AN11538

SCTimer/PWM cookbook

Rev. 3.1 — 18 February 2015 Application note

Document information

Info	Content
Keywords	LPC81x, LPC82x, LPC11U6x, LPC11E6x LPC15xx, LPC54xxx, LPC18S/43Sxx, LPC18/43xx State Configurable Timer, SCT, SCTimer/PWM
Abstract	This application note is a collection of examples and usage notes for the SCTimer/PWM block used in NXP microcontrollers

Revision history

Rev	Date	Description
3.1	20150218	Added support for LPC18S/43Sxx
3.0	20141104	Added support for LPC54xxx.
2.0	20140903	Updated with LPCOpen, added support for LPC82x.
1.0	20140821	Initial revision.

Contact information

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

1.1 Overview

The State Configurable Timer (SCTimer/PWM) is a peripheral that is unique to NXP Semiconductors. It can operate like most traditional timers, but also adds a state machine to give it a higher degree of configurability and control. This allows the SCT to be configured as multiple PWMs, a PWM with dead-time control, and a PWM with reset capability, as well as many other configurations that cannot be duplicated with traditional timers. Once the SCTimer/PWM has been configured, it can run totally autonomously from the microcontroller core, unless an SCTimer/PWM interrupt has been enabled which requires that the core service the interrupt.

Table 1 below gives an overview of the controller families that contain the SCTimer/PWM block (one or more) and the way they are synthesized (showing the available number of main resources like inputs, outputs, states, etc).

Table 1. SCTimer/PWM resources for each family

NXP part	Inputs	Outputs	States	Events	Match/ capture	SCTIPU	Dithering	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 (flashless)	8	16	32	16	16	☒	☒	☒
LPC18/43xx (flash)	8	16	32	16	16	☒	☑	☒
LPC18S/43Sxx (flashless)	8	16	32	16	16	☒	☒	☒
LPC18S/43Sxx (flash)	8	16	32	16	16	☒	☑	☒
LPC54xxx	8	8	13	13	13	☒	☒	☒

Additional features of the SCTimer/PWM block are:

- Inputs and outputs can be routed to external pins and internally to other peripherals.
- If more SCTs available (like on LPC15xx) then SCTimer/PWM outputs are internally connected to other SCTimer/PWM inputs.
- Each SCTimer/PWM can be used as one 32-bit counter or split into two 16-bit counters.
- Clocked by bus clock, selected input or separate SCTPLL (on LPC15xx SCT0/1).
- Up counters or up-down counters.
- State variable allows sequencing across multiple counter cycles.
- Input Pre-processor Unit (on LPC15xx) for processing SCTimer/PWM inputs and handling SCTimer/PWM aborts.
- The following conditions define an event: a counter match condition, an input (or output) condition, a combination of a match and/or and input/output condition in a specified state, and the count direction.

- Events control outputs, interrupts, DMA requests and the SCTimer/PWM states. Also:
  - Match register 0 can be used as an automatic limit.
  - In bi-directional mode, events can be enabled based on the count direction.
  - Match events can be held until another qualifying event occurs.
  - Selected events can limit, halt, start, or stop a counter.

This “cookbook” will give insight into the various ways that the SCTs can be used, but this is in no way an exhaustive list of potential applications for this unique peripheral.

Each example shows the SCTimer/PWM’s used resources, the configuration code and in addition to that, it configures, if available, any SCTimer/PWM inputs or outputs using the Switch Matrix, the SCTimer/PWM Input multiplexers and the SCTimer/PWM Input Processing Unit.

1.2 Terminology

The first time you look at the SCTimer/PWM, it may appear to be a very complex peripheral, but you will see that it is actually not that difficult to use, even when not using a design tool like Red State that is available in LPCXpresso IDE. It may be useful to review some terminology which you will see in this document, as well as the NXP User Manuals, that you may not encounter when dealing with other timers.

Limit – a limit is another name for a condition or event that causes the counter to be cleared to zero when operated in unidirectional mode, or to change the direction of count in bi-directional mode. For example, if a timer match occurs, this can (but does not have to) limit the counter. You can think of a limit condition as a kind of timer reset. The SCTimer/PWM limit register defines which events cause a limit condition.

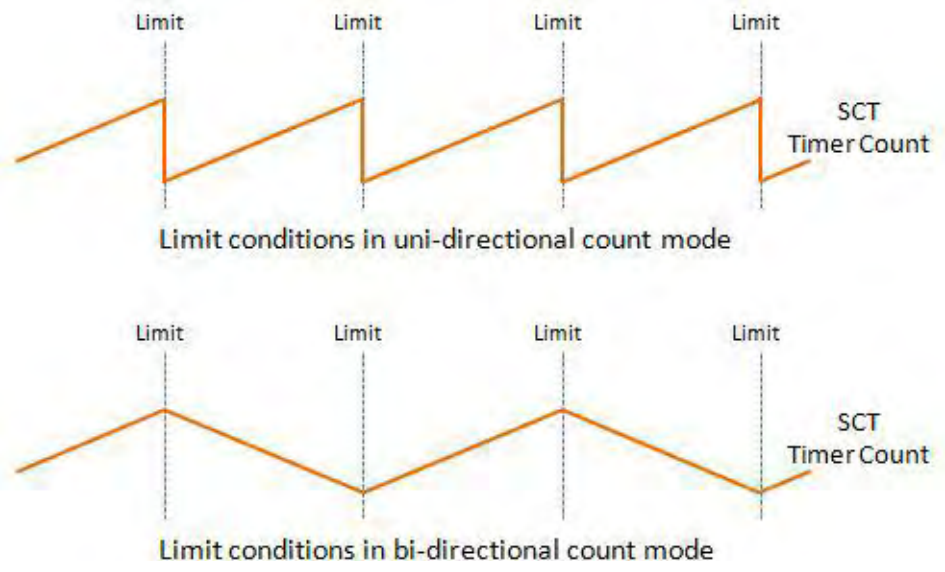


Fig 1. Timer limit in unidirectional and bidirectional counting mode

Event – understanding events is critical to understanding the SCTimer/PWM. The following conditions define possible events: a counter match condition, an input (or output) condition, a combination of a match and/or an input/output condition in a specified state, and the count direction. Events can control outputs, interrupts, DMA requests and the SCTimer/PWM states. They can also cause timer limit, halt, start, or stop conditions to occur.

STOP – when the SCTimer/PWM timer(s) have been stopped, the counter does not run, but I/O events related to the counter can still occur. If an event occurs that is enabled in the START register, the counter will resume running. The STOP condition is controlled by the STOP_L and STOP_H bits in the SCTimer/PWM control register. The STOP bits can be cleared by events or by software.

START – if the SCTimer/PWM has been stopped, it can be started again by an event. The START register determines which events can start the timer.

HALT – a HALT is similar to STOP; however, an event cannot restart the timer. Therefore, only software can be used to unhalt the timer. If you review the example code that is included with this document, you will see that user software needs to clear the halt condition in the control register to start the counting process.

Unified Timer – the SCTimer/PWM has one 32-bit counter. This counter can be configured as one 32-bit counter (also called a “unified” counter), or it can be used as two 16-bit counters.

State – The state variable is the main feature that distinguishes the SCTimer/PWM from other counter/timer/PWM blocks. Events can be made to occur only in certain states. Events, in turn, can perform the following actions:

- set and clear outputs
- limit, stop, and start the counter
- cause interrupts
- modify the state variable

The value of a state variable is completely under the control of the application. If an application does not use states, the value of the state variable remains zero, which is the default value. A state variable can be used to track and control multiple cycles of the associated counter in any desired operational sequence. The state variable is logically associated with a state machine diagram which represents the SCTimer/PWM configuration.

1.3 Target hardware

Most of the examples either use LPCXpresso V2 or LPCXpresso MAX board as target/test hardware. For the schematics of these boards please refer to:

http://www.lpcware.com/LPCXpressoBoards

2. Repetitive interrupt

2.1 Purpose

The SCTs can perform the same simple functions performed by a typical timer found in most microcontrollers. The timer in SCTimer/PWM can be configured to operate as two 16-bit timers, or as a “unified” 32-bit timer. This example uses the unified 32-bit timer mode to generate SCTimer/PWM interrupt every 10 milliseconds. The SCTimer/PWM interrupt handler (in user code) will count the number of times it has been called, and will toggle the GPIO (LED) every 20 interrupt cycles, or 200 milliseconds.

2.2 Configuration

This example (*SCTx_repetitive_irq*) uses Match register MATCH[0].U to trigger event0 which auto limits (resets) the counter and generates an interrupt (SCT_IRQ).

This example only uses 1 match and 1 event (no states, no inputs and no outputs).

```
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
}
```

Fig 2. Code for SCT_repetitive_irq

3. Blinky match

3.1 Purpose

In this example (*SCTx_blinky_match*), we use the unified 32-bit timer to toggle SCTx_OUT0 which is connected to an output port of the controller (for example connected to LED). The timer state will change every 100 milliseconds. Although it is not necessary to use multiple states to create a toggling output on SCTx_OUT0, the example illustrates the use of two states.

3.2 Configuration

- Match used: MATCH[0].U at 100 msec.
- Output used: SCTx_OUT0 connected to an LED that is illuminated when the output is low (during state 0).
- Event used: Event 0 and 1
- State used: State 0 and 1

Fig 3 below illustrates what we would like to achieve with the SCTimer/PWM. The red line shows the SCTimer/PWM counting up, until it reaches the match value, where it limits back to 0. After each limit, the SCTimer/PWM output should toggle. The state should be 0 when the output is low and it should be 1 when the output is high.

You can see that both event0 and event1 occur on a timer MATCH0. Event0 only happens in state0 and changes to state1. Event1 only happens in state1 and changes the state back to zero. The unified timer will limit (reset to zero) at both events, and the output SCTx_OUT0 (SCTx used in the example below) will be set or reset.

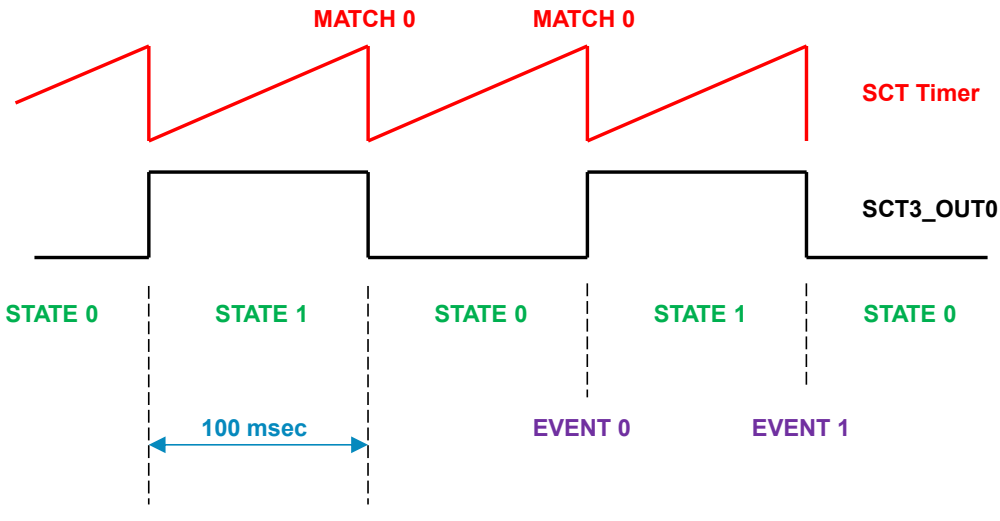


Fig 3. Blinky match illustration

3.3 Initialization code

```
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
}
```

Fig 4. Code for SCT_blinky_match

Remark: For LPC54xxx SCT_OUT[5] is used, since SCT_OUT[0] is not connected to LED on LPC54xxx LPCXPresso V2 board.

Match toggle

3.4 Purpose

The SCTimer/PWM has the capability to set or clear an output directly using the SET and CLR registers, but it does not have a way to directly toggle the outputs. We are going to look at how we can toggle an output using the Conflict Resolution register and demonstrate how the previous example can be built using only one event using no states. In addition this example (*SCTx_match_toggle*) will use only the lower 16-bit counter, rather than using the 32-bit unified counter used in the previous examples. The output should toggle SCTx_OUT0 every 100 milliseconds.

3.5 Configuration

- Match used: MATCH[0].L at 100 msec.
- Output used: SCTx_OUT0 toggling every time event 0 occurs
- Event used: Event 0 (triggered by match 0 condition only)
- State used: none

MATCH[0].L register is used to achieve a match every 100 msec. When a match occurs, the timer **auto** limits (resets) and generates event 0. Event 0 toggles SCTx_Out0, using the Conflict Resolution register.

The only issue with using a 12 MHz clock and a 16-bit counter is that the maximum delay time is about 5.5 milliseconds. Therefore, we will need to use the SCTimer/PWM prescaler. There is a separate 8-bit pre-divider for each of the 16-bit timers.

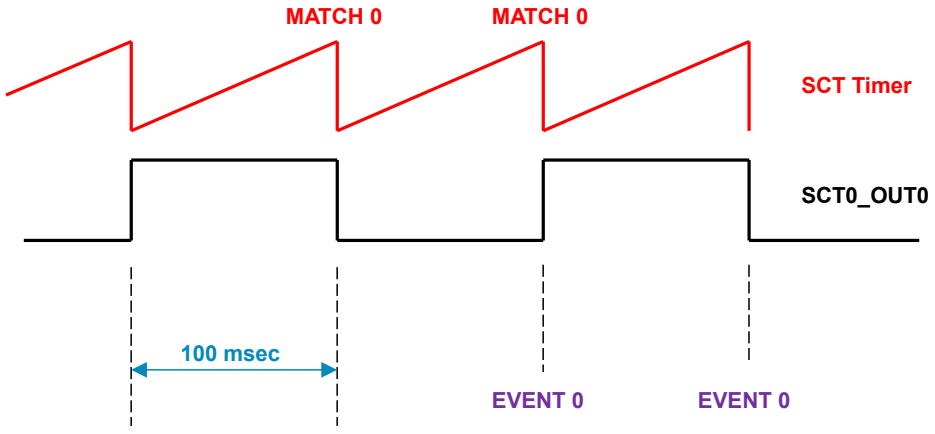


Fig 5. Match toggle illustration (SCT0 is used)

3.6 Setting the SCTimer/PWM prescaler

The SCTimer/PWM prescaler is used to allow a 100 millisecond match interval with the 16-bit LOW counter. To keep this simple, we set the SCTimer/PWM input clock to 100 kHz by dividing the 12 MHz main clock by 120.

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

3.7 Initialization code

```
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
```

Fig 6. Code for SCT_match_toggle

Remark: For LPC54xxx SCT_OUT[5] is used, since SCT_OUT[0] is not connected to LED on LPC54xxx LPCXPresso V2 board.

3.8 Using the conflict resolution register

As shown in Fig 6, the output pin 0 is both set and cleared by event 0. When an event does both set and clear an output, the conflict resolution register is used to decide what will happen for this conflict.

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.

Fig 7. Conflict resolution register

In the *SCTx_match_toggle* example, the conflict resolution register uses the value of 0x03 which tells the SCTimer/PWM to toggle the output.

4. Using the SCTPLL

4.1 Purpose

Like some other parts, the LPC15xx has a dedicated built-in PLL to create the clock for SCT0 and/or SCT1. This example (*SCT1_use_PLL*) is using the SCTPLL to generate a 72 MHz input clock to SCT1, while the system clock is at 12 MHz derived from the IRC. The SCTPLL input clock has a fixed connection to SCT1 input 7.

4.2 Configuration

The code is based on the previous example. It is using SCT1 and is tested on an LPCXpresso board with an LPC1549 running at 12 MHz. It uses SCT1_IN7 to receive a 72 MHz clock from the SCTPLL and the unified timer and MATCH[0].U register to achieve a match every 100 msec. When a match occurs, the timer auto limits (resets) and generates event 0. Event 0 toggles SCT1 output 0 (connected to P0_24 green LED).

4.3 Set up the SCTPLL

Power up and enable the SCTimer/PWM PLL running at 72 MHz:

```
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
```

Use the global CONFIG register to use the SCTimer/PWM PLL at input 7:

```
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)
```

4.4 Initialization code

```
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
}
```

Fig 8. PWM (using PLL) initialization code

5. Simple PWM

5.1 Purpose

This example (*SCTx_pwm*) uses the low 16-bit SCTimer/PWM timer to generate a 100 kHz PWM signal at SCTx_OUT0. Two pushbuttons (SW1 and SW2/SW3 on the LPCXpresso board) are used to decrease and increase the duty cycle of the PWM signal by updating the MATCHRELOAD register. Since the LPC812 LPCXpresso board doesn't have pushbuttons on it, the trimmer R38 is used to decrease (anticlock-wise) and increase (clock-wise) the duty cycle of the PWM signal. By connecting SCTx_OUT0 to a LED, this will adjust the brightness of the LED.

5.2 Configuration

- Match used: Match 0 for PWM period and Match 1 for PWM duty cycle.
- Output used: SCTx_OUT0 for PWM output signal
- Event used: Event 0 and Event 1
- State used: none

The SCTimer/PWM input clock is pre-scaled to 1 MHz. It uses MATCH[0].L = 10 (1 MHz / 100 kHz) to generate a 100 kHz timer match; this will auto limit (reset) the counter and generate event 0. Event 0 will then set SCTimer/PWM output 0 to a logic high level. The MATCH[0].L register defines the period length of the PWM signal. A second match register MATCH[1].L is used to define the duty cycle of the signal. When match event 1 is occurs, it will clear SCTimer/PWM output 0. Fig 9 shows the waveforms for this example.

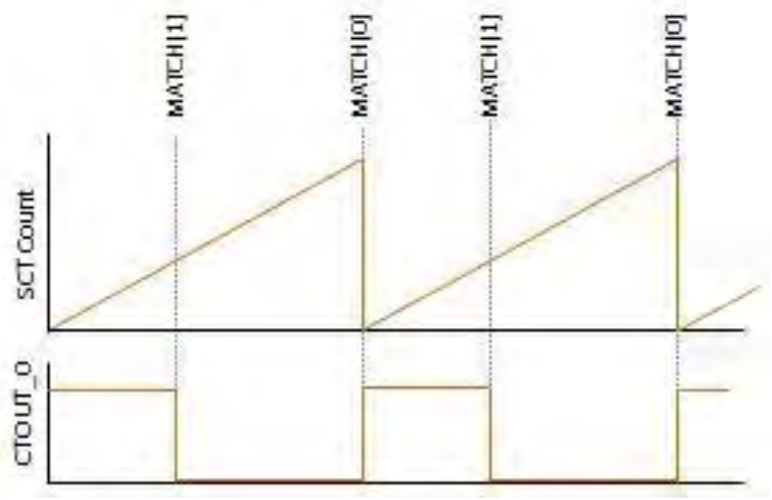


Fig 9. PWM counter operation and output

The application code uses (one GPIO input connected to a trimmer R38 on LPC812 LPCXpresso or two GPIO inputs connected to SW1 and SW2/SW3 on other LPCXpresso boards) to control the duty cycle of the PWM output signal. Every time SW1 goes high to low (falling edge) it increases the duty cycle (in 10 steps) intern decreases the LED brightness. And every time SW2/SW3 goes high to low it decreases the duty cycle (in 10 steps) intern increases the LED brightness.

Note: On LPC82x LPCXpresso board switch SW2 and red LED are connected on the same port pin, hence one might see unwanted red LED flashing while decreasing the duty cycle or increasing the LED brightness.

5.3 Configuration code

```
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
}
```

Fig 10. Simple PWM configuration code

Remark: For LPC54xxx SCT_OUT[5] is used, since SCT_OUT[0] is not connected to LED on LPC54xxx LPCXPresso V2 board.

6. Center aligned PWM

6.1 Purpose

The SCTimer/PWM has the capability to count up to a limit and then down to zero. In this case you can use the Output Direction Control register to specify (for each output) the impact of the counting direction on the meaning of set and clear operations on that output.

This example (*SCTx_pwm_center_aligned*) is using that feature to generate a center aligned PWM output. It demonstrates how the previous example can be built using just one event. It again uses the low 16-bit SCTimer/PWM timer to generate a 100 kHz PWM signal at SCTx_OUT0. Two pushbuttons (SW1 and SW2/SW3 on the LPCXpresso board) are used to decrease and increase the duty cycle of the PWM signal by updating the MATCHRELOAD register. Since the LPC812 LPCXpresso board doesn't have pushbuttons on it, the trimmer R38 is used to decrease (anticlock-wise) and increase (clock-wise) the duty cycle of the PWM signal. By connecting SCTx_OUT0 to an LED, this will adjust the brightness of the LED.

6.2 Configuration

The SCTimer/PWM input clock is now pre-scaled to 2 MHz. It uses MATCH[0].L = 10 to generate a timer limit that changes the counting direction from up to down counting. So the total PWM period is 20 clocks, 10 usec (100 kHz). A second match register MATCH[1].L is used to define the duty cycle of the signal. When match event 1 occurs, it will set SCTimer/PWM output 0 when up counting and clear (reverse) the output when down counting. Fig 11 shows the waveforms for this example.

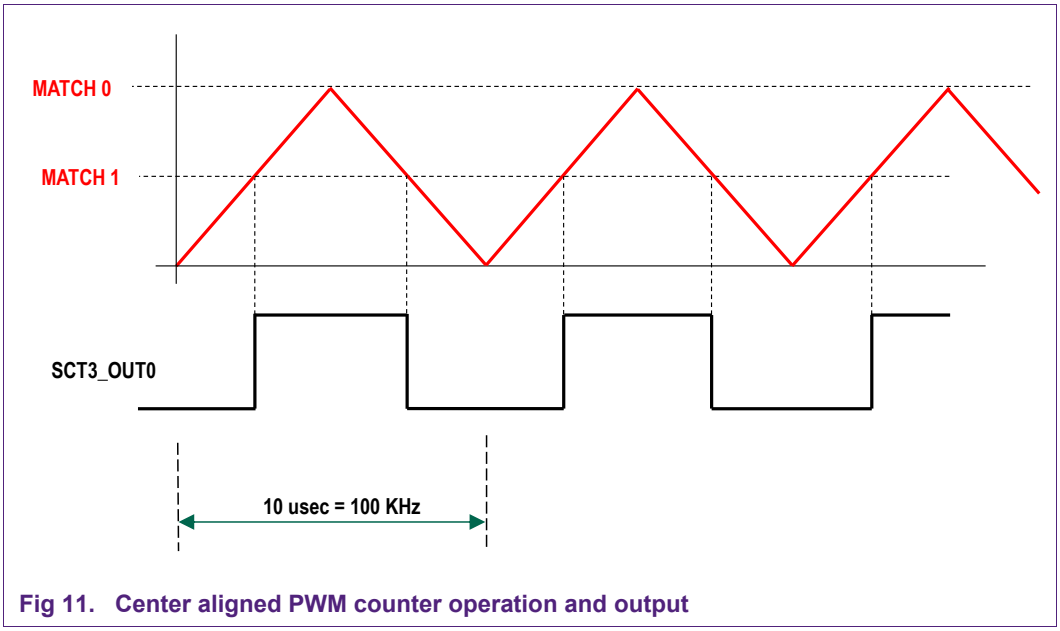


Fig 11. Center aligned PWM counter operation and output

The application code uses (one GPIO input connected to a trimmer R38 on LPC812 LPCXpresso or two GPIO inputs connected to SW1 and SW2/SW3 on the other LPCXpresso boards) to control the duty cycle of the PWM output signal. Every time SW1 goes high to low (falling edge) it increases the duty cycle (in 10 steps) intern decreases

the LED brightness. And every time SW2/SW3 goes high to low it decreases the duty cycle (in 10 steps) intern increases the LED brightness.

6.3 Configuration code

```
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
}
```

Fig 12. Center aligned PWM initialization code

Remark: For LPC54xxx SCT_OUT[4] is used, since SCT_OUT[0] is not connected to LED on LPC54xxx LPCXPresso V2 board.

6.4 Using bidirectional output control

In Fig 11, you will see that the timer is counting up and down. Output pin 0 must be set at MATCH1 during up counting, but must be reset at same match and event during down counting. This can be accomplished using the bidirectional output control register.

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.	

Fig 13. Conflict resolution register

In this example the OUTPUTDIRCTRL register uses the value of 0x1 which tells the SCTimer/PWM to reverse output 0 set and clear when counting down.

7. Two-channel PWM

7.1 Purpose

This example (*SCT_pwm_2ch*) shows the generation of two PWM signals with different duty cycles. It uses the unified 32-bit timer mode. A GPIO input assigned to SCTimer/PWM input 0 (SCT_IN0) selects which of the output signals is active. A trimmer (R38) on LPC812 LPCXpresso board or switch SW1 on other LPCXpresso boards is used to select between the green and red/blue flashing LEDs. While in some hardware boards (like LPC11U6x LPCXpresso) wherein GPIO (SW1) input can't be used for SCT_IN0, the output signal activation (red/blue and green LEDs flashing is time multiplexed). Initially red/blue LED flashes for few seconds and then the green.

This example is initially built using the graphical Red State tool, (see Fig 14). It is using the ALWAYS state (U_ALWAYS for unified counter and L_ALWAYS, H_ALWAYS for 16-bit implementations). The ALWAYS state is not included as one of the states by the SCTimer/PWM, so you still have all states available for each split timer. ALWAYS is a condition that can occur in any state.

Auto-limit is selected in the SCTimer/PWM configuration register to allow match register 0 to cause a limit condition. The green LED flashes with a short duty cycle, while the red/blue LED flashes with a long duty cycle.

7.2 Configuration

- Input(s) used: SCT_IN0 (SW1 or R38)
- Output(s) used: SCTx_OUT0 (green LED) and SCTx_OUT1 (red/blue LED)
- Match used: Match 0 to 4
- Event used: Event 0 to 5
- State used: State 0 and 1

7.3 Red State diagram

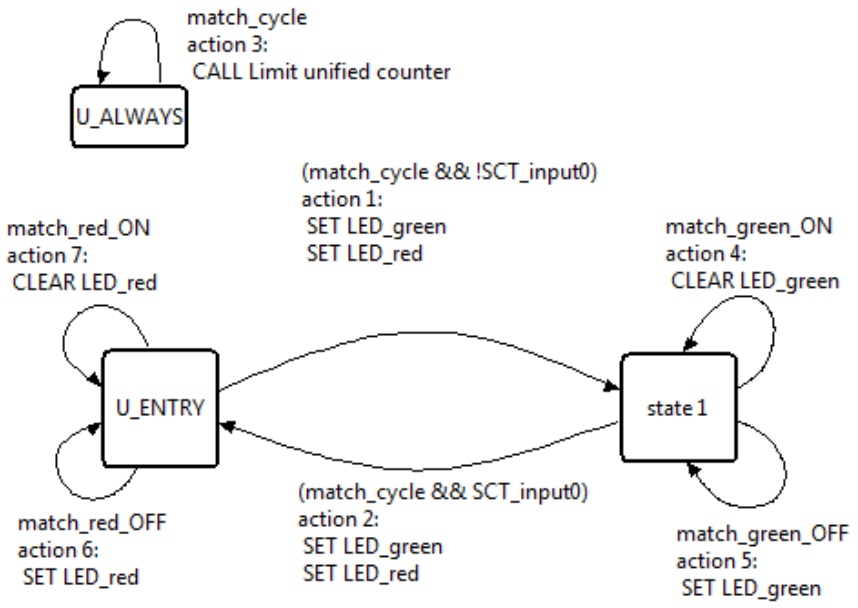


Fig 14. Red State diagram for SCT_pwm_2ch

7.4 Initialization code

See Fig 15. This code was initially generated by the Red State tool and afterwards cleaned up and restructured.

```
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
}
```

Fig 15. Cleaned up version of code generated by Red State tool

Remark: For LPC54xxx SCT_OUT[4] and SCT_OUT[5] are used, since SCT_OUT[0] and SCT_OUT[1] are not connected to LEDs on LPC54xxx LPCXPresso V2 board.

8. PWM with deadtime

8.1 Purpose

This example (*SCTx_pwm_deadtime*) demonstrates a two-channel double-edge controlled PWM generation, intended for use as a complementary PWM pair with dead-time control. It uses the split 16-bit timer mode (low counter). The high counter could be used to generate another complementary PWM pair with dead-time control, possibly with a phase shift relative to the first pair, or for another purpose. An Abort input has also been implemented on SCT_IN0 (the LPC15xx example uses the SCTimer/PWM Input Processing Unit). The Abort input drives the outputs to their off states (Out0 = HIGH, Out1 = LOW).

When the SCTimer/PWM detects a falling edge on the ABORT pin (SCT_IN0), it will call the SCTimer/PWM interrupt that resets the counter (see SCT_IRQHandler in main.c), and clears the STOP condition. An ABORT is generated by either by a GPIO pin that is connected to SW1 on some LPCXpresso boards (LPC15xx) or internally when the board hardware doesn't support it.

8.2 Configuration

- Input(s) used: SCT_IN0 used as ABORT (from the SCTIPU in case of the LPC15xx)
- Output(s) used: SCT_OUT0 (PWM1 blue LED) and SCT_OUT1 (PWM2 red/blue LED)
- Match used: Match 0 to 2
- Event used: Event 0 to 3
- State used: none

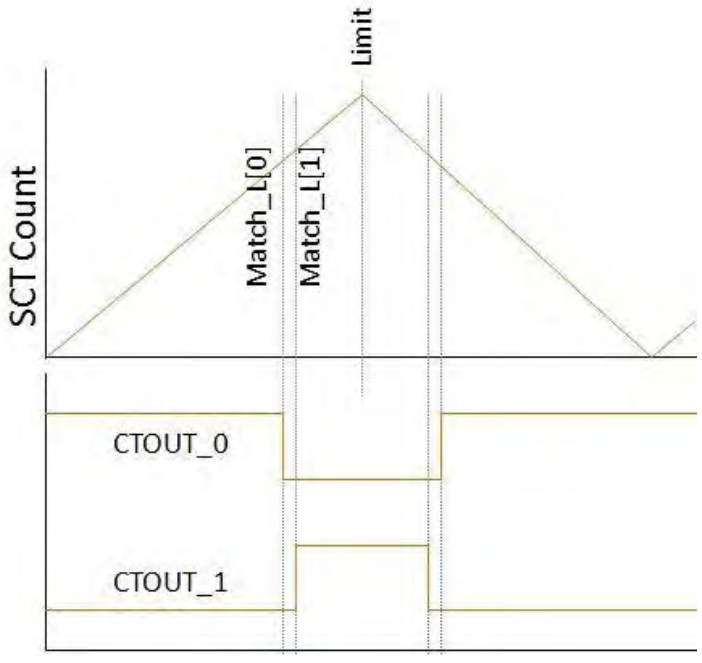


Fig 16. PWM with dead-time using bidirectional counter

8.3 LPC15xx input processing unit

To configure P1_9 as an ABORT input to SCT1_IN0 we use the code below. Note that by using the Switch Matrix any GPIO port pin can be assigned as an ABORT pin.

```
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)
```

8.4 Initialization code

Fig 17 shows the SCTimer/PWM initialization code using no states, four events and three match / match reload registers.

Remark: For LPC54xxx SCT_OUT[4] and SCT_OUT[5] are used, since SCT_OUT[0] and SCT_OUT[1] are not connected to LEDs on LPC54xxx LPCXPresso V2 board.

```
#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
}
```

Fig 17. PWM with dead-time initialization code

8.5 Adjusting the duty-cycle

The dead-time can be set by having a slight difference in the two duty cycles. Updating the duty cycle is done by:

Temporally disabling the update of the match registers of the low counter (set bit NORELOAD_L in register CONFIG).

Loading the match registers with their new values

Enabling the update of the low counter match registers again (clear bit NORELOAD_L in register CONFIG).

8.6 Result

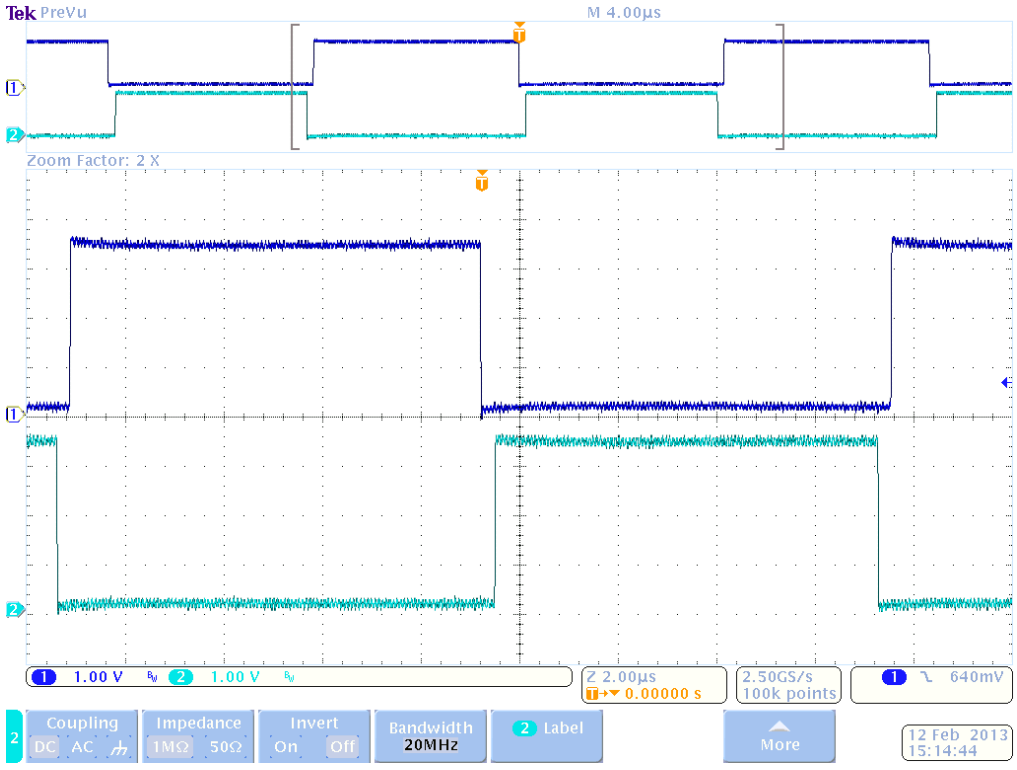


Fig 18. 2-channel PWM with dead-time control: There is a delay between one channel going low and the other channel going high

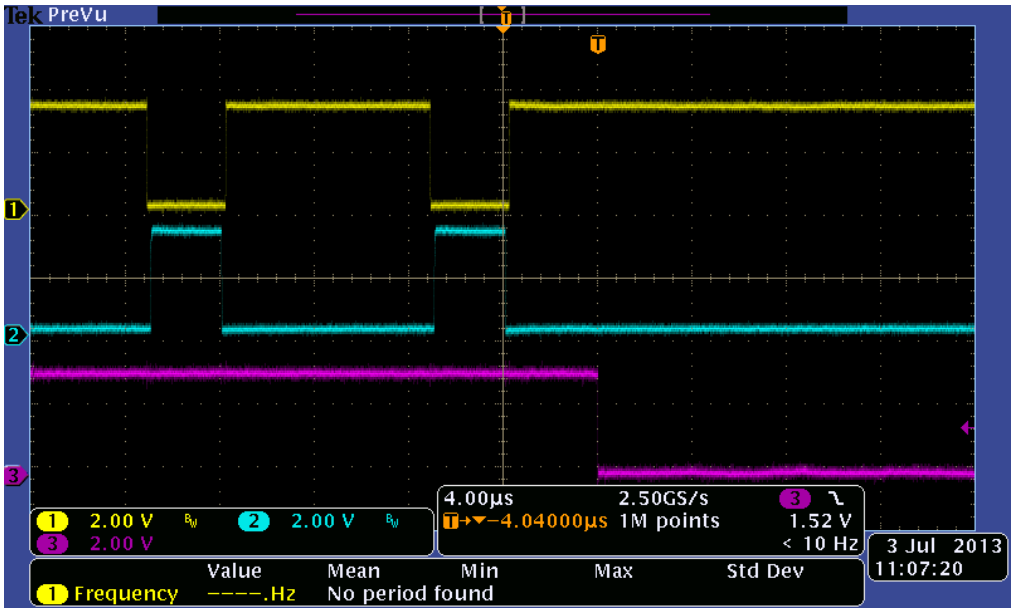


Fig 19. ABORT input goes low and stops the PWM output

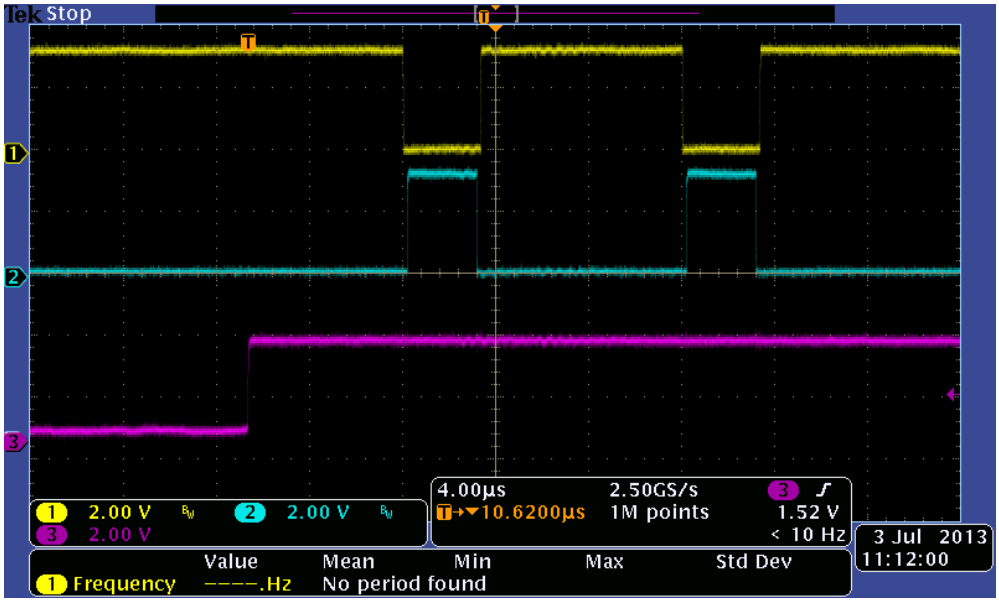


Fig 20. ABORT pin goes high and restarts the PWM

9. Match reload

9.1 Purpose

In the previous example, a PWM with a dead-time interval was implemented using the SCTimer/PWM (only L counter). Now we will demonstrate how to use the match reload registers to change the duty cycle of the two PWM signals and maintain their dead-time intervals using the NORELOAD_L bit in the SCTimer/PWM Configuration register.

9.2 Configuration

This example (*SCTx_pwm_reload*) is using the SysTick timer to generate a periodic interrupt every 20 msec. The match reload values are changed in the SysTick interrupt handler.

The application code is using GPIO input (SW1/SW3/R038) to control the duty cycle of the PWM output signal. As long as input is high it will increase the duty cycle (every 20 msec), and when input is low it will decrease the duty cycle.

- Output(s) used: SCT_OUT1 (PWM1 red LED) and SCT_OUT0 (PWM0 blue/green LED)

9.3 Initialization code

The initialization code is exactly the same as for the previous (SCTx_pwm_deadtime) example except the ABORT input is not implemented.

9.4 Updating the reload values

The updating of the reload registers occurs in the SysTick timer interrupt. The interrupt is configured to be generated every 20 msec. Setting the NORELOAD_L bit in the SCTimer/PWM Configuration register stops the match register from being updated. This allows us to update both the MATCHREL[1].L and MATCHREL[2].L, but both MATCH[1] and MATCH[2] registers do not get updated with the new values until the NORELOAD_L bit is reset to '0'. Fig 21 shows the complete SysTick timer interrupt code.

```
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
}
```

Fig 21. SysTick handler code for reloading match values

10. Four-channel PWM

10.1 Purpose

This example (SCTx_pwm_4ch) demonstrates simple four-channel PWM generation. It uses the unified 32-bit timer mode to generate single-edge aligned outputs. Channels can have different polarity. The demonstration state machine has been configured for positive pulses at SCT_OUT0/1 and negative pulses at SCT_OUT2/3.

SCT_IN0 (coming from SCTIPU and assigned to P2_5 in case of LPC15xx) is used as ABORT input. If low, it forces the outputs to their idle states, halts the timer, and generates an interrupt. For some hardware boards ABORT input is generated internally.

10.2 Configuration

- Input(s) used: SCT_IN0 (!ABORT)
- Output(s) used: SCT_OUT0 (green trace PWM1) and SCT_OUT1 (red trace PWM2)
- SCT_OUT2 (yellow trace PWM3) and SCT_OUT3 (blue trace PWM4)
- Match used: Match 0 to 4
- Event used: Event 0 to 5
- State used: none

10.3 Design

Fig 22 shows the Red State diagram for this example. However, the tool is not used to generate the SCTimer/PWM code, but just given as a reference.

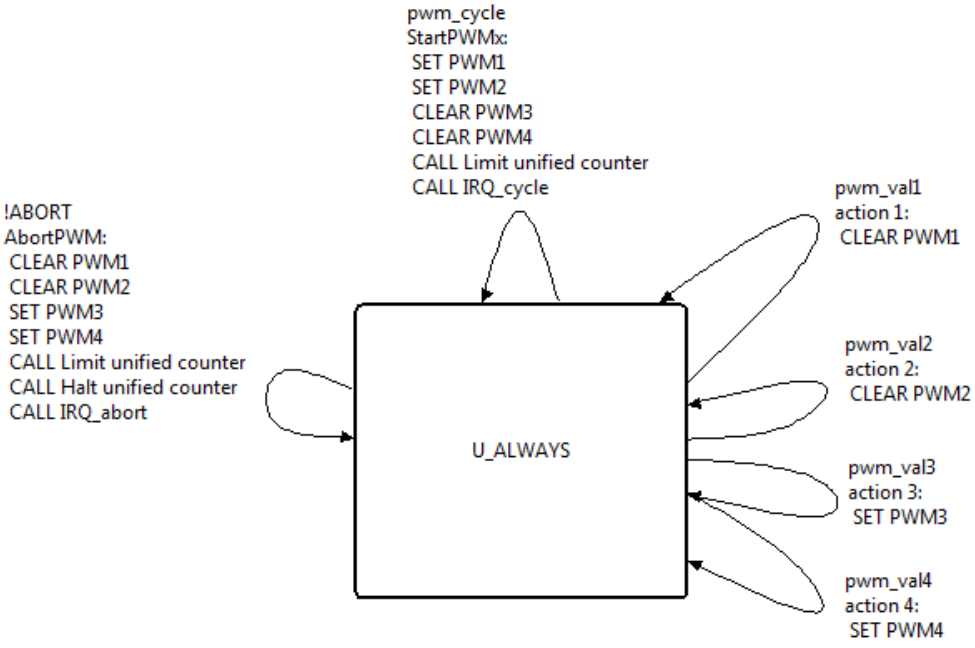


Fig 22. State diagram (Red State) 4-channel PWM generation

10.4 Initialization code

```
#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
}
```

Fig 23. 4-channel PWM Initialization code

10.5 Result

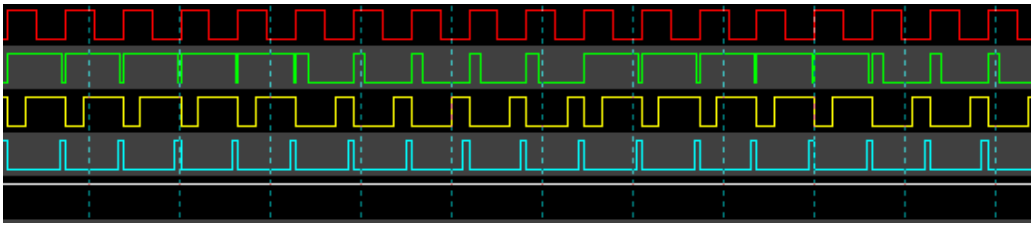


Fig 24. 4-channel PWM: duty cycles of two channels change every five PWM cycles

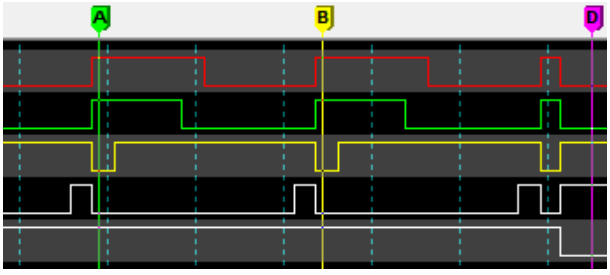
SCT_OUT1 (red trace) 50 %

SCT_OUT0 (green trace) 40 %

SCT_OUT2 (yellow trace) 10 %

SCT_OUT3 (blue trace) 90 %

ABORT (white)



Cursor positions A and B mark the early stage of two consecutive PWM cycles.

Cursor position D marks the abort state. Note that the idle level of SCT_OUT1 and SCT_OUT0 are low, while the idle level of SCT_OUT2 and SCT_OUT3 are high.

Fig 25. 4-channel PWM: duty cycles of two channels change every five PWM cycles

11. Decoding PWM

11.1 Purpose

This example (*SCTx_pwm_decode*) is using the capture and capture control features. It implements a PWM decoder which measures the duty cycle of a PWM signal and determines whether it is above (max_width) or below (min_width) a specific value. The PWM signal frequency is assumed to be 10 kHz. Two output signals (width_error and timeout) are included to indicate when the 10 kHz signal has an error or is missing.

11.2 Configuration

- Input(s) used: SCT_IN0 (apply the 10 kHz PWM signal here)
- Output(s) used:
  - SCT_OUT0, timeout indicator, low active. Output timeout activated if no edge is detected for three PWM periods.
  - SCT_OUT1, indicator for duty cycle out of bounds, low active. This output is also active when a timeout occurs.
- Match/Cap used: Match 0 to 2 and Capture 3 and 4
- Event used: Event 0 to 5
- State used: State 0 and 1

11.3 Design

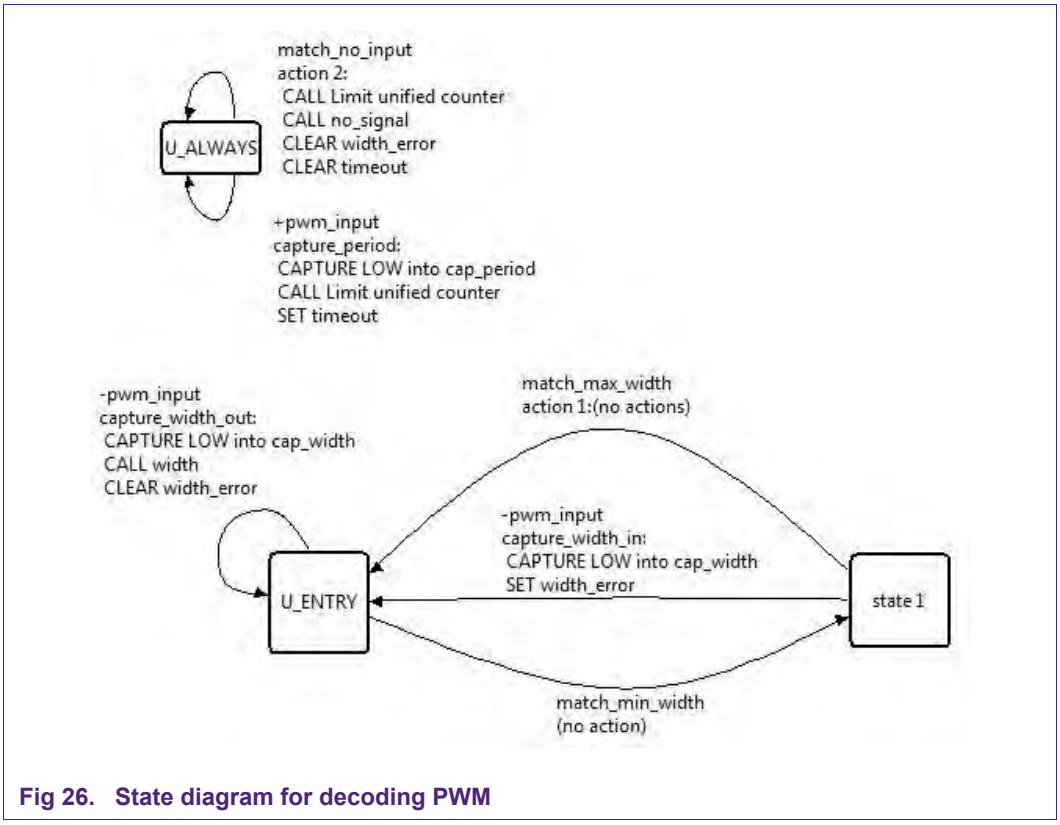


Fig 26. State diagram for decoding PWM

11.4 Initialization code

```
#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
}
```

Fig 27. Decoding PWM initialization code

12. RC5 transmission

12.1 Purpose

This example (*SCTx_rc5_send*) uses the SCTimer/PWM as an RC5 transmitter intended to drive an infrared LED for remote control. It's a very cost effective and low power alternative for older or even discontinued devices like the PCA84C122 and the SAA3010.

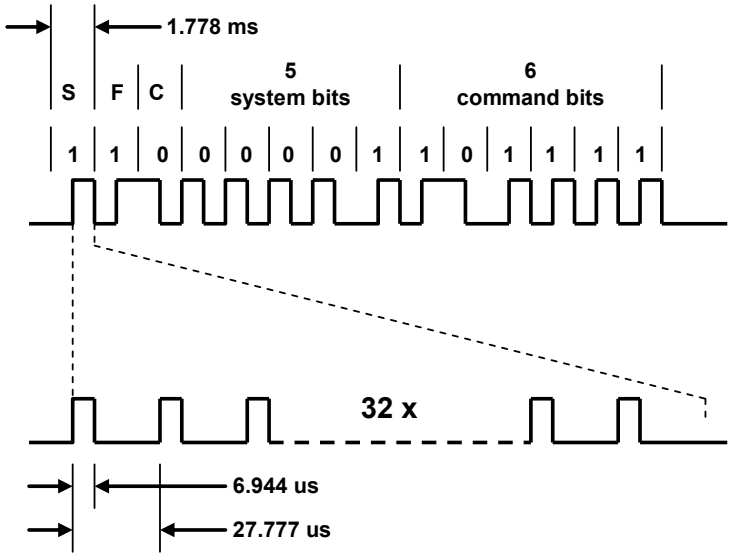


Fig 28. RC5 frame example

12.2 Configuration

Both halves of the SCTimer/PWM are used. The lower 16-bit timer is used to generate the 36 kHz modulated pulse with 25 % duty cycle. A (dummy) port pin is used as an input to the SCT. The software selects pull-up or pull-down at SCT_IN0 input pin to control the burst activation. The high part of the timer is used to send out the actual Manchester encoded data.

The MRT (Multi Rate Timer) interrupt handler (MRT_IRQHandler in main.c) is used to send the 14 data bits.

- Input(s): SCT_IN0 internally used (dummy) input that enables burst if high
- Output(s): SCT_OUT0 used as LED driver output, high active. Outputs a burst of a 36 kHz signal. Single 36 kHz pulses have 25 % duty cycle.
- Match used: Match 0 and 1
- Events used: 3
- States used: none

12.3 Design

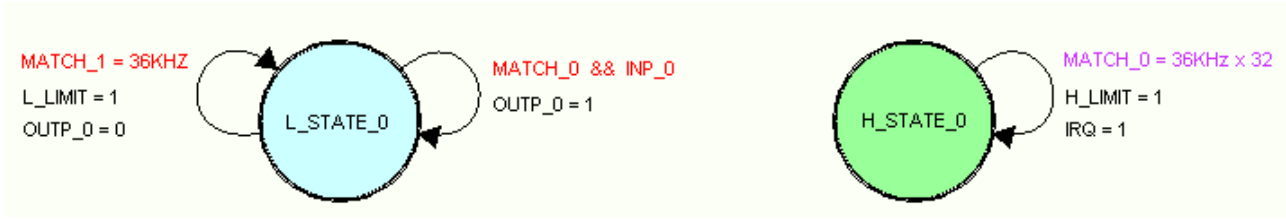


Fig 29. RC5 transmitter state diagram

12.4 Result

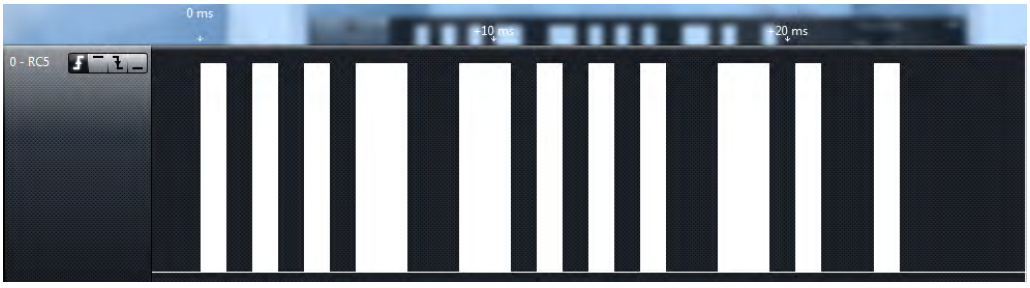


Fig 30. One complete RC5 frame measured at SCT_OUT0

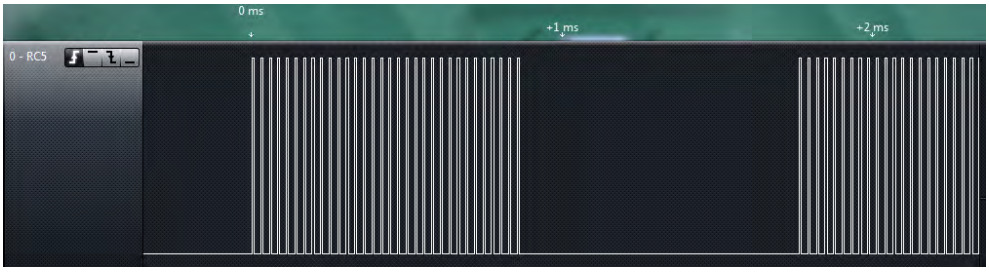


Fig 31. Burst of 32 carrier pulses for one high bit of a frame

13. RC5 receiving

13.1 Purpose

This example (*SCTx_rc5_receive*) uses the SCTimer/PWM low timer part as an RC5 receiver (Manchester decoding). Received RC5 frames are sent out over an RS232 interface using the U(S)ART0 of the microcontroller (at 19200 baud).

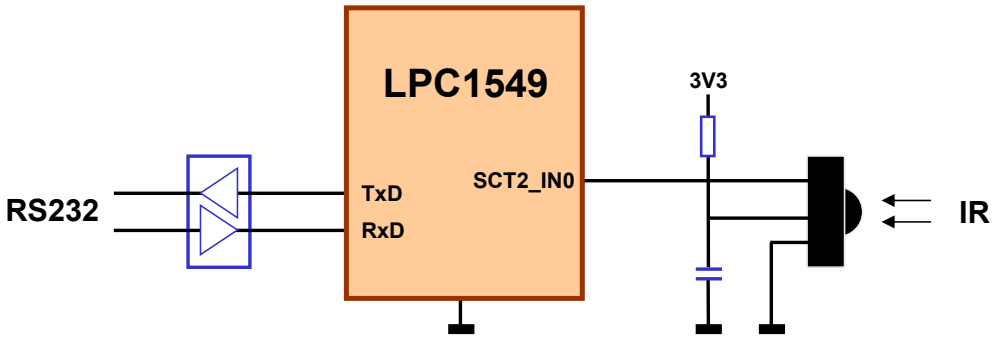


Fig 32. RC5 receiver hardware setup with LPC1549

13.2 Configuration

The hardware setup is shown in Fig 32. The SCTimer/PWM input clock is pre-scaled to 1 MHz. SCTimer/PWM input 0 is used to generate events on both rising and falling edge of the input signal. The events are used to capture the counter values, to limit (reset) the counter and to generate an interrupt at the rising edge. Inside the SCTimer/PWM interrupt handler the received data is decoded.

- Input(s): SCT_IN0 used to receive the RC5 data
- Output(s): none
- Match/Capture used: Match 0, Capture 1 and 2
- Events used: 3
- States used: none

13.3 Design

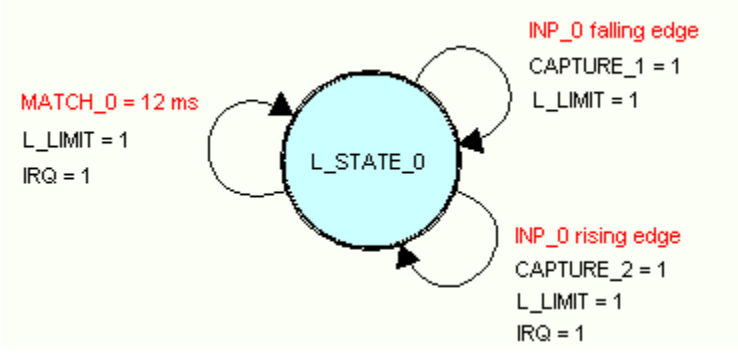


Fig 33. RC5 receiver state diagram

13.4 Initialization code

```
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
}
```

Fig 34. RC5 receiver SCTimer/PWM initialization code

13.5 Result

Received RC5 messages are send out over an RS232 interface using the U(S)ART0 of the micro. A PC running TeraTerm (19200 baud) is used to display the received data. The first value represents the RC5 system byte; the second value gives the RC5 command byte.

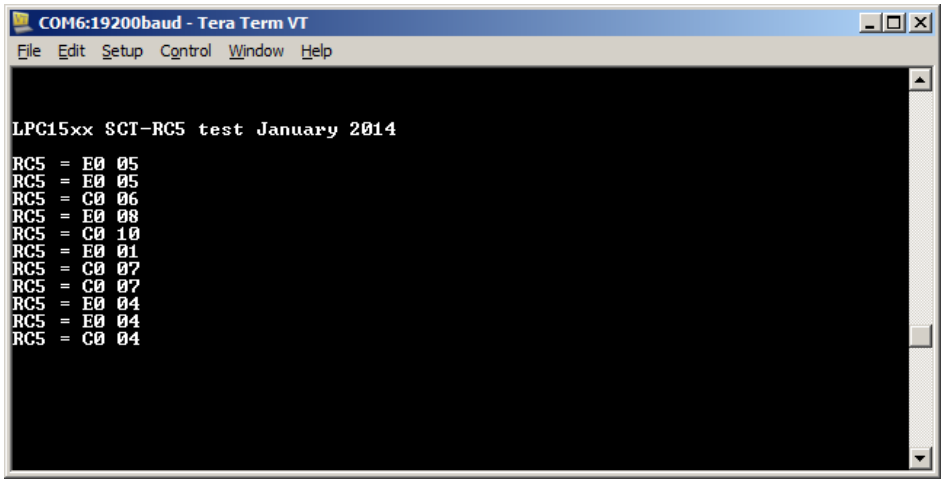


Fig 35. RC5 output at PC terminal

14. SCTimer/PWM start_stop

14.1 Purpose

This project (SCTx_start_stop) shows a possible usage of SCTimer/PWM start and stop events that can influence the other half of the same SCTimer.

In addition to the LPCXpresso code, this cookbook contains a sample application for the SCTimer/PWM Fizzim designer tool using the Keil compiler (see Fig 36).

14.2 Configuration

The timer in the SCTimer/PWM is configured in split mode (2 x 16-bit timers). Each half of the timer generates start and stop events, which alternatively starts and stops the other side of the state machine, in a ping-pong like fashion.

Note: for keeping one timer in stopped state while exiting reset, the HALT bit needs to be cleared in **the same write cycle as the** STOP bit.

14.3 Design

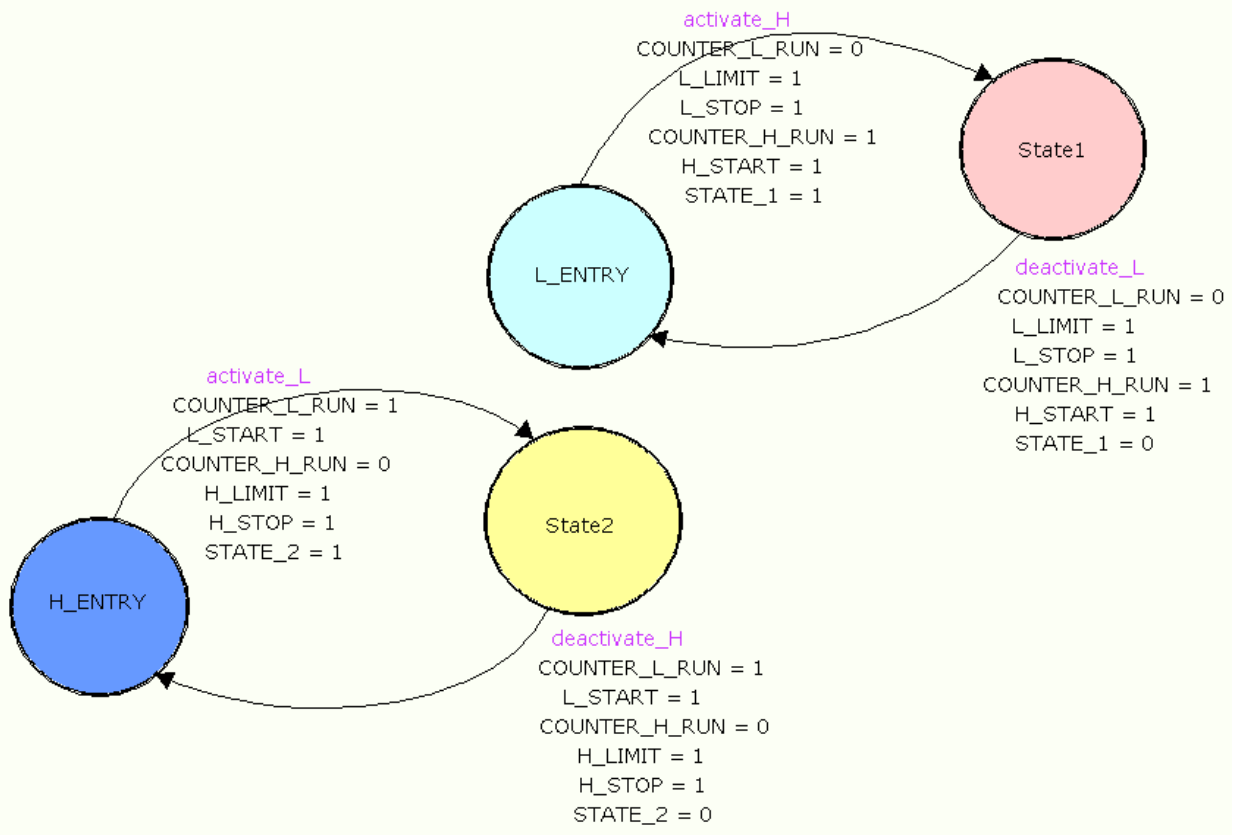


Fig 36. start_stop state diagram (Fizzim)

15. Input synchronization

The SCTs have an option to synchronize inputs to the SCTimer/PWM input clock, before the input is used to create an event.

Selecting the option is done in the global configuration register (bits 9 to 16).

If an input is synchronous to the SCTimer/PWM clock, you can select to have the unsync option for faster response.

If an input is asynchronous to the SCTimer/PWM clock, especially if inputs are edge sensitive, it is recommend to set the sync option, in order not to miss any inputs and/or input edges. See Fig 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

Fig 37. Configuration register INSYNC bits

Every rising edge of SCTIN_0 generates an event that toggles SCTOUT_0.

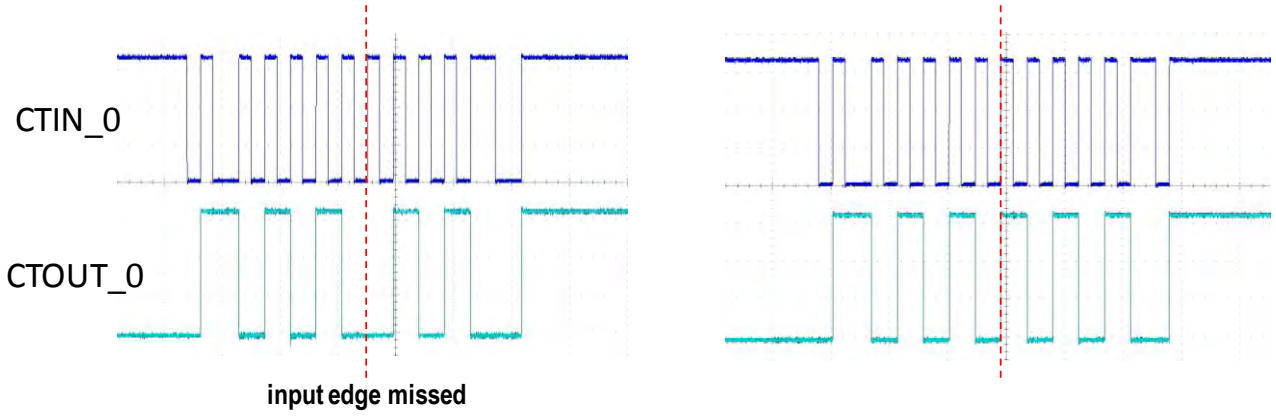


Fig 38. Left: no sync, inputs missed - Right: with sync okay

16. Dithering

16.1 Purpose

This example (SCTx_dithering) demonstrates the SCT's dithering feature available on some parts (see Table 1). By using this feature you can increase the average timer resolution with a factor 16.

16.2 Configuration

The example code is using SCT0 and has been tested on an LPCXpresso board with an LPC1549 running at 250 kHz (from IRC). SCT0 timer generates a 4 millisecond PWM output @ SCT0_OUT0 (see Fig 39). The Duty cycle of the PWM signal starts with 25 % (1 msec ON, 3 msec OFF).

SCT0_OUT0 is linked to P0_24 (green LED on LPCXpresso board).

Pressing pushbutton SW3 (P1_9) will change the LED brightness to 37.5 % by using the SCTimer/PWM dithering feature (giving an average of 1.5 msec ON and 2.5 msec OFF). Releasing pushbutton SW2 will change the LED brightness back to 25 %.

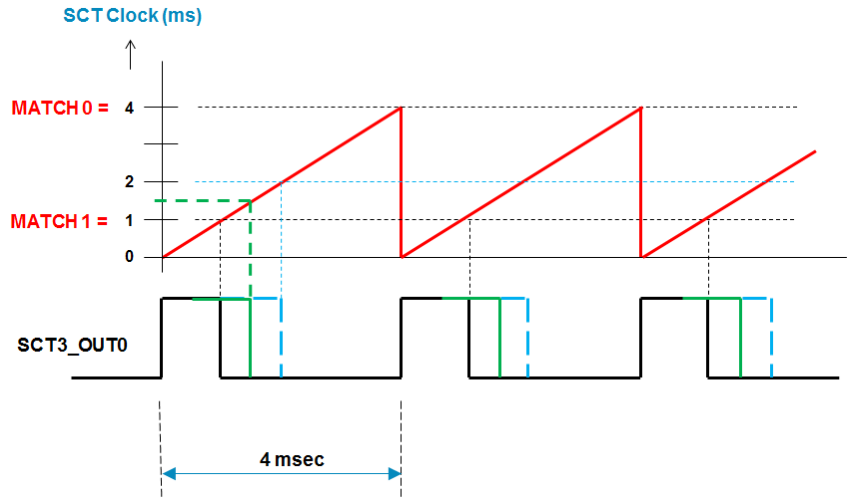


Fig 39. PWM output with dithering

16.3 Implementation

First thing needed is a very slow (1 msec) SCTimer/PWM input clock. To realize this, the System Clock (= IRC) is divided by 48, resulting in a 250 kHz system clock (this is done in module *system_LPC15xx.c*).

Next, the SCTimer/PWM prescaler is set to 250 to generate a 1 kHz SCTimer/PWM input clock.

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

16.4 Result

Fig 40 shows the SCTimer/PWM output using three different values for the fractional match register.

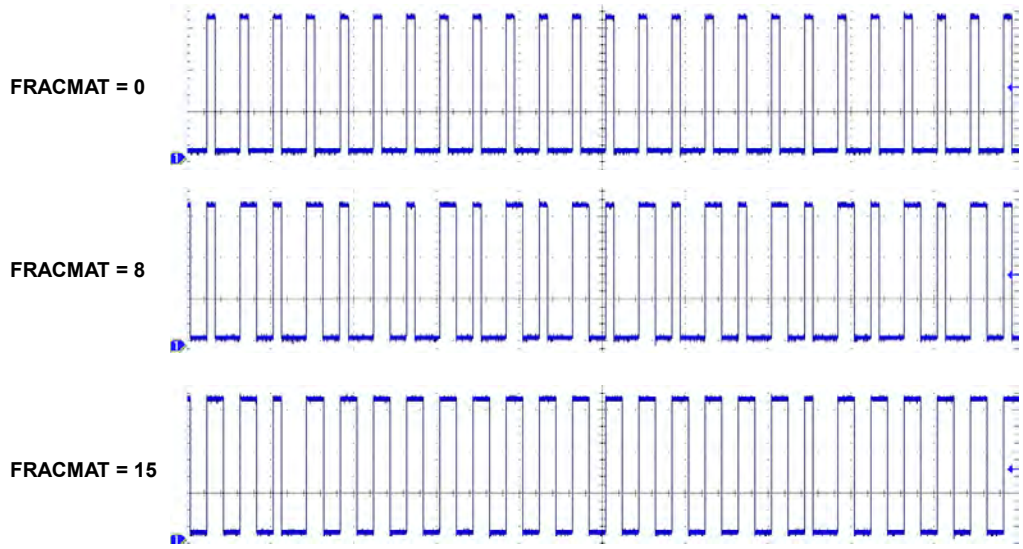


Fig 40. LPC15xx SCTimer/PWM dithering examples

17. WS2811 LED driver

The WS2811 LED drivers use a simple one-wire protocol for transferring 24-bit RGB values. Multiple WS2811 devices may be chained, and the RGB values for all of them are sent together to the first device in the chain. The first device takes the first 24-bit package to set its own RGB state, and retransmits the rest. Such blocks of RGB values are separated from each other by a “reset code” on the data line, which is a simple inactivity period of at least 50 µs in which the signal line is held low.

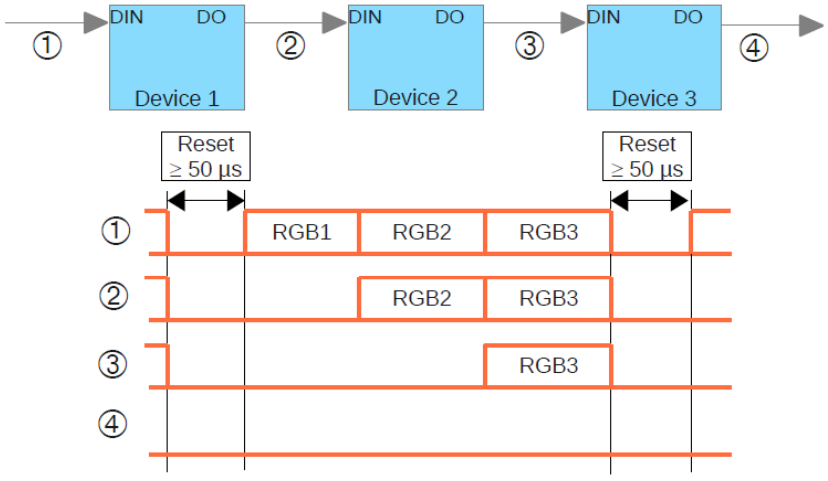


Fig 41. WS2811 one-wire communication protocol

A single 24-bit data package consists of 24-bit periods of either 1.25 µs or 2.5 µs, depending on whether the WS2811 is configured for 800 kHz or 400 kHz. Each bit is transmitted as a pulse with a duty cycle depending on the bit value. A “0” has a nominal duty cycle of 20 %, while a “1” has a nominal duty cycle of 48 %. There is a large timing tolerance when transmitting single bits, but accumulated jitter for a whole 24-bit package or multiple RGB values should be at a minimum. The general waveforms of data bits 0 and 1 are shown below. You can also see a full RGB frame which represents the RGB value 0xCA1722 (red channel = 0xCA, green channel = 0x17, blue channel = 0x22).

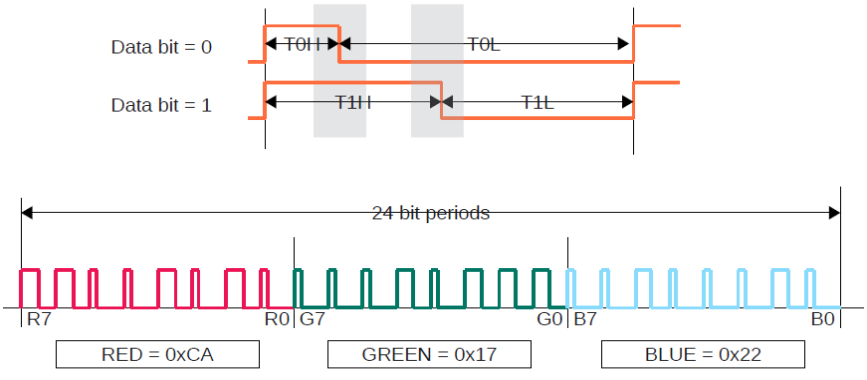


Fig 42. WS2811 frame and bit waveforms

You can see that each color channel is sent with MSB first.

The values for the on and off periods of the data bits depend on the operation frequency of the WS2811, which can be either 400 kHz or 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

Fig 43. WS2811 specification

17.1 Implementation

The WS2811 transmitter design demonstrates the efficient use of states and events in the SCT. It only uses one SCTimer/PWM half 16-bit timer, six events and 12 states (for resources check Table 1), leaving more than 50 % of the SCTimer/PWM resources available for another task.

Overview:

- Uses one 16-bit timer, leaving the other 16-bit timer free for other purposes.
- Uses the prescaler to run at a minimum clock frequency to save power.
- Autonomously send 24-bit frames, double-buffered.
- Interrupt after each frame transmission, leaving almost a full frame time for CPU to provide the next frame.
- Halt after last transmitted frame.
- Precede each multi-frame (block) transmission with a reset code of adjustable length.

17.2 Configuration

The SCTimer/PWM must be configured for split mode (CONFIG.UNIFY = 0). Conflict resolution for the data output must be set to “no action” (this is the default).

17.2.1 Match registers

MATCH0/MATCHREL0 holds the bit length (period).

MATCH1/MATCHREL1 holds the T1H time.

MATCH2/MATCHREL2 holds the T0H time.

17.2.2 Inputs/outputs

The data output can be assigned to any of the available SCTx_OUTx signals by configuring the corresponding SET and CLR registers of the output.

An auxiliary output is required for a double-buffering scheme. It can be assigned to any of the SCTOUT signals, but does not have to be connected to a pin (internal signal).

17.2.3 States

States form the heart of the data transmission. For parts that only have a maximum of 15 states (like the LPC1500) we decided to send a frame in two bursts of 12 bits (needing 12 states). The state machine begins in state 11 and decrements the state after each transmitted bit. After the last bit of a frame has been transmitted in state 0, the state machine is forced to state 11 upon start of the next 12 bits. State 11 corresponds to the first bit sent (MSB), and state 0 corresponds to the last bit sent (LSB).

At the start of a new bit, the data output is set. Two match registers set to time T0H (MATCH1, EV13) and T1H (MATCH2, EV14) trigger events which clear the data output. Without further events, this would always transmit a logical zero, since MATCH1 comes first, and the bit's active time would end at T0H.

We need another event to determine the value of the data bit in each of the 12 states 11...0. This event (EV12) is configured to set the data output at time T0H (MATCH1). Therefore, a conflict occurs at position T0H with the previously described event which wants to clear the data output. As the conflict resolution register for the data output says “no action”, the data output is not cleared at T0H, but rather remains set until MATCH2 triggers an event at T1H. When the transmit data word is written into the event state register of the new event, it acts as a mask which enables this event only in those states where the data word has a 1 in the corresponding bit position, so the SCTimer/PWM transmits a 1. A 0 in the data word disables the event in the corresponding state, and the SCTimer/PWM transmits a 0.

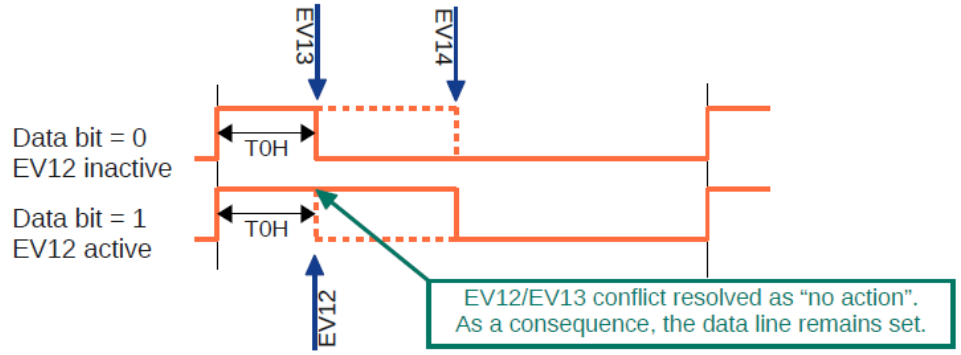


Fig 44. SCTimer/PWM data bit handling

17.2.4 Event details

Event 15 determines the start of a new bit. It is active in all states, and is triggered by a MATCH0 (bit time) event. This event sets the data output and decrements the state number (i.e. it adds 31 to it). It is important that the state is preset to 12 before this event occurs for the first time.

Event 14 determines the maximum output ON time, which is equivalent to T1H. It is active in all data bit states except state 0 (1...23), and is triggered by MATCH2. This event clears the data output. Event 10 (with extended functionality) replaces this event in state 0.

Event 13 determines the end of a zero data bit, which is equivalent to T0H. It is active in all data bit states, and is triggered by MATCH1. This event clears the data output. As this event occurs before event 14 in a timer cycle, we would only ever send logical zero bits. Therefore, events 12 and 11 can override the action of event 13. They occur at the same time as event 13 (if enabled!), and cancel the output action due to conflict resolution set to “no action”.

Event 12 forces transmission of a logical 1 from the first data buffer. The event is enabled in those states (0...11) in which the transmit data word has a 1 in that bit position, and is triggered by MATCH1. It is also qualified by the auxiliary flag (buffer selector) = 0. This means that this event occurs at the same time as event 13, and with conflict resolution set to “no action”, cancels the output clear action of event 13. This leads to the bit's ON time being extended to T1H (data output is eventually cleared by event 14).

Event 11 is equivalent to event 12, except that its trigger condition checks for the auxiliary flag = 1.

Event 10 determines the end of a frame transmission. It also takes the function of event 14 in state 0. It is active in state 0 only (LSB transmission), and is triggered by MATCH2 (the end of the last bit's ON period). It toggles the auxiliary bit, clears the data output, and triggers an interrupt. In response to that interrupt, the CPU shall read the auxiliary bit, and determine which buffer (= event state register 11 or 12) takes the next transmit frame (12 bits) data. The CPU shall write a pattern (1 << 10 for event 10) to the HALT_H register if it doesn't want another frame to be transmitted. This lets the transmission stop at the end of the frame that has just been started. This event sets the state number to 12.

17.3 Operation

The following steps are necessary once to prepare the SCTimer/PWM for this mode (when SCTimer/PWM is globally halted). We assume that the H counter is used for WS2811 mode.

1. Configure SCTimer/PWM for split mode.
2. Configure match registers:
   a. MATCHREL0 = SystemCoreClock/DATA_SPEED - 1
   b. MATCHREL1 = 20% of SystemCoreClock/DATA_SPEED - 1
   c. MATCHREL2 = 48% of SystemCoreClock/DATA_SPEED - 1
3. Configure events:
   a. Event 15: MATCH0, All states, DATA = 1 and STATE += 31
   b. Event 14: MATCH2, All states except state 0 and DATA = 0
   c. Event 13: MATCH1, All states and DATA = 0
   d. Event 12: MATCH1 && AUX==0, All of states [11:0] where a logical 1 shall be transmitted and DATA = 1
   e. Event 11: MATCH1 && AUX==1, All of states [11:0] where a logical 1 shall be transmitted, and DATA = 1
   f. Event 10: MATCH2, State 0, IRQ, AUX = toggle and STATE = 12

17.3.1 Transmission of a block of frames

1. Halt the H timer
2. Preset the reset time. Write the number of clock pulses required as a negative number to the counter COUNT_H.
3. Set STATE_H = 12.
4. Prime transmit buffer by writing first transmit frame to event 12 state register. Make sure bits [31:12] are zero.
5. Prime other transmit buffer by writing second transmit frame to event 11 state register. Write 0 if only one frame is to be transmitted.
6. Start H timer as up counter (clear DOWN_H and HALT_H in register CTRL_H).

17.3.2 Interrupt handling

Interrupts are triggered after a frame has been transmitted completely.

1. If this was the last transmit frame, stop the timer.
2. Read auxiliary output bit. If 1, write next frame to event 12 state register, else to event 11 state register.

Additional action may be required if the above procedure is not followed, and the data output is stuck high. You should not simply clear the output, since the access to the output register may present a race condition with hardware access to the outputs from the other (L) timer running a different application.

17.4 Result

Fig 45 shows the transmission of one 12 bit frame (0x123). The light blue trace is just used for debug. This GPIO signal toggles every frame (inside Event 10 interrupt service routine).

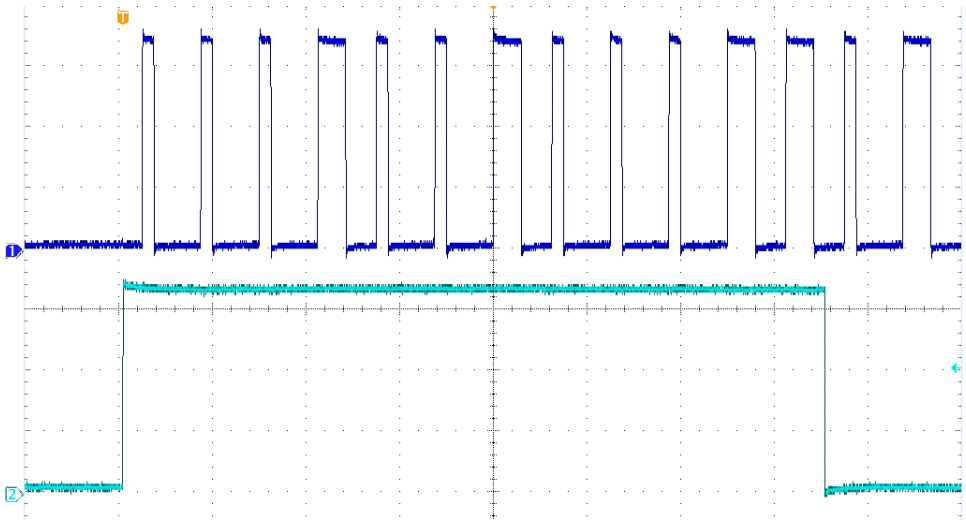


Fig 45. WS2811 LED driver one 12 bit frame transmission

Fig 46 shows the transmission of a block of four 24-bit RGB WS2811 LED driver values (split into 8 12 bit frames: 0x123, 0x456, 0xFF0, 0x0CC, 0x555, 0x555, 0x800, 0x001).

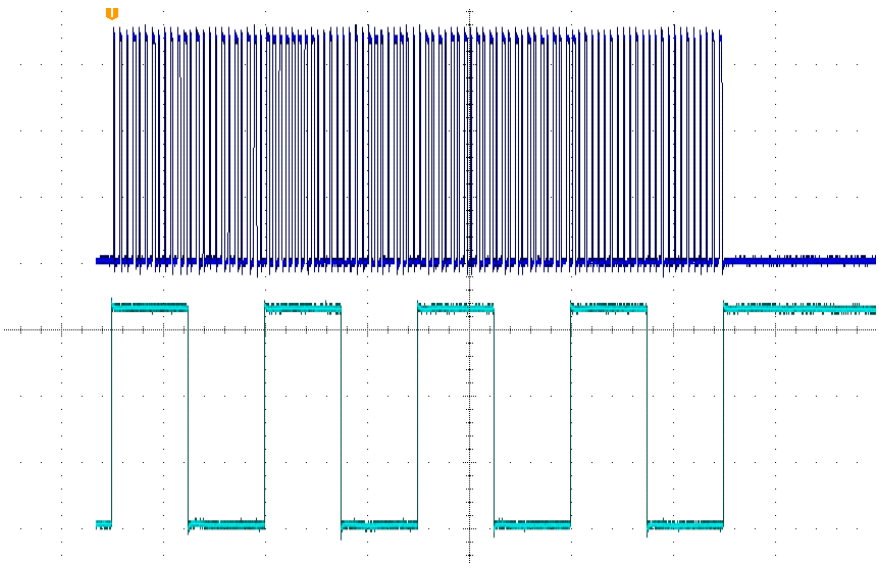


Fig 46. WS2811 transmission of a block of four 24 bit RGB values

Fig 47 shows the transmission of blocks of RGB values separated from each other by a “reset code” on the data line, which is a simple inactivity period of at least 50 µs in which the signal line is held low.

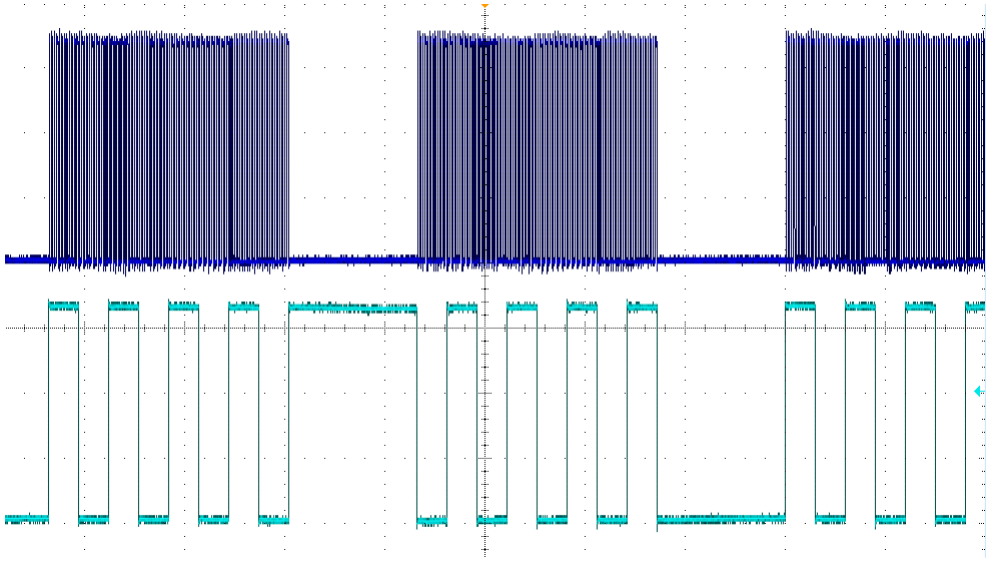


Fig 47. WS2811 transfer of blocks of RGB values separated by inactivity periods

18. Legal information

18.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

18.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

18.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

19. Contents

For more information, visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 18 February 2015
Document identifier: AN11538