Using hardware and software to make new stuff

Search

type, hit enter

Categories

- 3D Printing (1)
- Affinity Photo (1)
- Aide-memoir (1)
- Analog (1)
- Ansible (3)
- Dates to Remember (3)
- Docker (1)
- Electronics (144)
- ESP8266 (12)
- FPGA (4)
- Garden (1)
- General (8)
- Home Built CPU (6)
- Informatica (1)
- Internet of Things (8)
- iOS (2)
- KiCad (4)
- MSP430 (2)
- Netduino (49)
- NuttX (9)
- Photography (3)
- Pico (10)
- Raspberry Pi (17)
- Silverlight (6)
- Software Development (88)
- STM32 (5)
- STM8 (54)
- Tips (5)

Archives

- July 2024
- June 2024
- May 2024
- April 2024
- March 2024
- February 2024
- January 2024
- December 2023
- November 2023
- October 2023

- October 2011
- September 2011
- July 2011
- June 2011
- May 2011
- April 2011
- March 2011
- February 2011
- January 2011
- December 2010
- April 2010

Auto-Wakeup on the STM8S

A few days ago I was asked for advice about pausing the STM8S for a long time period (in this case 30 seconds). I had to admit that I was not sure how to achieve this without using a timer and counting interrupts until the 30 time period had expired. A quick examination of the STM8S Programming Reference reveals that there is a simpler way of doing this. This post examines the Auto-Wakeup (AWU) feature of the STM8S and shows how this feature can be used to pause for a time period which can range from 15.625uS to 30.720s, assuming the clock source is accurate.

Auto-Wakeup (AWU) Feature

The AWU feature wakes the STM8S after a predefined time period following the microcontroller going into the active halt state. This feature can only be used when the microcontroller is halted and the accuracy is dependent upon the clock source.

The clock source is fed into a prescalar. The output from the prescalar is then used as a clock for a counter. The counter will cause an interrupt to be generated when the preset counter value has been reached.

It is also possible to determine the accuracy of the clock source by using the capture compare feature of TIM1 or TIM3 to measure the clock frequency.

Auto-Wakeup Registers

The function of the AWU is determined by the values in the AWU control registers.

Auto-Wakeup Enable – AWU_CSR1_AWUEN

Setting this bit to 1 enables the AWU function. Setting this to 0 disables the function.

Auto-Wakeup Asynchronous Prescalar Divider – AWU_APR_APR and Auto-Wakeup Timebase Selection Register – AWU_TBR_AWUTB

These two registers together control the duration of the AWU. The exact duration of the wakeup period is determined by the AWU_APR_APR and AWU_TBR_AWUTB values from the following table.

$f_{LS} = f$	f_{LS} = 128 kHz	AWU_TBR_AWUTB	APR_{DIV} Formula	AWU_APR_APR Range
2/f – 64/f	0.015625 ms – 0.5 ms	0001	APR_{DIV}/f_{LS}	2 to 64
2×32/f – 2x2x32/f	0.5 ms – 1.0 ms	0010	2 x APR_{DIV}/f_{LS}	32 to 64

24×64/f – 24×128/f	8 ms – 16 ms	0110	25 x APR_{DIV}/f_{LS}	32 to 64
25×64/f – 25×128/f	16 ms – 32 ms	0111	26 x APR_{DIV}/f_{LS}	32 to 64
26×64/f – 26×128/f	32 ms – 64 ms	1000	27 x APR_{DIV}/f_{LS}	32 to 64
27×64/f – 27×128/f	64 ms – 128 ms	1001	28 x APR_{DIV}/f_{LS}	32 to 64
28×64/f – 28×128/f	128 ms – 256 ms	1010	29 x APR_{DIV}/f_{LS}	32 to 64
29×64/f – 29×128/f	256 ms – 512 ms	1011	210 x APR_{DIV}/f_{LS}	32 to 64
210×64/f – 210×128/f	512 ms – 1.024 s	1100	211 x APR_{DIV}/f_{LS}	32 to 64
211×64/f – 211×128/f	1.024 s – 2.048 s	1101	212 x APR_{DIV}/f_{LS}	32 to 64
211×130/f – 211×320/f	2.080 s – 5.120 s	1110	5 x 211 x APR_{DIV}/f_{LS}	26 to 64
211×330/f – 212×960/f	5.280 s – 30.720s	1111	30 x 211 x APR_{DIV}/f_{LS}	11 to 64

Where $f_{LS} = f$ is the formula which should be used when the inbuilt LSI is not being used.

It may not be possible to obtain an exact time period and the application may have to use the values which give the closest period.

When not in use, AWU_TBR_AWUB should be set to 0 in order to reduce power consumption.

Auto-Wakeup Flag – AWU_CSR1_AWUF

This flag is set when the AWU interrupt has been generated. The flag is cleared by reading the AWU control/status register (AWU_CSR1).

Auto-Wakeup Measurement Enable – AWU_CSR1_MSR

Setting this flag to 1 connects the output from the prescalar to one of the internal timers. This allows the application to accurately determine the clock frequency connected to the AWU prescalar. By measuring the clock frequency the application can adjust the values used for the prescalar divider and the timebase.

Software

Now we have the theory it is time to break out the STM8S Discovery board and the IAR compiler. The application starts pretty much the same as previous examples in this series, by setting the system clock and configuring a port for output:

```
1  //
2  //  This program demonstrates how to use the Auto-Wakeup feature of the STM8S
3  //  microcontroller.
4  //
5  //  This software is provided under the CC BY-SA 3.0 licence.  A
6  //  copy of this licence can be found at:
7  //
8  //  http://creativecommons.org/licenses/by-sa/3.0/legalcode
9  //
10 #include <iostm8S105c6.h>
11 #include <intrinsics.h>
```

```
//
void InitialiseSystemClock()
{
    CLK_ICKR = 0;                       //  Reset the Internal Clock Register
    CLK_ICKR_HSIEN = 1;                 //  Enable the HSI.
    CLK_ECKR = 0;                       //  Disable the external clock.
    while (CLK_ICKR_HSIRDY == 0);       //  Wait for the HSI to be ready for
    CLK_CKDIVR = 0;                     //  Ensure the clocks are running at
    CLK_PCKENR1 = 0xff;                 //  Enable all peripheral clocks.
    CLK_PCKENR2 = 0xff;                 //  Ditto.
    CLK_CCOR = 0;                       //  Turn off CCO.
    CLK_HSITRIMR = 0;                   //  Turn off any HSIU trimming.
    CLK_SWIMCCR = 0;                    //  Set SWIM to run at clock / 2.
    CLK_SWR = 0xe1;                     //  Use HSI as the clock source.
    CLK_SWCR = 0;                       //  Reset the clock switch control re
    CLK_SWCR_SWEN = 1;                  //  Enable switching.
    while (CLK_SWCR_SWBSY != 0);        //  Pause while the clock switch is b
}

//-----------------------------------------------------------------------------
//
//  Initialise the ports.
//
//  Configure all of Port D for output.
//
void InitialisePorts()
{
    PD_ODR = 0;             //  All pins are turned off.
    PD_DDR = 0xff;          //  All pins are outputs.
    PD_CR1 = 0xff;          //  Push-Pull outputs.
    PD_CR2 = 0xff;          //  Output speeds up to 10 MHz.
}
```

The next step is to provide a method to initialise the AWU function:

```
//-----------------------------------------------------------------------------
//
//  Initialise the Auto Wake-up feature.
//
//
//
void InitialiseAWU()
{
    AWU_CSR1_AWUEN = 0;     // Disable the Auto-wakeup feature.
    AWU_APR_APR = 38;
    AWU_TBR_AWUTB = 1;
    AWU_CSR1_AWUEN = 1;     // Enable the Auto-wakeup feature.
}
```

Firstly, the AWU function is disabled. The prescalar is set followed by the timebase. The final step is to re-enable the AWU.

```
//-----------------------------------------------------------------------------
//
```

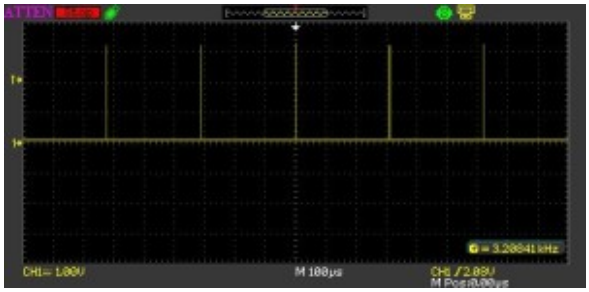
```
//
#pragma vector = AWU_vector
__interrupt void AWU_IRQHandler(void)
{
    volatile unsigned char reg;

    PD_ODR = 1;
    asm("nop;");
    asm("nop;");
    PD_ODR = 0;
    reg = AWU_CSR1;     // Reading AWU_CSR1 register clears the interrupt fla
}
```

This interrupt handler is triggered at the end of the AWU period. This handler simply pulses pin 0 on port D

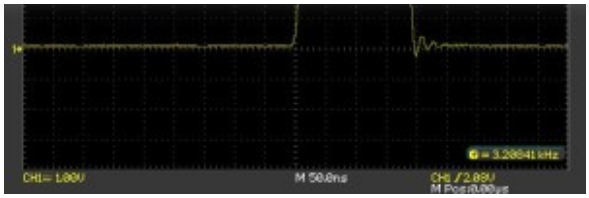
```
//-----------------------------------------------------------------------------
//
//  Main program loop.
//
int main(void)
{
    //
    //  Initialise the system.
    //
    __disable_interrupt();
    InitialiseSystemClock();
    InitialisePorts();
    InitialiseAWU();
    __enable_interrupt();
    //
    //  Main program loop.
    //
    while (1)
    {
        __halt();
    }
}
```

The AWU function is turned on when the microcontroller enters the active halt state. A halt instruction is generated by the *__halt()* method.

Running the above code on the STM8S Discovery board results in the following trace on the oscilloscope:



AWU 3208Hz Signal



AWU Expanded Pulse

Examining the table above and the values used for the precalar and timebase we should be seeing a pulse with a frequency of approximately 3,368Hz. The actual value measured on the oscilloscope is 3,208Hz. The measurement on the oscilloscope will alway be slightly out due to the time it takes for the ISR to be setup and called. An ISR on the STM8S takes 9 clock cycles to be established, this equates to about 560nS on a system running at 16MHz. The actual difference is 6.25mS. The remainder of the difference comes from the fact that the LSI has an accuracy of +/-12.5%. A quick calculation shows the the LSI was running at about 121KHz at the time this post was written.

Conclusion

AWU offers a simple way of triggering processing at predetermined time periods. The active halt state puts the microcontroller into a low power mode whilst the microcontroller is waiting for the time period to elapse.

It should also be noted that using the LSI results in a slightly variable time period.

Tags: Electronics, Software Development, STM8, The Way of the Register

Friday, June 20th, 2014 at 2:53 pm • Electronics, Software Development, STM8 • RSS 2.0 feed Both comments and pings are currently closed.

Comments are closed.

Pages

- About
- Making a Netduino GO! Module
- The Way of the Register
- Making an IR Remote Control
- Weather Station Project
- NuttX and Raspberry Pi PicoW

Support This Site

Buy me a coffee

Currently Reading