Timer/Counter And Interrupts

1 Introduction

The AVR32UC3A0512 has three individual Timer/Counter (TC) channels. They can be programmed in a variety of ways, for instance to generate processor interrupts at regular intervals. We will be using the Software Framework to program TC interrupts.

You will need the following drivers from the Software Framework:

- INTC - Interrupt Controller
- TC - Timer Counter

1.1 AVR32 Software Framework

Programming the AVR32 chip from scratch is time consuming and complex. To make it faster and easier to develop software there is a Software Framework which contain drivers for the EVK1100. The Framework provide good functionality at the expense of less flexibility of the more exotic options. A good enough trade-off in many cases.

Your task is to play around with the code below.

- Create a new Atmel Studio 6 project from template.
- To use the Software Framework, right click on the current project and select "ASF Wizard".

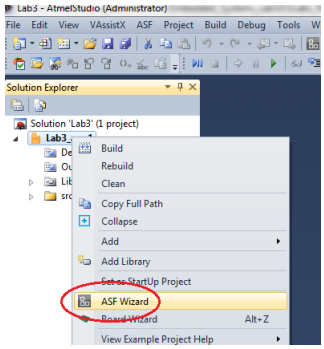


Figure 1: Open the ASF wizard

- In the ASF Wizard you see the available modules on the side and the selected modules on the rite side.
- To filter the available drivers out of all available modules you can use the dropdown menu "Show" and select "Drivers".

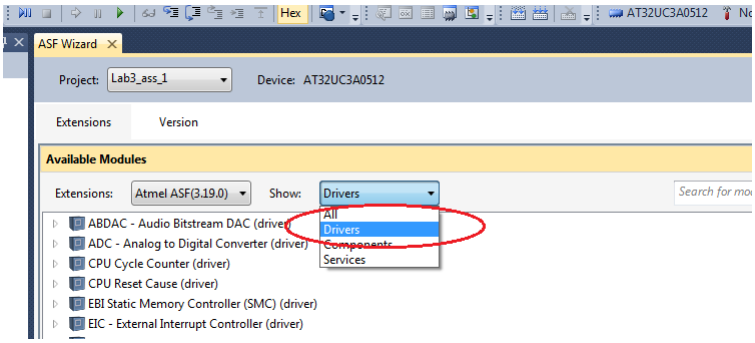


Figure 2: Filter the drivers

- Add the drivers you need:
  - To add an item select it in the left view (1) and press the "Add" button (2). It will show up in the right view.

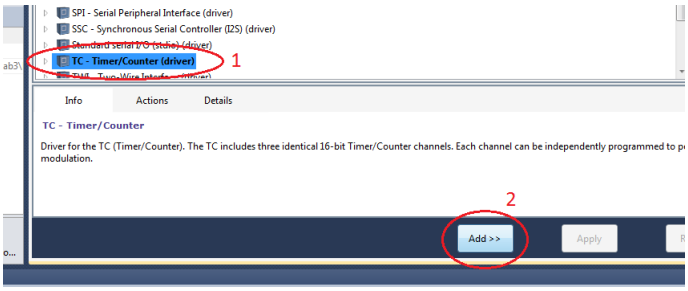


Figure 3: Software Framework - Add Driver

  - Add TC and INTC for this assignment.
    - TC is needed when using the Timer Counter.
    - INTC is needed for the exception handling when using the Software Framework.
    - The right side should now look like in the picture.

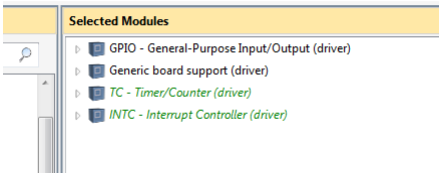


Figure 4: Software Framework - Select Driver

- Click "Apply" to add the selected drivers to the project. A new window will inform you about the added files. Click "Ok" to proceed.

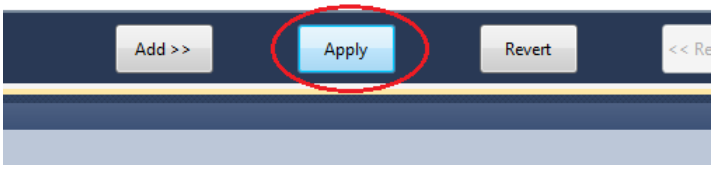


Figure 5: Software Framework - Apply changes

- Now you can find two new folders in the ASF directory of your project as shown in the picture.

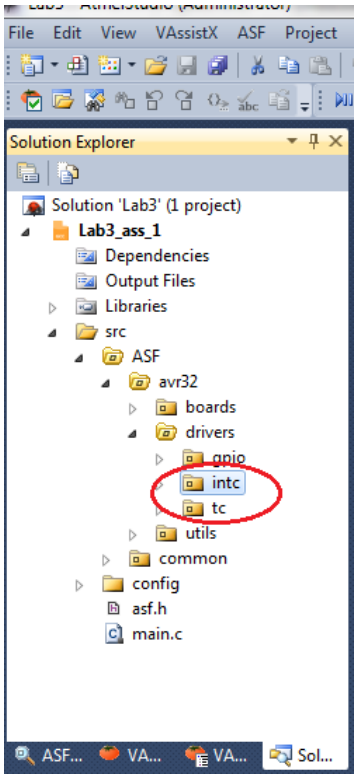
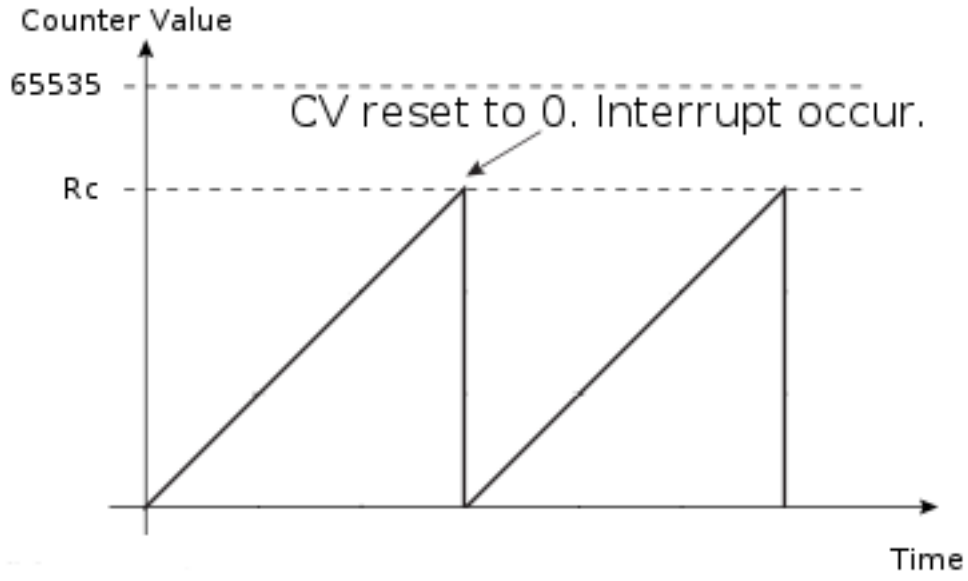


Figure 6: New files in the project

1.2 Timer/Counter

Each Timer channel is 16-bit and thus the Counter Value (CV) can count from 0 to 65535. The CV increases with the rate set in the Timer Counter Clock Select bits, TCCLKS, see below. The real time rate depends on the bus frequency, fPBA, which in our case is 115200 Hz. Once the counter reaches 65535 it starts all over at 0 again and repeats the cycle.

A compare value named Register C (RC) can be set to cause an interrupt upon match with the CV. The CV will be reset to 0 at the same time. Hence, we can make the timer signal interrupts at an arbitrary rate.



Setting the RC is done by the function *tc_write_rc()*, which take three parameters:

```
int tc_write_rc(volatile avr32_tc_t * tc,
                unsigned int channel,
                unsigned short value);
```

- *tc* - the address to the TC controller, &AVR32_TC.
- *channel* - the TC channel, 1-3, that the RC value should be altered on.
- *value* - the new RC value to set in the specified TC channel.

```
Example:
int tc_channel = 0;
int value = 14400;
tc_write_rc(&AVR32_TC, tc_channel, value);
```

The timer waveform option needs to be set up by defining a struct of the type *tc_waveform_opt_t* and calling the *tc_init_waveform()* function :

```
int tc_init_waveform(volatile avr32_tc_t * tc,
                     tc_waveform_opt_t * opt);
```

- *tc* - the address to the Timer/Counter controller, i.e. &AVR32_TC.
- *opt* - the address to the waveform option struct.

The more interesting options in the waveform struct are explained below:

Option	**Description**
Channel	Which of the three TC channel to use. Possible values: *0, 1 or 2.*
wavsel	Waveform selection. Decides how the counter value, CV, behaves. The UP modes increases the counter. The UPDOWN modes increase the counter until it hits the RC value and then decrease until CV is 0. Then the cycle repeats. Possible values: *TC_WAVEFORM_SEL_UP_MODE* *TC_WAVEFORM_SEL_UP_MODE_RC_TRIGGER* *TC_WAVEFORM_SEL_UPDOWN_MODE* *TC_WAVEFORM_SEL_UPDOWN_MODE_RC_TRIGGER* Note: Modes without _RC_TRIGGER doesn't cause an interrupt on RC match, but on CV overflow 65535->0.
tcclks	Clock Select. Which clock source to use for the timer. See table below.

The Clock Selection is made by tcclks. This parameter tells how fast the timer is going to tick. The PBA Clock is the frequency of the peripherial bus, fPBA, 115200 Hz in our case.

Source	Name	Connection
Internal	TC_CLOCK_SOURCE_TC2	fPBA / 2
"	TC_CLOCK_SOURCE_TC3	fPBA / 8
"	TC_CLOCK_SOURCE_TC4	fPBA / 32
"	TC_CLOCK_SOURCE_TC5	fPBA / 128
External	TC_CLOCK_SOURCE_XC0	12 Mhz crystal

Example:

```
// Define the options for waveform generation
static tc_waveform_opt_t waveform_opt =
{
    .channel = TC_CHANNEL0,         // Channel selection.
    .wavsel = TC_WAVEFORM_SEL_UP_MODE_RC_TRIGGER,
                                    // Waveform selection:
```

```
                                    // Up mode with automatic trigger
                                    // (reset) on RC compare.

    .enetrg   = FALSE,              // External event trigger enable.
    .eevt     = 0,                  // External event selection.
    .eevtedg  = TC_SEL_NO_EDGE,     // External event edge selection.
    .cpcdis   = FALSE,              // Counter disable when RC compare.
    .cpcstop  = FALSE,              // Counter clock stopped with RC
                                    // compare.
    .burst    = FALSE,              // Burst signal selection.
    .clki     = FALSE,              // Clock inversion.

    .tcclks   = TC_CLOCK_SOURCE_TC3 // Internal clock source 3,
                                    // connected to fPBA / 8.
};

// Initiate the TC with the waveform options
tc_init_waveform(&AVR32_TC, &waveform_opt);
...
```

1.2.1 Starting the timer

When the waveform has been set up the timer can be started by the *tc_start()* function:

```
int tc_start(volatile avr32_tc_t * tc, unsigned int channel);
```

Parameters:

- *tc* - Address to the TC unit, i.e. AVR32_TC
- *channel* - The TC channel to configure, *0, 1* or *2*.

Returns: Zero

1.3 Interrupts

Before any interrupt can be served the Interrupt Controller (INTC) must be configured. Every interrupt that can occur must be registered before normal execution starts to ensure a well-behaved system. During the time INTC is configured we do not want any interrupts to be served since this could happen at the same moment we are modifying some registers and a runtime failure could be caused. Hence, we need to disable all interrupts at a global level. When the configurations are done we enable interrupts globally again. This is done by:

Example:

```
...
Disable_global_interrupt();
INTC_init_interrupts();
...
Enable_global_interrupt();
...
```

1.3.1 Enabling TC interrupt

Enabling the TC interrupt by using the *INTC_register_interrupt()* function which takes three parameters:

```
void INTC_register_interrupt(__int_handler handler,
                             unsigned int irq,
                             unsigned int int_level);
```

- *handler* - The address to the interrupt handler that will be called when an interrupt is serviced. See the Interrupt handler section below.
- *irq* - The unique interrupt number that every peripherial has.
  - The three TC channels uses *AVR32_TC_IRQx* (x = *0, 1* or *2)*
- *int_level* is the priority level of the interrupt. We can satisfy with using priority level 0 for all lab work in this course, i.e. AVR32_INTC_INT0.

```
Example:
INTC_register_interrupt(&irq_handler,
                        AVR32_TC_IRQ0,
                        AVR32_INTC_INT0);
```

1.3.2 Configuring TC interrupt signals

The timer will not generate any interrupts until it has been configured properly. This is done by defining a struct of the type *tc_interrupt_t*, where the signals causing interrupt events are set to 1:

Example:

```
// Define which timer signals that will cause interrupts
static const tc_interrupt_t TC_INTERRUPT_OPT =
{
      // we interrupt on RC compare match with CV
      .cpcs  = 1,   // RC compare status
};
```

This struct is sent as a parameter to the *tc_configure_interrupts()* function:

```
int tc_configure_interrupts(volatile avr32_tc_t * tc,
                            unsigned int channel,
                            const tc_interrupt_t * bitfield);
```

Parameters:

- *tc* - Address to the TC unit, i.e. AVR32_TC
- *channel* - The TC channel to configure, *0, 1* or *2.*
- *bitfield* - Address to the TC_INTERRUPT_OPT struct

Returns: Zero (0).

1.3.3 Interrupt handler

An interrupt handler must be programmed to take care of the enabled interrupt requests. Interrupt handlers are much like normal functions in C, but with a few exceptions:

1. An interrupt handler must be attributed *__interrupt__* in order to get the compiler to produce correct code.
2. It is the responsibility of the interrupt handler to clear the interrupt flag before returning. If it is not cleared the interrupt remains active and will trigger again.
3. The interrupt handler can not *return* values as normal functions do. It does not really make sense. Using global variables are legal solutions to this inconvenience when the interrupt handler must signal something to the main program.

Example:

```
__attribute__((__interrupt__)) static void tc_irq_handler(void)
{
    ...
    // Before exiting:
    // Clear the interrupt flag, this is done
    // by reading the TC Status Register, SR.
    tc_read_sr(&AVR32_TC, TC_CHANNEL);
}
```

2 Assignments

1. **Stopwatch.** Create a stop watch. Use the serial terminal as an user interface. It should be possible to start, stop, and reset the clock via the keyboard on the serial terminal. The updated time should be displayed on terminal screen. Use a hardware timer to update the time value periodically when the timer is running. You should divide the program into several source files and create a well defined API for your main program. That is, all low-level code should be implemented as device drivers and its interface should be presented to the main program via descriptive function calls. The minimum requirement is that the clock should be able to handle hours, minutes and seconds.
   Hint: It might be useful to use the cursor movement ANSI escape sequences to position the cursor at the screen.

3 Optional Assignments

1. **Joystick debounce.** Use the timer/counter to prove bouncing from the push buttons. Unfortunately the quality of the push buttons is TOO GOOD, but the joystick bounces a little more. Write software debouncing for the joystick.

2. **IRQ serial driver.** Replace the *serialGetChar()* function in the Serial Driver API to use interrupts instead of polling.