**A**

# Thesis Report on

IMPLEMENTATION OF WATCH DOG TIMER AND ISR IN ESP8266

**Submitted to**

**CHADALAWADA RAMANAMMA ENGINEERING COLLEGE**

***In partial fulfillment of the requirements for the award of the Degree of***

# BACHELOR OF TECHNOLOGY IN ELECTRONICS & COMMUNICATION ENGINEERING

**By**

**V.LAVANYA 18P11A04A6**

**Under the Esteemed Guidance of**

**Dr. K. SOUNDARARAJAN M. Tech, Ph.D. Professor**



**DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING**

**CHADALAWADA RAMANAMMA ENGINEERING COLLEGE**

**(AUTONOMOUS)**

(Accredited by NAAC with ‘A’ Grade, Approved by AICTE, New Delhi, Affiliated to JNTU Anantapur) Renigunta road, Tirupati-517506, Andhra Pradesh, India

**2017-2021**

**CHADALAWADA RAMANAMMA ENGINEERING COLLEGE**

**(AUTONOMOUS)**

# Department of Electronics & Communication Engineering



# *Certificate*

This is to certify that the project work entitled **“**IMPLEMENTATION OF WATCH DOG TIMER AND ISR IN ESP8266**”** is a bonafide work done by **V. LAVANYA (18P11A04A6)U.GOUTAMI(18P11A0418)Y.VENKATASURESH (18P11A04B4 )**

**R. DEVENDRA KUMAR (18P11A0490)** in the Department of **“ELECTRONICS & COMMUNICATION ENGINEERING”**, and submitted to ***Chadalawada Ramanamma Engineering College (Autonomous), Tirupati*** is a project work carried out by them under my guidance during the academic year 2020-2021.

### GUIDE HEAD

**DR. K. SOUNDARARAJAN M. Tech, Ph.D. Dr. P. KRISHNA MURTHY M. Tech, Ph.D.**

Professor Associate Professor & Head

Department of ECE Department of ECE

**INTERNAL EXAMINER EXTERNAL EXAMINER**

## DECLARATION

**We are hereby declare that the project work on** “ IMPLEMENTATION OF WATCH DOG TIMER AND ISR IN ESP8266 done by us under the guidance of

### Dr . K. SOUNDARARAJAN, M. Tech, Ph. D in CHADALAWADA RAMANAMMA ENGINEERING COLLEGE (Autonomous) is submitted in partial fulfillment of the requirements of the requirements of the award of Degree of Bachelor of Technology.

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that, we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea / data / fact / source in our project report submission.

Signature of the Students

**V.LAVANYA 18P11A04A6**

**U. GOUTAMI 18P11A0418**

**Y. VENKATA SURESH 18P11A04B4**

**R. DEVENDRA KUMAR 18P11A0490**

## ACKNOWLEDGEMENT

A grateful thanks to **Dr. CHADALAWADA KRISHNAMURTHY**, Chairman, C.V.S. Krishna Murthy Teja Charities for providing education in this esteemed Institution.

We express our gratitude to our principal **Dr. P. SANJEEVA RAYUDU**, **M. Tech, Ph.D.**

Principal, Chadalawada Ramanamma Engineering College, Tirupati, for providing all the facilities and supporting us to complete this project work.

Our sincere thanks **to Dr. P. KRISHNA MURTHY**, **M. Tech, Ph.D.** Associate Professor and Head of the Department of Electronics & Communication Engineering, Chadalawada Ramanamma Engineering College, Tirupati, for providing infrastructural facilities to complete our project work.

We are greatly indebted to our guide, **Dr. k. SOUNDARARAJAN**, **M. Tech, Ph.D.**  Professor of the Department of Electronics & Communication Engineering, Chadalawada Ramanamma Engineering College, Tirupati, who encourages us in all spheres of activities, constant encouragement and keen interest enriched to complete this work.

We extend our thanks to all the Teaching staff of the Department of E.C.E. for their support, encouragement and inspired us. We also thank the Non-Teaching staff of the Department of E.C.E. for being helpful in many ways in successful completion of our work.

The chain of our gratitude would be definitely incomplete, if we forget to thank all my friends of Chadalawada Ramanamma Engineering College, Tirupati for their constant support and encouragement. Finally, we thank all those who helped us directly or indirectly in successful completion of our project work.

**V.LAVANYA 18P11A04A6**

**U. GOUTAMI 18P11A0418**

**Y. VENKATA SURESH 18P11A04B4**

**R. DEVENDRA KUMAR 18P11A0490**

## ABSTRACT

This work presents a brief explanation about the watch dog timer implementation in ESP8266 microcontroller using interrupt service routine. Usually watch dog is used in microcontrollers, embedded systems etc. we are using ESP8266 microcontroller. A microcontroller (MCU) is a compact processor for controlling electronic devices. Integrated into a wide variety of electronic devices, MCUs come pre-loaded with program software whose commands are used to control electronic devices.  
This makes safeguarding normal MCU operation essential. Should the MCU program, for some reason, run out of control or stop running altogether, the electronic device may behave erratically, which in the worst case could cause damage or an accident.

## TABLE OF CONTENTS

**TITLE** **Page No**

**LIST OF FIGURES** i

**LIST OF TABLES** iv

### ABBREVIATIONS v

**NOTATIONS** vii

### CHAPTER 1: INTRODUCTION TO WATCH DOG TIMERS

1.1 History of watch dog timers

1.2 1

1.3 2

1.4 3

1.5

4

1.6

9

### CHAPTER 2 : LITERATURE REVIEW

2.1 Introduction 11

2.2 Literature review 12

### CHAPTER 3: AN INTRODUCTION TO HFSS

3.1 Introduction 18

3.2 18

3. 19

3.4

24

26

29

### CHAPTER 4:

4.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

**CHAPTER 1**

## INTRODUCTION To WATCH DOG TIMERS

### 1.1 HISTORY OF WATCH DOG TIMERS

**1.2 INTRODUCTION**

A watchdog timer is an electronic timer that is used to detect and recover from computer malfunctions. During normal operation, the computer regularly restarts the watchdog timer to prevent it from elapsing, or "timing out". If, due to a hardware fault or program error, the computer fails to restart the watchdog, the timer will elapse and generate a timeout signal. The timeout signal is used to initiate corrective action or actions. The corrective actions typically include placing the computer system in a safe state and restoring normal system operation. Watchdog timers are commonly found in embedded systems and other computer-controlled equipment where humans cannot easily access the equipment or would be unable to react to faults in a timely manner. In such systems, the computer cannot depend on a human to reboot it if it hangs; it must be self-reliant. For example, remote embedded systems such as space probes are not physically accessible to human operators; these could become permanently disabled if they were unable to autonomously recover from faults. A watchdog timer is usually employed in cases like these. Watchdog timers may also be used when running untrusted code in a sandbox, to limit the CPU time available to the code and thus prevent some types of denial-of-service attacks

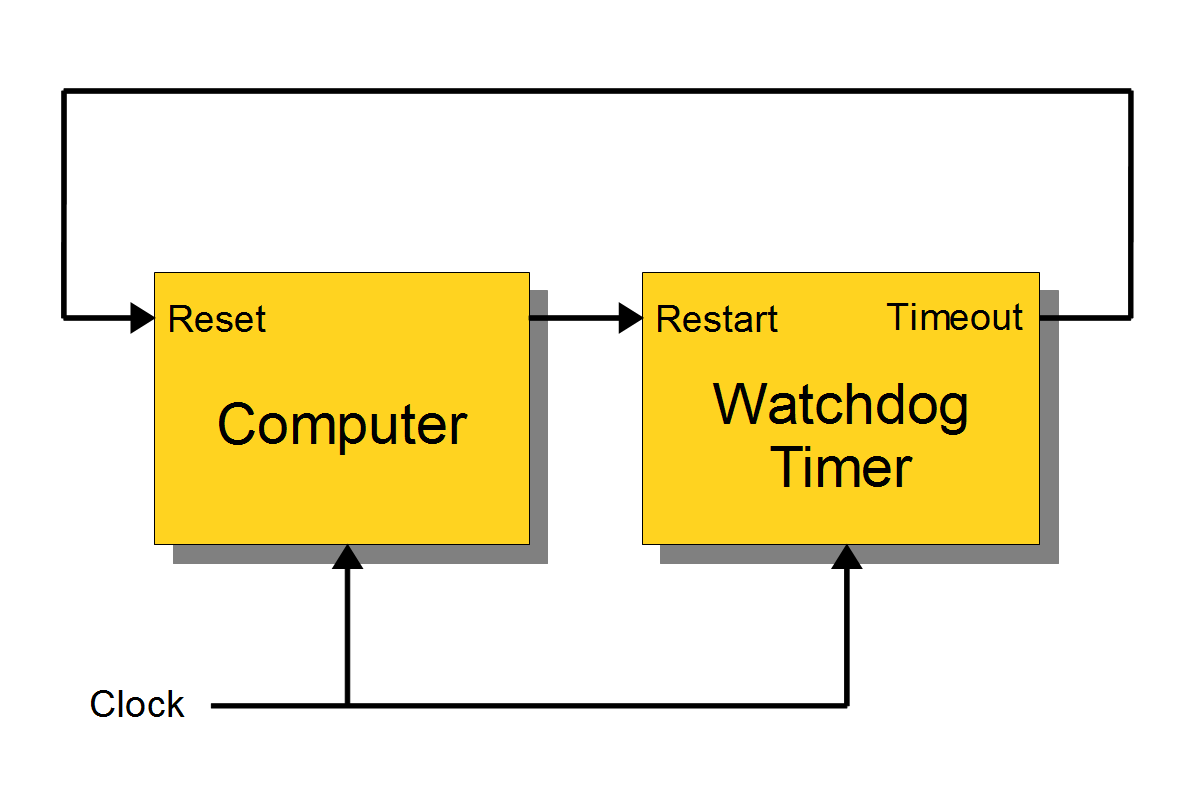
**1.3 ARCHITECTURE AND OPERATION**

**WATCH DOG TIMER RESTART**

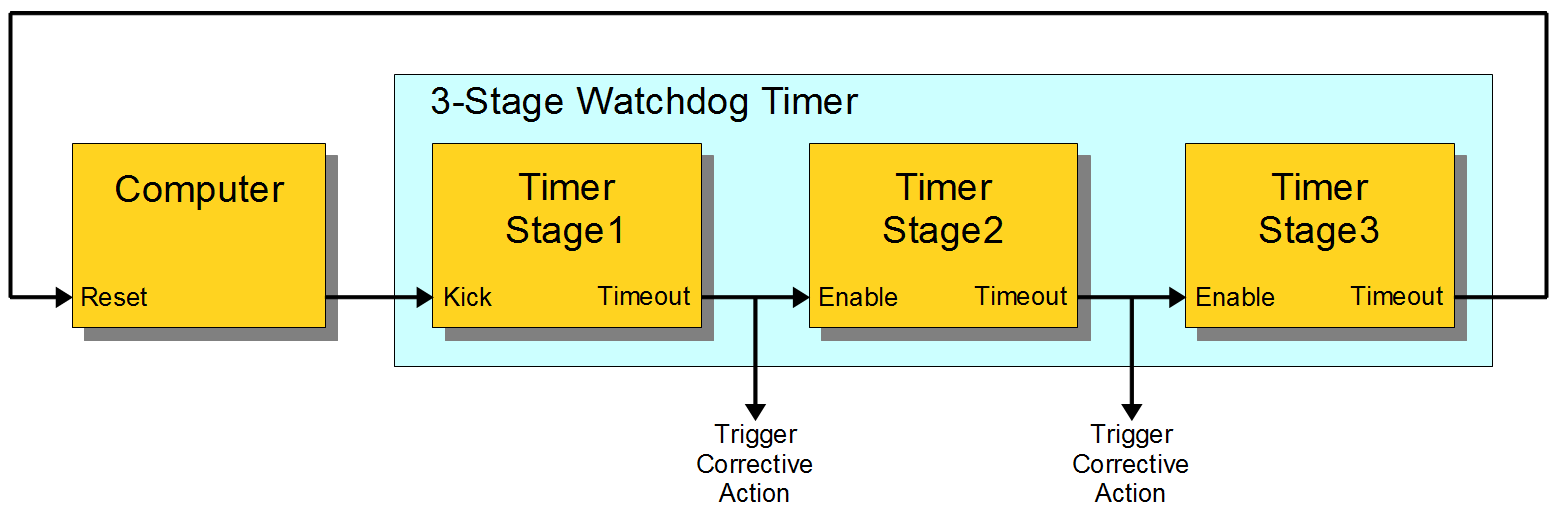
The act of restarting a watchdog timer is commonly referred to as "kicking the dog" or other similar terms; this is typically done by writing to a watchdog control port. Alternatively, in microcontrollers that have an integrated watchdog timer, the watchdog is sometimes kicked by executing a special machine language instruction. An example of this is the CLRWDT (clear watchdog timer) instruction found in the instruction set of some PIC microcontrollers. In computers that are running operating systems, watchdog resets are usually invoked through a device driver. For example, in the Linux operating system, a user space program will kick the watchdog by interacting with the watchdog device driver, typically by writing a zero character to /dev/watchdog. The device driver, which serves to abstract the watchdog hardware from user space programs, is also used to configure the time-out period and start and stop the timer.

**1.4 SINGLE STAGE WATCH DOG**

Watch dog timers come in many configurations, and many allow their configurations to be altered. Microcontrollers often include an integrated, on-chip watchdog. In other computers the watchdog may reside in a nearby chip that connects directly to the CPU, or it may be located on an external expansion card in the computer's chassis. The watchdog and CPU may share a common clock signal, as shown in the block diagram below, or they may have independent clock signals signal.

 1.5 MULTI STAGE WATCHDOG

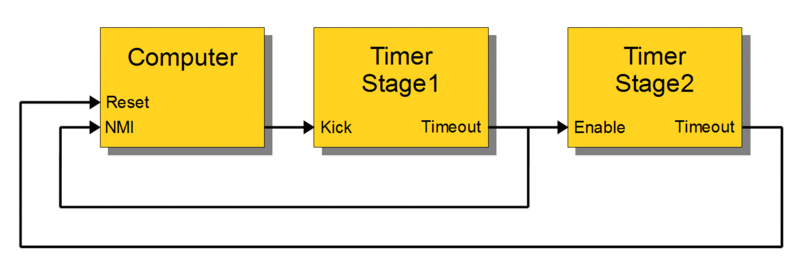
Two or more timers are sometimes cascaded to form a multistage watchdog timer, where each timer is referred to as a timer stage, or simply a stage. For example, the block diagram below shows a three stage watchdog. In a multistage watchdog, only the first stage is kicked by the processor. Upon first stage timeout, a corrective action is initiated and the next stage in the cascade is started. As each subsequent stage times out, it triggers a corrective action and starts the next stage. Upon final stage timeout, a corrective action is initiated, but no other stage is started because the end of the cascade has been reached. Typically, single-stage watchdog timers are used to simply restart the computer, whereas multistage watchdog timers will sequentially trigger a series of corrective actions, with the final stage triggering a computer restart.



1.6 TIME INTERVALS

Watchdog timers may have either fixed or programmable time intervals. Some watchdog timers allow the time interval to be programmed by selecting from among a few selectable, discrete values. In others, the interval can be programmed to arbitrary values. Typically, watchdog time intervals range from ten milliseconds to a minute or more. In a multistage watchdog, each timer it may have it’s unique time interval.

1.7 CORRECTIVE ACTIONS

A watchdog timer may initiate any of several types of corrective action, including processor reset, non maskable interrupt, maskable interrupt, power cycling, fail-safe state activation, or combinations of these. Depending on its architecture, the type of corrective action or actions that a watchdog can trigger may be fixed or programmable. Some computers require a pulsed signal to invoke a processor reset. In such cases, the watchdog typically triggers a processor reset by activating an internal or external pulse generator, which in turn creates the required reset pulses. In embedded systems and control systems, watchdog timers are often used to activate fail-safe circuitry. When activated, the fail-safe circuitry forces all control outputs to safe states (e.g., turns off motors, heaters, and high-voltages) to prevent injuries and equipment damage while the fault persists. In a twostage watchdog, the first timer is often used to activate fail-safe outputs and start the second timer stage; the second stage will reset the computer if the fault cannot be corrected before the timer elapses. Watchdog timers are sometimes used to trigger the recording of system state information—which may be useful during fault recovery—or debug information (which may be useful for determining the cause of the fault) onto a persistent medium. In such cases, a second timer—which is started when the first timer elapses—is typically used to reset the computer later, after allowing sufficient time for data recording to complete. This allows time for the information to be saved, but ensures that the computer will be reset even if the recording process fails. They have its own, unique time

For example, the above diagram shows a likely configuration for a two-stage watchdog timer. During normal operation the computer regularly kicks Stage1 to prevent a timeout. If the computer fails to kick Stage1 (e.g., due to a hardware fault or programming error), Stage1 will eventually timeout. This event will start the Stage2 timer and, simultaneously, notify the computer (by means of a non-maskable interrupt) that a reset is imminent. Until Stage2 times out, the computer may attempt to record state information, debug information, or both. The computer will be reset upon Stage2 timeout

CHAPTER 2

LITERATURE REVIEW

2.1 INTRODUCTION

A watchdog timer (WDT) is a timer that monitors microcontroller (MCU) programs to see if they are out of control or have stopped operating. It acts as a “watchdog” watching over MCU operation.

A microcontroller (MCU) is a compact processor for controlling electronic devices. Integrated into a wide variety of electronic devices, MCUs come pre-loaded with program software whose commands are used to control electronic devices.  
This makes safeguarding normal MCU operation essential. Should the MCU program, for some reason, run out of control or stop running altogether, the electronic device may behave erratically, which in the worst case could cause damage or an accident.

To proactively prevent such incidents, it falls to the role of the watchdog timer toconstantly watch over the MCU to ensure it is operating normally.

Watch dog timers are commonly found in embedded systems, microcontrollers and other computer equipment where humans cannot access the equipment or would be unable to react to faults in a timely manner. In such systems, the computer cannot depend on a human to reboot it if it hangs; it must be self reliant. A watch dog timer is usually employed in cases like these. It may be also used when running untrusted

During normal operation, the computer regularly restarts the watchdog timer to prevent it from elapsing, or "timing out". If, due to a hardware fault or program error, the computer fails to restart the watchdog, the timer will elapse and generate a timeout signal. The timeout signal is used to initiate corrective actions. The corrective actions typically include placing the computer and associated hardware in a safe state and invoking a computer reboot.

Every watchdog timer, however simple or sophisticated, must initiate two corrective actions. First, it must set the computer's control outputs to safe levels so that potentially dangerous devices such as motors and heaters will not pose threats to people or equipment. This is a high priority action that must occur as soon as a fault is detected. After setting the outputs to safe levels, the next order of business is to restore normal system operation. This can be as simple as restarting the computer, as if a human operator has pressed the computer's reset pushbutton, or it may involve a sequence of actions that ultimately ends with a computer restart.

A watchdog is a timer that, when not reset before expiring, triggers the reset of the system [1] that is monitoring. In our case, the system will be the ESP8266 microcontroller.

So, the main program needs to periodically reset the watchdog timer, to prevent the reset of the CPU and keep working normally.

The watchdog should be configured with a time greater that the worst case scenario delay in the program [1], so it only triggers in error / unpredicted problems that may make the main program to be locked and not recover on its own [2]. In those locking cases, the watchdog is not reset and, when it expires, resets the system.

This concept is of extreme importance, specially in microcontrollers, which may be affected by environmental conditions such as electrical noise, which can cause hardware malfunction that locks the execution. Additionally, it is  useful for problem in the code that may put the execution in an undesired infinite loop. The ESP8266 has two watch dogs one implemented in software and other implemented in hardware.

2.2THE SOFTWARE WATCH DOG FUNCTIONS

In order to access the watchdog functions on the ESP8266, we have the **Esp Class**, which can be analysed in more detail [here](https://github.com/esp8266/Arduino/blob/4897e0006b5b0123a2fa31f67b14a3fff65ce561/cores/esp8266/Esp.cpp#L82). We can access the functionality of this class by using an extern variable called ESP, which is declared [here](https://github.com/esp8266/Arduino/blob/4897e0006b5b0123a2fa31f67b14a3fff65ce561/cores/esp8266/Esp.h) in the libraries.

In order to disable the software watchdog, we just need to call the **wdt Disable** method on the ESP object, as indicated bellow.

|  |  |
| --- | --- |
| 1 | ESP.wdtDisable(); |

Although this disables the software watchdog, the hardware watchdog will still remain active, causing a reset after some time. As indicated in the  of the **wdtDisable** method, if we stop the software watchdog by more that 6 seconds, the hardware watchdog will trigger. This is approximately what I got when testing it.

We can re-enable the software watchdog by calling the **wdtEnable** method, as indicated bellow.

|  |  |
| --- | --- |
| 1 | ESP.wdtEnable(1000); |

It’s important to note that an integer value needs to be passed as input argument of this method, which would correspond to the number of milliseconds for the watchdog to trigger. Nevertheless, this value makes no effect and it is not used in the internal call of the SDK ESP8266 functions, as can be seen by the source code [here](https://github.com/esp8266/Arduino/blob/4897e0006b5b0123a2fa31f67b14a3fff65ce561/cores/esp8266/Esp.cpp#L84).

Additionally, in the [header file](https://github.com/esp8266/Arduino/blob/4897e0006b5b0123a2fa31f67b14a3fff65ce561/cores/esp8266/Esp.h#L89), it’s clearly stated that, at the time, setting the timeout is not implemented. Unfortunately, it’s not clear what is the default value of the watchdog timer when we call this function, and neither it is documented in the [SDK](http://www.espressif.com/sites/default/files/2c-esp8266_non_os_sdk_api_guide_en_v1.5.4.pdf) of the ESP8266.

In order to explicitly restart the watchdog, we can call the **watchdog Feed** method, specified [here](https://github.com/esp8266/Arduino/blob/4897e0006b5b0123a2fa31f67b14a3fff65ce561/cores/esp8266/Esp.cpp#L102). Fortunately, the ESP libraries implicitly reset the watchdog in many of the functions, so most of the time we don’t need to worry about feeding the watchdog. Nevertheless, it’s important to know that it exists, in order to troubleshoot spontaneous reboots of our programs.

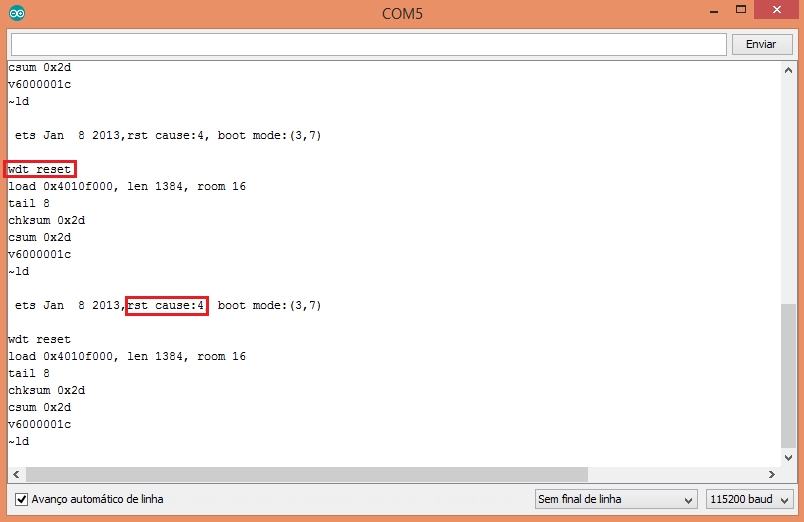
|  |  |
| --- | --- |
| 1 | ESP.wdtFeed(); |

**2.3Triggering software watchdog**

To trigger the hardware watchdog, we just need to disable the software watchdog timer and do an infinite loop. We can do all of these actions in the setup function, as indicated bellow. Note that we will do an infinite loop with a **while** loop.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | void setup() {     ESP.wdtDisable();   while (1){};    }    void loop(){} |

After uploading the code, open the serial port to check the output. When the watchdog fires, a crash log is printed to the serial port, as indicated in the figure 1.



**Figure 1** – Crash log of the hardware watchdog reset.

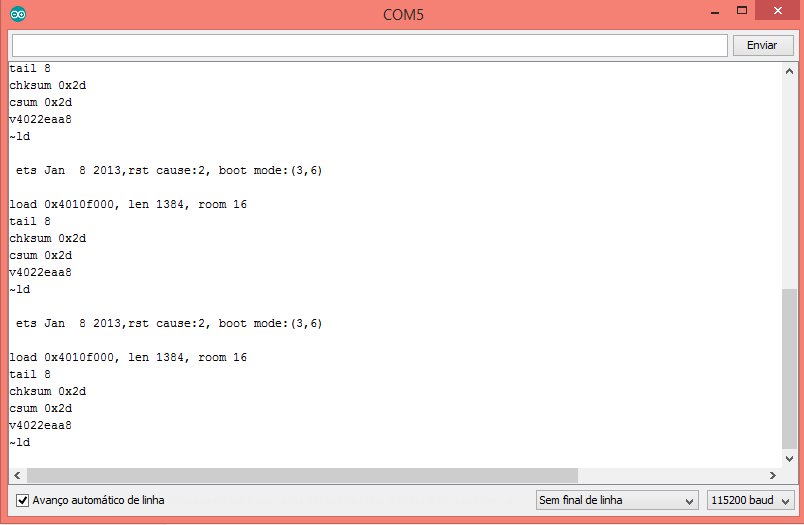
As indicated in the figure, there is a mention to the cause (**wdt reset**) and the reset cause has the number 4, which corresponds to the hardware watchdog reset [4]. You can check other reset code [here](http://www.esp8266.com/viewtopic.php?p=2096#p2112). This result is consistent with the indicated in [this](https://github.com/esp8266/Arduino/blob/4897e0006b5b0123a2fa31f67b14a3fff65ce561/doc/faq/a02-my-esp-crashes.md) document.

2.4TRIGGERING THE WATCH DOG

In order to trigger the software watchdog reset, we just use the same code as before and remove the disabling of the software watchdog (enabled by default).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | void setup() {     while (1){};    }    void loop(){} |

In this case, after uploading the code and opening the serial port, we will just get the result illustrated in figure 2, when the software watchdog triggers.



**Figure 2** – Crash log of the software watchdog reset.

The result is also coherent with the one shown [here](https://github.com/esp8266/Arduino/blob/4897e0006b5b0123a2fa31f67b14a3fff65ce561/doc/faq/a02-my-esp-crashes.md).

## 

2.5FEEDING THE WATCH DOG

Finally, we will use the code indicated bellow to test the function of restarting the watchdog. So, we will call the**wdtFeed** method inside our infinite loop, which will ensure the reset of the watchdog.

It works both for the software and hardware watchdog, so if we uncomment the call to the **wdtDisable**method, the program will still keep running without being reset by the hardware watchdog.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | void setup() {     //ESP.wdtDisable();   while (1){ESP.wdtFeed();};    }    void loop(){} |

## 

**CHAPTER 3**

**INTRODUCTION TO ESP8266**

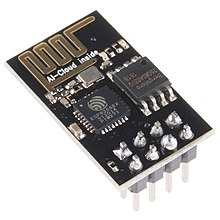
**3.1**

The **ESP8266** is a low-cost [Wi-Fi](https://en.wikipedia.org/wiki/Wi-Fi) microchip, with a full [TCP/IP stack](https://en.wikipedia.org/wiki/TCP/IP_stack) and [microcontroller](https://en.wikipedia.org/wiki/Microcontroller) capability, produced by [Espressif Systems](https://en.wikipedia.org/w/index.php?title=Espressif_Systems&action=edit&redlink=1)[[1]](https://en.wikipedia.org/wiki/ESP8266#cite_note-Espressif_ESP8266-1) in Shanghai, China.

The chip first came to the attention of Western [makers](https://en.wikipedia.org/wiki/Maker_culture) in August 2014 with the **ESP-01** module, made by a third-party manufacturer Ai-Thinker. This small module allows microcontrollers to connect to a Wi-Fi network and make simple TCP/IP connections using [Hayes](https://en.wikipedia.org/wiki/Hayes_command_set)-style commands. However, at first, there was almost no English-language documentation on the chip and the commands it accepted.[[2]](https://en.wikipedia.org/wiki/ESP8266#cite_note-2) The very low price and the fact that there were very few external components on the module, which suggested that it could eventually be very inexpensive in volume, attracted many hackers to explore the module, the chip, and the software on it, as well as to translate the Chinese documentation.[[3]](https://en.wikipedia.org/wiki/ESP8266#cite_note-3)

The **ESP8285** is an ESP8266 with 1 MiB of built-in flash, allowing the building of single-chip devices capable of connecting to Wi-Fi.[[4]](https://en.wikipedia.org/wiki/ESP8266#cite_note-esp8285-4)

These microcontroller chips have been succeeded by the [ESP32](https://en.wikipedia.org/wiki/ESP32) family of devices, including the pin-compatible ESP32-C3.



THE NODEMCU OR ESP8266

Is a complex device, which combines some features of the ordinary Arduino board with the possibility of connecting to the internet. Arduino Modules and Microcontrollers have always been a great choice to incorporate automation into the relevant project.

But these modules come with a little drawback as they don’t feature a built-in Wi Fi capability, subsequently, we need to add external Wi Fi protocol into these devices to make them compatible with the internet channel. This is the famous Node MCU which is based on ESP8266 Wi Fi So C. This is version 3 and it is based on ESP-12E (An ESP8266 based Wi Fi module).

Node MCU is also an open-source firmware and development kit that helps you to prototype your IOT product within a few LUA script lines, and of course you can always program it with Arduino IDE. In this article,We will try present useful details related to this Wi Fi Development Kit, its main features, pinout and everything we need to know about this module and the application domain.

Introduction Node MCU V3 Node MCU V3 is an open-source firmware and development kit that plays a vital role in designing an IoT product using a few script lines. Multiple GPIO pins on the board allow us to connect the board with other peripherals and are capable of generating PWM, I2C, SPI, and UART serial communications.

The interface of the module is mainly divided into two parts including both Firmware and Hardware where former runs on the ESP8266 Wi-Fi SoC and later is based on the ESP-12 module. The firmware is based on Lua – A scripting language that is easy to learn, giving a simple programming environment layered with a fast scripting language that connects you with a well-known developer community.

And open source firmware gives you the flexibility to edit, modify and rebuilt the existing module and keep changing the entire interface until you succeed in optimizing the module as per your requirements.

• USB to UART converter is added on the module that helps in converting USB data to UART data which mainly understands the language of serial communication. Instead of the regular USB port, MicroUSB port is included in the module that connects it with the computer for dual purposes: programming and powering up the board.

• The board incorporates status LED that blinks and turns off immediately, giving you the current status of the module if it is running properly when connected with the computer. The ability of module to establish a flawless WiFi connection between two channels makes it an ideal choice for incorporating it with other embedded devices like Raspberry Pi.

3.2 FEATURES OF ESP8266

* Processor: L106 32-bit [RISC](https://en.wikipedia.org/wiki/Reduced_instruction_set_computing) microprocessor core based on the [Tensilica](https://en.wikipedia.org/wiki/Tensilica) Xtensa Diamond Standard 106Micro running at 80 MHz[[5]](https://en.wikipedia.org/wiki/ESP8266#cite_note-5)
* Memory:[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]
  + 32 KiB instruction RAM
  + 32 KiB instruction cache RAM
  + 80 KiB user-data RAM
  + 16 KiB ETS system-data RAM
* External QSPI flash: up to 16 MiB is supported (512 KiB to 4 MiB typically included)
* [IEEE 802.11](https://en.wikipedia.org/wiki/IEEE_802.11) b/g/n [Wi-Fi](https://en.wikipedia.org/wiki/Wi-Fi)
  + Integrated [TR switch](https://en.wikipedia.org/wiki/Duplexer#Transmit-receive_switch), [balun](https://en.wikipedia.org/wiki/Balun), [LNA](https://en.wikipedia.org/wiki/Low-noise_amplifier), [power amplifier](https://en.wikipedia.org/wiki/RF_power_amplifier) and [matching network](https://en.wikipedia.org/wiki/Matching_network)
  + [WEP](https://en.wikipedia.org/wiki/Wired_Equivalent_Privacy) or [WPA/WPA2](https://en.wikipedia.org/wiki/Wi-Fi_Protected_Access) authentication, or open networks
* 17 [GPIO](https://en.wikipedia.org/wiki/General-purpose_input/output) pins[[6]](https://en.wikipedia.org/wiki/ESP8266#cite_note-6)
* [SPI](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus)
* [I²C](https://en.wikipedia.org/wiki/I%C2%B2C) (software implementation)[[7]](https://en.wikipedia.org/wiki/ESP8266#cite_note-EspressifBBS_I2C-7)
* [I²S](https://en.wikipedia.org/wiki/I%C2%B2S) interfaces with DMA (sharing pins with GPIO)
* [UART](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter) on dedicated pins, plus a transmit-only UART can be enabled on GPIO2
* 10-bit [ADC](https://en.wikipedia.org/wiki/Analog-to-digital_converter) ([successive approximation ADC](https://en.wikipedia.org/wiki/Successive_approximation_ADC))

**3.3APPLICATIONS OF ESP8266**

a)Implementation of Esp8266-01 in Home automation:

Some trends that we foresee for this phase of the industry are Philips , Siemens & scheider will eventually bring out fairly mass market automation products with appealing user interface but at a lower price point today, and more people will be able to afford the products. Some foreign players will have niche in high and automation.

b) Wearable device:

The ESP8266 board used was Node MCU 1.0 (ESP-12E Module) with CPU Frequency:

80MHz and Flash Size: 4M (3M SPIFFS). the development of prototype that enables monitoring of heart rate

and inter beat interval for several subjects. The prototype was realized using ESP8266 hardware modules,

WebSocket library, nodes and JavaScript. System architecture is described where nodes server acts as the

signal processing and GUI code provider for clients. Signal processing algorithm was implemented in

JavaScript. Application GUI is presented which can be used on mobile devices. Several important parts of the

code are described which illustrate the communication between ESP8266 modules, server and clients.

Developed prototype shows one of the possible realizations of group monitoring of biomedical data[9] .

Health is the most important part of any human’s life without health it is useless to any treasure of life. Most humans

live a busy life in which going to a doctor for weekly or even monthly checkup is an impossible task. Without

monitoring your health it is not possible to whether you are a healthy or sick person. This problem leads to the design

of a product which monitors your health every day without going to a doctor. In this paper, a system is designed as a

prototype for monitoring alerting based on the health of a person. This system is fully automated little or no human

help is needed. Any doctor can monitor this person from anywhere through the internet[10] .

**CCCCHACH**

CHAPTER 4

INTRODUCTION TO ISR

4.1 INTERRUPT SERVICE ROUTINE

Stands for "Interrupt Service Routine." An ISR (also called an interrupt handler) is a [software](https://techterms.com/definition/software) process invoked by an interrupt request from a [hardware](https://techterms.com/definition/hardware) device. It handles the request and sends it to the [CPU](https://techterms.com/definition/cpu), interrupting the active [process](https://techterms.com/definition/process). When the ISR is complete, the process is resumed.

A basic example of an ISR is a routine that handles [keyboard](https://techterms.com/definition/keyboard) events, such as pressing or releasing a key. Each time a key is pressed, the the ISR processes the [input](https://techterms.com/definition/input). For example, if you press and hold the right arrow key in a text file, the ISR will signal to the CPU that the right arrow key is depressed.

The CPU sends this information to the active [word processor](https://techterms.com/definition/wordprocessor) or text editing program, which will move the [cursor](https://techterms.com/definition/cursor) to the right. When you let go of the key, the ISR handles the "key up" event. This interrupts the previous "key down" state, which signals to the program to stop moving the cursor.

Similar to Newton's law of inertia (an object in motion tends to stay in motion), computer processes continue to run unless interrupted. Without an interrupt request, a computer will remain in its currentstate. Each input signal causes an interrupt, forcing the CPU to process the corresponding event.

Many types of hardware devices, including internal [components](https://techterms.com/definition/component) and external [peripherals](https://techterms.com/definition/peripheral) can sent interrupts to the CPU. Examples include keyboards, [mice](https://techterms.com/definition/mouse), [sound cards](https://techterms.com/definition/soundcard), and [hard drives](https://techterms.com/definition/harddrive). A [device driver](https://techterms.com/definition/driver) enables communication between each of these devices and the CPU. ISRs prioritize interrupt requests based on the [IRQ](https://techterms.com/definition/irq) setting of the device (or [port](https://techterms.com/definition/port)). Typically the keyboard is at the top of the IRQ list, while devices like hard drives are further down.

**so technically,**Interrupts is a mechanism by which an I/O or instruction can suspend the normal execution of the processor and gets itself serviced like it has higher priority**. For example, a processor doing a normal execution can be interrupted by some sensor to execute a particular process that is present in ISR (Interrupt Service Routine). After executing the ISR processor can again resume the normal execution.**

### 4.2Types of Interrupts

There are two types of interrupts:

**Hardware Interrupt:**It happens when an external event occurs like an external interrupt pin changes its state from LOW to HIGH or HIGH to LOW.

**Software Interrupt:**It happens according to the instruction from the software. For example **Timer interrupts are software interrupt**.

**Some Conditions while using Interrupt**

* Interrupt Service Routine function (ISR) must be as short as possible.
* Delay () function doesn’t work inside ISR and should be avoided.

EXAMPLES OF INTERRUPTS

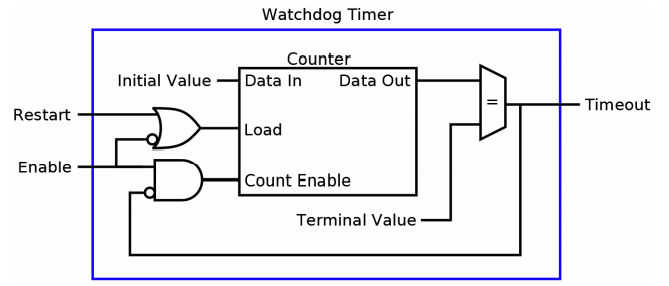
Consider a fast-moving car, if it suddenly gets hit by another car in opposite direction, the first thing that happens is that, the accelerometer sensor present in the car senses a sudden de-acceleration and triggers an external interrupt to the microcontroller present in the car. Then based on that interrupt the microcontroller produces an electric signal to deploy the airbags immediately. Microcontrollers present in the car monitor many things simultaneously like sensing the speed of the car, checking other sensors, controlling air conditioner temperature etc. So what makes a sudden opening of an airbag in seconds? The answer is **interrupts**, an**interrupt signal** is used here which has the highest priority of all.

Another simple **example** of Interrupts is touch screen mobile phones which have the highest priority to the “Touch” sense. Almost every electronic device has some kind to interrupts to ‘interrupt’ the regular process and do some higher priority things on a particular event. The regular process is resumed after serving the Interrupt.

**CHAPTER 5**

**5.1PROPOSED METHOD OF WATCH DOG TIMER**

As its name implies, a watchdog timer performs a timing function. The timing function is performed by a circuit that produces a delayed output signal in response to an input trigger signal. The circuit may be implemented as an analog delay circuit, which typically employs a monostable multivibrator, or as a digital delay circuit, which uses a digital counter to control the delay time. This application note focuses on digital watchdog timers, though many of the principles discussed here also apply to analog delay circuits. In general, a watchdog timer (or just “watchdog”) consists of a digital counter that counts from an initial value to a terminal value at a rate determined by a fixed-frequency clock. Typically the counter counts down from the initial value to zero, and the initial value is programmable so the program can configure how long it takes to count to zero. The watchdog will “timeout” when the counter reaches zero, causing it to assert its timeout signal and halt counting; this is known as a watchdog “event.” The timeout signal is connected to external circuitry so that it can initiate corrective action.

****

A program can restart the timer at any time by loading the initial value into the counter; this is commonly called “kicking” the watchdog. The watchdog is kicked by momentarily asserting its restart input, usually by writing to a watchdog “kick” port. During normal operation, the application program regularly kicks the timer to keep it from reaching zero, typically as part of a control loop.

If a fault condition prevents the program from kicking the timer, the watchdog will timeout and initiate corrective action. 1 Figure 1: A basic digital watchdog timer In many watchdog designs, the timer will be restarted regardless of the value written to the kick port, so that any arbitrary value will cause a timer restart. In other designs, however, a specific value must be written to the kick port; the timer will not be restarted unless the correct value is written to the port.

This is useful in various situations. For example, in a multi-threaded program it may be necessary to detect faults in more than one thread. In such cases, each thread can be designed to contribute a part of the kick port value, so that the correct value will be formed only if all of the threads are operating normally. Some watchdogs can be enabled and disabled by software, making it possible for a program to enable the watchdog only when its services are needed.

Other watchdogs are automatically enabled upon system boot and cannot be disabled at all; these are typically used to detect and recover from boot faults. In some cases, a computer may employ both types of watchdogs. In theory, there is no upper limit to the number of watchdogs used in a computer. Software-enabled watchdogs are typically disabled upon system reset.

When a control program starts executing, it enables its watchdog before it begins to control the output signals. Once the watchdog is enabled, the program must regularly kick the watchdog to prevent timeouts. When the application is preparing to terminate, it first sets all outputs to safe states and ceases output control and then it disables the watchdog; the application can terminate after disabling the watchdog.

5.2 **ESP Watchdog Timers**  
All ESP’s are ‘fitted’ with 2 watchdog timers (WDT), a software WDT and a hardware WDT. If these ‘dogs’ are not regularly ‘fed’ then the ESP is ‘bitten’ and it resets. This feeding process goes automatically and you usually don’t have to do anything with it. As far as I know there are at least 3 functions during which the dog is fed: loop(), delay() and yield().

If however during a loop it takes too long to start the next loop (i’ll get to the times in a bit) then the dog bites and the ESP resets. The simplest way to test this is the following sketch:

**void** **setup**() {

**while** (1) {};

}

**void** **loop**() {}

This will initiate an endless loop and after a while the WDT kicks in and the ESP is reset.

The same will happen if you e.g. do a readout of 10 sensors, taking one second per sensor, without taking care of the dog.

**5.3Feeding the dogs**  
So how to take care of the dog? Two ways;

1. The proper way: feed it regularly by inserting this in strategic places:

ESP.wdtFeed();

This will reset the WDT timer and keep the dog happy.

1. The ‘if you know what you’re doing way’: lock it up with

ESP.wdtDisable();

Note that you can take it out again with

ESP.wdtEnable(0);

The ‘0’ is an arbitrary number that is required but not used). HOWEVER there’s still the hardware dog and you cannot disable that one!

**5.4Timers**  
So what times are we talking about? For an ESP8266 the:  
Software WDT = 3.2 seconds (cannot be changed)  
Hardware WDT = 8.2 seconds (cannot be changed)  
Blynk Heartbeat = 10/15 seconds (can be changed). The device sends a heartbeat every 10 seconds, the server requires a ping every 15 seconds.

Here in this project we are using watch dog timer along with interrupt service routine to trouble shoot the errors in the ESP8266 Microcontroller.