

AVRAssist - AVR Helpers

Norman Dunbar

7th May 2019

Table of Contents

1. Introduction	1
2. Timer 0	3
2.1. Timer Initialisation	3
2.1.1. Initialisation Function	4
2.1.2. Timer Modes	4
2.1.3. Clock Sources	4
2.1.4. Compare Match Options	5
2.1.5. Interrupts	6
2.1.6. Force Compare Options	8
3. Timer 1	9
3.1. Timer Initialisation	9
3.1.1. Initialisation Function	10
3.1.2. Timer Modes	10
3.1.3. Clock Sources	11
3.1.4. Compare Match Options	11
3.1.5. Interrupts	12
3.1.6. Force Compare Options	13
3.1.7. Input Capture	14
4. Timer 2	17
4.1. Timer Initialisation	17
4.1.1. Initialisation Function	18
4.1.2. Timer Modes	18
4.1.3. Clock Sources	18
4.1.4. Compare Match Options	19
4.1.5. Interrupts	20
4.1.6. Force Compare Options	21
5. Analogue Comparator	23
5.1. Comparator Initialisation	23
5.1.1. Initialisation Function	23
5.1.2. Reference Voltage	24
5.1.3. Sample Voltage	24
5.1.4. Interrupts	25
6. Watchdog Timer	27
6.1. Watchdog Timer Initialisation	27
6.1.1. Initialisation Function	27
6.1.2. Timeout Setting	28
6.1.3. Watchdog Mode	28
Appendix A: Foibles	31

A.1. General - Interrupts	31
A.2. General - Timers	31
A.3. Timer 0	32
A.3.1. Timer 0 - General	32
A.3.2. Timer 0 - Interrupts	32
A.3.3. Timer 0 - Overflow Interrupt	32
A.4. Timer 1	32
A.4.1. Timer 0 - General	32
A.4.2. Timer 0 - Interrupts	33
A.5. Timer 2	33
A.5.1. Timer 0 - General	33
A.5.2. Timer 0 - Interrupts	33
A.6. Analogue Comparator	33
A.6.1. Analogue Comparator - Interrupts	33
A.7. Watchdog	33
A.7.1. Watchdog - Interrupts	33

Chapter 1. Introduction

AVRAssist is a *first attempt* at making (my) life easier when it comes to trying to remember what bits in which registers need to be set to enable/initialise certain features on my Arduino boards' Atmega328P micro controller. After all, which of these is easier to remember, this:

```
#include <timer0.h>

using namespace AVRAssist;

...

Timer0::initialise(Timer0::MODE_FAST_PWM_255,    // Timer mode;
                  Timer0::CLK_PRESCALE_64,       // Prescaler;
                  Timer0::OC0X_DISCONNECTED,      // Don't touch OCOA or OCOB;
                  Timer0::INT_COMP_MATCH_A |      // Interrupt(s) to enable;
                  Timer0::INT_COMP_MATCH_B);

...
```

or this:

```
TCCR0A = (1 << WGM00) | (1 << WGM01);
TCCR0B = (1 << CS01) | (1 << CS00);
TIMSK0 = (1 << OCIE0A) | (1 << OCIE0B);
```

And it gets worse when I decide to add in more features and have to look up the additional bits and set them correctly. Get the computer to do it for you is my motto - or one of them!

At the moment, each header file creates a namespace and you (or I) can use that namespace to initialise one feature - timer, comparator, watchdog, ADC etc - of the micro-controller. At the moment, everything lives in the **AVRAssist::<feature>** namespace, **AVRAssist::Timer0** for example, but if, as I might, I decide to add my ATtiny85s into the mix, it is possible that some future update will break things and add another level - **AVRAssist::<chip>::<feature>** namespace, **AVRAssist::Atmega328::Timer0** and **AVRAssist::Attiny85::Timer0** perhaps. Stay tuned.



It *should* be a simple edit to change one line, **using namespace AVRAssist;** to something like **using AVRAssist::Atmega328;** instead, to use the micro-controller specific later versions of the code. Hey, I have to change my code as well, so the easier the better.

At the moment, I'm working mostly with ATmega328P devices, so the whole thing is designed (for certain values of *designed*) exclusively for that micro controller.

It could be that I extend this code to make actual objects out of the current namespaces, so that I can do other things, without needing the bit names and registers all the time. Time will tell, that's its job after all.

It is intended that this code will work in the Arduino IDE as well as with other development systems, such as PlatformIO or even AVR Studio etc, however, the Arduino IDE does a lot of jiggery-pokery in the background, setting up timers and PWM and such like, that some things might not be fully compatibly with the Arduino. Hopefully, this will be pointed out in the documentation.

Chapter 2. Timer 0

This AVR Assistant allows the simple setup and configuration of Timer/Counter 0 on your AVR (specifically, ATmega328) micro controller. This code works in the Arduino IDE or free standing for use with some other development system, such as PlatformIO. It has not been tested on other micro controllers. (Although the ATmega168 should work.)

To use the assistant, you must include the `timer0.h` header file:

```
#include "timer0.h"
```

Following this, you may, optionally, use the `AVRAssist` namespace:

```
using namespace AVRAssist;
```



The spelling of `AVRAssist` must be as shown above.

If you choose not to do this, you must prefix everything with `AVRAssist::` or the code will not work.

2.1. Timer Initialisation

Once the header file has been included, timer/counter 0 can be initialised as follows:

```
#include <timer0.h>

using namespace AVRAssist;

...

Timer0::initialise(Timer0::MODE_FAST_PWM_255,           // Timer mode;
                  Timer0::CLK_PRESCALE_64,              // Clock source;
                  Timer0::OC0X_DISCONNECTED,             // OCOA, OCOB actions on
compare match;
                  Timer0::INT_COMPARE_MATCH_A |         // Interrupt(s) to enable;
                  Timer0::INT_COMPARE_MATCH_B,
                  Timer0::FORCE_COMPARE_NONE            // Force Compare required?
                  );

...
```

The above sets Timer/Counter 0 into fast PWM mode with `TOP` = to 255, with a clock prescaler of 64. When `OCR0A` and `OCR0B` match `TCNT0` nothing happens to pins `OC0A` or `OC0B` but an interrupt will be fired when the comparison happens to match.

2.1.1. Initialisation Function

The header file exposes a single `initialise` function which is defined as follows:

```
void initialise(const uint8_t timerMode,
               const clockSource_t clockSource,
               const compareMatch_t compareMatch = OCOX_DISCONNECTED,
               const interrupt_t enableInterrupts = INT_NONE
               const forceCompare_t forceCompare = FORCE_COMPARE_NONE) {
```

2.1.2. Timer Modes

Timer/counter 0 can be initialised to run in one of 6 modes, as follows.

Mode	Parameter	Description
0	MODE_NORMAL	Normal mode.
1	MODE_PC_PWM_255	Phase Correct PWM with TOP at 255.
2	MODE CTC_OCR0A	Clear Timer on Compare (CTC) with TOP at OCR0A.
3	MODE_FAST_PWM_255	Fast PWM with TOP at 255.
4	MODE_RESERVED_4	Reserved, do not use.
5	MODE_PC_PWM_OCR0A	Phase Correct PWM with TOP at OCR0A.
6	MODE_RESERVED_6	Reserved, do not use.
7	MODE_FAST_PWM_OCR0A	Fast PWM with TOP at OCR0A.

You use this parameter to define the mode that you wish the timer/counter to run in. It should be obvious, I hope, that only one of the above modes can be used, however, if you wish to `or` them together, be it on your own head!

```
Timer0::initialise(Timer0::MODE_FAST_PWM_255,           ①
                   Timer0::CLK_PRESCALE_64,
                   Timer0::OCOX_DISCONNECTED,
                   Timer0::INT_COMPARE_MATCH_A |
                   Timer0::INT_COMPARE_MATCH_B,
                   Timer0::FORCE_COMPARE_NONE
                   );
```

① The timer mode parameter in action enabling the timer/counter in fast PWM mode with TOP defined by the value 255.

2.1.3. Clock Sources

The timer/counter needs a clock source to actually start it running as a timer, or as a counter. The following options are available for Timer/counter 0:

Parameter	Description
-----------	-------------

CLK_DISABLED	The timer/counter will stopped.
CLK_PRESCALE_1	The timer/counter will be running at F_CPU/1
CLK_PRESCALE_8	The timer/counter will be running at F_CPU/8
CLK_PRESCALE_64	The timer/counter will be running at F_CPU/64
CLK_PRESCALE_256	The timer/counter will be running at F_CPU/256
CLK_PRESCALE_1024	The timer/counter will be running at F_CPU/1024
CLK_T0_FALLING	The timer/counter will be clocked from pin 6, aka T0, Arduino pin D4 on a falling edge.
CLK_T0_RISING	The timer/counter will be clocked from pin 6, aka T0, Arduino pin D4 on a rising edge.

You use this mode to define how fast the timer/counter will count, or, to disable the timer.

```

Timer0::initialise(Timer0::MODE_FAST_PWM_255,
                   Timer0::CLK_PRESCALE_64,           ①
                   Timer0::OC0X_DISCONNECTED,
                   Timer0::INT_COMPARE_MATCH_A |
                       Timer0::INT_COMPARE_MATCH_B,
                   Timer0::FORCE_COMPARE_NONE
                   );

```

- ① The clock source parameter in action showing that the timer/counter will be running at a speed defined by the system clock, F_CPU, divided by 64.

2.1.4. Compare Match Options

This parameter allows you to indicate what actions you want the AVR micro controller to perform on pins OC0A (pin 12, Arduino pin D6) and/or OC0B (pin 11, Arduino pin D5) when the value in TCNT0 matches OCR0A or OCR0B. The allowed values are:

Parameter	Description
OC0X_DISCONNECTED	The two OC0x pins will not be affected when the timer count matches either OCR0A or OCR0B.
OC0A_TOGGLE	Pin OC0A will toggle when TCNT0 matches OCR0A.
OC0A_CLEAR	Pin OC0A will be reset LOW when TCNT0 matches OCR0A.
OC0A_SET	Pin OC0A will be set HIGH when TCNT0 matches OCR0A.
OC0B_TOGGLE	Pin OC0B will toggle when TCNT0 matches OCR0B. You cannot use OC0B_TOGGLE in anything but NORMAL and CTC modes.
OC0B_CLEAR	Pin OC0B will be reset LOW when TCNT0 matches OCR0B.
OC0B_SET	Pin OC0B will be set HIGH when TCNT0 matches OCR0B.

An example of initialising the timer/counter using this parameter is:

```

Timer0::initialise(Timer0::MODE_FAST_PWM_255,
                  Timer0::CLK_PRESCALE_64,
                  Timer0::OC0X_DISCONNECTED,
                  Timer0::INT_COMPARE_MATCH_A | ①
                  Timer0::INT_COMPARE_MATCH_B,
                  Timer0::FORCE_COMPARE_NONE
                );

```

- ① The compare match parameter in action showing that when `TCNT0` matches `OCR0A` or `OCR0B`, that no special effects take place. The pins `OC0A` (pin 12, Arduino `D6`) and `OC0B` (pin 11, Arduino `D5`) are not affected.

2.1.5. Interrupts

Timer/counter 0 has three interrupts that can be enabled. Sadly though, if you are using the Arduino IDE to write your code, it will sneakily use the overflow interrupt on this timer/counter, to facilitate the `millis()` function, and from that `delay()` etc are defined. You cannot define the Timer/counter 0 overflow interrupt *handler* in your own code if you compile within the Arduino IDE.

The various interrupt options are:

Parameter	Description
INT_NONE	No interrupts are required on this timer/counter. This is the default.
INT_COMPARE_MATCH_A	The <code>TIMER0 COMPA</code> interrupt is to be enabled. You are required to create an ISR function to handle it - <code>ISR(TIMER0_COMPA_vect)</code> .
INT_COMPARE_MATCH_B	The <code>TIMER0 COMPB</code> interrupt is to be enabled. You are required to create an ISR function to handle it - <code>ISR(TIMER0_COMPB_vect)</code> .
INT_OVERFLOW	The <code>TIMER0 OVF</code> interrupt is to be enabled. You are required to create an ISR function to handle it. This interrupt's handler cannot be defined in your own code in an Arduino IDE development system as the handler, <code>ISR(TIMER0_OVF_vect)</code> has been created internally and added to your code behind the scenes.

An example of initialising the timer/counter with interrupts enabled, would be:

```

ISR(TIMER0_COMPA_vect) {
    ...
}

ISR(TIMER0_COMPB_vect) {
    ...
}

Timer0::initialise(Timer0::MODE_FAST_PWM_255,
                  Timer0::CLK_PRESCALE_64,
                  Timer0::OC0X_DISCONNECTED,
                  Timer0::INT_COMPARE_MATCH_A | ①
                  Timer0::INT_COMPARE_MATCH_B, ②
                  Timer0::FORCE_COMPARE_NONE
                  );

```

- ① The interrupts parameter in action showing that the 'compare match A' and 'compare match B' interrupts are to be enabled, while the other interrupt, the timer/counter overflow interrupt, is not to be enabled here.



On Arduino systems, disabling the overflow interrupt for timer/counter 0 will stop `millis()` etc from working. The `delay()` function will also fail to work. In addition, attempting to define the timer/counter 0 overflow interrupt handler in your own (Arduino) code will fail to compile as the Arduino system already defines a handler for that interrupt. You need to write your code in something like *Atmel Studio* or *PlatformIO* to be able to use that interrupt as those development systems do not interfere with the code that you write!

You *can* obviously still enable the interrupts with `INT_OVERFLOW` enabled, you just cannot control what happens in the interrupt handler for it. The Arduino IDE has full control, you have none, but if your code relies on the Arduino `millis()` or calls `delay()` then you should always enable that interrupt.

Of course, if you change this timer/counter's initialisation away from that of the Arduino, you will affect those functions anyway.

- ② You can **or** various values together to create the full set of required interrupts, as in this example.



On an Arduino board, global interrupts are enabled as part of the Arduino initialisation code. Under other development systems, PlatformIO for example, this is not the case. Therefore, if you are developing on a system other than the Arduino IDE, and you wish to use interrupts with this timer/counter, then your code must enable global interrupts by calling the `sei()` function. `Timer0.h` will not automatically enable interrupts for you, as it is possible that this could interfere with other code in your application.

2.1.6. Force Compare Options

Timer/counter 0 can be forced to run a compare of **TCNT0** against **OCR0A** and/or **OCR0B** at any time. However, it is unlikely that this will be useful (Famous last words?) - the output pins **OC0A** (pin 12, Arduino **D6**) and **OC0B** (pin 11, Arduino **D5**) will be toggled or set according to the [compare match options](#) as long as that parameter is not set to **OCOX_DISCONNECTED** and the pin(s) in question are set to toggle, clear or set.

When the forced comparison is carried out, no interrupts will fire, if configured, and **TCNT0** will not be cleared in CTC mode with **OCR0A** as **TOP**. (Timer mode **MODE_CTC_OCR0A**.)

Setting these bits at timer initialisation is perhaps not so useful, but at least the option is there. These bits are cleared after the forced compare has taken place.

The options are:

Parameter	Description
FORCE_COMPARE_NONE	No forced comparisons will take place. This is the default.
FORCE_COMPARE_MATCH_A	A forced compare of TCNT0 against OCR0A will be carried out. You cannot use any force compare modes in anything but NORMAL and CTC modes.
FORCE_COMPARE_MATCH_B	A forced compare of TCNT0 against OCR0B will be carried out. You cannot use any force compare modes in anything but NORMAL and CTC modes.

While the default for this parameter is to have no force compares enabled, **FORCE_COMPARE_NONE**, you can be explicit if you wish, and call the **initialise()** function as follows:

```
Timer0::initialise(Timer0::MODE_FAST_PWM_255,  
                  Timer0::CLK_PRESCALE_64,  
                  Timer0::OCOX_DISCONNECTED,  
                  Timer0::INT_COMPARE_MATCH_A |  
                    Timer0::INT_COMPARE_MATCH_B,  
                  Timer0::FORCE_COMPARE_NONE    ①  
                );
```

① The force compare parameter in action showing that we are not requiring a force compare as soon as the timer is initialised.

You can, of course, initialise the timer as above, and then, in your code at any time, simply set one or other of the **FOC0A** and **FOC0B** bits in register **TCCR0B** to force a compare to affect the output pins at that point, but remember, no interrupts will fire for the compare match in that case.

Chapter 3. Timer 1

This AVR Assistant allows the simple setup and configuration of the 16 bit Timer/counter 1 on your AVR (specifically, ATmega328) micro controller. This code works in the Arduino IDE or free standing for use with some other development system, such as PlatformIO. It has not been tested on other micro controllers. (Although the ATmega168 should work.)

To use the assistant, you must include the `timer1.h` header file:

```
#include "timer1.h"
```

Following this, you may, optionally, use the `AVRAssist` namespace:

```
using namespace AVRAssist;
```



The spelling of `AVRAssist` must be as shown above.

If you choose not to do this, you must prefix everything with `AVRAssist::` or the code will not work.

3.1. Timer Initialisation

Once the header file has been included, Timer/counter 1 can be initialised as follows:

```
#include <timer1.h>

using namespace AVRAssist;

...

Timer1::initialise(Timer1::MODE CTC_OCR1A,           // Timer mode;
                  Timer1::CLK_PRESCALE_1024,        // Clock source;
                  Timer1::OC1A_TOGGLE,              // OC1A, OC1B actions on
compare match;
                  Timer1::INT_NONE,
                  Timer1::FORCE_COMPARE_NONE,        // Force Compare required?
                  Timer1::INPCAP_NOISE_CANCEL_OFF_FALLING_EDGE // Disable input
capture.
                  );

...
```

The above sets Timer/counter 1 into Clear Timer on Compare match mode with **TOP** at **OCR1A**, with a clock prescaler of 1,024. When **OCR1A** matches **TCNT1** pin **OC1A** will toggle but when **OCR1B** matches **TCNT1** nothing happens to pin **OC1B**. No interrupts will be fired for this timer.

3.1.1. Initialisation Function

The header file exposes a single `initialise` function which is defined as follows:

```
void initialise(const uint8_t timerMode,
               const clockSource_t clockSource,
               const compareMatch_t compareMatch = OC1X_DISCONNECTED,
               const interrupt_t enableInterrupts = INT_NONE,
               const forceCompare_t forceCompare = FORCE_COMPARE_NONE,
               const inputCapture_t inputCapture =
INPCAP_NOISE_CANCEL_OFF_FALLING_EDGE)
```

3.1.2. Timer Modes

Timer/counter 1 can be initialised to run in one of 15 modes, as follows.

Mode	Parameter	Description
0	MODE_NORMAL = 0	Normal mode.
1	MODE_PC_PWM_255	8 bit Phase Correct PWM with TOP at 255.
2	MODE_PC_PWM_511	9 bit Phase Correct PWM with TOP at 511.
3	MODE_PC_PWM_1023	10 bit Phase Correct PWM with TOP at 1023.
4	MODE_CTC_OCR1A	Clear Timer on Compare (CTC) with TOP at OCR1A.
5	MODE_FAST_PWM_255	8 bit Fast PWM with TOP at 255.
6	MODE_FAST_PWM_511	9 bit Fast PWM with TOP at 511.
7	MODE_FAST_PWM_1023	10 bit Fast PWM with TOP at 1023.
8	MODE_PC_FC_PWM_ICR1	Phase and Frequency Correct PWM with TOP at ICR1.
9	MODE_PC_FC_PWM_OCR1A	Phase and Frequency Correct PWM with TOP at OCR1A.
10	MODE_PC_PWM_ICR1	Phase Correct PWM with TOP at ICR1.
11	MODE_PC_PWM_OCR1A	Phase Correct PWM with TOP at OCR1A.
12	MODE_CTC_ICR1	Clear Timer on Compare (CTC) with TOP at ICR1.
13	MODE_RESERVED_13	Reserved, do not use.
14	MODE_FAST_PWM_ICR1	Fast PWM with TOP at ICR1.
15	MODE_FAST_PWM_OCR1A	Fast PWM with TOP at OCR1A.

You use this parameter to define the mode that you wish the timer/counter to run in. It should be obvious, I hope, that only one of the above modes can be used, however, if you wish to `or` them together, be it on your own head!

```

Timer1::initialise(Timer1::MODE_CTC_OCR1A,
                  Timer1::CLK_PRESCALE_1024,
                  Timer1::OC1A_TOGGLE,
                  Timer1::INT_NONE,
                  Timer1::FORCE_COMPARE_NONE,
                  Timer1::INPCAP_NOISE_CANCEL_OFF_FALLING_EDGE
                  );

```

① The timer mode parameter in action enabling the timer/counter in CTC mode with TOP at OCR1A.

3.1.3. Clock Sources

The timer/counter needs a clock source to actually start it running as a timer, or as a counter. The table below lists the available options for Timer/counter 1.

Parameter	Description
CLK_DISABLED	The timer/counter will stopped.
CLK_PRESCALE_1	The timer/counter will be running at F_CPU/1
CLK_PRESCALE_8	The timer/counter will be running at F_CPU/8
CLK_PRESCALE_64	The timer/counter will be running at F_CPU/64
CLK_PRESCALE_256,	The timer/counter will be running at F_CPU/256
CLK_PRESCALE_1024	The timer/counter will be running at F_CPU/1024
CLK_T1_FALLING	The timer/counter will be clocked from pin 11, aka T1, Arduino pin D5 on a falling edge.
CLK_T1_RISING	The timer/counter will be clocked from pin 11, aka T1, Arduino pin D5 on a rising edge.

You use this mode to define how fast the timer/counter will count, or, to disable the timer.

```

Timer1::initialise(Timer1::MODE_CTC_OCR1A,
                  Timer1::CLK_PRESCALE_1024,
                  Timer1::OC1A_TOGGLE,
                  Timer1::INT_NONE,
                  Timer1::FORCE_COMPARE_NONE,
                  Timer1::INPCAP_NOISE_CANCEL_OFF_FALLING_EDGE
                  );

```

① The clock source parameter in action showing that the timer/counter will be running at a speed defined by the system clock, F_CPU, divided by 1,024.

3.1.4. Compare Match Options

This parameter allows you to indicate what actions you want the AVR micro controller to perform on pins OC1A (pin 15, Arduino pin D9) and/or OC1B (pin 16, Arduino pin D10) when the value in TCNT1 matches OCR1A or OCR1B. The allowed values are:

Parameter	Description
OC1X_DISCONNECTED	The two OC1x pins will not be affected when the timer count matches either OCR1A or OCR1B . This is the default.
OC1A_TOGGLE	Pin OC1A will toggle when TCNT1 matches OCR1A .
OC1A_CLEAR	Pin OC1A will be reset LOW when TCNT1 matches OCR1A .
OC1A_SET	Pin OC1A will be set HIGH when TCNT1 matches OCR1A .
OC1B_TOGGLE	Pin OC1B will toggle when TCNT1 matches OCR1B .
OC1B_CLEAR	Pin OC1B will be reset LOW when TCNT1 matches OCR1B .
OC1B_SET	Pin OC1B will be set HIGH when TCNT1 matches OCR1B .

An example of initialising the timer/counter using this parameter is:

```
Timer1::initialise(Timer1::MODE CTC_OCR1A,
                  Timer1::CLK_PRESCALE_1024,
                  Timer1::OC1A_TOGGLE,           ①
                  Timer1::INT_NONE,
                  Timer1::FORCE_COMPARE_NONE,
                  Timer1::INPCAP_NOISE_CANCEL_OFF_FALLING_EDGE
                );
```

- ① The compare match parameter in action showing that when **TCNT1** matches **OCR1A**, pin **OC1A** (pin 15, Arduino **D9**) will toggle while **OC1B** (pin 16, Arduino **D10**) will not be affected when **TCNT1** matches **OCR1B**.

3.1.5. Interrupts

Timer/counter 1 has four interrupts that can be enabled and these are:

Parameter	Description
INT_NONE	No interrupts are required on this timer/counter. This is the default.
INT_CAPTURE	The TIMER1_CAPT (input capture) interrupt is to be enabled. You are required to create an ISR function to handle it - ISR(TIMER1_CAPT_vect) .
INT_COMPARE_MATCH_A	The TIMER1_COMP A interrupt is to be enabled. You are required to create an ISR function to handle it - ISR(TIMER1_COMPA_vect) .
INT_COMPARE_MATCH_B	The TIMER1_COMP B interrupt is to be enabled. You are required to create an ISR function to handle it - ISR(TIMER1_COMPB_vect) .
INT_OVERFLOW	The TIMER1_OVF interrupt is to be enabled. You are required to create an ISR function to handle it - ISR(TIMER1_OVF_vect) .

An example of initialising the timer/counter with one interrupt enabled, would be:


```

Timer1::initialise(Timer1::MODE_CTC_OCR1A,
                  Timer1::CLK_PRESCALE_1024,
                  Timer1::OC1A_TOGGLE,
                  Timer1::INT_NONE,
                  Timer1::FORCE_COMPARE_NONE,
                  Timer1::INPCAP_NOISE_CANCEL_OFF_FALLING_EDGE
                  );

```

① The interrupts parameter in action showing that there are no interrupts to be enabled for this timer. If required, you can **or** various values together to create the full set of required interrupts.



You don't have to activate the input capture interrupt (**INT_CAPTURE**) if you don't wish to, you can poll (not always a good idea) bit **ICF1** in register **TIFR1** and when it is set, an event has occurred.



On an Arduino board, global interrupts are enabled as part of the Arduino initialisation code. Under other development systems, PlatformIO for example, this is not the case. Therefore, if you are developing on a system other than the Arduino IDE, and you wish to use interrupts with this timer/counter, then your code must enable global interrupts by calling the **sei()** function. **Timer1.h** will not automatically enable interrupts for you, as it is possible that this could interfere with other code in your application.

3.1.6. Force Compare Options

Timer/counter 1 can be forced to run a compare of **TCNT1** against **OCR1A** and/or **OCR1B** at any time. However, it is unlikely that this will be particularly useful - Famous last words? When actioned, the output pins **OC1A** (pin 15, Arduino **D9**) and **OC1B** (pin 16, Arduino **D10**) will be toggled or set according to the **compare match options** as long as that parameter is not set to **OC1X_DISCONNECTED** and the pin(s) in question are set to toggle, clear or set.

These options, if enabled, are only ever actioned when the timer/counter is running in a mode other than any of the PWM modes.

When a forced comparison is carried out, no interrupts will fire, even if configured, and **TCNT1** will not be cleared in CTC mode with **OCR1A** as **TOP**. (Timer mode **MODE_CTC_OCR1A**.)

Setting these bits at timer initialisation is perhaps not so useful, but at least the option is there. These bits are cleared immediately after the forced compare has taken place.

The options are:

Parameter	Description
FORCE_COMPARE_NONE	No forced comparisons will take place. This is the default.
FORCE_COMPARE_MATCH_A	A forced compare of TCNT1 against OCR1A will be carried out.
FORCE_COMPARE_MATCH_B	A forced compare of TCNT1 against OCR1B will be carried out.

While the default for this parameter is to have no force compares enabled, `FORCE_COMPARE_NONE`, you can be explicit if you wish, and call the `initialise()` function as follows:

```
Timer1::initialise(Timer1::MODE_CTC_OCR1A,  
                  Timer1::CLK_PRESCALE_1024,  
                  Timer1::OC1A_TOGGLE,  
                  Timer1::INT_NONE,  
                  Timer1::FORCE_COMPARE_NONE, ①  
                  Timer1::INPCAP_NOISE_CANCEL_OFF_FALLING_EDGE  
                  );
```

① The force compare parameter in action showing that we are not requiring a force compare as soon as the timer is initialised.

You can, of course, initialise the timer as above, and then, in your code at any time, simply set one or other of the `FOC1A` and `FOC1B` bits in register `TCCR1C` to force a compare to affect the output pins at that point, but remember, no interrupts will fire for the compare match in that case.

3.1.7. Input Capture

Timer/counter 1 has an input capture facility which allows it to record a 'timestamp' when an event happens on pin 14, `ICP1`, Arduino pin `D8`. This parameter allows the timer to be configured as required, or for the input capture to be disabled - the default setting.

The permitted values are shown in the following table.

Parameter	Description
<code>INPCAP_NOISE_CANCEL_OFF_FALLING_EDGE</code>	The input capture is running with no noise cancelling and will be triggered on a falling edge on <code>ICP1</code> .
<code>INPCAP_NOISE_CANCEL_OFF_RISING_EDGE</code>	The input capture is running with no noise cancelling and will be triggered on a rising edge on <code>ICP1</code> .
<code>INPCAP_NOISE_CANCEL_ON_FALLING_EDGE</code>	The input capture is running with noise cancelling enabled and will be triggered on a falling edge on <code>ICP1</code> .
<code>INPCAP_NOISE_CANCEL_ON_RISING_EDGE</code>	The input capture is running with noise cancelling enabled and will be triggered on a rising edge on <code>ICP1</code> .



When **ICR1** is used as the **TOP** value in timer mode **MODE_PC_FC_PWM_ICR1**, **MODE_PC_PWM_ICR1**, **MODE CTC_ICR1** or **MODE_FAST_PWM_1CR1**, then the **ICP1** (pin 14, Arduino pin **D8**) is disconnected from the input capture circuitry meaning that the input capture function is disabled.

You can still set the bits in *any* timer mode, obviously, but they won't work if the mode is one of the PWM modes.

Yes, I know, they *are* long names!

```
Timer1::initialise(Timer1::MODE CTC_OCR1A,  
                  Timer1::CLK_PRESCALE_1024,  
                  Timer1::OC1A_TOGGLE,  
                  Timer1::INT_NONE,  
                  Timer1::FORCE_COMPARE_NONE,  
                  Timer1::INPCAP_NOISE_CANCEL_OFF_FALLING_EDGE ①  
                  );
```

- ① The input capture parameter in action showing that we wish to have input capture noise cancelling turned off, and the input to be triggered on a falling edge on **ICP1**. As no interrupts have been enabled for the input capture, the code is assumed to be polling bit **ICF1** in register **TIFR1** to determine when an event occurred.

Chapter 4. Timer 2

This AVR Assistant allows the simple setup and configuration of the 8 bit Timer/Counter 2 on your AVR (specifically, ATmega328) micro controller. This code works in the Arduino IDE or free standing for use with some other development system, such as PlatformIO. It has not been tested on other micro controllers. (Although the ATmega168 should work.)

To use the assistant, you must include the `timer2.h` header file:

```
#include "timer2.h"
```

Following this, you may, optionally, use the `AVRAssist` namespace:

```
using namespace AVRAssist;
```



The spelling of `AVRAssist` must be as shown above.

If you choose not to do this, you must prefix everything with `AVRAssist::` or the code will not work.

4.1. Timer Initialisation

Once the header file has been included, Timer/counter 2 can be initialised as follows:

```
#include <timer2.h>

using namespace AVRAssist;

...

Timer2::initialise(Timer2::MODE_FAST_PWM_255,           // Timer mode;
                  Timer2::CLK_PRESCALE_64,              // Clock source;
                  Timer2::OC2X_DISCONNECTED,             // OC2A, OC2B actions on
compare match;
                  Timer2::INT_COMPARE_MATCH_A |         // Interrupt(s) to enable;
                  Timer2::INT_COMPARE_MATCH_B,
                  Timer2::FORCE_COMPARE_NONE            // Force Compare required?
                  );

...
```

The above sets Timer/counter 2 into fast PWM mode with `TOP` = to 255, with a clock prescaler of 64. When `OCR2A` and `OCR2B` match `TCNT2` nothing happens to pins `OC2A` or `OC2B` but an interrupt will be fired when the comparison happens to match.

4.1.1. Initialisation Function

The header file exposes a single `initialise` function which is defined as follows:

```
void initialise(const uint8_t timerMode,
               const clockSource_t clockSource,
               const compareMatch_t compareMatch = OC2X_DISCONNECTED,
               const interrupt_t enableInterrupts = INT_NONE
               const forceCompare_t forceCompare = FORCE_COMPARE_NONE) {
```

4.1.2. Timer Modes

Timer/counter 2 can be initialised to run in one of 6 modes, as follows.

Mode	Parameter	Description
0	MODE_NORMAL	Normal mode.
1	MODE_PC_PWM_255	Phase Correct PWM with TOP at 255.
2	MODE CTC_OCR2A	Clear Timer on Compare (CTC) with TOP at OCR2A.
3	MODE_FAST_PWM_255	Fast PWM with TOP at 255.
4	MODE_RESERVED_4	Reserved, do not use.
5	MODE_PC_PWM_OCR2A	Phase Correct PWM with TOP at OCR2A.
6	MODE_RESERVED_6	Reserved, do not use.
7	MODE_FAST_PWM_OCR2A	Fast PWM with TOP at OCR2A.

You use this parameter to define the mode that you wish the timer/counter to run in. It should be obvious, I hope, that only one of the above modes can be used, however, if you wish to `or` them together, be it on your own head!

```
Timer2::initialise(Timer2::MODE_FAST_PWM_255,           ①
                   Timer2::CLK_PRESCALE_64,
                   Timer2::OC2X_DISCONNECTED,
                   Timer2::INT_COMPARE_MATCH_A |
                   Timer2::INT_COMPARE_MATCH_B,
                   Timer2::FORCE_COMPARE_NONE
                   );
```

① The timer mode parameter in action enabling the timer/counter in fast PWM mode with TOP defined by the value 255.

4.1.3. Clock Sources

The timer/counter needs a clock source to actually start it running as a timer, or as a counter. The following options are available for Timer/counter 2 and are slightly different from Timer/counter 0, the other 8 bit timer/counter, as there are no facilities to clock this timer/counter externally. It also has an additional two prescaler options over Timer/counter 0.

Parameter	Description
CLK_DISABLED	The timer/counter will stopped.
CLK_PRESCALE_1	The timer/counter will be running at F_CPU/1
CLK_PRESCALE_8	The timer/counter will be running at F_CPU/8
CLK_PRESCALE_32	The timer/counter will be running at F_CPU/32
CLK_PRESCALE_64	The timer/counter will be running at F_CPU/64
CLK_PRESCALE_128	The timer/counter will be running at F_CPU/128
CLK_PRESCALE_256,	The timer/counter will be running at F_CPU/256
CLK_PRESCALE_1024	The timer/counter will be running at F_CPU/1024

You use this mode to define how fast the timer/counter will count, or, to disable the timer.

```

Timer2::initialise(Timer2::MODE_FAST_PWM_255,
                  Timer2::CLK_PRESCALE_64,           ①
                  Timer2::OC2X_DISCONNECTED,
                  Timer2::INT_COMPARE_MATCH_A |
                    Timer2::INT_COMPARE_MATCH_B,
                  Timer2::FORCE_COMPARE_NONE
                  );

```

① The clock source parameter in action showing that the timer/counter will be running at a speed defined by the system clock, **F_CPU**, divided by 64.

4.1.4. Compare Match Options

This parameter allows you to indicate what actions you want the AVR micro controller to perform on pins **OC2A** (pin 17, Arduino pin **D11**) and/or **OC2B** (pin 5, Arduino pin **D3**) when the value in **TCNT2** matches **OCR2A** or **OCR2B**. The allowed values are:

Parameter	Description
OC2X_DISCONNECTED	The two OC2x pins will not be affected when the timer count matches either OCR2A or OCR2B . This is the default.
OC2A_TOGGLE	Pin OC2A will toggle when TCNT2 matches OCR2A .
OC2A_CLEAR	Pin OC2A will be reset LOW when TCNT2 matches OCR2A .
OC2A_SET	Pin OC2A will be set HIGH when TCNT2 matches OCR2A .
OC2B_TOGGLE	Pin OC2B will toggle when TCNT2 matches OCR2B . You cannot use OC2B_TOGGLE in anything but NORMAL and CTC modes.
OC2B_CLEAR	Pin OC2B will be reset LOW when TCNT2 matches OCR2B .
OC2B_SET	Pin OC2B will be set HIGH when TCNT2 matches OCR2B .

An example of initialising the timer/counter using this parameter is:

```

Timer2::initialise(Timer2::MODE_FAST_PWM_255,
                  Timer2::CLK_PRESCALE_64,
                  Timer2::OC2X_DISCONNECTED,           ①
                  Timer2::INT_COMPARE_MATCH_A |
                  Timer2::INT_COMPARE_MATCH_B,
                  Timer2::FORCE_COMPARE_NONE
                );

```

- ① The compare match parameter in action showing that when **TCNT2** matches **OCR2A** or **OCR2B**, that no special effects take place. The pins **OC2A** (pin 17, Arduino **D11**) and **OC2B** (pin 5, Arduino **D3**) are not affected.

4.1.5. Interrupts

Timer/counter 2 has three interrupts that can be enabled and these are:

Parameter	Description
INT_NONE	No interrupts are required on this timer/counter. This is the default.
INT_COMPARE_MATCH_A	The TIMER2 COMPA interrupt is to be enabled. You are required to create an ISR function to handle it - ISR(TIMER2_COMPA_vect) .
INT_COMPARE_MATCH_B	The TIMER2 COMPB interrupt is to be enabled. You are required to create an ISR function to handle it - ISR(TIMER2_COMPB_vect) .
INT_OVERFLOW	The TIMER2 OVF interrupt is to be enabled. You are required to create an ISR function to handle it - ISR(TIMER2_OVF_vect) .

An example of initialising the timer/counter with interrupts enabled, would be:

```

ISR(TIMER2_COMPA_vect) {
    ...
}

ISR(TIMER2_COMPB_vect) {
    ...
}

Timer2::initialise(Timer2::MODE_FAST_PWM_255,
                  Timer2::CLK_PRESCALE_64,
                  Timer2::OC2X_DISCONNECTED,
                  Timer2::INT_COMPARE_MATCH_A |           ①
                  Timer2::INT_COMPARE_MATCH_B,           ②
                  Timer2::FORCE_COMPARE_NONE
                );

```

- ① The interrupts parameter in action showing that the 'compare match A' and 'compare match B' interrupts are to be enabled, while the other interrupt, the timer/counter overflow interrupt, is not to be enabled.
- ② You can **or** various values together to create the full set of required interrupts, as in this

example.



On an Arduino board, global interrupts are enabled as part of the Arduino initialisation code. Under other development systems, PlatformIO for example, this is not the case. Therefore, if you are developing on a system other than the Arduino IDE, and you wish to use interrupts with this timer/counter, then your code must enable global interrupts by calling the `sei()` function. `Timer2.h` will not automatically enable interrupts for you, as it is possible that this could interfere with other code in your application.

4.1.6. Force Compare Options

Timer/counter 2 can be forced to run a compare of `TCNT2` against `OCR2A` and/or `OCR2B` at any time. However, it is unlikely that this will be particularly useful - Famous last words? When actioned, the output pins `OC2A` (pin 17, Arduino `D11`) and `OC2B` (pin 5, Arduino `D3`) will be toggled or set according to the [compare match options](#) as long as that parameter is not set to `OC2X_DISCONNECTED` and the pin(s) in question are set to toggle, clear or set.

When the forced comparison is carried out, no interrupts will fire, if configured, and `TCNT2` will not be cleared in CTC mode with `OCR2A` as `TOP`. (Timer mode `MODE_CTC_OCR2A`.)

Setting these bits at timer initialisation is perhaps not so useful, but at least the option is there. These bits are cleared immediately after the forced compare has taken place.

The options are:

Parameter	Description
<code>FORCE_COMPARE_NONE</code>	No forced comparisons will take place. This is the default.
<code>FORCE_COMPARE_MATCH_A</code>	A forced compare of <code>TCNT2</code> against <code>OCR2A</code> will be carried out. You cannot use any force compare modes in anything but NORMAL and CTC modes.
<code>FORCE_COMPARE_MATCH_B</code>	A forced compare of <code>TCNT2</code> against <code>OCR2B</code> will be carried out. You cannot use any force compare modes in anything but NORMAL and CTC modes.

While the default for this parameter is to have no force compares enabled, `FORCE_COMPARE_NONE`, you can be explicit if you wish, and call the `initialise()` function as follows:

```
Timer2::initialise(Timer2::MODE_FAST_PWM_255,  
                  Timer2::CLK_PRESCALE_64,  
                  Timer2::OC2X_DISCONNECTED,  
                  Timer2::INT_COMPARE_MATCH_A |  
                    Timer2::INT_COMPARE_MATCH_B,  
                  Timer2::FORCE_COMPARE_NONE  
                  );
```

- ① The force compare parameter in action showing that we are not requiring a force compare as soon as the timer is initialised.

You can, of course, initialise the timer as above, and then, in your code at any time, simply set one or other of the `FOC2A` and `FOC2B` bits in register `TCCR2B` to force a compare to affect the output pins at that point, but remember, no interrupts will fire for the compare match in that case.

Chapter 5. Analogue Comparator

This AVR Assistant allows the simple setup and configuration of the Analogue Comparator on your AVR (specifically, ATmega328) micro controller. This code works in the Arduino IDE or free standing for use with some other development system, such as PlatformIO. It has not been tested on other micro controllers. (Although the ATmega168 should work.)

The comparator output will be **HIGH** whenever the reference voltage is higher than the sampled voltage, and will be **LOW** when the reference voltage is lower or equal to the sampled voltage. The comparator output is simply bit **ACO** in register **ACSR**.

To use this assistant, you must include the **comparator.h** header file:

```
#include "comparator.h"
```

Following this, you may, optionally, use the **AVRAssist** namespace:

```
using namespace AVRAssist;
```



The spelling of **AVRAssist** must be as shown above.

If you choose not to do this, you must prefix everything with **AVRAssist::** or the code will not work.

5.1. Comparator Initialisation

Once the header file has been included, the comparator can be initialised as follows:

```
#include <comparator.h>

using namespace AVRAssist;

...

Comparator::initialise(Comparator::REFV_EXTERNAL,
                       Comparator::SAMPLE_AIN1,
                       Comparator::INT_TOGGLE);

...
```

The above sets the comparator to use **AIN0**, pin 12, Arduino pin **D6** as the reference voltage, **AIN1**, pin 13, Arduino pin **D7** as the voltage to be sampled and compared, and an interrupt to fire whenever the comparator output toggles.

5.1.1. Initialisation Function

The header file exposes a single **initialise** function which is defined as follows:

```
void initialise(const reference_t referenceSource,
               const sample_t sampleSource,
               const interrupt_t interruptMode = 0);
```

5.1.2. Reference Voltage

The comparator can be configured to use one of two separate voltage sources as the reference voltage, these being:

Parameter	Description
REFV_EXTERNAL	Pin 12, AIN0 , Arduino pin D6 will be used as the reference voltage.
REFV_INTERNAL	The internal 1.1 volt bandgap reference voltage will be used.



The data sheet advises strongly, that you do not change to **REFV_INTERNAL** if the **AREF** pin is connected to any external source of voltage. You will let the magic blue smoke out if you do.

Many of the 'breadboard Arduinos' on the internet, show the **AREF** pin connected to 5V - this is a bad thing if you ever configure the Analogue Comparator or the ADC to use the internal voltage as a reference.

The only thing you should connect to the **AREF** pin is a 100nF capacitor to ground.

You use this parameter to define the reference voltage to be used, as follows:

```
Comparator::initialise(Comparator::REFV_EXTERNAL, ①
                       Comparator::SAMPLE_AIN1,
                       Comparator::INT_TOGGLE);
```

① The comparator will be set up using the **AIN0** pin as its reference voltage.

5.1.3. Sample Voltage

The comparator needs a second voltage source, this one is to be compared with the reference voltage described above. The following values are permitted:

Parameter	Description
SAMPLE_ADC0	Compare the voltage on pin PC0 (Arduino A0) with the reference voltage.
SAMPLE_ADC1	Compare the voltage on pin PC1 (Arduino A1) with the reference voltage.
SAMPLE_ADC2	Compare the voltage on pin PC2 (Arduino A2) with the reference voltage.
SAMPLE_ADC3	Compare the voltage on pin PC3 (Arduino A3) with the reference voltage.
SAMPLE_ADC4	Compare the voltage on pin PC4 (Arduino A4) with the reference voltage.
SAMPLE_ADC5	Compare the voltage on pin PC5 (Arduino A5) with the reference voltage.

SAMPLE_ADC6	Compare the voltage on pin ADC6 with the reference voltage. (SMD version only.)
SAMPLE_ADC7	Compare the voltage on pin ADC7 with the reference voltage. (SMD version only.)
SAMPLE_AIN1	Compare the voltage on pin AIN1 with the reference voltage.



The various dual inline versions of the ATmega328 do not have pins **ADC6** and **ADC7**, those two are only present on the surface mount versions. Some Arduino Uno clones have been built with a surface mount version of the ATmega328, and on those boards, *some* manufacturers have connected these two pins to a header while others leave them unconnected.

You use this parameter to define which pin will be use to source the voltage to be compared with the reference voltage.

```
Comparator::initialise(Comparator::REFV_EXTERNAL,
                      Comparator::SAMPLE_AIN1,
                      Comparator::INT_TOGGLE);
```

①

① The comparator will be set up using the **AIN1** pin as its sample voltage source.

5.1.4. Interrupts

The comparator has a single interrupt, but it has thee separate manners of firing it. These are:

Parameter	Description
INT_NONE	The comparator will not raise any interrupts. The code is assumed to be monitoring bit ACO in register ACSR .
INT_TOGGLE	When bit ACO in register ACSR toggles, the ANALOG_COMP interrupt will be fired, you are required to create an interrupt handler for it - ISR(ANALOG_COMP) .
INT_FALLING	When bit ACO in register ACSR changes from HIGH to LOW , the ANALOG_COMP interrupt will be fired, you are required to create an interrupt handler for it - ISR(ANALOG_COMP) .
INT_RISING	When bit ACO in register ACSR changes from LOW to HIGH , the ANALOG_COMP interrupt will be fired, you are required to create an interrupt handler for it - ISR(ANALOG_COMP) .

An example of initialising the comparator using this parameter is:

```
ISR(ANALOG_COMP) {  
    ...  
}  
  
Comparator::initialise(Comparator::REFV_EXTERNAL,  
                       Comparator::SAMPLE_AIN1,  
                       Comparator::INT_TOGGLE); ①
```

① The comparator will be set up so that the interrupt will be fired whenever the **AC0** bit changes.



On an Arduino board, global interrupts are enabled as part of the Arduino initialisation code. Under other development systems, PlatformIO for example, this is not the case. Therefore, if you are developing on a system other than the Arduino IDE, and you wish to use interrupts with the comparator, then your code must enable global interrupts by calling the `sei()` function. `Comparator.h` will not automatically enable interrupts for you, as it is possible that this could interfere with other code in your application.

Chapter 6. Watchdog Timer

This AVR Assistant allows the simple setup and configuration of the Watchdog Timer on your AVR (specifically, ATmega328) micro controller. This code works in the Arduino IDE or free standing for use with some other development system, such as PlatformIO. It has not been tested on other micro controllers. (Although the ATmega168 should work.)

To use this assistant, you must include the `watchdog.h` header file:

```
#include "watchdog.h"
```

Following this, you may, optionally, use the `AVRAssist` namespace:

```
using namespace AVRAssist;
```



The spelling of `AVRAssist` must be as shown above.

If you choose not to do this, you must prefix everything with `AVRAssist::` or the code will not work.

6.1. Watchdog Timer Initialisation

Once the header file has been included, the watchdog can be initialised as follows:

```
#include <watchdog.h>

using namespace AVRAssist;

...

Watchdog::initialise(Watchdog::WDT_TIMEOUT_8S,
                    Watchdog::WDT_MODE_INTERRUPT);

...
```

The above sets the watchdog to timeout every 8 seconds, and to run in Watchdog Interrupt mode, WDI. In this mode, there is no need to call the `wdt_reset()` function before the timeout occurs, that is only required in `WDT_MODE_RESET` and `WDT_MODE_BOTH` modes, and is simply to prevent the watchdog from resetting the device.

6.1.1. Initialisation Function

The header file exposes a single `initialise` function which is defined as follows:

```
void initialise(const timeout_t timeout,
               const mode_t mode);
```

6.1.2. Timeout Setting

The watchdog timer has 10 available timeouts ranging from 16 milliseconds up to 8 seconds. These are as follows:

Parameter	Description
WDT_TIMEOUT_16MS	Watchdog times out after 16 milliseconds.
WDT_TIMEOUT_32MS	Watchdog times out after 32 milliseconds.
WDT_TIMEOUT_64MS	Watchdog times out after 64 milliseconds.
WDT_TIMEOUT_125MS	Watchdog times out after 125 milliseconds.
WDT_TIMEOUT_250MS	Watchdog times out after 250 milliseconds.
WDT_TIMEOUT_500MS	Watchdog times out after 500 milliseconds.
WDT_TIMEOUT_1S	Watchdog times out after 1 Seconds.
WDT_TIMEOUT_2S	Watchdog times out after 2 Seconds.
WDT_TIMEOUT_4S	Watchdog times out after 4 Seconds.
WDT_TIMEOUT_8S	Watchdog times out after 8 Seconds.

You use this parameter to define the timeout period of the watchdog timer, as follows:

```
Watchdog::initialise(Watchdog::WDT_TIMEOUT_8S,           ①  
                    Watchdog::WDT_MODE_INTERRUPT);
```

① The watchdog will be set up so that it times out every 8 seconds.

6.1.3. Watchdog Mode

The watchdog timer can run in one of three modes, as described in the following table:

Parameter	Description
WDT_MODE_RESET	The device will be reset when the watchdog times out, unless reset.
WDT_MODE_INTERRUPT	The device will execute an interrupt, <code>ISR(WDT_vect)</code> , when it times out. It will not reset the device.
WDT_MODE_BOTH	The device will fire the interrupt, <code>ISR(WDT_vect)</code> , when it times out the first time. It will then reset the device when it times out for the second time - unless reset.

You use this parameter to define the mode in which the watchdog timer is to run.

```
Watchdog::initialise(Watchdog::WDT_TIMEOUT_8S,  
                    Watchdog::WDT_MODE_INTERRUPT);           ①
```

① The watchdog will be set up so that it times out every 8 seconds, firing an interrupt but allowing the device to continue operation without being reset.



If the mode contains a reset, `WDT_MODE_RESET` and `WDT_MODE_BOTH`, then there must be a call to `wdt_reset()` before the timeout period expires, otherwise the watchdog will reset the system - on the very first timeout in `WDT_MODE_RESET` or on the second in `WDT_MODE_BOTH`.



On an Arduino board, global interrupts are enabled as part of the Arduino initialisation code. Under other development systems, PlatformIO for example, this is not the case. Therefore, if you are developing on a system other than the Arduino IDE, and you wish to use interrupts with the watchdog, then your code must enable global interrupts by calling the `sei()` function. `Watchdog.h` will not automatically enable interrupts for you, as it is possible that this could interfere with other code in your application, however, it will preserve the existing state of the global interrupt flag in the status register when `Watchdog::initialise()` is called. If global interrupts are enabled they will remain enabled, and if disabled, they will remain that way too.

Appendix A: Foibles

The following foibles have been detected with the `AVRAssist` code. Some of these *may* be able to be fixed in future, and if so, this appendix will be updated to show that such a thing has happened. Some might not be fixable.

A.1. General - Interrupts

When using interrupts in your code, there is one major difference when using the Arduino IDE and other development systems such as PlatformIO. In the Arduino IDE, all sketches are started with the global interrupts enabled. This is done so that the `Timer/Counter 0 overflow interrupt` can be used to count up and maintain the `millis()` function and everything else that relies on it. The `Serial` interface also uses interrupts.

With other development systems, which do not have the *hand holding* of the Arduino IDE, this is not done. If your code needs interrupts, then it is up to your code to enable global interrupts at the appropriate time, by calling the `sei()` function.

`AVRAssist` cannot simply enable interrupts - for example, when you select that an interrupt be used - because there may be further initialisation in your code that needs to be done before the interrupts are enabled. For this reason, it is still the responsibility of your code to enable interrupts as and when required.

A.2. General - Timers

For some, currently unknown, reason, setting up timers in the Arduino IDE differs from setting them up in other development environments. Obviously, the Arduino *hand holding* code, that happens in the background, has its own requirements for the various timer/counter and sets them up accordingly. `AnalogWrite()` for example, relies on the three timer/counters being set up correctly. However, it should be the case that calling `Timer1::initialise()`, for example, will *override* all the Arduino settings and define the settings that your code requires.

Strangely, this does not appear to happen correctly, although I have examined the generated assembly code, and it does appear to do the required setup, overwriting the setup created by the Arduino IDE, it causes some code not to work when compiled in the Arduino IDE. The following, for example, fails in the Arduino IDE:

```
OCR1A = 31249;

Timer1::initialise( Timer1::MODE_CTC_OCR1A,           // Timer Mode
                    Timer1::CLK_PRESCALE_256,        // Clock Source
                    Timer1::OC1A_TOGGLE               // Compare Match
                    default.                           // Remaining parameters
                    );
```

However, it works perfectly in other development environment. The fix, is to initialise the

time/counter first, then set the required value in **OCR1A** (in this case), as follows, which works in all environments:

```
Timer1::initialise( Timer1::MODE_CTC_OCR1A,           // Timer Mode
                    Timer1::CLK_PRESCALE_256,        // Clock Source
                    Timer1::OC1A_TOGGLE              // Compare Match
                    default.                           // Remaining parameters
                    );

OCR1A = 31249;
```

It is possible that other timer/counter settings will be similarly affected.

A.3. Timer 0

A.3.1. Timer 0 - General

See the [General - Timers](#) section for details.

A.3.2. Timer 0 - Interrupts

See the [General - Interrupts](#) section for details.

A.3.3. Timer 0 - Overflow Interrupt

The ISR for this timer 0 overflow *cannot* be used when compiling code within the Arduino IDE. This is down to the fact that the Arduino IDE sets up its own ISR for this interrupt and if you (or I) define one for our own code, it will cause duplicate definition errors in the linker.

This problem cannot be fixed in the **AVRAssist** code as it is caused by the Arduino IDE. The problem does not occur in PlatformIO, for example, where you can define your own ISR for this interrupt without any problems.



Redefining this interrupt in the Arduino IDE, if it actually was possible, would lead to all sorts of problems as you would be messing with the interrupt that works the **millis()** function, and from that, the **delay()** and all the other functions that depend upon **millis()**.

A.4. Timer 1

A.4.1. Timer 0 - General

See the [General - Timers](#) section for details.

A.4.2. Timer 0 - Interrupts

See the [General - Interrupts](#) section for details.

A.5. Timer 2

A.5.1. Timer 0 - General

See the [General - Timers](#) section for details.

A.5.2. Timer 0 - Interrupts

See the [General - Interrupts](#) section for details.

A.6. Analogue Comparator

A.6.1. Analogue Comparator - Interrupts

See the [General - Interrupts](#) section for details.

A.7. Watchdog

While this is not a foible, as such, it is repeated here for information - the Data Sheet doesn't make things exactly clear on the matter.

When using the watchdog in WDI, Watchdog Interrupt, mode, there is no need to call the `wdt_reset()` function. If you are running in this mode, the WDT will not reset the device after the timeout period. If, on the other hand, you are running the WDT in either of the modes that do involve potentially resetting the device, then you must call `wdt_reset()` within the timeout period to avoid said resets. In summary then:

- WDT_MODE_INTERRUPT - there is no reset, so no need to call `wdt_reset()`;
- WDT_MODE_RESET - there is a possibility of a reset, so your code *must* call `wdt_reset()` before the defined timeout expires;
- WDT_MODE_BOTH - there is a possibility of a reset, so your code *must* call `wdt_reset()` before the defined timeout expires;

A.7.1. Watchdog - Interrupts

See the [General - Interrupts](#) section for details.