ESP8266 Watchdogs in Arduino

August 26, 2017
Updated: August 27, 2017

Arduino Sketch Managed ESP8266 Watchdog

While the hardware and software watchdog timers of the ESP8266 are essential, they are not sufficient to ensure the kind of reliability needed in an IoT device. In the next post, I discuss how to implement yet a third watchdog to further improve the dependability of the firmware programmed into this chip. In the mean time, I thought it would be useful to discuss watchdogs in general and to delve into some of the details of the ESP8266 watchdogs.

Table of Contents

toc

1. What is a Watchdog Timer?

If the man stops kicking the dog, the dog will take advantage of the hesitation and bite the man.
Niall Murphy, *Watchdog Timers*.

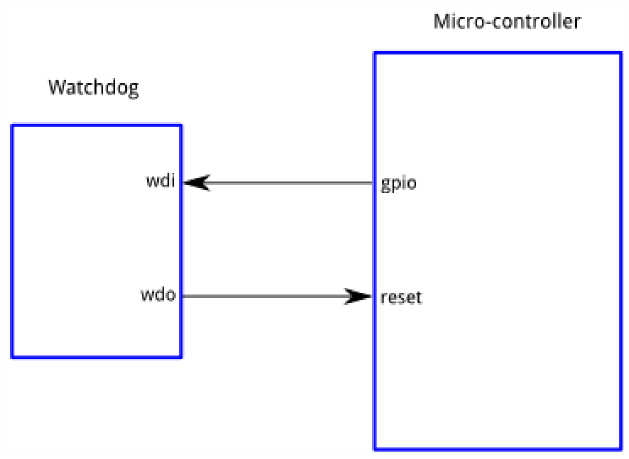
If you are upset with kicking software dogs, you could substitute a "feed" metaphor which I think is even more eloquent.

The role of a watchdog timer (abbreviated WDT) is to react to a hardware or software malfunction in a timely manner by returning a device to normal operation most often by performing a reset.

When resetting a frozen computer with a three-finger salute or by pulling the power plug or removing the battery or whatever other incantation we have stumbled upon, we are the watchdog. In embedded systems such as IoT devices where human intervention cannot be relied upon for correcting errors, a well-designed WDT can be viewed as the ultimate tool in defensive design.

A watchdog can be viewed as a counter that is regularly decremented. Properly functioning software will "feed" the watchdog which involves resetting the counter value. Presumably, runaway software will not do that and the watchdog counter will reach zero, when it is said to "bite" by resetting the device.

Better watchdog timers are actual integrated circuits that are independent of the CPU or micro-controllers that they monitor.Examples include the Texas Instrument UCC3946, Maxim MAX823/4/5 and the STMicroelectronics STWD100 chips. They operate on the following principle.



On a regular basis, the micro-controller must toggle the watchdog input pin **wdi** connected to one of its **gpio** pin. If this is not done, the watchdog times out and asserts its output pin **wdo** which can be wired to the micro-controller **reset** pin. There are more sophisticated versions of these chips less likely to be spoofed by runaway software (see Jack Ganssle *Great Watchdog Timers for Embedded Systems*).

When using an off-the-shelf device without an independent watchdog circuit, it will be necessary to rely on the built-in watchdog of its micro-controller. The watchdog timer works pretty much in the same manner. An internal register is decremented at regular intervals. If it is not reset, the processor will reboot when the register reaches zero.

References:

Barr, Michael (2001), *Introduction to Watchdog Timers*.
Murphy, Niall (2000), *Watchdog Timers*.
Ganssle, Jack (2016), *Great Watchdog Timers for Embedded Systems*.

toc

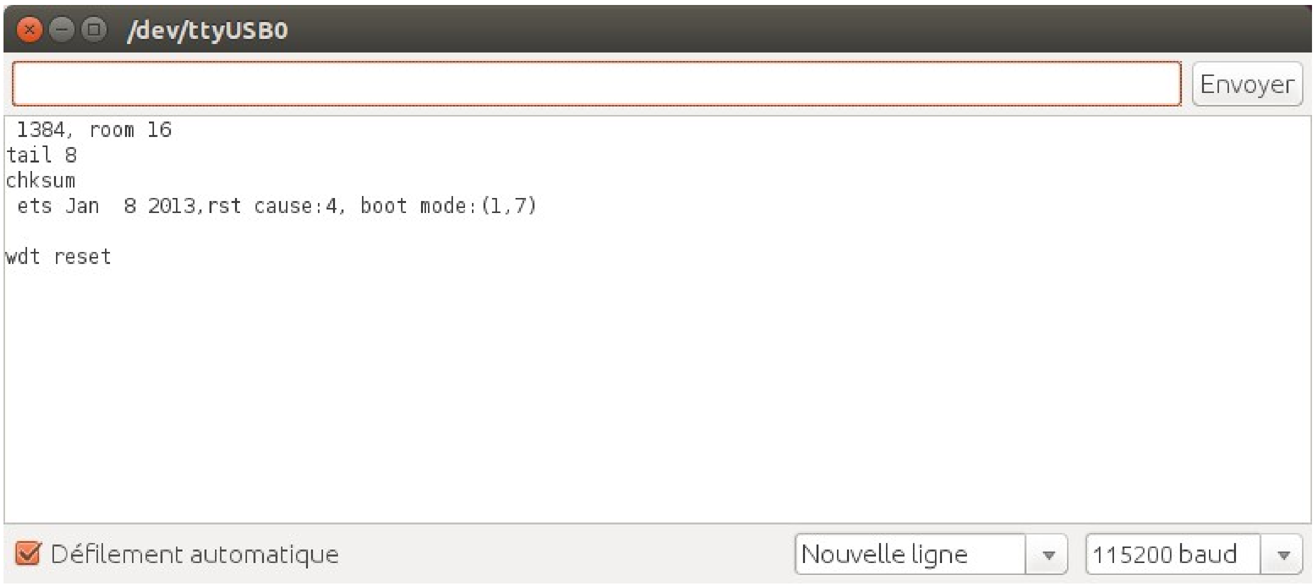
2. ESP8266 Hardware Watchdog

The ESP8266 software watchdog, which has a time-out period about half as long as that of the ESP8266 hardware watchdog feeds the latter. The following sketch makes the hardware watchdog trigger a reset out by suspending the software watchdog and then throwing the ESP into an endless empty loop.

```
void setup() {

  ESP.wdtDisable();
  while (1) {};

}

void loop() {}
```

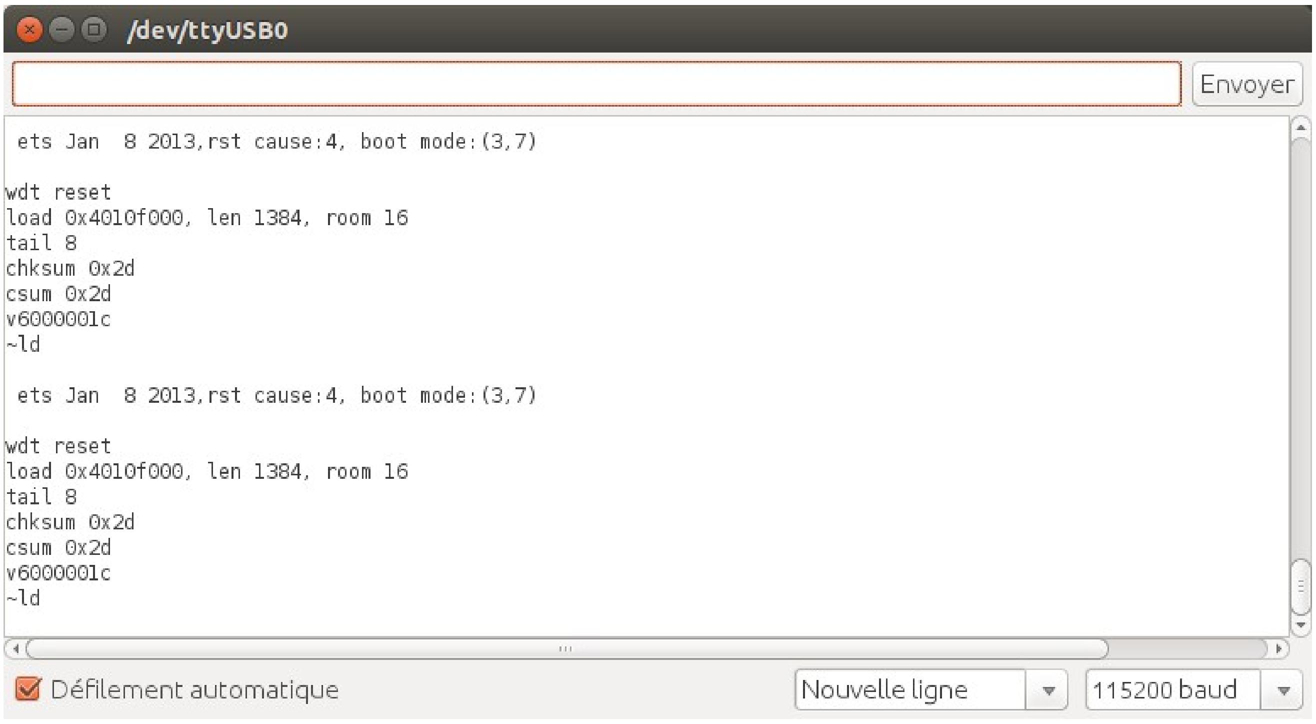
ESP is an **EspClass** object declared in **ESP.h** which is part of the ESP8266/Arduino core included in every Arduino sketch.

Open the serial monitor and upload the sketch. Wait about 8 seconds, and see something like this displayed in the serial monitor window:



```
 1384, room 16
tail 8
chksum
 ets Jan  8 2013,rst cause:4, boot mode:(1,7)

wdt reset
```

And then... nothing happens. The system is frozen! This is not good. Actually, this is a well-known problem that only occurs on the first watchdog reset after a firmware download by serial link (see esp8266/Arduino FAQ). Manually reset the ESP8266 and from then on the hardware watchdog timeouts and resets the ESP without fail.



```
 ets Jan  8 2013,rst cause:4, boot mode:(3,7)

wdt reset
load 0x4010f000, len 1384, room 16
tail 8
chksum 0x2d
csum 0x2d
v6000001c
~ld

 ets Jan  8 2013,rst cause:4, boot mode:(3,7)

wdt reset
load 0x4010f000, len 1384, room 16
tail 8
chksum 0x2d
csum 0x2d
v6000001c
~ld
```

This initial failure of the hardware watchdog is not a big deal. Presumably an ESP based device will be turned off after uploading its Arduino sketch with a computer and will be powered up again when it is installed in its final position. If uploading occurred *in situ* as it were, then remember to turn the ESP off and then back on or to reset it manually. It is said that the initial failure does not occur after an OTA upload of a sketch. I have yet to test this.

At first blush, one might desire that there be a way to hook the hardware watchdog timeout routine to save information in non-volatile memory before the reset is performed. As far as I know, this is not easily done. On second thought, only those of us that create bug-free software at first try should lament this lack of flexibility. For my part, I am glad the hardware watchdog works dependably no matter what I might do.

Since there is really nothing much more that can be done with the hardware watchdog, time to go on to the software watchdog.

Reference:

Santos, Nuno (2017), *ESP8266: Watchdog functions*.

The hardware watchdog biting code can be found on many sites on the Web. I happen to like Nuno Santos' clear exposition.

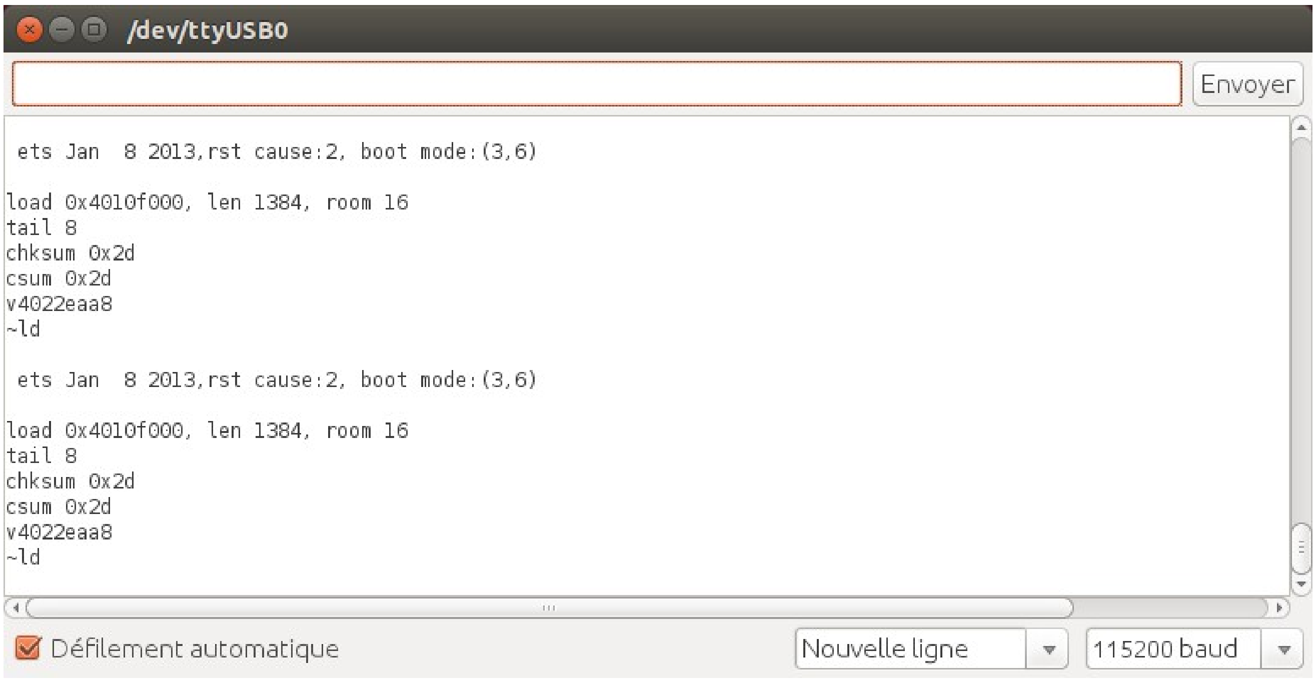
toc

3. ESP8266 Sofware Watchdog

Getting the software watchdog to bite on the ESP8266 is even easier.

```
void setup() {

  while (1) {};

}

void loop() {}
```

Again the system will freeze if the sketch is downloaded to the ESP over a serial link. After an initial power cycle or reset, the device will be locked in an endless loop of resets cause by the software watchdog every 3.2 seconds or so.



```
 ets Jan  8 2013,rst cause:2, boot mode:(3,6)

load 0x4010f000, len 1384, room 16
tail 8
chksum 0x2d
csum 0x2d
v4022eaa8
~ld

 ets Jan  8 2013,rst cause:2, boot mode:(3,6)

load 0x4010f000, len 1384, room 16
tail 8
chksum 0x2d
csum 0x2d
v4022eaa8
~ld
```

The **EspClass** object has a watchdog enable routine with a timeout parameter. Here is part of the class definition that will be useful in this blog.

```
class EspClass {
   public:
      // TODO: figure out how to set WDT timeout
      void wdtEnable(uint32_t timeout_ms = 0);
      // note: setting the timeout value is not implemented at the moment
      void wdtEnable(WDTO_t timeout_ms = WDTO_0MS);

      void wdtDisable();
      void wdtFeed();

      String getResetReason();
      String getResetInfo();
      struct rst_info * getResetInfoPtr();
```

```
     void reset();
     void restart();
```

But as the comment in the code says, the watchdog enable routine does not implement the timeout value: **`wdtEnable(0)`** and **`wdtEnable(20000)`** have exactly the same effect. They both enable the software watchdog with the same default timeout period.

There is not much one can do with the software watchdog. It can be enabled, disabled and fed with **`wdtFeed`**. We can use the latter to try to get the length of time before the watchdogs bite.

toc

4. ESP8266 Watchdogs Timeout

There seems to be conflicting information about the timeout period for the ESP8266 software watchdog. Dave Evans reports a software watchdog timeout occurs between 1.5 and 2.5 seconds on an Adafruit Huzzah ESP8286. My own test with similar code but run on a Wemos D1 mini, to puts it at between 1.62 seconds and 1.63 seconds. However pasko_zh (paško from Zurich, I am sorry but that contributor's real name is not readily available to credit him properly) is adamant that the software watchdog timeout period is 3.2 seconds. It's a value that I have seen elsewhere, so I decided to investigate the matter a bit further.

I used the following sketch (available for download esp_watchdog_timing.ino) to obtain the period of the hardware and software watchdogs.

```
/*
 * Test to establish the timeout period of the ESP8266 watchdog timers
 */

//#define SOFT_WATCHDOG     // comment out to test hardware watchdog time-out

#ifdef SOFT_WATCHDOG
  #define LOOPS 15
  #define STEP 10
  unsigned long interval = 3150;
#else
  #define LOOPS 10
  #define STEP 100
  unsigned long interval = 7900;
#endif

unsigned long etime;

void setup() {
  Serial.begin(115200);
  Serial.printf("\n\nSdk version: %s\n", ESP.getSdkVersion());
  Serial.printf("Core Version: %s\n", ESP.getCoreVersion().c_str());
  Serial.printf("Boot Version: %u\n", ESP.getBootVersion());
  Serial.printf("Boot Mode: %u\n", ESP.getBootMode());
  Serial.printf("CPU Frequency: %u MHz\n", ESP.getCpuFreqMHz());
  Serial.printf("Reset reason: %s\n", ESP.getResetReason().c_str());
  Serial.println("---------------------------");
  Serial.println("-- WATCHDOG TIMING TESTS --");
  Serial.println("---------------------------\n");

  while (1) {

    interval += STEP;

    Serial.print("  Testing ");
    #ifdef SOFT_WATCHDOG
      Serial.print("software");
    #else
```

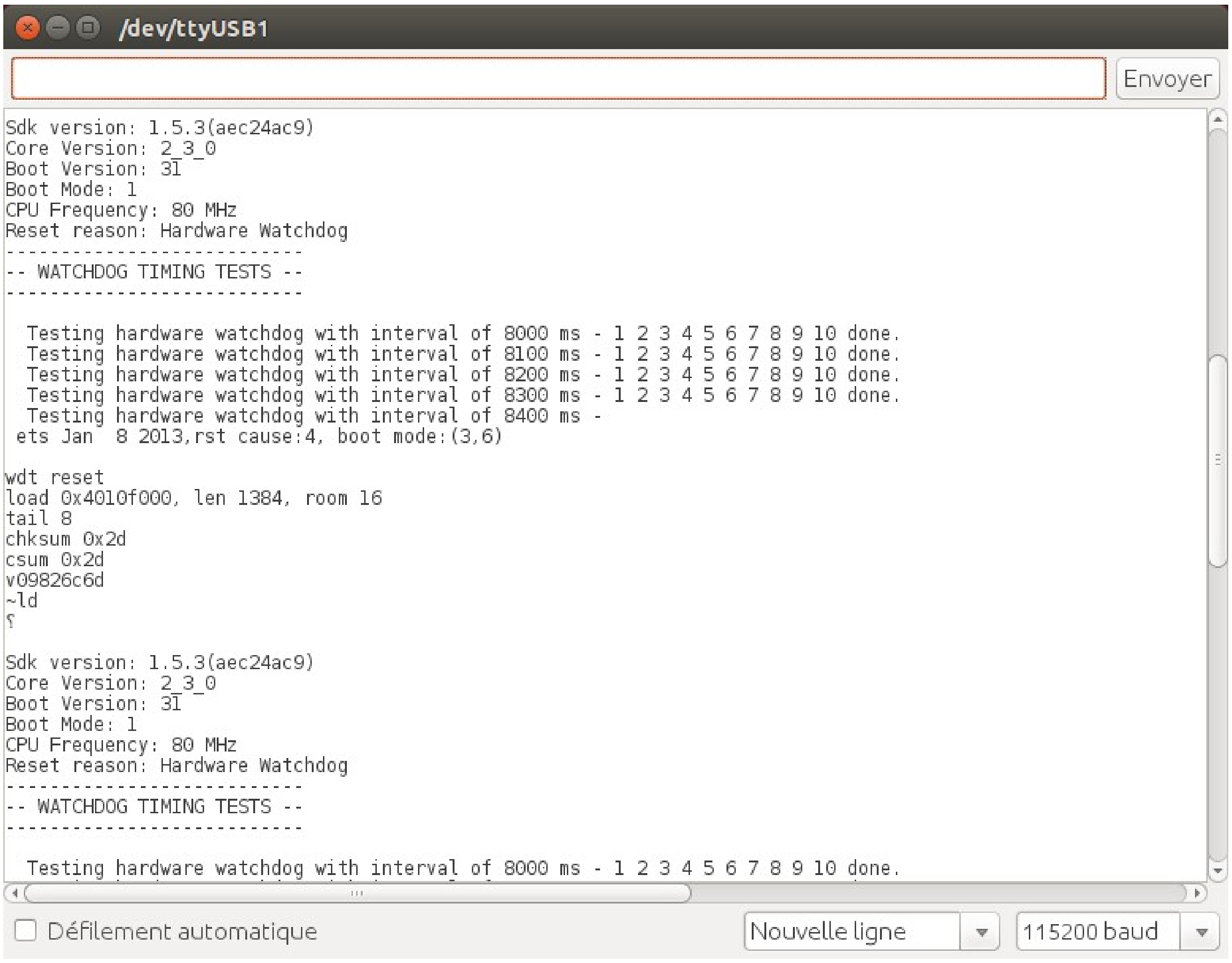
```
      Serial.print("hardware");
    #endif
    Serial.printf(" watchdog with interval of %u ms - ", interval);
    Serial.flush();

    ESP.wdtDisable();
    #ifdef SOFT_WATCHDOG
      ESP.wdtEnable(0);
    #endif

    for (int i = 0; i < LOOPS; i++) {

      ESP.wdtFeed();

      ESP.wdtDisable();
      #ifdef SOFT_WATCHDOG
        ESP.wdtEnable(0);
      #endif

      etime = millis();
      while (millis() - etime < interval) { }

      ESP.wdtDisable();
      ESP.wdtEnable(0);
      ESP.wdtFeed();
      Serial.printf("%d ", i+1);
    }
    Serial.println("done.");
  }
}

void loop() { }
```

The idea is simple enough. Do an empty loop for a length of time set with the **`interval`** variable in milliseconds:

```
  etime = millis();
  while (millis() - etime < interval) { }
```

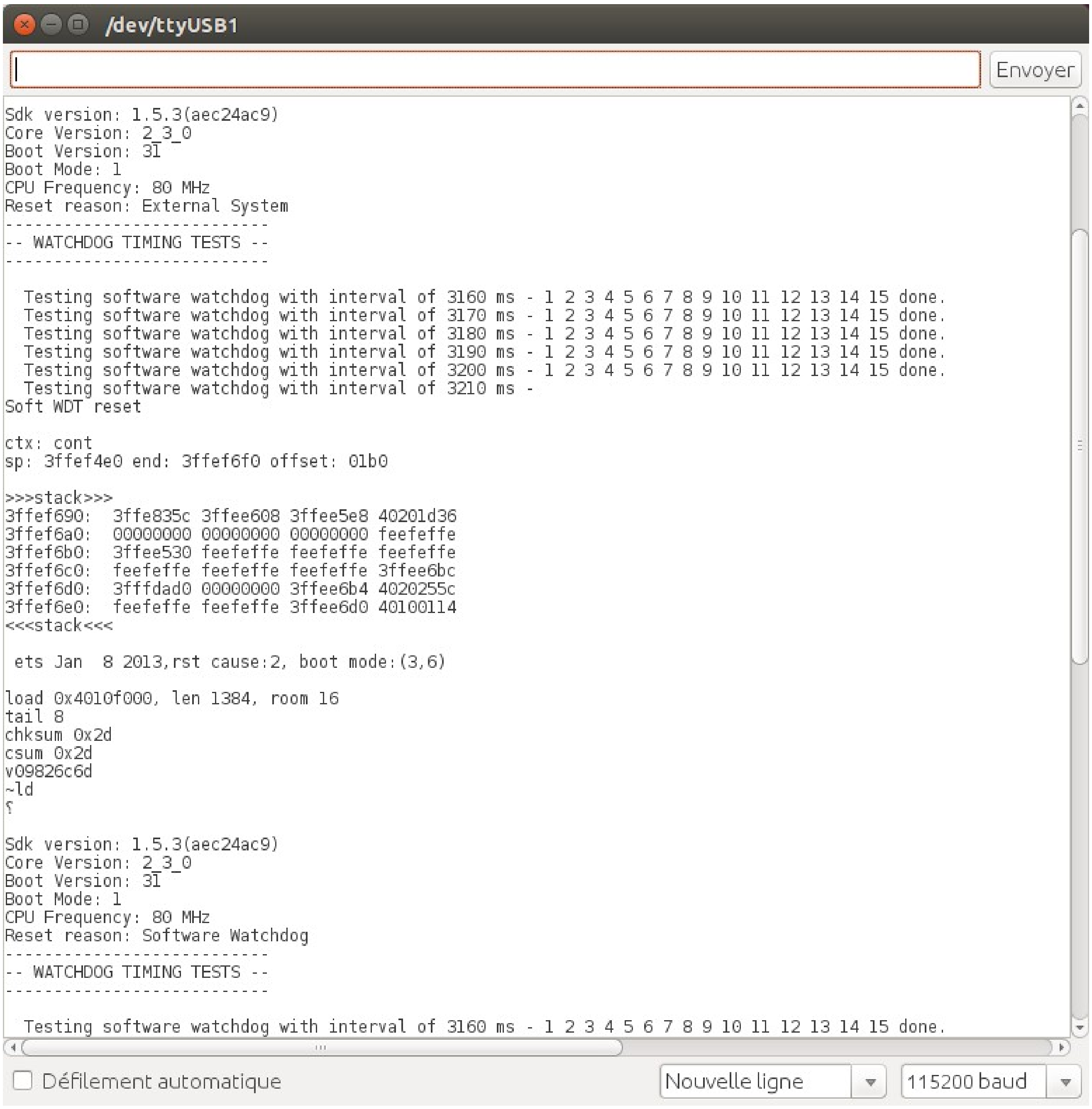
During that time the watchdog is not being fed. If the watchdog does not bite, increase the interval and try again until the watchdog does reset the ESP. Since the value of the **`interval`** variable is printed to the serial monitor before the beginning of the empty loop, it will be easy to figure out the length of the watchdog time-out period. There are two complications. First, each interval is tested a number of times because the ESP may perform background task which makes timing somewhat unpredictable. The number of iterations of the empty loop test is set with the LOOPS directive. The second complication is the care and feeding of the watchdogs at the end of each iteration, just before printing the iteration number to the serial monitor. It seems that the **`Serial`** library interacts with the watchdogs in unpredictable fashion. This is not too surprising as both transmission and reception of serial data is interrupt-driven (see ESP8266 Arduino Core documentation.)

Here is the Arduino serial monitor output of the sketch when run on a Wemos D1 mini to test the hardware watchdog:



```
Sdk version: 1.5.3(aec24ac9)
Core Version: 2_3_0
Boot Version: 31
Boot Mode: 1
CPU Frequency: 80 MHz
Reset reason: Hardware Watchdog
---------------------------
-- WATCHDOG TIMING TESTS --
---------------------------

  Testing hardware watchdog with interval of 8000 ms - 1 2 3 4 5 6 7 8 9 10 done.
  Testing hardware watchdog with interval of 8100 ms - 1 2 3 4 5 6 7 8 9 10 done.
  Testing hardware watchdog with interval of 8200 ms - 1 2 3 4 5 6 7 8 9 10 done.
  Testing hardware watchdog with interval of 8300 ms - 1 2 3 4 5 6 7 8 9 10 done.
  Testing hardware watchdog with interval of 8400 ms -
 ets Jan  8 2013,rst cause:4, boot mode:(3,6)

wdt reset
load 0x4010f000, len 1384, room 16
tail 8
chksum 0x2d
csum 0x2d
v09826c6d
~ld
ʕ

Sdk version: 1.5.3(aec24ac9)
Core Version: 2_3_0
Boot Version: 31
Boot Mode: 1
CPU Frequency: 80 MHz
Reset reason: Hardware Watchdog
---------------------------
-- WATCHDOG TIMING TESTS --
---------------------------

  Testing hardware watchdog with interval of 8000 ms - 1 2 3 4 5 6 7 8 9 10 done.
```

This is the output when the SOFT_WATCHDOG directive is defined:



```
Sdk version: 1.5.3(aec24ac9)
Core Version: 2_3_0
Boot Version: 31
Boot Mode: 1
CPU Frequency: 80 MHz
Reset reason: External System
---------------------------
-- WATCHDOG TIMING TESTS --
---------------------------

  Testing software watchdog with interval of 3160 ms - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 done.
  Testing software watchdog with interval of 3170 ms - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 done.
  Testing software watchdog with interval of 3180 ms - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 done.
  Testing software watchdog with interval of 3190 ms - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 done.
  Testing software watchdog with interval of 3200 ms - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 done.
  Testing software watchdog with interval of 3210 ms -
Soft WDT reset

ctx: cont
sp: 3ffef4e0 end: 3ffef6f0 offset: 01b0

>>>stack>>>
3ffef690:  3ffe835c 3ffee608 3ffee5e8 40201d36
3ffef6a0:  00000000 00000000 00000000 feefeffe
3ffef6b0:  3ffee530 feefeffe feefeffe feefeffe
3ffef6c0:  feefeffe feefeffe feefeffe 3ffee6bc
3ffef6d0:  3fffdad0 00000000 3ffee6b4 4020255c
3ffef6e0:  feefeffe feefeffe 3ffee6d0 40100114
<<<stack<<<

 ets Jan  8 2013,rst cause:2, boot mode:(3,6)

load 0x4010f000, len 1384, room 16
tail 8
chksum 0x2d
csum 0x2d
v09826c6d
~ld
ʕ

Sdk version: 1.5.3(aec24ac9)
Core Version: 2_3_0
Boot Version: 31
Boot Mode: 1
CPU Frequency: 80 MHz
Reset reason: Software Watchdog
---------------------------
-- WATCHDOG TIMING TESTS --
---------------------------

  Testing software watchdog with interval of 3160 ms - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 done.
```

The 8.2 seconds timeout for the hardware watchdog and 3.2 seconds for the software watchdog timeout, concord with the times reported by pasko_zh.

It turns out that before starting an empty loop to see if the watchdog will timeout, both pasko_zh and I feed and restart the software watchdog, while Dave Evans only feeds it. Note that pasko_zh uses the lower level C functions in **`user_interface.h`** so that he begins each test with

```
system_soft_wdt_feed();
system_soft_wdt_restart();
```

The same result can be obtained with the methods in **`ESPclass`**:

```
ESP.wdtFeed();
ESP.wdtEnable(0);
```

Indeed, I modified **`test1`** of the Evans sketch (available for download evans_soft.ino) to verify that not enabling the software watchdog is the reason it bites when not fed it for 1.6 to 1.8 seconds. These timing results could lead us to conclude that the initial watchdog timer value set when restarting the watchdog is greater than the value set by a watchdog feed routine. However, changing the order in which the watchdog is fed and restarted does not change the timing results.

I did not care to push my investigation further. The lesson seems to be that the software watchdog should not be starved for more than a second to avoid its bite.

References:

Evans, Dave (2016), *Watchdog Timer Tests*.
paško from Zurich (pasko_zh) (2016), *I'm getting these Watchdog Resets "wdt reset" - How can I avoid them?*.

toc

5. Care and Feeding of the ESP8266 Watchdogs

In his discussion of the ESP watchdogs, pasko_zh makes an important point: **`ESP.wdtFeed()`** (or **`system_soft_wdt_feed()`**) actually feeds both the hardware and the software watchdogs. So the two watchdogs are not cascaded. They are two independent watchdogs with different (and non modifiable) timeout periods that happen to be fed by the same low level function.

In most Arduino sketches, the feeding of the watchdogs is done implicitly. It is done at the beginning of each iteration of the program loop. It is done in the **`yield()`** and **`delay()`** functions (but not the **`delayMicroseconds()`** function). Certain blocking functions, such as those found in the WiFi module also feed the watchdogs. So it would appear that the 'Arduino way' of preventing software and hardware resets is to ensure that any loop in a sketch lasts less than say 1 second (1.5 second at most) or, if necessary, that it calls regularly on **`yield()`** or **`delay()`** as necessary.

Calls to **`yield()`** or **`delay()`** have a salutary side effect. They are integrated with the WiFi module so that a connection to a WiFi network is not lost as long as the software watchdog is fed. Unfortunately, I have noticed that things are not quite as smooth with regard to the MQTT module **`pubsub`**. So in those sketches that use that module, I call on **`local_yield()`** or **`local_delay()`** as defined below if needs be.

```
/*
   Need a local yield that that calls yield() and mqttClient.loop()
   The system yield() routine does not call mqttClient.loop()
*/

void local_yield()
{
  yield();
  mqttClient.loop();
}

/*
   Need a local delay calls yield() and mqttClient.loop()
   The system delay() routine does not call mqttClient.loop()
*/
void local_delay(unsigned long millisecs)
{
  unsigned long start = millis();
  local_yield();
  if (millisecs > 0)
  {
    while (elapsed_time(start) < millisecs)
    {
      local_yield();
    }
  }
}
```

Note that the following declarations are part of the sketch that contains the above functions

```
#include <ESP8266WiFi.h>
#include <PubSubClient.h>

WiFiClient    espClient;
PubSubClient  mqttClient(espClient);
```

toc

6. Recovery

That is a rather ambiguous if not ambitious title. As it happens, it is possible to obtain the reason why the ESP8266 has come to be started or restarted at the start of a sketch. By recovery, I just mean that it is therefore possible to take different actions based on the cause of the restart.

Quoting the Espressif document ESP8266 Reset Causes and Common Fatal Exception Causes, "Each time [the] ESP8266 reboots, the ROM code will print out a number corresponding to the reset cause." It also prints out something called the boot mode which indicates from where the code was taken. This was visible on the screen shots of the serial monitor window:

```
   ets Jan 8 2013, rst cause:2, boot mode: (1,6)
or
   ets Jan 8 2013, rst cause:4, boot mode: (3,6)
```

To the right of "rst cause", the 2 indicates a software watchdog reset and the 4 marks a hardware watchdog reset. As for the first digit of the boot mode, a 1 indicates that the code was just uploaded to the ESP, a 3 indicates that the code is from the SPI flash memory.

The explanation for the boot mode comes from a document by Max Filippov on the web called Boot Process. It states that the ESP boot mode is in the lower three bits of the first number of the boot mode. Perhaps calling this the "boot mode" is unfortunate. Others use "boot device". Actually, the values of the three bits correspond to three pins of the integrated circuit and determine the source of the code run by the ESP8266. As the following table taken from the document shows, there are three possibilities:

GPIO15	GPIO0	GPIO2	Dec.	Device	Description
0	0	1	1	UART	Code downloaded from UART (programming)
0	1	1	3	Flash	Boot from code in SPI Flash (normal)
1	x	x	4-7	SDIO	Boot from SD-card

Of course, to boot from code on an SD card, it would be necessary to add an external card reader to the ESP8266.

While this information is useful when programming an ESP8266 from the Arduino IDE, the information must be retrieved if the sketch is to make use of it. The class EspClass contains a method that returns the cause of the reset as a string. The method, **`ESP.getResetReason()`**, was used in the initial **`setup`** block of the watchdog timing sketch. There is another function

ESP.getResetInfoPtr() that returns a pointer to a structure with information about system reset including the reason for it, which is more useful than the string function.

As far as I can ascertain, there is no **EspClass** method to get the boot device. It can be retrieved from memory. A contributor to the ESP8266 Community Forum, who goes by the moniker 0ff, posted an assembler routine that does that.

The following sketch (available for download: esp_boot.ino) retrieves the boot information in the **setup** section and then goes on to restart.

```
/*
 * esp_boot.ino
 *
 */
#define BM_WDT_SOFTWARE 0
#define BM_WDT_HARDWARE 1
#define BM_ESP_RESTART 2
#define BM_ESP_RESET 3

#define BOOT_MODE BM_WDT_SOFTWARE

extern "C" {
#include "user_interface.h"
}

struct bootflags
{
  unsigned char raw_rst_cause : 4;
  unsigned char raw_bootdevice : 4;
  unsigned char raw_bootmode : 4;

  unsigned char rst_normal_boot : 1;
  unsigned char rst_reset_pin : 1;
  unsigned char rst_watchdog : 1;

  unsigned char bootdevice_ram : 1;
  unsigned char bootdevice_flash : 1;
};

struct bootflags bootmode_detect(void) {
  int reset_reason, bootmode;
  asm (
    "movi %0, 0x60000600\n\t"
    "movi %1, 0x60000200\n\t"
    "l32i %0, %0, 0x114\n\t"
    "l32i %1, %1, 0x118\n\t"
    : "+r" (reset_reason), "+r" (bootmode) /* Outputs */
    : /* Inputs (none) */
    : "memory" /* Clobbered */
  );

  struct bootflags flags;

  flags.raw_rst_cause = (reset_reason & 0xF);
  flags.raw_bootdevice = ((bootmode >> 0x10) & 0x7);
  flags.raw_bootmode = ((bootmode >> 0x1D) & 0x7);

  flags.rst_normal_boot = flags.raw_rst_cause == 0x1;
  flags.rst_reset_pin = flags.raw_rst_cause == 0x2;
  flags.rst_watchdog = flags.raw_rst_cause == 0x4;

  flags.bootdevice_ram = flags.raw_bootdevice == 0x1;
  flags.bootdevice_flash = flags.raw_bootdevice == 0x3;

  return flags;
}
```

```
void setup() {
  Serial.begin(115200);
  Serial.println("\n\nSketch Setup\n");

  rst_info* rinfo = ESP.getResetInfoPtr();

  Serial.printf("rinfo->reason:   %d, %s\n", rinfo->reason, ESP.getResetReason().c_str());
  Serial.printf("rinfo->exccause: %d\n", rinfo->exccause);
  Serial.printf("rinfo->epc1:     %d\n", rinfo->epc1);
  Serial.printf("rinfo->epc2:     %d\n", rinfo->epc2);
  Serial.printf("rinfo->epc3:     %d\n", rinfo->epc3);
  Serial.printf("rinfo->excvaddr: %d\n", rinfo->excvaddr);
  Serial.printf("rinfo->depc:     %d\n", rinfo->depc);

  struct bootflags bflags = bootmode_detect();

  Serial.printf("\nbootflags.raw_rst_cause: %d\n", bflags.raw_rst_cause);
  Serial.printf("bootflags.raw_bootdevice: %d\n", bflags.raw_bootdevice);
  Serial.printf("bootflags.raw_bootmode: %d\n", bflags.raw_bootmode);

  Serial.printf("bootflags.rst_normal_boot: %d\n", bflags.rst_normal_boot);
  Serial.printf("bootflags.rst_reset_pin: %d\n", bflags.rst_reset_pin);
  Serial.printf("bootflags.rst_watchdog: %d\n", bflags.rst_watchdog);

  Serial.printf("bootflags.bootdevice_ram: %d\n", bflags.bootdevice_ram);
  Serial.printf("bootflags.bootdevice_flash: %d\n", bflags.bootdevice_flash);

  Serial.printf("\n\nrinfo->reason=%d\n\n", ESP.getResetInfoPtr()->reason);

  if (bflags.raw_bootdevice == 1) {
    Serial.println("The sketch has just been uploaded over the serial link to the ESP8266");
    Serial.println("Beware: the device will freeze after it reboots in the following step.");
    Serial.println("It will be necessary to manually reset the device or to power cycle it");
    Serial.println("and thereafter the ESP8266 will continuously reboot.");
  }

#if BOOT_MODE == BM_ESP_RESTART
  Serial.println("\n\nRebooting with ESP.restart()");
  ESP.restart();
#elif BOOT_MODE == BM_ESP_RESET
  Serial.println("\n\nRebooting with ESP.reset()");
  ESP.reset();
#else
  #if BOOT_MODE == BM_WDT_HARDWARE
    Serial.println("\n\nRebooting with hardware watchdog timeout reset");
    ESP.wdtDisable();
  #else
    Serial.println("\n\nRebooting with sofware watchdog timeout reset");
  #endif;
  while (1) {}  // timeout!
#endif;
}

void loop() { }
```

The way the ESP8266 is rebooted is set by letting the preprocessor directive **`BOOT_MODE`** directive to one of four values

Value	Description
BM_WDT_SOFTWARE	Software watchdog timeout
BM_WDT_HARDWARE	Hardware watchdog timeout

BM_ESP_RESTART	Software restart with ESP.restart()
BM_ESP_RESET	Software restart with ESP.reset()

Running the sketch confirms that in all four ways of provoking a reboot of the ESP, it will hang on the first reboot after the sketch is downloaded. Consequently, the values in the following table show the result of running the sketch only after that first manual reset after downloading it to the ESP.

	`ESP.restart()`	`ESP.reset()`	`soft WDT`	`hard WDT`
rst cause	2	2	2	4
boot mode	(3,6)	(3,6)	(3,6)	(3,6)
rinfo->reason	4	4	3	1
rfinfo->exccause	0	0	4	4
raw_rst_cause	2	2	2	4
raw_bootdevice	3	3	3	3
raw_bootmode	6	6	6	6

The first two rows of results are the "ets" values sent to UART0 at the start of the reboot, before the sketch in flash memory is run. The next two rows show the partial content of the rst_info struct returned by the **`EspClass`** method **`ESP.getResetInfoPtr()`**. The last three rows are the values read from the ESP memory by Off assembly code. They correspond exactly to the "ets" start up message.

Clearly, **`rinfo->reason`** and **`bootflags.raw_bootdevice`** is most useful to decide on how to start a sketch. I will end this discussion of the standard ESP8266 watchdogs with an example sketch (download: esp_bootex.ino) showing how this could be done.

```
/*
 * esp_bootex.ino
 *
 */

extern "C" {
#include "user_interface.h"
}

int getBootDevice(void) {
  int bootmode;
  asm (
    "movi %0, 0x60000200\n\t"
    "l32i %0, %0, 0x118\n\t"
    : "+r" (bootmode) /* Output */
    : /* Inputs (none) */
    : "memory" /* Clobbered */
  );
  return ((bootmode >> 0x10) & 0x7);
}

void setup() {
  Serial.begin(115200);
  Serial.println();

  if ( getBootDevice() == 1 ) {
    Serial.println("\nThis sketch has just been uploaded over the UART.");
    Serial.println("The ESP8266 will freeze on the first restart.");
    Serial.println("Press the reset button or power cycle the ESP");
    Serial.println("and operation will be resumed thereafter.");
  }

  char buff[32];

  switch (ESP.getResetInfoPtr()->reason) {

    case REASON_DEFAULT_RST:
      // do something at normal startup by power on
      strcpy_P(buff, PSTR("Power on"));
      break;

    case REASON_WDT_RST:
      // do something at hardware watch dog reset
      strcpy_P(buff, PSTR("Hardware Watchdog"));
      break;

    case REASON_EXCEPTION_RST:
      // do something at exception reset
      strcpy_P(buff, PSTR("Exception"));
      break;

    case REASON_SOFT_WDT_RST:
      // do something at software watch dog reset
      strcpy_P(buff, PSTR("Software Watchdog"));
      break;

    case REASON_SOFT_RESTART:
      // do something at software restart ,system_restart
      strcpy_P(buff, PSTR("Software/System restart"));
      break;

    case REASON_DEEP_SLEEP_AWAKE:
      // do something at wake up from deep-sleep
      strcpy_P(buff, PSTR("Deep-Sleep Wake"));
      break;

    case REASON_EXT_SYS_RST:
      // do something at external system reset (assertion of reset pin)
      strcpy_P(buff, PSTR("External System"));
      break;

    default:
      // do something when reset occured for unknown reason
      strcpy_P(buff, PSTR("Unknown"));
      break;
  }
  Serial.printf("\n\nReason for reboot: %s\n", buff);
  Serial.println("----------------------------------------------");
}

char user_input() {
  while (1) {
    Serial.println();
    Serial.println("To select how to reset the ESP, press");
    Serial.println("  1 - for ESP.restart()");
    Serial.println("  2 - for ESP.reset()");
    Serial.println("  3 - for hardware watchdog timeout");
    Serial.println("  4 - for software watchdog timeout");
    Serial.println("or press the reset button.");
    Serial.println("In any case, press the reset button if");
    Serial.println("if this sketch has just been uploaded.");
    while (!Serial.available()) {
      delay(1);
    }
    char ch = Serial.read();
    if (ch >= '1' && ch <= '4') {
      return ch;
    } else {
      Serial.println("\nInvalid entry");
    }
  }
}

void loop() {
  switch (user_input()) {
    case '1':
      Serial.println("\n\nRebooting with ESP.restart()");
      ESP.restart();
      break;
    case '2':
      Serial.println("\n\nRebooting with ESP.reset()");
      ESP.reset();
      break;
    case '3':
      Serial.println("\n\nRebooting with hardware watchdog timeout reset");
      ESP.wdtDisable();
      while (1) {} // timeout
      break;
    case '4':
      Serial.println("\n\nRebooting with sofware watchdog timeout reset");
      while (1) {} // timeout
      break;
  }
}
```

As shown the sketch is kind of useless since all it does is replicate the **`EspClass`** method getResetReason(). Of course, the idea is that something more useful than preparing a string for a print statement of the cause of the reset should be done in the **`switch/case`** code.

Arduino Sketch Managed ESP8266 Watchdog