Impulszähler mit MicroPhython auf dem Raspberry Pi Pico

Bernd Laquai, 18. Juni 2025

Die Halbleitertechnologie schreitet unweigerlich in Richtung kleinerer Strukturen, höherer Taktraten und größerer Leistungsfähigkeit voran. Die Folge sind immer leistungsfähigere Prozessoren unter anderem auch Mikrocontroller. Darüber freut sich auch die Maker Szene, die mittlerweile zu einem gewaltigen Business herangewachsen ist. Was mit dem Arduino anfing (2005) und vom Raspberry Pi (ab 2012) in höhere Leistungsklassen voranschritt, ist heute eine schier unüberschaubare Vielzahl an Mikrocontroller-Boards geworden mit der sich Hardware-Peripherie extrem flexibel kontrollieren und sich Signale umfangreich verarbeiten lassen.

Während bisher die Programmiersprache C bzw. C++ als kompilierte Sprache die Programmierung von Mikrocontrollern dominierte, ist nun mit den neueren deutlich leistungsfähigeren Controllern ein neuer Trend hin zu der interpretierten Sprache MicroPython zu erkennen. Dies ist sicher auch die Folge davon, dass im Bereich der weniger hardware-nahen allgemeinen Software-Entwicklung die Sprache Python bereits äußerst beliebt geworden ist und sowohl den kompilierten Sprachen wie auch den bisher schon umfangreich eingesetzten interpretierten Sprachen wie z.B. Java den Rang abzulaufen scheint. Jedenfalls gibt es mittlerweile eine unglaublich schnell wachsende Fan-Gemeinde, die sich über viele Anwendungs-Disziplinen hinweg erstreckt und dafür sorgt, dass es für die allgemeine Programmiersprache Python eine breite Unterstützung durch Literatur, Internet-Tutorials und Anwendungsforen usw. gibt. Und diese Entwicklung scheint sich nun auch für MicroPython als Programmiersprache speziell für die Mikrocontroller in gleicher Weise fortzusetzen. Das bedeutet, wenn man am Puls der Zeit bleiben will, kommt man kaum umhin sich auch mit MicroPython und ihrer Derivate zur Programmierung von Mikrocontrollern zu befassen.

Da jedoch MicroPython eine gewisse Einschränkung im Sprachumfang gegenüber der allgemeinen Programmiersprache Python darstellt und auch die Hardware-Nähe von MicroPython bei speziellen Anwendungen Probleme gegenüber einer Implementierung in C bzw. C++ bedeuten könnte, tut man gut daran, die gewünschte konkrete Funktionalität, die häufig bei einer Ziel-Anwendung vorkommt in einer elementaren Form auszutesten, bevor man umsteigt. Dafür ist die asynchrone Impulszählung mit einem Hardware-Interrupt in der Strahlungs-Messtechnik ein gutes Beispiel. Sowohl ein Geigermüller-Zählrohr wie ein Szintillationsdetektor liefert Zählimpulse als Ausgangssignal. Die Anzahl an Zählimpulsen pro Zeiteinheit, oder die Zeitdifferenz bei vorgegebener Zählimpulsanzahl ergibt die Zählrate, welche zu einer Strahlungs-Dosisleistung proportional ist und die gemessen werden muss. Die Strahlungs-Dosisleistung ist schließlich die Messgröße, die man in der Regel mit einem Strahlungsmessgerät bestimmen möchte. Nun könnte es aber sein, dass MicroPython zum Beispiel, weil es eine interpretierte Sprache ist und nicht wirklich bis auf Maschinensprache des Controllers kompiliert wird, nur niedrige maximale Zählraten erlaubt, was einen Einsatz als Impulszähler in Anwendungen der Strahlungsmesstechnik unbrauchbar machen würde.

Dieser Frage soll hier exemplarisch anhand des derzeit wohl bekanntesten MicroPython tauglichen Mikrocontroller-Boards, dem Pi Pico von Raspberry Pi, untersucht werden. Der Pi Pico (Version 1) basiert auf dem Mikrocontroller RP2040, der von Raspberry Pi selbst entwickelt wurde. Der hier vorgestellte Test kann allerdings auch genauso auf andere MicroPython-fähige Mikrocontroller übertragen werden.

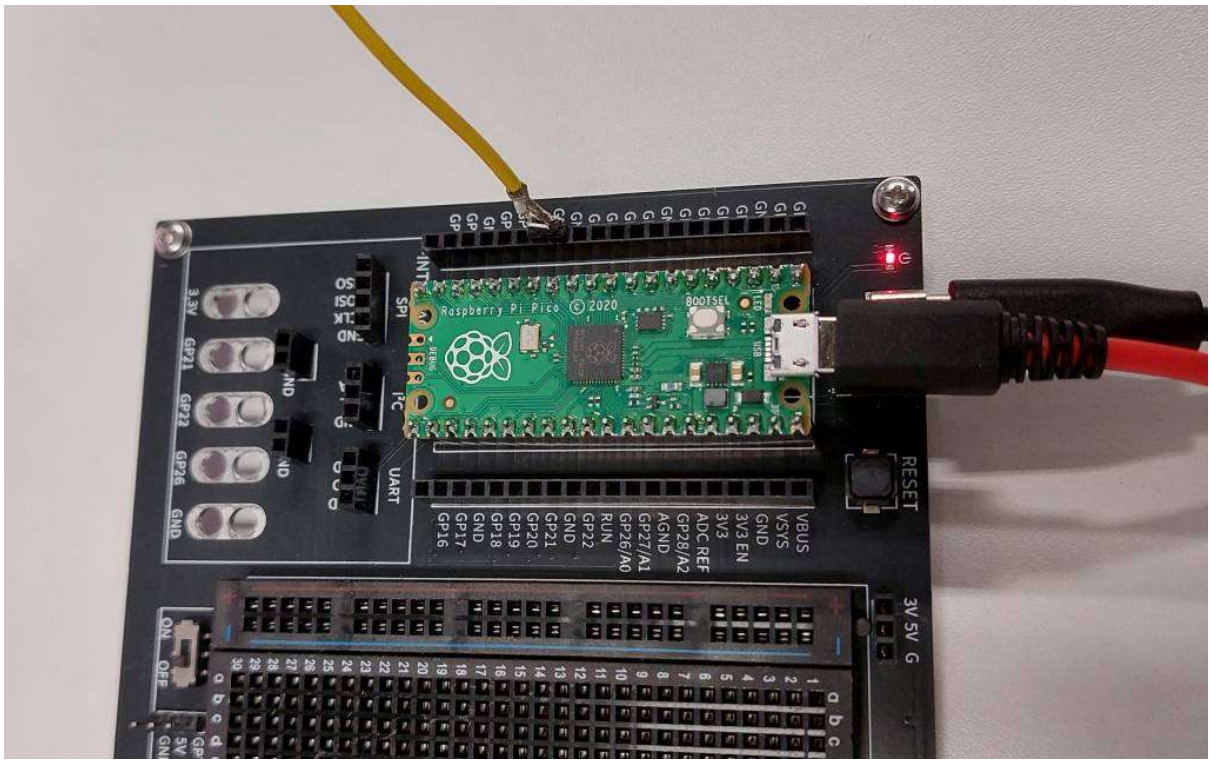

Abb. 1: Raspberry Pi Pico auf einem Pico Explorer Board von Joy-IT

Der Pi Pico Mikrocontroller wurde zur Evaluierung auf ein RB-P-XPLR Pico Explorer Board von Joy-IT aufgesteckt und mit einem Keysight 33600A Waveform Generator verbunden, der so eingestellt wurde, dass er Zählpulse mit 3V und 10us Impulsdauer mit variabler Frequenz erzeugt. Das Zählimpulssignal wurde über den GP10-Pin und den darüberliegenden GND-Pin mit einer 50Ohm Terminierung an den Pins eingespeist um eine gute Signalqualität zu erreichen.

Der Pi Pico wurde mit der MicroPython v1.25.0 Firmware versehen (RPI_PICO-20250415-v1.25.0.uf2) und mit dem Thonny Editor (Version 4.1.7 https://thonny.org/) programmiert.

Das zufällige Eintreffen der Zählimpulse erfordert im Microcontroller eine interrupt-gesteuerte Verarbeitung der Zählimpulse. Der Umgang mit Interrupts unter MicroPython ist in dem Artikel “Raspberry Pi Pico with Interrupts: External and Timer Interrupts (MicroPython)“ von Random Nerd Tutorials sehr gut beschrieben:

https://randomnerdtutorials.com/raspberry-pi-pico-interrupts-micropython/

Diese Anleitung wurde benutzt um einen Minimal-Code zu erzeugen, der lediglich die Impulszählung und optional eine Dosisratenberechnung ausführt.

Der eigentlich zeitkritische Teil in der Signalverarbeitung ist lediglich die Interrupt-Handler Routine countPulse(pin). Die einzige Aufgabe, die diese Routine ausführen muss, ist die Variable counter hochzuzählen. Diese Variable muss als global vereinbart werden und zwar in der Interrupt Handler-Routine selbst, nicht im Hauptprogram. Interessant ist, dass MicroPython bei der Erklärung des Interrupt-Handlers countPulse einen Parameter verlangt, der hier „pin“

genannt wurde (Listing 1). Er wird jedoch nicht weiter verwendet. Während der Entwickung des Programmcodes kann man sich optional noch den Zählerstand mit einer print()-Anweisung ausgeben lassen. Allerdings muss man sich im Klaren sein, dass jede Anweisung, die man in den Interrupt-Handler schreibt eine Totzeit bedeuten kann, da die Anzahl an neu eintreffenden Interrupts, die sich der Mikrocontroller während er etwas anderes tut, per Hardware in einer Interrupt Request Queue merken kann, äußerst begrenzt ist.

```
from machine import Pin
import time
convOfst = 0  #in cps
convSlope = 1 #in (uSv/h)/cps
maxcnt = 1000
intrIn = 10 #interrupt input
counter = 0

# Initialize interrupt
intrOne = Pin(intrIn, Pin.IN, Pin.PULL_DOWN)

# Define interrupt handler functions
def countPulse(pin):
    global counter
    counter += 1
    #print(counter)

# Attach interrupt handler to interrupt input
intrOne.irq(trigger=Pin.IRQ_RISING, handler=countPulse)
oldTime = time.ticks_ms()
while True:
    if (counter >= maxcnt):
        newTime = time.ticks_ms()
        dt = time.ticks_diff(newTime, oldTime) #measure time difference
        rate = float(maxcnt)*1000.0/float(dt) #in cps
        dose = (rate-convOfst)/convSlope; #in uSv/h
        oldTime = newTime;
        counter = 0;
        print(rate)
```

Listing 1: Minimal Code in MicroPython für die Berechnung der Zählrate

Das Hauptprogramm ist so aufgebaut, dass die Anzahl Zählimpulse, auf die gewartet wird, vorgegeben wird, und dazu die Zeitdifferenz gemessen wird, die dafür gebraucht wurde, um die Zählrate zu bestimmen (Impulsvorwahl). Das hat den großen Vorteil, dass dann die Streuung der Zählrate und der daraus bestimmten Dosisleistung konstant bleibt. Der Nachteil ist, dass bei sehr niedriger Impulsrate lange gewartet werden muss, bis ein Messwert entsteht. Man könnte natürlich auch eine gewisse Messzeit abwarten und die Anzahl der in dieser Zeit eintreffenden Zählimpulse zählen um die Zählrate zu bestimmen (Zeitvorwahl). Das hat dann wiederum den Nachteil, dass bei niederer Impulsrate, also schwacher Strahlungsintensität, die Impulsrate aus nur sehr wenigen Zählimpulsen berechnet wird und damit so stark streuen kann, dass ein einzelner Messwert sehr weit vom wahren Wert abweichen kann. Wenn man auso auf eine gewisse Qualität des Messergebnisses schaut, ist die Impulsvorwahl die bessere Strategie.

Die Anzahl der Impulse auf die hier gewartet wird, wird daher in Listing 1 mit maxcnt = 1000 vereinbart. Damit ergibt sich die Streuung des Messwerts zu Wurzel(1000) also etwa 30 Zählimpulse. Damit erreicht man eine statistische Messunsicherheit von grob 3%. Würde man maxcnt mit 100 vereinbaren, würde man eine statistische Messunsicherheit von nur 10% erreichen. Eine optionale Umrechnung der Zählrate wurde hier mit den Variablen convSlope und

convOfst nur vorbereitet. convSlope enthält später den Kalibrierwert für die Empfindlichkeit in (uSv/h)/cps und convOfst einen Kalibrierwert für die Nullrate in der Einheit cps, was einer Frequenz in der SI-Einheit Hertz entspricht.

Als Zählimpulseingang wurde hier der GPIO-Pin 10 verwendet, was sich darin wiederspiegelt dass die Variablen intrIn = 10 gesetzt wurde, welche der Hardware mitteilt, dass an diesem Pin auf externe Interrupts gewartet werden soll.

Die eigentliche Anweisung zur Vereinbarung der Eigenschaften des Interrupt Pins ist aber diese Zeile:

intrOne = Pin(intrIn, Pin.IN, Pin.PULL_DOWN)

Die Methode Pin() wird von dem Bibliotheks-Modul machine in der allerersten Zeile des Programcodes importiert. An sie wird der Parameter IntrIn übergeben, zusammen mit der Anforderung, dass dieser Pin als Signaleingang arbeiten soll und in der Hardware mit einem kleinen Pull-Down Strom versehen werden soll. Das bedeutet, dass der Zählimpuls-Ausgang des Detektors diesen Strom aufbringen können muss, wenn er auf High geht. Das hilft Störungen zu unterdrücken. Schließlich muss natürlich auch noch die Zählervariable counter mit Null intialisiert werden.

Die eigentliche Aktion, die ablaufen soll, wenn ein Zählimpuls eintreffen soll wird hier vereinbart:

intrOne.irq(trigger=Pin.IRQ_RISING, handler=countPulse)

Das bedeutet,dass auf eine steigende Flanke am Interrupt-Pin gewartet wird, und wenn diese eintrifft, wird die Interrupt Request Routine (der Interrupt-Handler) mit dem Namen countPulse ausgeführt. Man sieht hier auch, dass kein Parameter an die Handler-Routine übergeben wird, was sprachlich gesehen etwas merkwürdig anmutet, wenn man auf die Vereinbarung der Handler-Routine selbst schaut, wo der Parameter pin in den Klammern steht.

In der while-Schleife in der die Code-Ausführung im Hauptprogramm nach den Initialisierungen endlos läuft, werden zwei Variable für die Messung der Zeitdifferenz verwendet, oldTime und newTime. oldTime stellt den Startwert dar und newTime den Endwert. Zu Beginn eines Schleifen-Durchlaufs wird vorausgesetzt, dass oldTime einen Wert besitzt. Beim Eintritt in die Endlos-Schleife wird dies durch die Anweisung oldTime = time.ticks_ms() erricht. Wenn dann die erwartete Anzahl an Zählimpulsen erreicht ist, wird die Variable newTime mit Hilfe der Methode time.ticks_ms() aus dem Bibliotheksmodul time auf die aktuelle Zeit gesetzt und mit der Methode time.ticks_diff(newTime, oldTime) die Zeitdifferenz berechnet. Die Zeitdifferenz wird in die Variable dt gespeichert. Die spezielle Methode time.ticks_diff() wird nun verwendet um auch im Falle des Überlaufs des internen Hardware-Zeitzählers immer noch die richtige Zeitdifferenz zu erhalten und keinen negativen Wert. Die Berechnung der Zählrate wird dann explizit in Float-Zahlen gemacht. Danach übernimmt die Variable oldTime den Wert von newTime und die Zählvariable für die Zählimpulse wird zurückgesetzt.

Betrachtet man nun die print(rate) Ausgabe der Implementierung des Impulszählers nach Listing 1, so kann man erkennen, dass die gemessene Impulsrate bis über 5000cps sehr exakt ist. Danach tauchen im Bereich 10kcps bis 20kcps nur marginale Abweichungen auf, die aber tendentiell etwas größer werden. Ab etwa 30kcps kann man dann deutlich erkennen, dass das Messergebnis keinen Sinn mehr macht, weil der Mikrocontroller mit dem Zählen der Zählimpulse nicht mehr nachkommt. Die Messwerte werden dann deutlich zu klein.

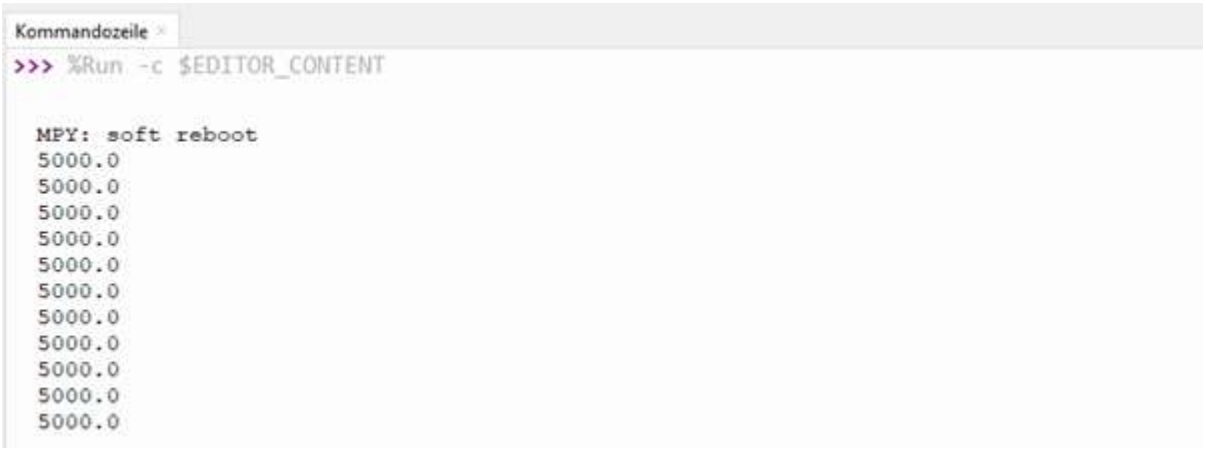

Abb. 2a: Ausgabe für Listing 1 (maxcnt = 1000) bei einer Zählrate von 5000 cps

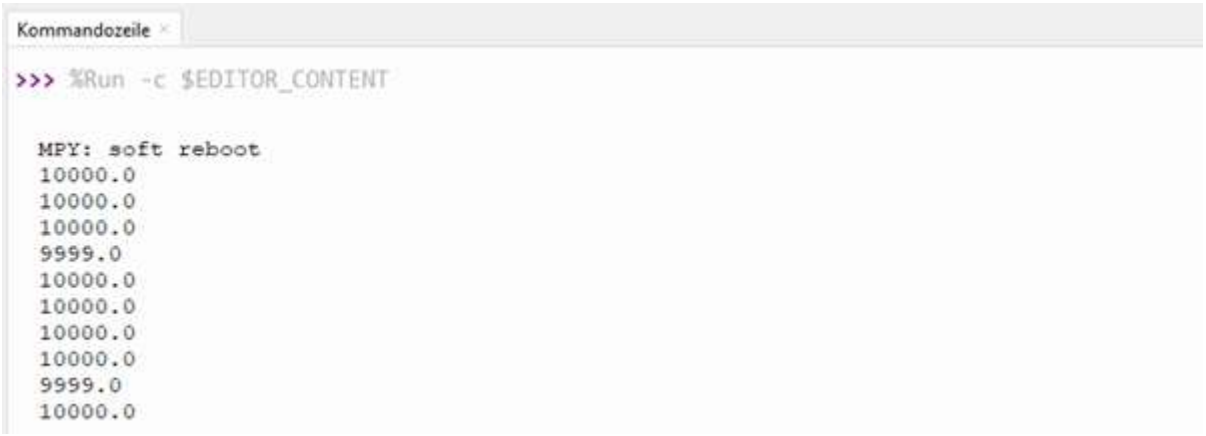

Abb. 2b: Ausgabe für Listing 1 (maxcnt = 100000) bei einer Zählrate von 10000 cps

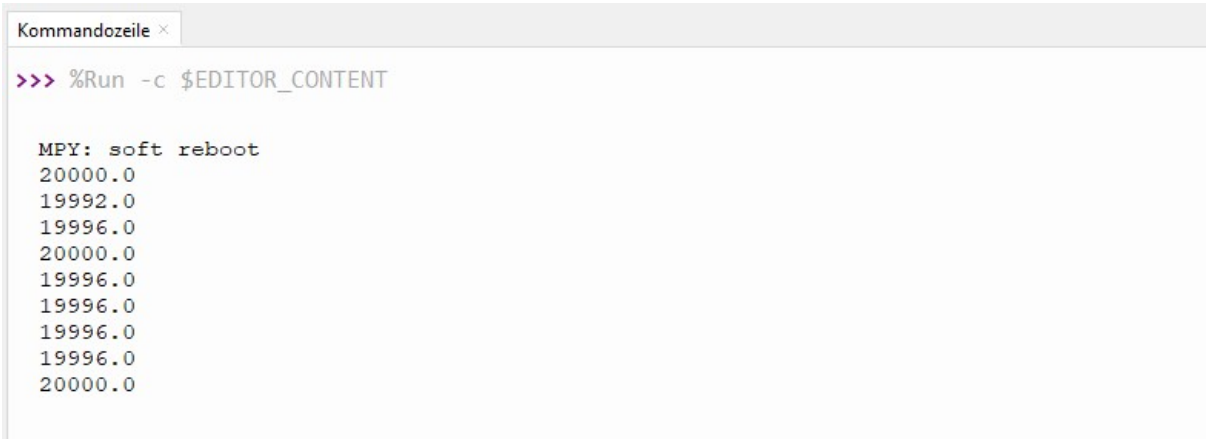

Abb. 2c: Ausgabe für Listing 1 (maxcnt = 100000) bei einer Zählrate von 20000 cps

Man kann das nun bis etwas über 30000cps so weitertreiben, und man erhält immer noch hinreichend genaue Ergebnisse. Bei 40000cps erhält man jedoch schließlich Ergebnisse, die so stark abweichen, dass sie keinen Sinn mehr ergeben.

Man kann nun noch die Frage stellen, was passiert eigentlich, wenn während der Berechnung der Zählrate (und Dosisleistung) und vor allem während der Ausgabe des Ergebnisses auf dem

Display weiterhin Zählimpulse eintreffen. Diese Zählimpulse müssten eigentlich unberücksichtigt bleiben, bis die Anweisung oldTime = newTime ausgeführt wird, was die Messung der Zeitdifferenz wieder neu initialisiert. Streng genommen müsste man daher nach Erreichen der erwarteten Anzahl an Zählimpulsen den Interrupt-Mechanismus für die Zeit der Auswertung deaktivieren. Dies ist in Listing 2 gezeigt.

```
#see: PyInterruptsRandomNerd.docx
# link https://randomnerdtutorials.com/raspberry-pi-pico-interrupts-micropython/

from machine import Pin
import time
convOfst = 0  #in cps
convSlope = 1 #in (uSv/h)/cps
maxcnt = 1000
intrIn = 10 #interrupt input
counter = 0

# Initialize interrupt
intrOne = Pin(intrIn, Pin.IN, Pin.PULL_DOWN)

# Define interrupt handler functions
def countPulse(pin):
    global counter
    counter += 1
    #print(counter)

# Attach interrupt handler to interrupt input
intrOne.irq(trigger=Pin.IRQ_RISING, handler=countPulse)
oldTime = time.ticks_ms()
while True:
    if (counter >= maxcnt):
        intrOne.irq(handler=None) #detach interrupt
        newTime = time.ticks_ms()
        dt = time.ticks_diff(newTime, oldTime) #measure time difference
        rate = float(maxcnt)*1000.0/float(dt) #in cps
        dose = (rate-convOfst)/convSlope; #in uSv/h
        oldTime = newTime;
        counter = 0;
        print(rate)
        intrOne.irq(trigger=Pin.IRQ_RISING, handler=countPulse)
```

Listing 2: Aussetzen des Interrupts

Bei dem in Listing 2 gezeigten Programm-Code wird unmittelbar nach dem Erreichen das erwarteten Zählerstands (counter >= mxacnt) mit der Anweisung intrOne.irq(handler=None) die dem Interrupt-Pin zugeordnete Interrupt Handler-Routine temporär „ins Leere“ gesetzt. Dies entspricht einer detach-interrupt() Routine in anderen hardware-nahen Sprachimplementierungen (z.B. dem C++ von Arduino). Nach der Berechnung und der Anzeige des Messergebnisses wird dann ganz am Ende der While-Schleife die Interrupt-Handler Routine wieder neu aufgesetzt. Damit wird sichergestellt, dass sich die Inerrupt-Request-Queue der Hardware in der Zeit der Verarbeitung und Ausgabe nicht füllt und so verhindert wird, dass die Zählervariable nach dem Zurücksetzen mit den gespeicherten Interrupts hochgezählt wird.

Allerdings zeigte der Vergleich der Programmausführung zwischen Listing 1 und Listing 2 in der Praxis keinen merklichen Unterschied. Möglicherweise liegt es eben daran, dass die Hardware nicht allzu viele Interrupts speichern kann, ohne dass die Queue überläuft, so dass das erst spürbar werden würde, wenn die Zahl der Zählimpulse, auf die gewartet wird, sehr klein ist. In jedem Fall aber ist Listing 2 die sicherere Programm-Code Variante, die auch unter extremen Bedinungen genaue Ergebnisse liefern müsste.

Da im Citizen Science und Hobby-Bereich für Anwendungen in der Strahlungsmesstechnik eine Detektoreinheit selten Zählraten über 10000 cps liefert, kann man also zusammenfassend sagen, dass man mit MicroPython trotz der Tatsache, dass nur auf einen Bytecode compiliert wird, welcher erst bei der Ausführung auf dem Mikrocontroller interpretiert wird, ohne große Probleme einen Impulszähler implementieren kann. Die heutigen modernen Mikrocontroller wie z.B. der Pico Pi von Raspberry sind ganz offensichtlich schnell genug um den Bytecode bis zu einer Zählrate von über 20kcps korrekt zu interpretieren.