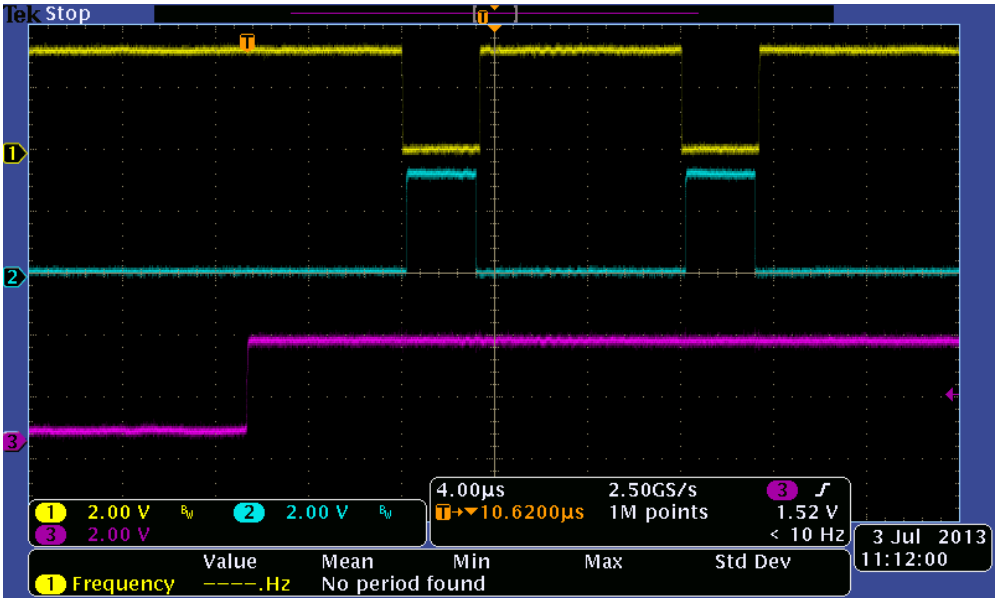


图20. ABORT 引脚变成高电平并重新启动 PWM

10. 匹配重新加载

10.1 目的

在前面的示例中，使用 SCTimer/PWM 置位带死区间隔的 PWM（仅限 L 计数器）。现在，我们将演示如何通过匹配重新加载寄存器使用 SCTimer/PWM 配置寄存器中的 NORELOAD_L 位更改两个 PWM 信号的占空比，并保持它们的死区间隔。

10.2 配置

此示例 (*SCTx_pwm_reload*) 正在使用 SysTick 定时器每隔 20 毫秒生成一次周期性中断。在 SysTick 中断处理器中更改了匹配重新加载值。

应用代码正在使用 GPIO 输入 (SW1/SW3/R038) 来控制 PWM 输出信号的占空比。只要输入处于高电平，就会增加占空比（每隔 20 毫秒一次），当输入处于低电平时，会减少占空比。

- 已用的输出：SCT_OUT1（PWM1 红色 LED）和 SCT_OUT0（PWM0 蓝色/绿色 LED）。

10.3 初始化代码

除非未置位 ABORT 输入，否则初始化代码与之前的 (SCTx_pwm_deadtime) 示例完全相同。

10.4 更新重新加载值

更新重新加载寄存器发生在 SysTick 定时器中断时。中断可配置为每 20 毫秒生成一次。置位 SCTimer/PWM 配置寄存器中的 NORELOAD_L 位可停止更新匹配寄存器。这允许我们更新 MATCHREL[1].L 和 MATCHREL[2].L，但在将 NORELOAD_L 位重置为“0”之前，MATCH[1] 和 MATCH[2] 寄存器不会更新为新值。图 21 显示完整的 SysTick 定时器中断代码。

```
void SysTick_Handler(void)
{
    LPC_SCT->CONFIG |= (1 << 7);                    // stop reload process for L counter

    if (LPC_GPIO->PIN[2] & (1 << 5))                // P2_5 high?
    {
        if (LPC_SCT->MATCHREL[2].L < hperiod-1)     // check if DC2 < Period of PWM
        {
            LPC_SCT->MATCHREL[1].L ++;
            LPC_SCT->MATCHREL[2].L ++;
        }
    }
    else if (LPC_SCT->MATCHREL[1].L > 1)            // check if DC1 > 1
    {
        LPC_SCT->MATCHREL[1].L --;
        LPC_SCT->MATCHREL[2].L --;
    }
    LPC_SCT->CONFIG &= ~(1 << 7);                   // enable reload process for L counter
}
```

图21. 用于重新加载匹配值的 SysTick 处理程序代码

11. 4 通道 PWM

11.1 目的

此示例 (SCTx_pwm_4ch) 演示简单的四通道 PWM 生成。它使用统一的 32 位定时器模式来生成单边对齐的输出。各通道可拥有不同的极性。演示状态机已针对 SCT_OUT0/1 时的正脉冲和 SCT_OUT2/3 时的负脉冲进行配置。

SCT_IN0（在 LPC15xx 的情况下，来自 SCTIPU 并分配至 P2_5）用作中止输入。如果是低电平，会强制输出进入闲置状态、暂停定时器并生成中断。对于一些硬件电路板，ABORT 输入是在内部生成的。

11.2 配置

- 已用的输入：SCT_IN0 (!ABORT)。
- 已用的输出：SCT_OUT0（绿色轨迹 PWM1）和 SCT_OUT1（红色轨迹 PWM2）。
- SCT_OUT2（黄色轨迹 PWM3）和 SCT_OUT3（蓝色轨迹 PWM4）。
- 已用的匹配：匹配 0 至 4。
- 已用的事件：事件 0 至 5。
- 已用的状态：无。

11.3 设计

图 22 显示此示例的 Red State 图。但是，该工具没有用于生成 SCTimer/PWM 代码，但仅用作参考。

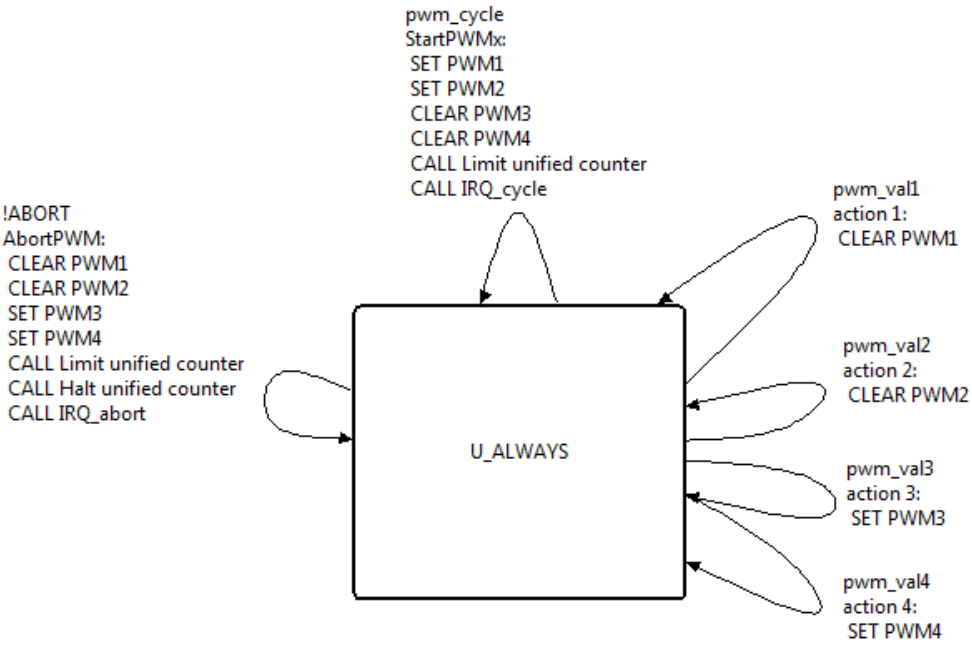


图22. 状态图 (Red State) 4 通道 PWM 生成

11.4 初始化代码

```
#define pwm_val1        (400000)                        // duty cycle PWM1
#define pwm_val2        (500000)                        // duty cycle PWM2
#define pwm_val3        (100000)                        // duty cycle PWM3
#define pwm_val4        (900000)                        // duty cycle PWM4
#define pwm_cycle       (1000000)

void SCT_Init(void)
{
    LPC_SCT->CONFIG           |= (1 << 0) | (1 << 17);     // unified timer, auto limit

    LPC_SCT->MATCH[0].U        = pwm_cycle;               // match 0 on PWM cycle
    LPC_SCT->MATCHREL[0].U     = pwm_cycle;
    LPC_SCT->MATCH[1].U        = pwm_val1;                // match 1 on val1 (PWM1)
    LPC_SCT->MATCHREL[1].U     = pwm_val1;
    LPC_SCT->MATCH[2].U        = pwm_val2;                // match 2 on val2 (PWM2)
    LPC_SCT->MATCHREL[2].U     = pwm_val2;
    LPC_SCT->MATCH[3].U        = pwm_val3;                // match 3 on val3 (PWM3)
    LPC_SCT->MATCHREL[3].U     = pwm_val3;
    LPC_SCT->MATCH[4].U        = pwm_val4;                // match 4 on val4 (PWM4)
    LPC_SCT->MATCHREL[4].U     = pwm_val4;

    LPC_SCT->EVENT[0].STATE    = 0xFFFFFFFF;              // event 0 happens in all states
    LPC_SCT->EVENT[0].CTRL     = (0 << 0) | (1 << 12);    // match 0 (pwm_cycle) only condition

    LPC_SCT->EVENT[1].STATE    = 0xFFFFFFFF;              // event 1 happens in all states
    LPC_SCT->EVENT[1].CTRL     = (1 << 0) | (1 << 12);    // match 1 (pwm_val1) only condition

    LPC_SCT->EVENT[2].STATE    = 0xFFFFFFFF;              // event 2 happens in all states
    LPC_SCT->EVENT[2].CTRL     = (2 << 0) | (1 << 12);    // match 2 (pwm_val2) only condition

    LPC_SCT->EVENT[3].STATE    = 0xFFFFFFFF;              // event 3 happens in all states
    LPC_SCT->EVENT[3].CTRL     = (3 << 0) | (1 << 12);    // match 3 (pwm_val3) only condition

    LPC_SCT->EVENT[4].STATE    = 0xFFFFFFFF;              // event 4 happens in all states
    LPC_SCT->EVENT[4].CTRL     = (4 << 0) | (1 << 12);    // match 4 (pwm_val4) only condition

    LPC_SCT->EVENT[5].STATE    = 0xFFFFFFFF;              // event 5 happens in all states
    LPC_SCT->EVENT[5].CTRL     = (0 << 10) | (2 << 12);   // IN_0 LOW only condition

    LPC_SCT->OUT[0].SET        = (1 << 0);                // event 0       sets  OUT0 (PWM1)
    LPC_SCT->OUT[0].CLR        = (1 << 1) | (1 << 5);     // event 1 and 5 clear OUT0 (PWM1)
    LPC_SCT->OUT[1].SET        = (1 << 0);                // event 0       sets  OUT1 (PWM2)
    LPC_SCT->OUT[1].CLR        = (1 << 2) | (1 << 5);     // event 2 and 5 clear OUT1 (PWM2)
    LPC_SCT->OUT[2].SET        = (1 << 3) | (1 << 5);     // event 3 and 5 set   OUT2 (PWM3)
    LPC_SCT->OUT[2].CLR        = (1 << 0);                // event 0       clear OUT2 (PWM3)
    LPC_SCT->OUT[3].SET        = (1 << 4) | (1 << 5);     // event 4 and 5 set   OUT3 (PWM4)
    LPC_SCT->OUT[3].CLR        = (1 << 0);                // event 0       clear OUT3 (PWM4)
    LPC_SCT->OUTPUT            = 0x0000000C;              // default clear OUT0/1 and set OUT2/3
    LPC_SCT->RES               = 0x0000005A;              // conflict: Inactive state takes precedence
                                                          // SCT2_OUT0/1: Inactive state low
                                                          // SCT2_OUT2/3: Inactive state high


    LPC_SCT->HALT_L            = (1 << 5);                // event 5 will halt the timer
    LPC_SCT->LIMIT_L           = (1 << 5);                // event 5 will limit the timer
    LPC_SCT->EVEN              = (1 << 0 ) | (1 << 5);    // event 0 and 5 will generate an irq

    NVIC_EnableIRQ(SCT_IRQn);                             // enable SCTimer/PWM interrupt

    LPC_SCT->CTRL_L           &= ~(1 << 2);               // start timer
}
```

图23. 4 通道 PWM 初始化代码

11.5 结果

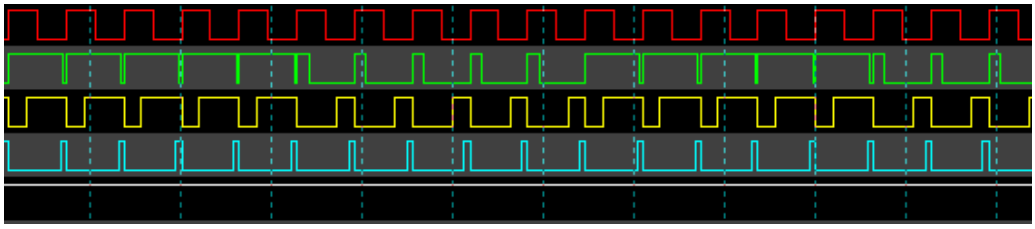


图24. 4 通道 PWM：两个通道的占空比每隔 5 个 PWM 周期更改一次

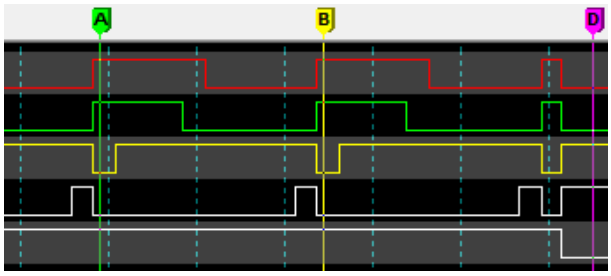
SCT_OUT1（红色轨迹）50 %

SCT_OUT0（绿色轨迹）40 %

SCT_OUT2（黄色轨迹）10 %

SCT_OUT3（蓝色轨迹）90 %

ABORT（白色）



游标位置 A 和 B 标记两个连续 PWM 周期的早期阶段。

游标位置 D 标记中止状态。注意：SCT_OUT1 和 SCT_OUT0 的闲置状态为低电平，而 SCT_OUT2 和 SCT_OUT3 的闲置状态为高电平。

图25. 4 通道 PWM：两个通道的占空比每隔 5 个 PWM 周期更改一次

12. 解码 PWM

12.1 目的

此示例 (*SCTx_pwm_decode*) 使用捕获和捕获控制功能。它实现 PWM 解码器来测量 PWM 信号的占空比并决定是高于 (max_width) 还是低于 (min_width) 某个特定值。PWM 信号频率假定为 10 kHz。包含两个输出信号（width_error 和超时）来指示 10 kHz 信号是否有误或缺失。

12.2 配置

- 已用的输入：SCT_IN0（在这里应用 10 kHz PWM 信号）。
- 已用的输出：
  - SCT_OUT0，超时指示器，低电平有效。如果没有检测到三个 PWM 周期的边缘，则有效输出超时。
  - SCT_OUT1，占空比溢出的指示器，低电平有效。如果发生超时，也会有效此输出。
- 已用的匹配/捕获：匹配 0 至 2 和捕获 3 和 4。
- 已用的事件：事件 0 至 5。
- 已用的状态：状态 0 和 1。

12.3 设计

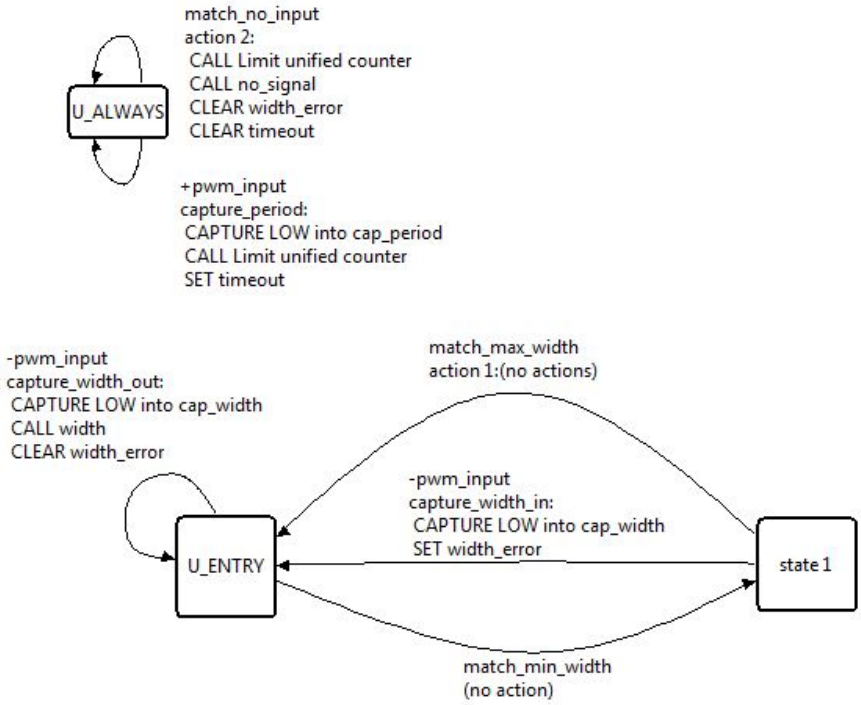


图26. 解码 PWM 的状态图

12.4 初始化代码

```
#define PWM_FREQUENCY          10000                    // PWM frequency in Hz
#define PWM_RESOLUTION_NS      1000                     // Timer resolution in ns
#define PWM_MIN_DUTY_PERCENT   25                       // Minimum allowed duty cycle in %
#define PWM_MAX_DUTY_PERCENT   70                       // Maximum allowed duty cycle in %

#define SCT_PRESCALER          (((SystemCoreClock / 1000u) * PWM_RESOLUTION_NS) / 1000000u - 1u)
#define match_min_width        ((10000000u * PWM_MIN_DUTY_PERCENT) / (PWM_FREQUENCY * PWM_RESOLUTION_NS))
#define match_max_width        ((10000000u * PWM_MAX_DUTY_PERCENT) / (PWM_FREQUENCY * PWM_RESOLUTION_NS))
#define match_no_input         ((10000000u * 300                ) / (PWM_FREQUENCY * PWM_RESOLUTION_NS))

void SCT_Init(void)
{
    LPC_SCT->CONFIG           |= (1 << 0) | (1 << 17);       // unified, auto limit

    LPC_SCT->CTRL_U           |= (SCT_PRESCALER << 5);       // set prescaler
    LPC_SCT->REGMODE_L         = 0x00000018;                 // 3x MATCH, 2x CAPTURE used

    LPC_SCT->MATCH[0].U        = match_max_width;            // match_max_width
    LPC_SCT->MATCHREL[0].U     = match_max_width;
    LPC_SCT->MATCH[1].U        = match_min_width;            // match_min_width
    LPC_SCT->MATCHREL[1].U     = match_min_width;
    LPC_SCT->MATCH[2].U        = match_no_input;             // match_no_input
    LPC_SCT->MATCHREL[2].U     = match_no_input;

    LPC_SCT->EVENT[0].STATE    = 0xFFFFFFFF;                 // event 0 happens in all states
    LPC_SCT->EVENT[0].CTRL     = (2 << 0)  | (1 << 12);      // related to match_no_input only
    LPC_SCT->EVENT[1].STATE    = 0xFFFFFFFF;                 // event 1 happens in all states
    LPC_SCT->EVENT[1].CTRL     = (1 << 10) | (2 << 12);      // IN_0 rising edge condition only
    LPC_SCT->EVENT[2].STATE    = (1 << 0);                   // event 2 happens in state 0
    LPC_SCT->EVENT[2].CTRL     = (1 << 0)  |                 // related to match_min_width
                                 (1 << 12) |                 // match condition only
                                 (1 << 14) |                 // STATEV is loaded into state
                                 (1 << 15);                  // new state is 1
    LPC_SCT->EVENT[3].STATE    = (1 << 1);                   // event 3 happens in state 1
    LPC_SCT->EVENT[3].CTRL     = (2 << 10) |                 // IN_0 falling edge
                                 (2 << 12) |                 // IO condition only
                                 (1 << 14) |                 // STATEV is loaded into state
                                 (0 << 15);                  // new state is 0
    LPC_SCT->EVENT[4].STATE    = (1 << 1);                   // event 4 happens in state 1
    LPC_SCT->EVENT[4].CTRL     = (0 << 0)  |                 // related to match_max_width
                                 (1 << 12) |                 // match condition only
                                 (1 << 14) |                 // STATEV is loaded into state
                                 (0 << 15);                  // new state is 0
    LPC_SCT->EVENT[5].STATE    = (1 << 0);                   // event 5 happens in state 0
    LPC_SCT->EVENT[5].CTRL     = (2 << 10) | (2 << 12);      // IN_0 falling edge condition only

    LPC_SCT->CAPCTRL[3].U      = (1 << 1);                   // event 1 is causing capture 3
    LPC_SCT->CAPCTRL[4].U      = (1 << 3) | (1 << 5);        // event 3 and 5 cause capture 4
    LPC_SCT->OUT[0].SET        = (1 << 1);                   // event 1 set   OUT0 (no timeout)
    LPC_SCT->OUT[0].CLR        = (1 << 0);                   // event 0 clear OUT0 (timeout)
    LPC_SCT->OUT[1].SET        = (1 << 3);                   // event 3 set   OUT1 (no width error)
    LPC_SCT->OUT[1].CLR        = (1 << 0) | (1 << 5);        // event 0 and 5 clear OUT1 (width error)
    LPC_SCT->OUTPUT           |= 3;                          // default set OUT0 and OUT1
    LPC_SCT->LIMIT_L           = (1 << 0) | (1 << 1);        // event 0 and 1 limit the timer
    LPC_SCT->EVEN              = (1 << 0) | (1 << 5);        // event 0 and 5 generate an irq

    NVIC_EnableIRQ(SCT_IRQn);                                // enable SCTimer/PWM interrupt
    LPC_SCT->CTRL_U           &= ~(1 << 2);                  // start timer
}
```

图27. 解码 PWM 初始化代码

13. RC5 传输

13.1 目的

此示例 (*SCTx_rc5_send*) 将 SCTimer/PWM 用作 RC5 发送器，旨在驱动遥控器的红外 LED。这是针对较早甚至停产设备（如 PCA84C122 和 SAA3010）的高性价比、低功率替代方案。

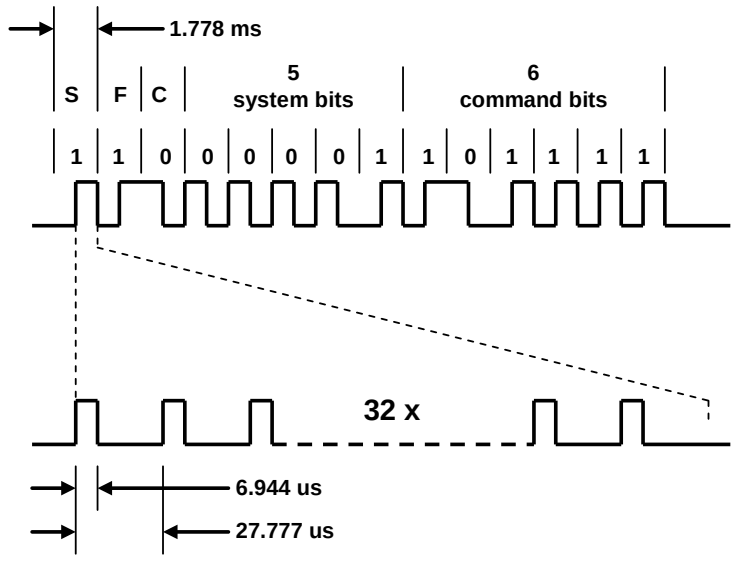


图28. RC5 帧示例

13.2 配置

使用了 SCTimer/PWM 的两个定时器。低电平 16 位定时器用于生成带 25 %占空比的 36 kHz 调制脉冲。（空）端口引脚用作 SCT 的输入。软件在 SCT_IN0 输入引脚中选择上拉电阻或下拉电阻以控制猝发有效。定时器的高位部分用于发送实际的曼彻斯特编码数据。

MRT（多速率定时器）中断处理程序（MRT_IRQHandler 在 main.c 中）用于发送 14 个数据位。

- 输入：SCT_IN0 内部使用的（空）输入可以在高电平时启用猝发。
- 输出：SCT_OUT0 用作 LED 驱动器输出，高电平有效。输出 36 kHz 的猝发信号。36 kHz 信号脉冲拥有 25 %的占空比。
- 已用的匹配：匹配 0 和 1。
- 已用的事件：3。
- 已用的状态：无。

13.3 设计

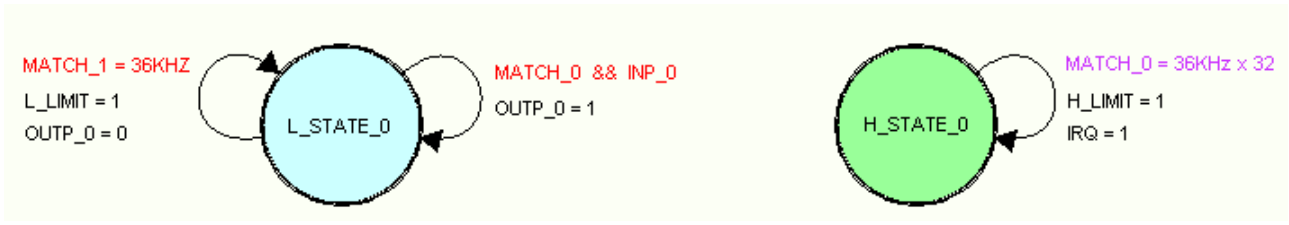


图29. RC5 发送器状态图

13.4 结果

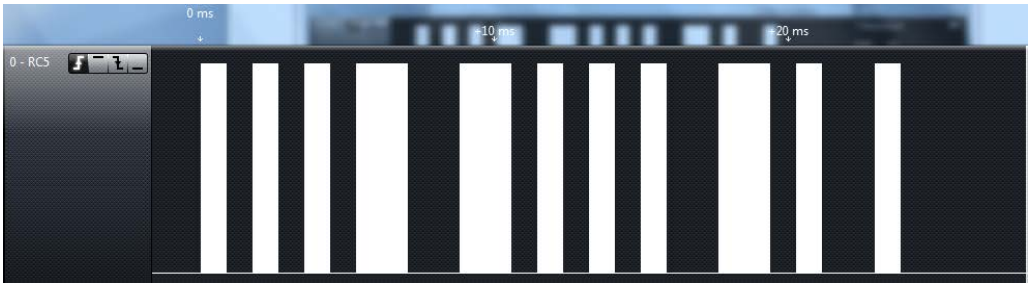


图30. 在 SCT_OUT0 中测量一个完整的 RC5 帧

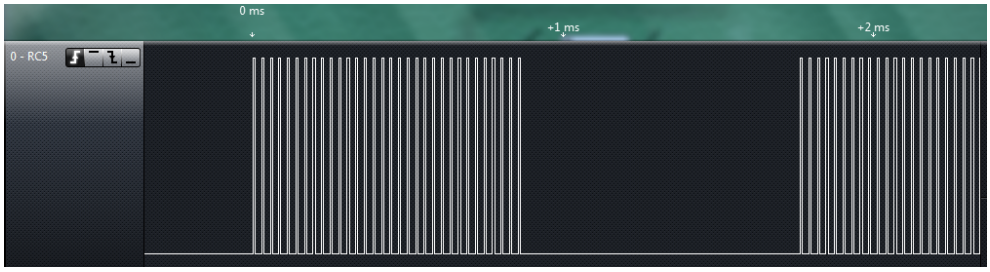


图31. 针对一个高电平帧位的 32 载波脉冲猝发

14. RC5 接收

14.1 目的

此示例 (*SCTx_rc5_receive*) 将 SCTimer/PWM 低电平定时器器件用作 RC5 接收器（曼彻斯特解码）。使用微控制器的 U(S)ART0 通过 RS232 接口发送接收的 RC5 帧（19200 波特率）。

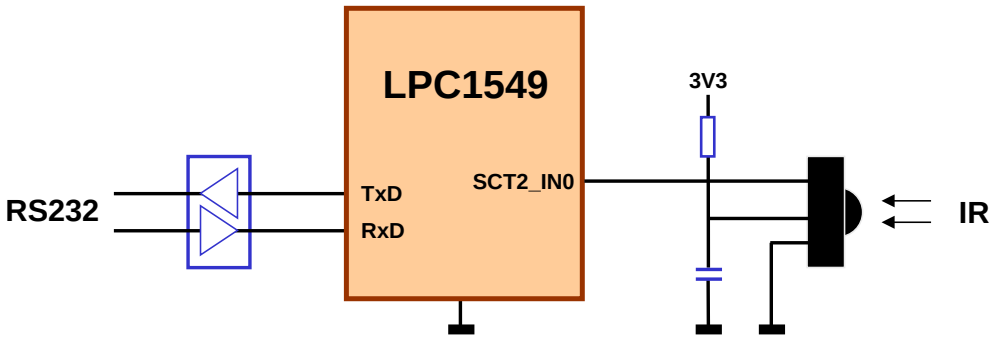


图32. 带 LPC1549 的 RC5 接收器硬件置位

14.2 配置

硬件置位显示在图 32 中。SCTimer/PWM 输入时钟已经预分频至 1 MHz。SCTimer/PWM 输入 0 用于在输入信号的上升沿和下降沿生成事件。这些事件用于捕获计数器值，限制（重置）计数器并在上升沿生成中断。在 SCTimer/PWM 中断处理中，解码收到的数据。

- 输入：SCT_IN0 用于接收 RC5 数据。
- 输出：无。
- 已用的匹配/捕捉：匹配 0，捕捉 1 和 2。
- 已用的事件：3。
- 已用的状态：无。

14.3 设计

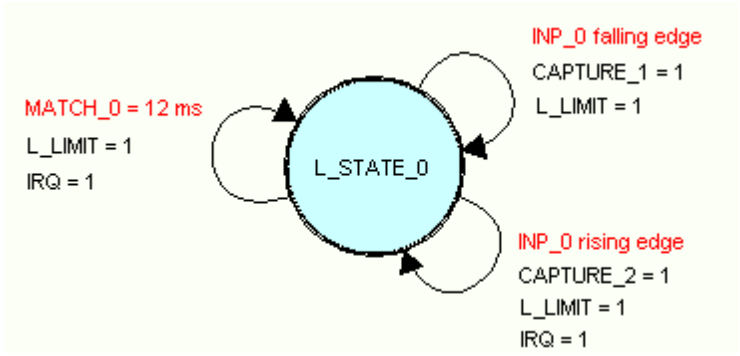


图33. RC5 接收器状态图

14.4 初始化代码

```
void RC5_Init(void)
{
    LPC_SYSCON->SYSAHBCLKCTRL1 |= EN1_SCT2;                // enable the SCT2 clock
    LPC_SCT2->CTRL_L |= (SystemCoreClock/1000000-1) << 5;  // set prescaler, SCTimer/PWM clock = 1 MHz
    LPC_SCT2->REGMODE_L      = (1 << 1) | (1 << 2);        // register pair 1 and 2 are capture
    LPC_SCT2->MATCH[0].L     = 12000;                      // match 0 @ 12000/1MHz = 12 msec (timeout)
    LPC_SCT2->MATCHREL[0].L  = 12000;
    LPC_SCT2->EVENT[0].STATE = 0x00000001;                 // event 0 only happens in state 0
    LPC_SCT2->EVENT[0].CTRL  = (0 << 0)  |                 // MATCHSEL[3:0]   = related to match 0
                               (1 << 12) |                 // COMBMODE[13:12] = uses match condition only
                               (1 << 14) |                 // STATELD [14]    = STATEV is loaded into state
                               (0 << 15);                  // STATEV  [15]    = new state is 0
    LPC_SCT2->EVENT[1].STATE = 0x00000001;                 // event 1 only happens in state 0
    LPC_SCT2->EVENT[1].CTRL  = (0 << 6)  |                 // IOSEL   [9:6]   = SCT_IN0
                               (2 << 10) |                 // IOCOND  [11:10] = falling edge
                               (2 << 12) |                 // COMBMODE[13:12] = uses IO condition only
                               (1 << 14) |                 // STATELD [14]    = STATEV is loaded into state
                               (0 << 15);                  // STATEV[ 15]     = new state is 0
    LPC_SCT2->EVENT[2].STATE = 0x00000001;                 // event 2 only happens in state 0
    LPC_SCT2->EVENT[2].CTRL  = (0 << 6)  |                 // IOSEL   [9:6]   = SCT_IN0
                               (1 << 10) |                 // IOCOND  [11:10] = rising edge
                               (2 << 12) |                 // COMBMODE[13:12] = uses IO condition only
                               (1 << 14) |                 // STATELD [14]    = STATEV is loaded into state
                               (0 << 15);                  // STATEV  [15]    = new state is 0
    LPC_SCT2->CAPCTRL[1].L   = (1 << 1);                   // event 1 causes capture 1 to be loaded
    LPC_SCT2->CAPCTRL[2].L   = (1 << 2);                   // event 2 causes capture 2 to be loaded
    LPC_SCT2->LIMIT_L        = 0x0007;                     // events 0, 1 and 2 are used as counter limit
    LPC_SCT2->EVEN           = 0x00000005;                 // events 0 and 2 generate interrupts
    NVIC_EnableIRQ(SCT2_IRQn);                             // enable SCTimer/PWM interrupt
    LPC_SCT2->CTRL_L        &= ~(1 << 2);                  // unhalt it
}
```

图34. RC5 接收器 SCTimer/PWM 初始化代码

14.5 结果

使用微控制器的 U(S)ART0 通过 RS232 接口发送接收的 RC5 消息。运行 TeraTerm（19200 波特率）的 PC 用于显示收到的数据。第一个值代表 RC5 系统字节；第二个值代表 RC5 命令字节。

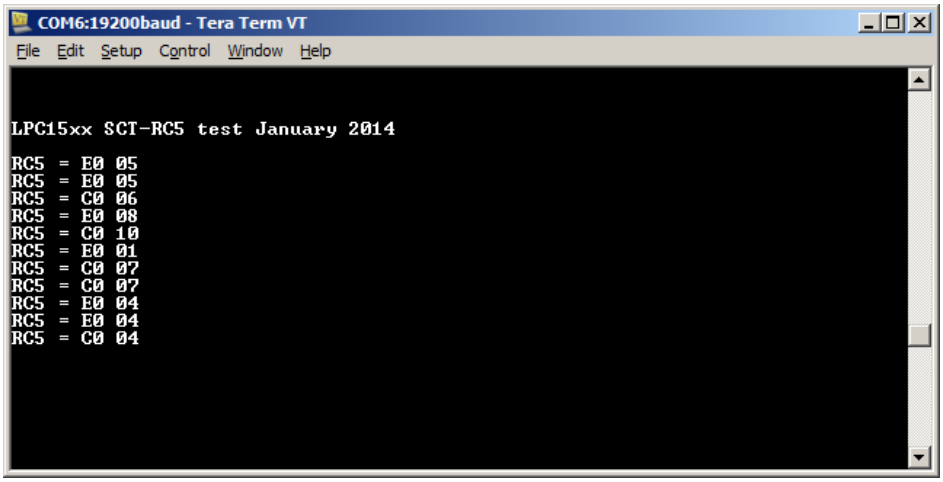


图35. PC 终端的 RC5 输出

15. SCTimer/PWM start_stop

15.1 目的

此项目 (SCTx_start_stop) 显示 SCTimer/PWM 启动和停止事件的可能使用情况，这些事件可影响同一 SCTimer 的另一半。

除了 LPCXpresso 代码，这本操作手册还包含使用 Keil 编译器的 SCTimer/PWM Fizzim 设计师工具的示例应用（参见图 36）。

15.2 配置

SCTimer/PWM 中的定时器可以在分拆模式（2 个 16 位定时器）中进行配置。定时器的每一半生成启动和停止事件，这会以类似乒乓的方式交替启动或停止状态机的另一侧。

注意：要在退出重置时将一个定时器保持在已停止状态， 需要在与 **STOP 位相同的写周期**清除 HALT 位。

15.3 设计

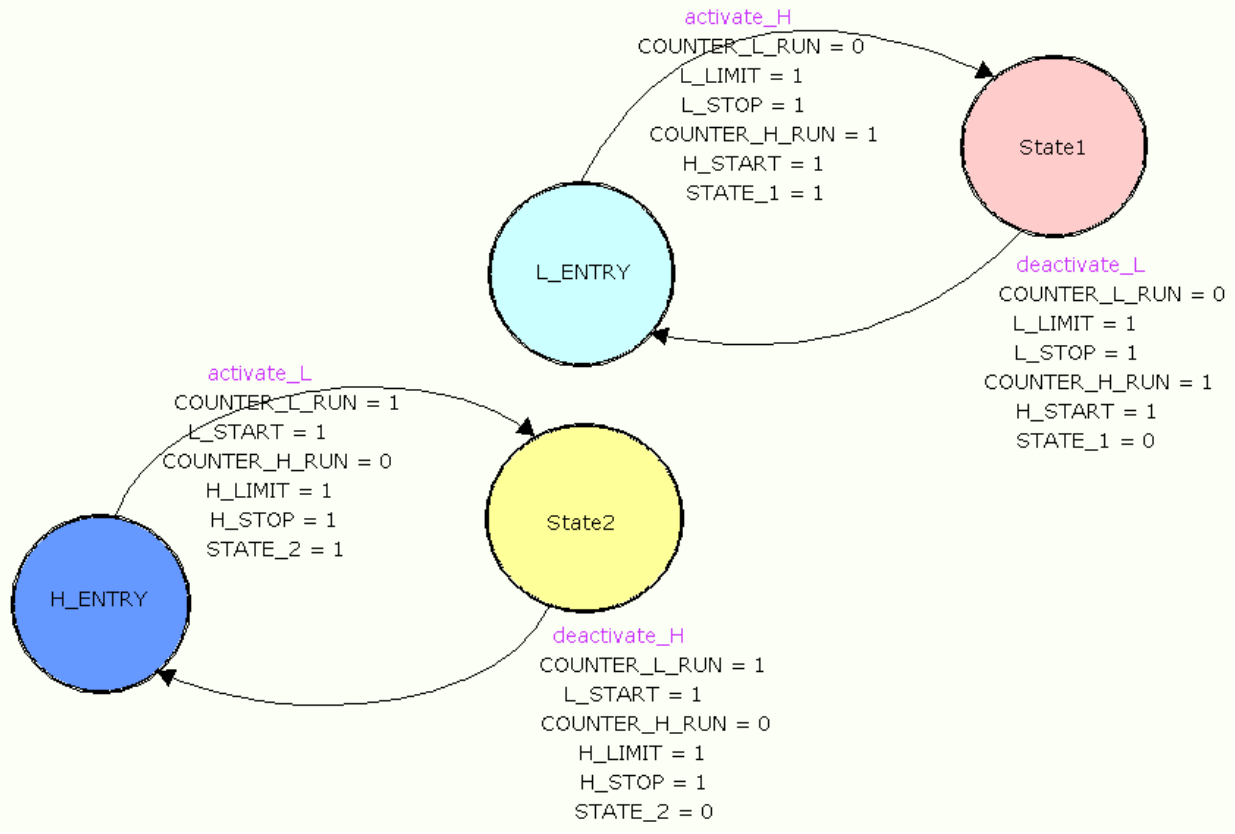


图36. start_stop 状态图 (Fizzim)

16. 输入同步

SCT 可以在输入用于创建事件之前同步 SCTimer/PWM 输入时钟的输入。

在全局配置寄存器（9 至 16）中完成选项选择。

如果输入同步至 SCTimer/PWM 时钟，您可选择非同步选项实现更快响应。

如果输入与 SCTimer/PWM 时钟异步，特别是当输入是边缘敏感型时，建议置位同步选项，以便不会错过任何输入与/或输入边缘。参见图 38。

Table 175. SCT configuration register (CONFIG, address 0x1C01 8000) bit description *...continued*

Bit	Symbol	Value	Description	Reset value
16:9	INSYNC	-	Synchronization for input n (bit 9 = input 0, bit 10 = input 1,..., bit 16 = input 7). A 1 in one of these bits subjects the corresponding input to synchronization to the SCT clock, before it is used to create an event. If an input is synchronous to the SCT clock, keep its bit 0 for faster response. When the CKMODE field is 1x, the bit in this field, corresponding to the input selected by the CKSEL field, is not used.	1

图37. 配置寄存器 INSYNC 位

SCTIN_0 的每个上升沿都会生成一个事件来切换 SCTOUT_0。

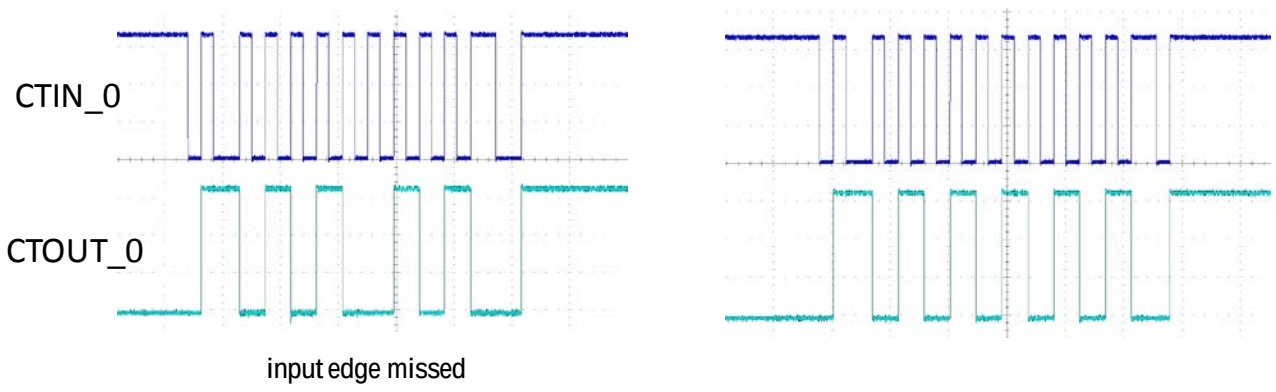


图38. 左侧：没有同步，缺少输入 - 右侧：同步正常

17. 去抖

17.1 目的

此示例 (SCTx_dithering) 显示一些器件上提供的 SCT 的去抖功能（参见表 1）。借助此功能，您可将平均定时器分辨率增加 16 倍。

17.2 配置

此示例代码正在使用 SCT0，并且已经通过在 250 kHz 下运行的 LPC1549（来自 IRC）在 LPCXpresso 板上进行测试。SCT0 定时器在 SCT0_OUT0 上生成一个 4 毫秒 PWM 输出（参见图 39）。PWM 信号的占空比从 25 %起（1 毫秒开启，3 毫秒关闭）。

SCT0_OUT0 可链接至 P0_24（LPCXpresso 板上的绿色 LED）。

按住 SW3 (P1_9) 按钮可以使用 SCTimer/PWM 去抖功能将 LED 的亮度改成 37.5 %（平均 1.5 毫秒开启，2.5 毫秒关闭）。松开 SW2 按钮可让 LED 亮度返回至 25 %。

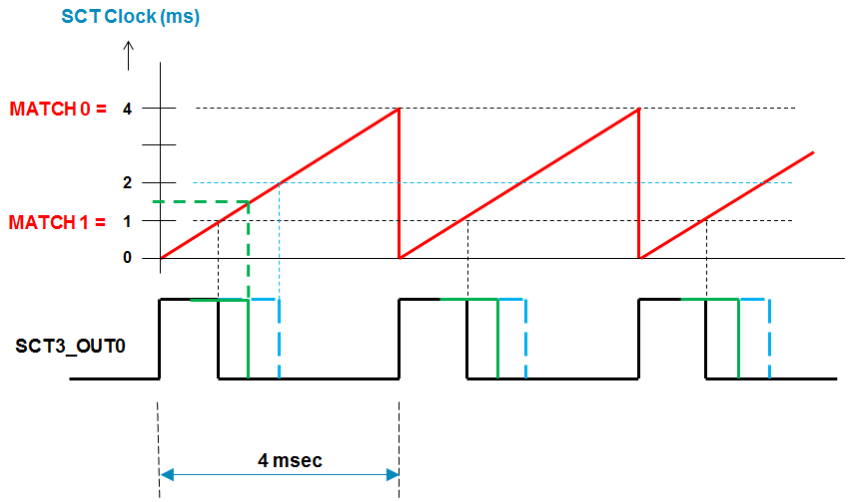


图39. 带去抖功能的 PWM 输出

17.3 置位

首先需要的是非常低（1 毫秒）的 SCTimer/PWM 输入时钟。为实现这点，对系统时钟 (= IRC) 进行 48 分频，得到一个 250 kHz 的系统时钟（这在模块 *system_LPC15xx.c* 中完成）。

然后，SCTimer/PWM 预分频器置位为 250，生成一个 1 kHz 的 SCTimer/PWM 输入时钟。

```
LPC_SCT0->CTRL_U |= (249 << 5);        // SCT0 clock input is:
                                       // 250KHZ/(249+1) = 1kHz (1msec)
```

17.4 结果

图 40 显示使用小数匹配寄存器的三个不同值的 SCTimer/PWM 输出。

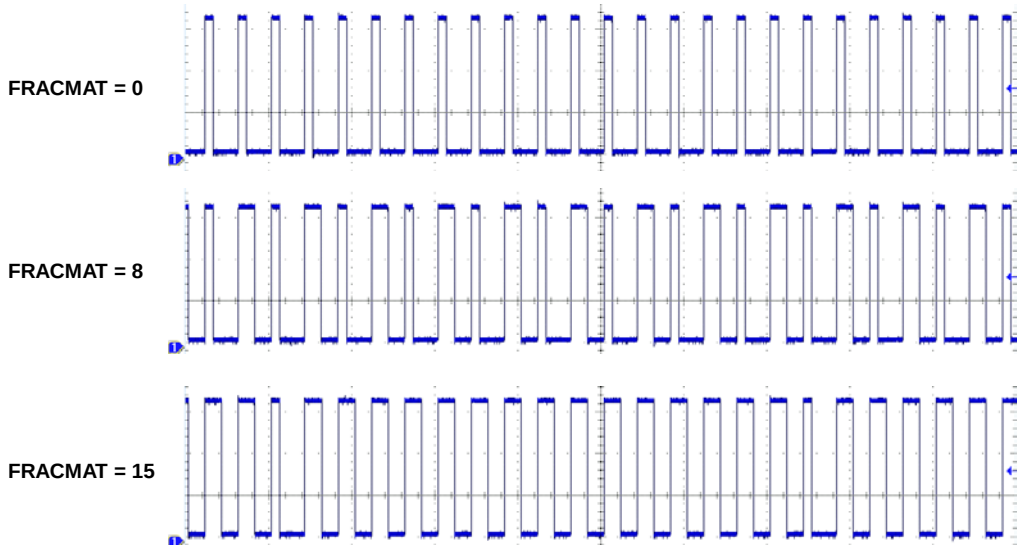


图40. LPC15xx SCTimer/PWM 去抖示例

18. WS2811 LED 驱动器

WS2811 LED 驱动器使用一个简单的单线协议来传输 24 位 RGB 值。可以链接多个 WS2811 设备，并且所有这些设备的 RGB 值可以一起发送至链路中的第一个设备。第一个设备利用第一个 24 位数据包来置位自己的 RGB 状态，并重新发送剩余数据包。这类 RGB 值的数据块通过数据行的“复位代码”彼此分离，这是一个至少 50 µs 的简单无效期，在此期间，信号线路将保持在低电平。

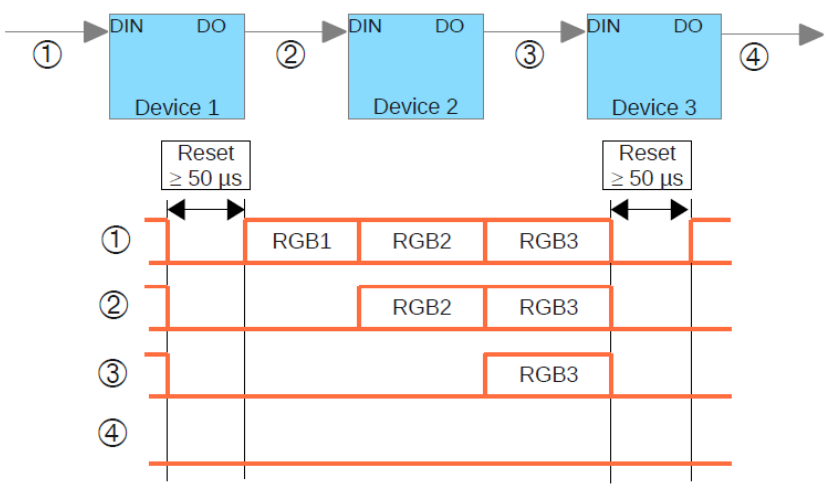


图41. WS2811 单线式通信协议

单个 24 位数据包包含 1.25 µs 或 2.5 µs 的 24 位周期，具体取决于 WS2811 是配置为 800 kHz 还是 400 kHz。每个位都可作为一个带占空比的脉冲发送，具体取决于位值。“0”拥有 20 %的标称占空比，而“1”则有 48 %的标称占空比。发送每个位时都有较大的时间容限，但整个 24 位数据包或多个 RGB 值的累加抖动应保持在最小值。数据位 0 和 1 的常规波形如下所示。您还可以看到完整的 RGB 帧，这代表 RGB 值 0xCA1722（红色通道 = 0xCA，绿色通道 = 0x17，蓝色通道 = 0x22）。

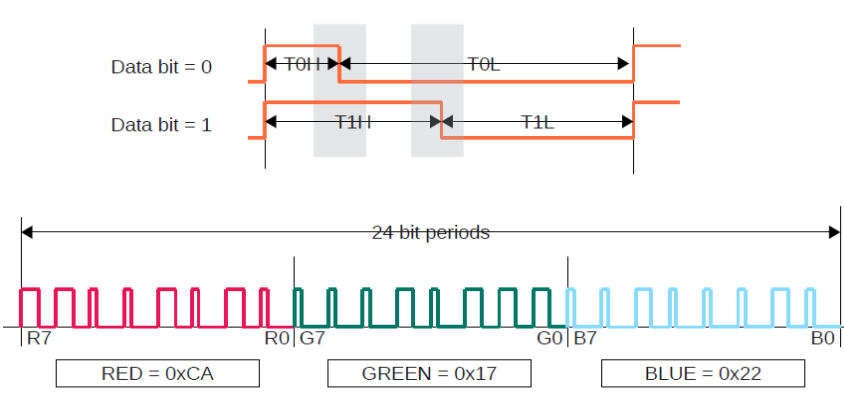


图42. WS2811 帧和位波形

您会看到发送的每个颜色通道都以 MSB 开始。

数据位打开和关闭期间的值取决于 WS2811 的操作频率，可以是 400 kHz 或 800 kHz。

	Operation Frequency 400 kHz	Operation Frequency 800 kHz
T0H	0.5 (± 0.15) µs	0.25 (± 0.15) µs
T0L	2.0 (± 0.15) µs	1.0 (± 0.15) µs
T1H	1.2 (± 0.15) µs	0.6 (± 0.15) µs
T1L	1.3 (± 0.15) µs	0.65 (± 0.15) µs
Bit	2.5 µs	1.25 µs
Frame	60 µs	30 µs
Reset	≥ 50 µs	≥ 50 µs

图43. WS2811 规格

18.1 实现

WS2811 发送器的设计显示了状态和事件在 SCT 中的高效利用。它只使用 1 个 SCTimer/PWM 半 16 位定时器、6 个事件和 12 个状态（资源检查见 表 1），余下超过 50 % 的 SCTimer/PWM 资源用于其他任务。

概述：

- 使用 1 个 16 位定时器，将其他 16 位定时器留作他用。
- 使用该预分频器运行最低时钟频率以节约功耗。
- 自主发送 24 位帧，双缓冲。
- 每次发送帧后中断操作，留下几乎整个帧时间供 CPU 提供下一个帧。
- 最后一个发送帧后减半。
- 在每个多帧（数据块）发送之前加一个可调节长度的复位代码。

18.2 配置

SCTimer/PWM 必须配置为分离模式 (CONFIG.UNIFY = 0)。必须将数据输出的冲突解决寄存器置位为“无操作”（这是默认置位）。

18.2.1 匹配寄存器

MATCH0/MATCHREL0 保持位长（周期）。

MATCH1/MATCHREL1 保持 T1H 时间。

MATCH2/MATCHREL2 保持 T0H 时间。

18.2.2 输入/输出

可通过配置输出对应的 SET 和 CLR 寄存器，将数据输出分配至任何可用的 SCTx_OUTx 信号。

双缓冲方案需要使用辅助输出。它可分配至任意 SCTOUT 信号，但不需要连接至引脚（内部信号）。

18.2.3 状态

状态构成数据传输的核心。针对最多只有 15 个状态的器件（如 LPC1500），我们决定在 2 个 12 位猝发中发送一个帧（需要 12 个状态）。状态机从状态 11 开始，并在每个发送位之后递减状态。在状态 0 中发送最后一个帧位后，启动后面 12 个位时，状态机会强制进入状态 11。状态 11 对应于第一个发送的位 (MSB)，状态 0 对应于最后一个发送的位 (LSB)。

在新位开始时，可置位数据输出。两个匹配寄存器置位为可清除数据输出的时间 T0H (MATCH1, EV13) 和 T1H (MATCH2, EV14) 触发器事件。如果没有其他事件，这将始终发送逻辑 0，因为 MATCH1 首先出现，位的活动时间将在 T0H 时结束。

我们需要其他事件来决定 12 个状态 11...0 中每个状态的数据位的值。此事件 (EV12) 经过配置可置位时间 T0H (MATCH1) 时的数据输出。因此，会在 T0H 位置发生冲突，带之前描述的想要清除数据输出的事件。作为“无操作”数据输出的冲突解决寄存器，数据输出不会在 T0H 处清除，而会保留置位，直到 MATCH2 在 T1H 时触发一个事件。将发送数据字写入到新事件的事件状态寄存器中时，它会作为掩模，并且仅在数据字在对应的位位置有 1 的状态下启用此事件，因此 SCTimer/PWM 会发送 1。数据字中的 0 可在对应的状态中禁用此事件，SCTimer/PWM 会发送 0。

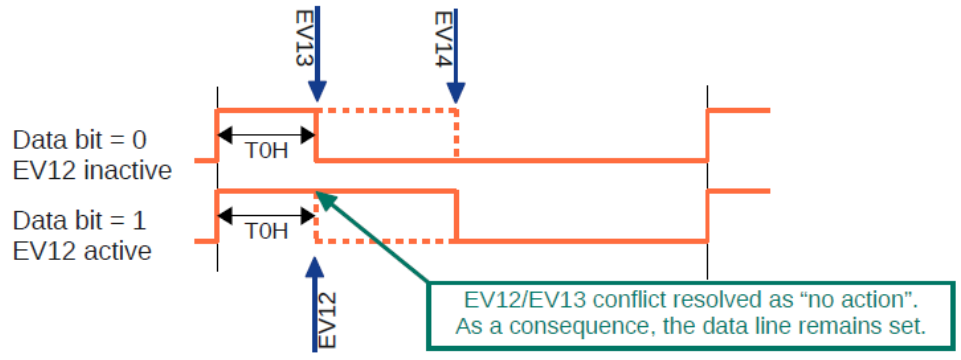


图44. SCTimer/PWM 数据位处理

18.2.4 事件详情

事件 15 决定起始位。它在所有状态中都有效，并可通过一个 MATCH0（位时间）事件触发。此事件可置位数据输出并递减状态数字（例如，添加 31）。首次发生此事件之前，将状态预设为 12，这非常重要。

事件 14 决定最大输出启动时间，这等于 T1H。它在除状态 0 外的所有数据位状态 (1...23) 中有效，并由 MATCH2 触发。此事件可清除数据输出。事件 10（带扩展功能）可以在状态 0 中取代此事件。

事件 13 决定零数据位的结束位，这等于 T0H。它在所有数据位状态中有效，并由 MATCH1 触发。此事件可清除数据输出。由于在定时器周期中，此事件早于事件 14 发生，因此我们可能只会发送逻辑零位。因此，事件 12 和 11 可覆盖事件 13 的操作。它们与事件 13（如果启用！）同时发生，并且由于冲突解决寄存器置位为“无操作”，它们可能会取消输出操作。

事件 12 会从第一个数据缓冲区强制发送逻辑 1。该事件在发送数据字在该位位置有一个 1 的状态 (0...11) 下启用，并且由 MATCH1 触发。它还通过辅助标志（缓冲区选择器） = 0 来限定。这意味着该事件与事件 13 同时发生，并且将冲突解决寄存器置位为“无操作”可取消事件 13 的输出清除操作。这导致将位启动时间延长至 T1H（数据输出最终由事件 14 清除）。

事件 11 等于事件 12，除非其触发器条件检查了辅助标志 = 1。

事件 10 决定帧传输的结束。它还在状态 0 时发挥事件 14 的作用。它仅在状态 0 有效（LSB 发送），并由 MATCH2（最后位启动周期的结束位）触发。它可切换辅助位，清除数据输出并触发中断。为响应此中断，CPU 应读取辅助位并决定哪个缓冲区（= 事件状态寄存器 11 或 12）接收下一个发送帧（12 位）数据。如果不想要发送其他帧，CPU 应该向 HALT_H 寄存器写入一种模式（1 << 10，针对事件 10）。这会导致发送操作在刚刚开始的帧结束时停止。此事件将状态数字置位为 12。

18.3 操作

准备此模式的 SCTimer/PWM 时（当 SCTimer/PWM 全局暂停时），以下步骤必不可少。我们假设 H 计数器用于 WS2811 模式。

1. 针对分拆模式配置 SCTimer/PWM。
2. 配置匹配寄存器：
   a. MATCHREL0 = SystemCoreClock/DATA_SPEED - 1
   b. MATCHREL1 = 20% of SystemCoreClock/DATA_SPEED - 1
   c. MATCHREL2 = 48% of SystemCoreClock/DATA_SPEED - 1
3. 配置事件：
   a. 事件 15：MATCH0，所有状态，DATA = 1 且 STATE += 31
   b. 事件 14：MATCH2，除状态 0 外的所有状态，且 DATA = 0
   c. 事件 13：MATCH1，所有状态且 DATA = 0
   d. 事件 12：MATCH1 && AUX==0，应发送逻辑 1 的所有状态 [11:0]，且 DATA = 1
   e. 事件 11：MATCH1 && AUX==1，应发送逻辑 1 的所有状态 [11:0]，且 DATA = 1
   f. 事件 10：MATCH2，状态 0，IRQ，AUX = 切换且 STATE = 12

18.3.1 发送帧数据块

1. 暂停 H 定时器
2. 预设重置时间。以负数的形式在计数器 COUNT_H 中写入需要的时钟脉冲数。
3. 置位 STATE_H = 12。
4. 通过向事件 12 状态寄存器写入第一个发送帧，启动发送缓冲区。确保位 [31:12] 为零。
5. 通过向事件 11 状态寄存器写入第二个发送帧，启动其他发送缓冲区。如果只要发送一个帧，则写入 0。
6. 启动 H 定时器作为向上计数器（在寄存器 CTRL_H 中清除 DOWN_H 和 HALT_H）。

18.3.2 中断处理

可以在完全发送帧之后触发中断。

1. 如果这是最后一个发送帧，则停止定时器。
2. 读取辅助输出位。如果是 1，将下一个帧写入事件 12 状态寄存器，否则写入事件 11 状态寄存器。

如果未遵守上述流程并且数据输出停滞在高电平，则可能需要其他操作。您不应该简单地清除输出，因为对输出寄存器的访问可能会带来竞争条件，影响通过硬件访问来自其他运行不同应用程序的 (L) 定时器的输出。

18.4 结果

图 45 显示一个 12 位帧 (0x123) 的发送。淡蓝色轨迹仅用于调试。此 GPIO 信号会在每个帧处切换（位于事件 10 中断服务程序中）。

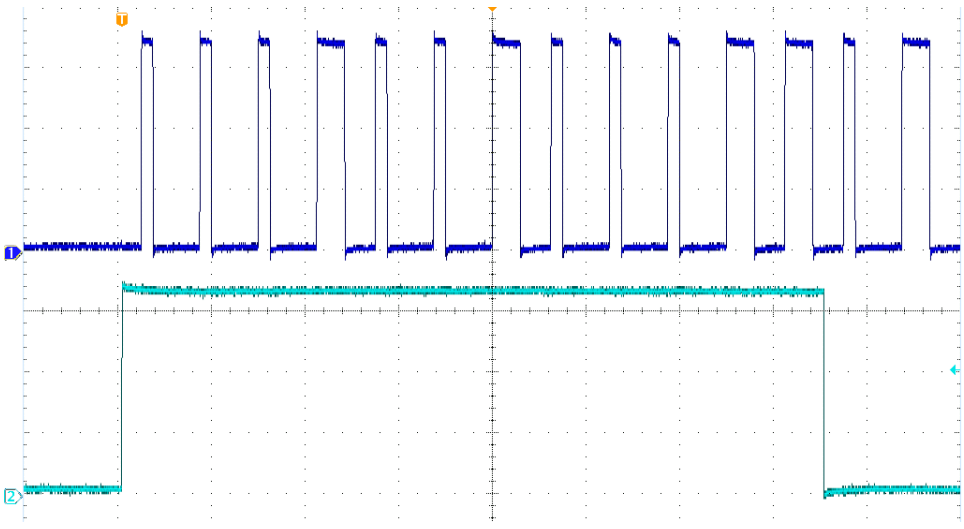


图45. WS2811 LED 驱动器的一个 12 位帧传输

图 46 显示发送一个带四个 24 位 RGB WS2811 LED 驱动器值的数据块（拆分成 8 个 12 位帧：0x123、0x456、0xFF0、0x0CC、0x555、0x555、0x800、0x001）。

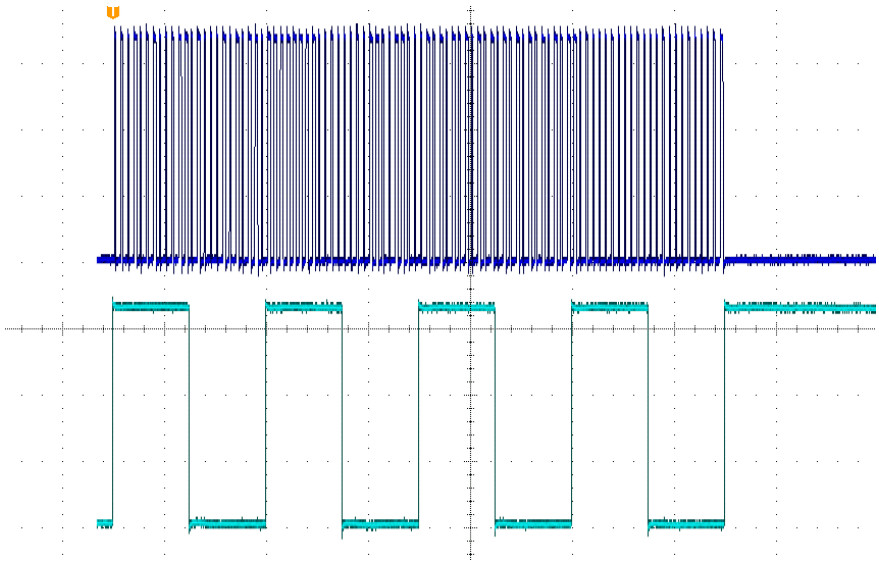


图46. WS2811 发送一个带四个 24 位 RGB 值的数据块

图 47 显示发送的 RGB 值的数据块通过数据行的“重置代码”彼此分离，这是一个至少 50 µs 的简单无效期，在此期间，信号线路将保持在低电平。

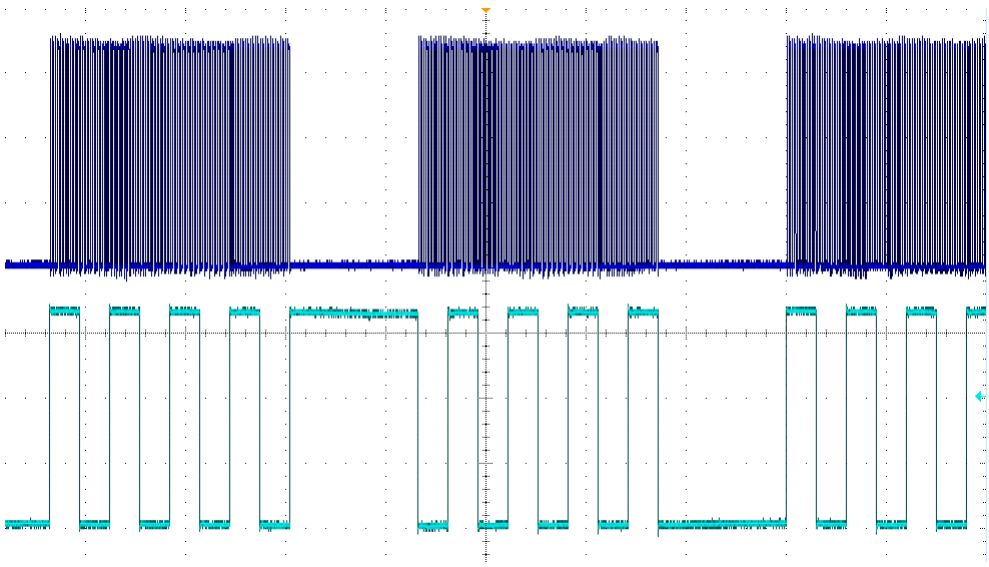


图47. WS2811 发送通过无效周期隔离的 RGB 值的数据块

19. WS2812 LED 驱动器

19.1 目的

此示例显示使用带较少资源（如 LPC81x 中）的更小 SCT 构建之前示例所示的串行接口的替代方法。这次我们将使用不同的、但非常类似的器件，即拥有稍微不同的时间规范的 WS2812。WS2812 的工作频率始终为 800 kHz。

T0H	0 code ,high voltage time	0.35us	±150ns
T1H	1 code ,high voltage time	0.7us	±150ns
T0L	0 code , low voltage time	0.8us	±150ns
T1L	1 code ,low voltage time	0.6us	±150ns
RES	low voltage time	Above 50μs	

图48. WS2812 数据发送时间 (TH + TL = 1.25μs ± 600ns)

19.2 配置

此方法是使用 SCT **重塑** SPI 外设的输出数据。因此，开关矩阵用于重定向 SPI 的 MOSI 和 SCK 信号至 SCT 输入。编程的 SPI 位速率为 800 kHz，并且可将一个 WS2812（24 位）RGB LED 帧作为三个 8 位 SPI 帧发送。

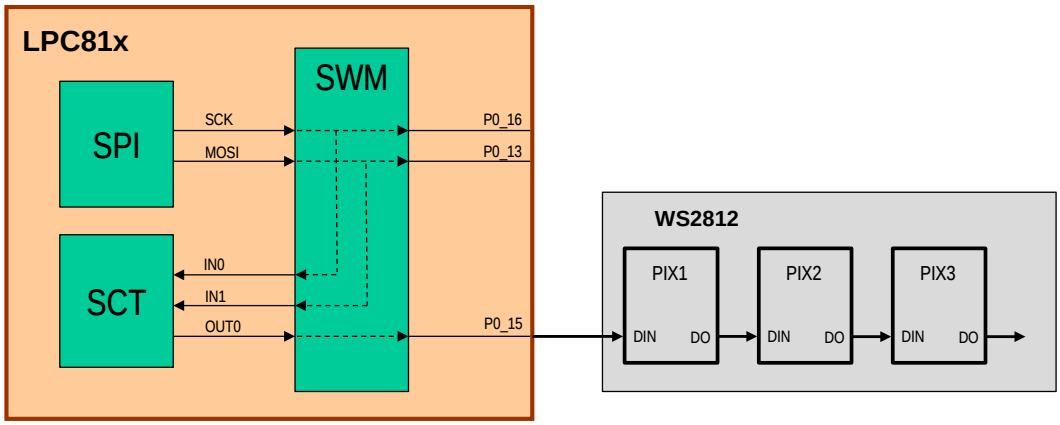


图49. LPC812 – SPI / SCTimer / WS2812 硬件配置

发生 72 位猝发 (3 x 24) 时，演示示例代码将 RGB 数据写入 WS2812 的前三个 LED，如同它是 SPI 外设一般。此代码在 MCORE48 板上使用 24 MHz 下运行的 LPC812（从使用 PLL 的片上 IRC）进行测试。

19.3 实现

WS2812 发送器的设计显示了状态和事件在 SCT 中的高效利用。它在统一的（32 位）定时器模式、6 个事件和 2 个状态中使用 SCTimer/PWM。

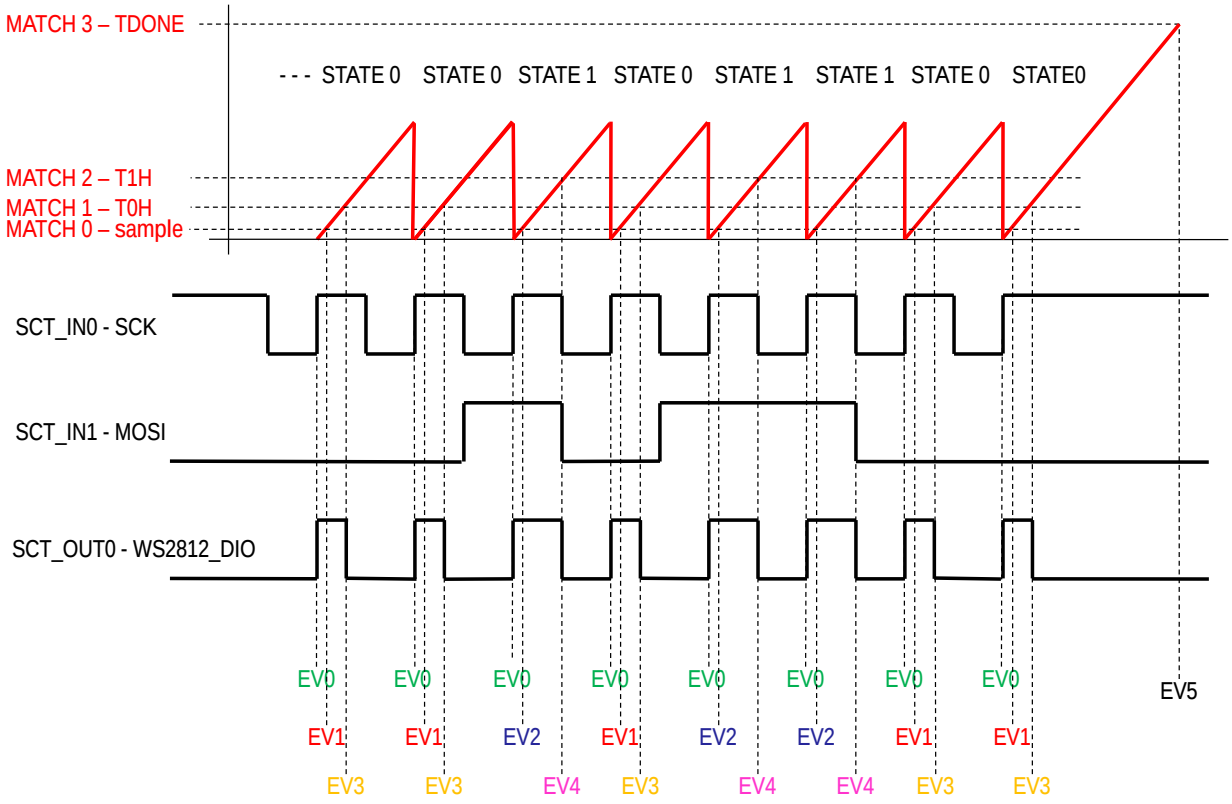


图50. SCT - WS2812 驱动器置位

19.3.1 匹配寄存器

MATCH0 用于保持采样时间（始终是一个 SCT 时钟）。它生成事件 1 或 2，具体取决于 SCT_IN1 (MOSI) 输入的电平。

MATCH1 保持 T0H 时间 (0.35 us)，并且如果 SCT 处于状态 0，它会触发事件 3。

MATCH2 保持 T1H 时间 (0.70 us)，并且如果 SCT 处于状态 1，它会触发事件 4。

MATCH3 保持发送完成时间 (> 50 us) 并且用于 STOP SCTimer。

19.3.2 输入/输出

SPI_SCK 信号分配至输入 SCT_IN0。上升沿在所有状态下生成事件 0。

SPI_MOSI 信号分配至输入 SCT_IN1。此输入的电平用于决定是发送“0”还是“1”。

发送数据输出可分配至 SCT_OUT0。

19.3.3 状态

两个状态用于决定发送逻辑 0 位还是逻辑 1 位。

19.3.4 事件详情

事件 0 决定起始位发送。它在所有状态下有效，并且可由输入 0(SPI_SCK) 的上升沿触发。这是一个限制和启动事件。在这个事件中，输出 0（WS2812 数据）驱动至高电平。它不会更改状态。

事件 1 强制更改状态 0。它在所有状态下有效并由 MATCH0（采用时间）和输入 1 (SPI_MOSI) 上的一个低电平触发。

事件 2 强制更改状态 1。它在所有状态下有效并由 MATCH0（采用时间）和输入 1 (SPI_MOSI) 上的一个高电平触发。

事件 3 决定逻辑零数据位的结束位，这等于 T0H。它仅在状态 0 有效，并由 MATCH1 触发。此事件可清除数据输出但不会更改状态。

事件 4 决定逻辑 1 数据位的结束位，这等于 T1H。它仅在状态 1 有效，并由 MATCH2 触发。此事件可清除数据输出但不会更改状态。

事件 5 在帧传输结束时生成。它在所有状态中有效，并由 MATCH3 触发。此事件可停止 SCTimer，触发中断并且不会更改状态。

19.3.5 中断处理

完成帧传输、经过至少 50 us 的延迟并且 SCTimer 停止后，可由事件 5 触发中断。为响应该中断，演示应用会置位一个标志以表明其准备好发送下一个帧。如果新数据已由 SPI 外设发送，则会发送下一个帧。

19.4 结果

下图显示 WS2812 帧的发送。深蓝色轨迹是 SCK，淡蓝色轨迹是 MOSI，紫色轨迹是输出数据。

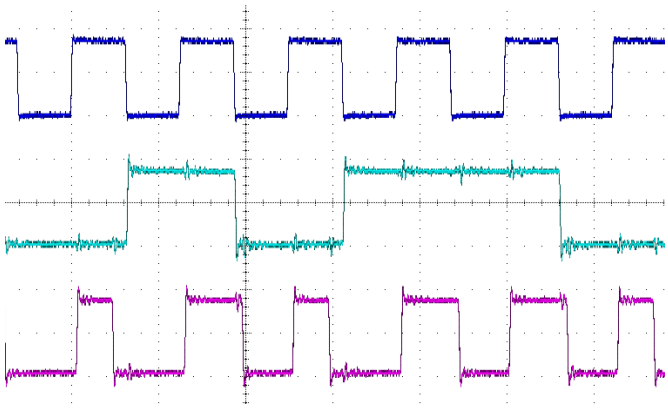


图51. WS2812 LED 驱动器位传输。

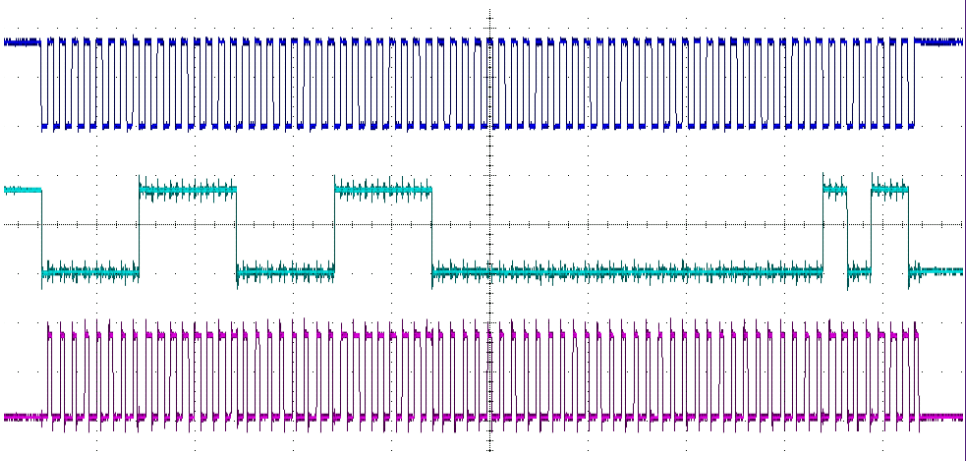


图52. WS2812 发送一个带三个 24 位 RGB 值的数据块。

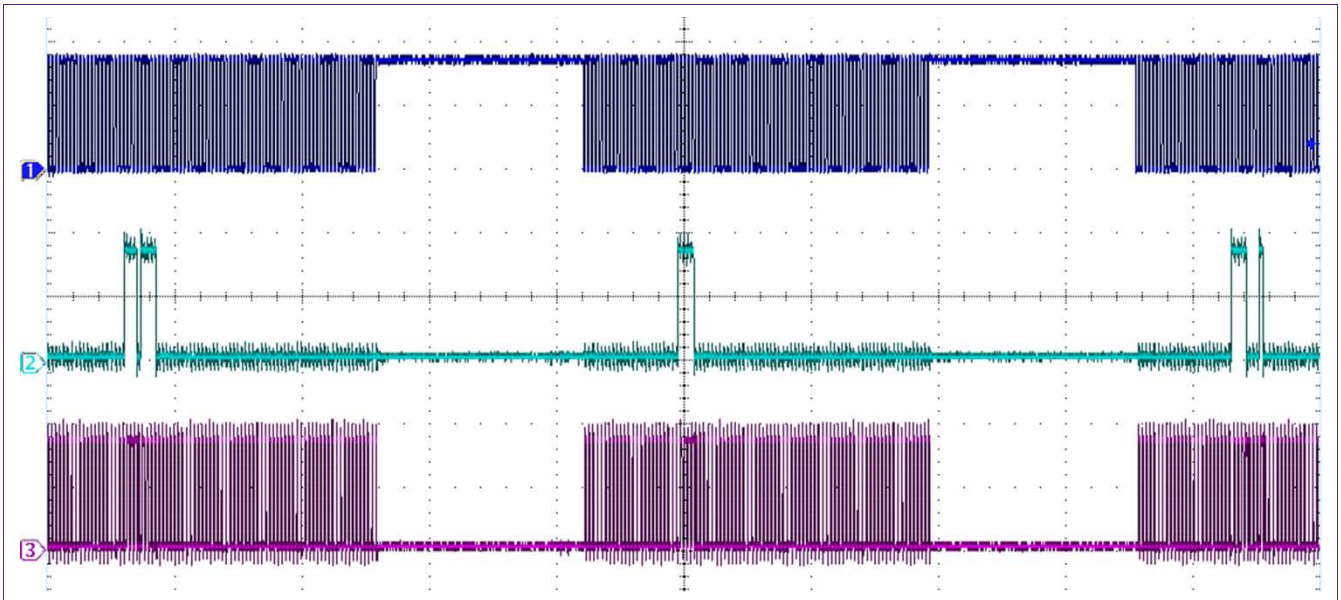


图53. WS2812 发送通过无效周期隔离的 RGB 值的数据块

20. 带 0 - 100%占空比的 PWM

20.1 目的

此示例 (*SCT_PWM_0_100*) 显示如何生成一个带 0 - 100%占空比的居中对齐 PWM 输出。它会使用低电平 16 位 SCTimer/PWM 定时器在 SCT_OUT0 上生成 10 kHz 的 PWM 信号。SysTick 定时器用于通过更新 MATCHRELOAD 寄存器来定期减少和增加 PWM 信号的占空比。通过将 SCT_OUT0 连接至 LED，这可调整 LED 的亮度。

20.2 配置

SCTimer/PWM 使用 MATCH[0].L 生成可更改自上向下计数的计数方向的定时器限制。第二个匹配寄存器 MATCH[1].L 用于定义信号的占空比。发生匹配事件 1 时，它会在向上计数时置位 SCTimer/PWM 输出 0，并在向下计数时清除（反转）输出。输出 0 可通过开关矩阵模块分配至 P0_14。

MATCH[1].L = 0 生成 0%占空比（信号关闭）。

MATCH[1].L = MATCH[0].L - 1 生成 100%占空比（信号打开）。

0 < MATCH[1].L < MATCH[0].L - 1 生成 1 至 99%占空比。

20.3 配置代码

```
/*****************************************************************************
*  SCT_L - is used for center aligned PWM at SCT_OUT0
*
*       |       PWM0 Period       |                       |
*       |                         |                       |
*                 +-----+                   +-----+                   +-----+
*                 |     |                   |     |                   |     |      OUT0
*    ---+---------+     +---------+---------+     +---------+---------+     +----
*
*                 |     |                   |     |                   |
*                EV0   EV0                 EV0   EV0                 EV0
*
*  MATCH0_L used for PWM0 frequency period
*  EV0 - all states - on MATCH1_L - set OUT0 if up counting, clear OUT0 if down counting
*
*  P0.14 [O] - SCT_OUT0 : PWM
*
*****************************************************************************/
#include "LPC8xx.h"

#define PWM             14                                  // PWM at port pin P0_14
#define OUT0            0                                   // SCT_OUT0 as PWM output
#define PWM_FREQ        10000                               // PWM required frequency = 10KHz
#define PWM_PERIOD      (SystemCoreClock / (PWM_FREQ * 2))  // PWM counter period (*2 because of bi-dir mode)
                                                            // example 24MHz/10KHz*2 = 1200 SCT clocks

void PWM_set(uint8_t val)                                   // set PWM duty cycle (from 0 to 100%)
{
#define PWM_STEP        (PWM_PERIOD / 100)                  // PWM resolution in 100 steps

    if (val == 0)                                           // check val between 0% and 100%
       LPC_SCT->MATCHREL[1].L = 0;
    else if (val < 100)
       LPC_SCT->MATCHREL[1].L = (PWM_STEP * val) - 1;
    else
       LPC_SCT->MATCHREL[1].L = PWM_PERIOD - 2;             // set to 100% duty cycle
}

void SCT_Init(void)
{
    LPC_SYSCON->SYSAHBCLKCTRL |= (1 << 7) | (1 << 8);       // enable the SWM and SCT clock

    LPC_SWM->PINASSIGN6 &= ((PWM << 24) | 0x00FFFFFF);      // SCT_OUT0 = PWM

    LPC_SCT->CONFIG           |= (1 << 17);                 // auto limit _L (on match 0)
    LPC_SCT->OUTPUT           |= (0 << OUT0);               // preset OUT0 low
//  LPC_SCT->OUTPUT           |= (1 << OUT0);               // preset OUT0 high (for low active signal)
    LPC_SCT->OUTPUTDIRCTRL    |= (0x1 << 0);                // reverse OUT0 set/clr when counter _L is
                                                            // down counting (center aligned mode)
    LPC_SCT->CTRL_L           |= (1 << 4);                  // bi-dir count mode

    LPC_SCT->MATCH[0].L        = PWM_PERIOD - 1;            // match 0 @ PWM freq
    LPC_SCT->MATCH[1].L        = 0;                         // use match 1 for PWM duty cycle

    LPC_SCT->EVENT[0].STATE    = 0x00000003;                // event 0 happens in all states (both 0 and 1)
    LPC_SCT->EVENT[0].CTRL     = (1 << 0)  |                // MATCHSEL[3:0]   = related to match 1
                                 (0 << 4)  |                // HEVENT[4]       = ev 0 belongs to the L timer
                                 (1 << 12) |                // COMBMODE[13:12] = match condition only
                                 (0 << 14) |                // STATELD[14]     = STATEV is added to state
                                 (0 << 15);                 // STATEV[15]      = 0 (no change)

    LPC_SCT->OUT[OUT0].SET     = (1 << 0);                  // ev 0 sets the OUT0 signal
//  LPC_SCT->OUT[OUT0].CLR     = (1 << 0);                  // ev 0 clears the OUT0 signal (low active mode)
```

```
    LPC_SCT->CTRL_L           &= ~(1 << 2);                 // start the _L counter
}
```

图54. 带 0 – 100%占空比的居中对齐 PWM

21. PWM at _L and _H

21.1 目的

此示例 (*SCT_PWML_PWMH*) 使用 SCT_L 一半来生成 PWM0 信号，并使用 SCT_H 一半来创建第二个独立的 PWM1 信号。这两个信号居中对齐，带 0 至 100%占空比。

21.2 配置

```
/*****************************************************************************
*  SCT_L - is used for PWM0 at SCT_OUT0
*  SCT_H - is used for PWM1 at SCT_OUT1
*
*  Both PWM0/1 are center aligned PWM signals, Pins used in this application:
*
*  P0.14 [O] - SCT_OUT0 : PWM0
*  P0.15 [O] - SCT_OUT1 : PWM1
*
*****************************************************************************/
#include "LPC8xx.h"

void SCT_Init(void)
{
    LPC_SYSCON->SYSAHBCLKCTRL |= (1 << 7) | (1 << 8);       // enable the SWM and SCT clock

    LPC_SWM->PINASSIGN6 &= ((PWM0 << 24) | 0x00FFFFFF);     // SCT_OUT0 = PWM0
    LPC_SWM->PINASSIGN7 &= ((PWM1 << 0)  | 0xFFFFFF00);     // SCT_OUT1 = PWM1

/*****************************************************************
*  SCT_L: low part configuration:
*****************************************************************/

    LPC_SCT->CONFIG           |= (1 << 17);                 // auto limit _L (on match 0)
    LPC_SCT->OUTPUT           |= (0 << OUT0);               // preset OUT0 low
//  LPC_SCT->OUTPUT           |= (1 << OUT0);               // preset OUT0 high (for low active signal)
    LPC_SCT->OUTPUTDIRCTRL    |= (0x1 << 0);                // reverse OUT0 set/clr when counter _L is
                                                            // down counting (center aligned mode)
    LPC_SCT->CTRL_L           |= (1 << 4);                  // bi-dir count mode

    LPC_SCT->MATCH[0].L        = PWM0_PERIOD - 1;           // match 0 @ PWM0 freq
    LPC_SCT->MATCHREL[0].L     = PWM0_PERIOD - 1;
    LPC_SCT->MATCH[1].L        = 0;                         // use match 1 for PWM0 duty cycle
    LPC_SCT->MATCHREL[1].L     = 0;                         // PWM0 off after power-up/reset

    LPC_SCT->EVENT[0].STATE    = 0x00000003;                // event 0 happens in all states (both 0 and 1)
    LPC_SCT->EVENT[0].CTRL     = (1 << 0)  |                // MATCHSEL[3:0]   = related to match 1
```

```
                                (0 << 4)  |                 // HEVENT[4]       = ev 0 belongs to the L timer
                                (1 << 12) |                 // COMBMODE[13:12] = match condition only
                                (0 << 14) |                 // STATELD[14]     = STATEV is added to state
                                (0 << 15);                  // STATEV[15]      = 0 (no change)

    LPC_SCT->OUT[OUT0].SET     = (1 << 0);                  // event 0 sets the OUT0 signal
//  LPC_SCT->OUT[OUT0].CLR     = (1 << 0);                  // ev 0 clears the OUT0 signal (low active mode)

/*********************************************************************
*  SCT_H: high part configuration:
*********************************************************************/

    LPC_SCT->CONFIG           |= (1 << 18);                 // auto limit _H (on match 0)
    LPC_SCT->OUTPUT           |= (0 << OUT1);               // preset OUT1 low
//  LPC_SCT->OUTPUT           |= (1 << OUT1);               // preset OUT1 high (for low active signal)
    LPC_SCT->OUTPUTDIRCTRL    |= (0x2 << 2);                // reverse OUT0 set/clr when counter _H is
                                                            // down counting (center aligned mode)
    LPC_SCT->CTRL_H           |= (1 << 4);                  // bi-dir count mode

    LPC_SCT->MATCH[0].H        = PWM1_PERIOD - 1;           // match 0 @ PWM0 freq
    LPC_SCT->MATCHREL[0].H     = PWM1_PERIOD - 1;
    LPC_SCT->MATCH[1].H        = 0;                         // use match 1 for PWM1 duty cycle
    LPC_SCT->MATCHREL[1].H     = 0;                         // PWM1 off after power-up/reset

    LPC_SCT->EVENT[1].STATE    = 0x00000003;                // event 1 happens in all states (both 0 and 1)
    LPC_SCT->EVENT[1].CTRL     = (1 << 0)  |                // MATCHSEL[3:0]   = related to match 1
                                 (1 << 4)  |                // HEVENT[4]       = ev 1 belongs to the H timer
                                 (1 << 12) |                // COMBMODE[13:12] = match condition only
                                 (0 << 14) |                // STATELD[14]     = STATEV is added to state
                                 (0 << 15);                 // STATEV[15]      = 0 (no change)

    LPC_SCT->OUT[OUT1].SET     = (1 << 1);                  // event 1 sets the OUT1 signal
//  LPC_SCT->OUT[OUT1].CLR     = (1 << 1);                  // ev 1 clears the OUT1 signal (low active mode)

    LPC_SCT->CTRL_L           &= ~(1 << 2);                 // start the _L counter
    LPC_SCT->CTRL_H           &= ~(1 << 2);                 // start the _H counter
}
```

图55.　在_L 和_H 时双通道居中对齐的 PWM

22. 法律信息

22.1 定义

初稿 — 本文仅为初稿版本。内容仍在内部审查，尚未正式批准，可能会有进一步修改或补充。恩智浦半导体对本文信息的准确性或完整性不做任何说明或保证，并对因使用此信息而带来的后果不承担任何责任。

22.2 免责声明

有限担保和责任 — 本文中的信息据信是准确和可靠的。但是，恩智浦半导体对此处所含信息的准确性或完整性不做任何明示或暗示的说明或保证，并对因使用此信息而导致的后果不承担任何责任。恩智浦半导体不对本文中非源自恩智浦半导体的信息内容负责。

在任何情况下，对于任何间接、意外、惩罚性、特殊或衍生性损害（包括但不限于利润损失、积蓄损失、业务中断、因拆卸或更换任何产品而产生的开支或返工费用），无论此等损害是否基于侵权行为（包括过失）、担保、违约或任何其他法理，恩智浦半导体均不承担任何责任。

对于因任何原因给客户带来的任何损害，恩智浦半导体对本文所述产品的总计责任和累积责任仅限于恩智浦商业销售条款和条件所规定的范围。

修改权利 — 恩智浦半导体保留对本文所发布的信息（包括但不限于规格和产品说明）随时进行修改的权利，恕不另行通知。本文件将取代并替换之前就此提供的所有信息。

适宜使用 — 恩智浦半导体产品并非设计、授权或担保适合用于生命保障、生命关键或安全关键系统或设备，亦非设计、授权或担保适合用于在恩智浦半导体产品失效或故障时会导致人员受伤、死亡或严重财产或环境损害的应用。恩智浦半导体及其供应商对在此类设备或应用中加入与/或使用恩智浦半导体产品不承担任何责任，客户需自行承担因加入与/或使用恩智浦半导体产品而带来的风险。

应用 — 本文件所载任何产品的应用只用于例证目的。此类应用如不经进一步测试或修改用于特定用途，恩智浦半导体对其适用性不做任何说明或保证。

客户负责自行利用恩智浦半导体的产品进行设计和应用，对于应用或客户产品设计，恩智浦半导体无义务提供任何协助。客户须自行判断恩智浦半导体的产品是否适用于其应用和设计计划，以及是否适用于其第三方客户的规划应用。客户须提供适当的设计和操作安全保障措施，以降低与应用和产品相关的风险。

对于因客户应用或产品的任何缺陷或故障，或者客户的第三方客户的应用或使用导致的任何故障、损害、开支或问题，恩智浦半导体均不承担任何责任。客户负责对自己基于恩智浦半导体的产品的应用和产品进行所有必要测试，以避免这些应用和产品或者客户的第三方客户的应用或使用存在任何缺陷。恩智浦不承担与此相关的任何责任。

出口管制 — 本文件以及此处所描述的产品可能受出口法规的管制。出口可能需要事先经相关主管部门批准。

评估版产品 — 该产品仅用于评估目的，保持“原样”并“包含所有缺陷”。恩智浦半导体、其关联公司及其供应商明确拒绝承担任何担保（无论是明确、暗示或明文的），包括但不限于默认的不侵权性、适销性和特殊用途适用性担保。由于本产品的质量、使用和性能所引起的一切风险均由客户自行承担。

任何情况下，对于任何因产品的使用或无法使用造成的特殊、间接、衍生、惩罚性或其他损害（包括但不限于业务损失、业务中断、无法使用、数据或信息丢失以及类似损失），无论是否基于侵权行为（包括过失）、严格的法律责任、合同违反、担保违反或其他任何理论，恩智浦、其关联公司或供应商均不对客户承担责任，即使已获悉此类损害的可能性亦不例外。

对于因任何原因给客户带来的任何损害（包括但不限于以上列出的所有损害以及所有直接或普通损害），恩智浦半导体、其关联公司及供应商和客户对所有上述损害的唯一补救措施仅限于客户产生的实际损失，并应基于合理的信赖，不超过客户实际上为产品所支付的金额或 5 美元 (US$5.00) 中的较大者。前述限制、例外情况和免责声明将适用于适用法律允许的最大范围，即使补救措施不能达到其根本目的。

22.3 商标

注意：所有引用的品牌、产品名称、服务名称以及商标均为其各自所有者的财产。

23. 内容

注意：关于本文及相关产品的重要说明详见“法律信息”一节。

欲了解更多信息，请访问：http://www.nxp.com
欲咨询销售办事处地址，请发送电子邮件至：salesaddresses@nxp.com

发布日期：2016 年 9 月
文档号：AN11538

欲了解更多信息，请访问：http://www.nxp.com
欲咨询销售办事处地址，请发送电子邮件至：salesaddresses@nxp.com

发布日期：2016 年 9 月
文档号：AN11538