Schreiben von Interrupt-Handlern¶

Auf geeigneter Hardware bietet MicroPython die Möglichkeit, Interrupt-Handler in Python zu schreiben. Interrupt-Handler - auch bekannt als Interrupt-Service-Routinen (ISRs) - sind als Callback-Funktionen definiert. Diese werden als Reaktion auf ein Ereignis wie einen Timer-Trigger oder eine Spannungsänderung an einem Pin ausgeführt. Solche Ereignisse können an jedem beliebigen Punkt der Ausführung des Programmcodes auftreten. Dies hat erhebliche Konsequenzen, von denen einige spezifisch für die Sprache MicroPython sind. Andere sind allen Systemen gemein, die auf Echtzeitereignisse reagieren können. In diesem Dokument werden zunächst die sprachspezifischen Probleme behandelt, gefolgt von einer kurzen Einführung in die Echtzeitprogrammierung für diejenigen, die damit noch nicht vertraut sind.

In dieser Einführung werden vage Begriffe wie "langsam" oder "so schnell wie möglich" verwendet. Dies ist beabsichtigt, da Geschwindigkeiten anwendungsabhängig sind. Die akzeptable Dauer einer ISR hängt von der Häufigkeit der Unterbrechungen, der Art des Hauptprogramms und dem Vorhandensein anderer gleichzeitiger Ereignisse ab.

Tipps und empfohlene Praktiken¶

Hier werden die unten aufgeführten Punkte zusammengefasst und die wichtigsten Empfehlungen für den Interrupt-Handler-Code aufgeführt.

- Halten Sie den Code so kurz und einfach wie möglich.
- Vermeiden Sie die Speicherzuweisung: kein Anhängen an Listen oder Einfügen in Wörterbücher,
- keine Fließkommazahlen. Erwägen Sie die Verwendung von `micropython.schedule`, um die obige Einschränkung zu umgehen.
- Wenn eine ISR mehrere Bytes zurückgibt, verwenden Sie ein vorab zugewiesenes `Bytearray`. Wenn mehrere ganze Zahlen zwischen einer ISR und dem Hauptprogramm ausgetauscht werden sollen, sollte ein Array (`array.array`) verwendet werden.
- Wenn Daten zwischen dem Hauptprogramm und einem ISR ausgetauscht werden, sollten Sie erwägen, Unterbrechungen zu deaktivieren, bevor Sie im Hauptprogramm auf die Daten zugreifen, und sie unmittelbar danach wieder zu aktivieren (siehe Kritische Abschnitte).
- Weisen Sie einen Notfall-Ausnahmepuffer zu (siehe unten).

MicroPython-Themen¶

Der Notfall-Ausnahmepuffer¶

Wenn ein Fehler in einer ISR auftritt, kann MicroPython keinen Fehlerbericht erstellen, es sei denn, es wird ein spezieller Puffer für diesen Zweck erstellt. Das Debugging wird vereinfacht, wenn der folgende Code in einem Programm enthalten ist, das Interrupts verwendet.

```
micropython importieren
micropython.alloc_emergency_exception_buf(100)
```

Einfachheit¶

Aus einer Reihe von Gründen ist es wichtig, den ISR-Code so kurz und einfach wie möglich zu halten. Er sollte nur das tun, was unmittelbar nach dem Ereignis, das ihn ausgelöst hat, getan werden muss: Operationen, die aufgeschoben werden können, sollten an die Hauptprogrammschleife delegiert werden. Normalerweise kümmert sich ein ISR um das Hardware-Gerät, das die Unterbrechung verursacht hat, und macht es für die nächste Unterbrechung bereit. Sie kommuniziert mit der Hauptprogrammschleife, indem sie die gemeinsamen Daten aktualisiert, um anzuzeigen, dass die Unterbrechung aufgetreten ist, und kehrt dann zurück. Eine ISR sollte die Kontrolle so schnell wie möglich an die Hauptschleife zurückgeben. Dies ist kein spezifisches MicroPython-Problem und wird daher weiter unten ausführlicher behandelt.

Kommunikation zwischen einem ISR und dem Hauptprogramm¶

Normalerweise muss eine ISR mit dem Hauptprogramm kommunizieren. Am einfachsten geht das über ein oder mehrere gemeinsam genutzte Datenobjekte, die entweder als global deklariert oder über eine Klasse gemeinsam genutzt werden (siehe unten). Dabei gibt es verschiedene Einschränkungen und Gefahren, auf die im Folgenden näher eingegangen wird. Integer-, Byte- und Bytearray-Objekte werden üblicherweise für diesen Zweck verwendet, ebenso wie Arrays (aus dem Array-Modul), die verschiedene Datentypen speichern können.

Die Verwendung von Objektmethoden als Rückrufe¶

MicroPython unterstützt diese leistungsstarke Technik, die es einer ISR ermöglicht, Instanzvariablen mit dem darunter liegenden Code zu teilen. Außerdem ermöglicht es einer Klasse, die einen Gerätetreiber implementiert, mehrere Geräteinstanzen zu unterstützen. Das folgende Beispiel veranlasst zwei LEDs, mit unterschiedlichen Raten zu blinken.

```
import pyb, micropython
micropython.alloc_emergency_exception_buf(100)
class Foo(object):
    def __init__(self, timer, led):
        self.led = led
        timer.callback(self.cb)
    def cb(self, tim):
        self.led.toggle()

rot = Foo(pyb.Timer(4, freq=1), pyb.LED(1)) grün
= Foo(pyb.Timer(2, freq=0.8), pyb.LED(2))
```

In diesem Beispiel verbindet die `rote` Instanz den Timer 4 mit der LED 1: Wenn eine Unterbrechung des Timers 4 auftritt, wird `red.cb()` aufgerufen, wodurch die LED 1 ihren Zustand ändert. Die `grüne` Instanz funktioniert ähnlich: eine Unterbrechung durch Timer 2 führt zur Ausführung von `green.cb()` und schaltet LED 2 um. Die Verwendung von Instanzmethoden bringt zwei Vorteile mit sich. Erstens ermöglicht eine einzige Klasse die gemeinsame Nutzung von Code durch mehrere Hardware-Instanzen. Zweitens ist bei einer gebundenen Methode das erste Argument der Callback-Funktion `self`. Dadurch kann der Callback auf Instanzdaten zugreifen und den Zustand zwischen aufeinanderfolgenden Aufrufen speichern. Wenn die obige Klasse beispielsweise eine Variable `self.count` im Konstruktor auf Null gesetzt hat, könnte `cb()` den Zähler erhöhen. Die `roten` und `grünen` Instanzen würden dann unabhängig voneinander zählen, wie oft jede LED ihren Zustand geändert hat.

Erstellung von Python-Objekten¶

ISRs können keine Instanzen von Python-Objekten erzeugen. Das liegt daran, dass MicroPython den Speicher für das Objekt aus einem freien Speicherblock, dem Heap, zuweisen muss. Dies ist in einem Interrupt-Handler nicht zulässig, da die Heap-Zuweisung nicht re-entrant ist. Mit anderen Worten, die Unterbrechung könnte auftreten, wenn das Hauptprogramm gerade dabei ist, eine Zuweisung durchzuführen - um die Integrität des Heaps zu wahren, verbietet der Interpreter Speicherzuweisungen in ISR-Code.

Dies hat zur Folge, dass ISRs keine Fließkomma-Arithmetik verwenden können, da Floats Python-Objekte sind. Ebenso kann eine ISR nicht ein Element an eine Liste anhängen. In der Praxis kann es schwierig sein, genau zu bestimmen, welche Codekonstrukte versuchen werden, eine Speicherzuweisung durchzuführen und eine Fehlermeldung zu provozieren: ein weiterer Grund, ISR-Code kurz und einfach zu halten.

Eine Möglichkeit, dieses Problem zu vermeiden, besteht darin, dass die ISR vorab zugewiesene Puffer verwendet. Ein Klassenkonstruktor erzeugt zum Beispiel eine Bytearray-Instanz und ein boolesches Flag. Die ISR-Methode ordnet die Daten den Positionen im Puffer zu und setzt das Flag. Die Speicherzuweisung erfolgt im Hauptprogrammcode, wenn das Objekt instanziiert wird, und nicht in der ISR.

Die E/A-Methoden der MicroPython-Bibliothek bieten in der Regel eine Option zur Verwendung eines vorab zugewiesenen Puffers. Zum Beispiel
`pyb.i2c.recv()` kann einen veränderbaren Puffer als erstes Argument akzeptieren: Dies

ermöglicht die Verwendung in einer ISR. Eine Möglichkeit, ein Objekt zu erzeugen, ohne eine

Klasse oder Globals zu verwenden, ist wie folgt:

```
def set_volume(t, buf=bytearray(3)):
    buf[0] = 0xa5
    buf[1] = t >> 4
    buf[2] = 0x5a
    return buf
```

Der Compiler instanziiert das Standardargument `buf`, wenn die Funktion zum ersten Mal geladen wird (normalerweise, wenn das Modul, in dem sie enthalten ist, importiert wird).

Eine Instanz der Objekterstellung tritt auf, wenn ein Verweis auf eine gebundene Methode erstellt wird. Dies bedeutet, dass eine ISR eine gebundene Methode nicht an eine Funktion übergeben kann. Eine Lösung besteht darin, einen Verweis auf die gebundene Methode im Klassenkonstruktor zu erstellen und diesen Verweis in der ISR zu übergeben. Zum Beispiel:

```
class Foo():
    def __init__(self):
        self.bar_ref = self.bar # Zuweisung erfolgt hier
        self.x = 0.1
        tim = pyb.Timer(4)
        tim.init(freq=2)
        tim.callback(self.cb)
```

```
def bar(self, _):
    self.x *= 1.2
    print(self.x)

def cb(self, t):
```

```
        # Die Übergabe von self.bar würde eine Zuordnung verursachen.
        micropython.schedule(self.bar_ref, 0)
```

Andere Techniken sind die Definition und Instanziierung der Methode im Konstruktor oder die Übergabe von `Foo.bar()` mit dem Argument *self*.

Verwendung von Python-Objekten¶

Eine weitere Einschränkung für Objekte ergibt sich aus der Arbeitsweise von Python. Wenn eine Importanweisung ausgeführt wird, wird der Python-Code in Bytecode kompiliert, wobei eine Codezeile in der Regel mehreren Bytecodes entspricht. Wenn der Code ausgeführt wird, liest der Interpreter jeden Bytecode und führt ihn als eine Reihe von Maschinencodeanweisungen aus. Da zwischen den Maschinencodeanweisungen jederzeit eine Unterbrechung auftreten kann, wird die ursprüngliche Python-Codezeile möglicherweise nur teilweise ausgeführt. Folglich kann ein Python-Objekt wie ein Set, eine Liste oder ein Wörterbuch, das in der Hauptschleife geändert wird, zum Zeitpunkt der Unterbrechung nicht mehr konsistent sein.

Ein typisches Ergebnis ist wie folgt. In seltenen Fällen wird die ISR genau zu dem Zeitpunkt ausgeführt, zu dem das Objekt teilweise aktualisiert wird. Wenn die ISR versucht, das Objekt zu lesen, kommt es zu einem Absturz. Da solche Probleme typischerweise bei seltenen, zufälligen Gelegenheiten auftreten, können sie schwer zu diagnostizieren sein. Es gibt Möglichkeiten, dieses Problem zu umgehen, die in den folgenden kritischen Abschnitten beschrieben werden.

Es ist wichtig, sich darüber klar zu werden, was die Änderung eines Objekts ausmacht. Eine Änderung eines eingebauten Typs wie eines Wörterbuchs ist problematisch. Die Änderung des Inhalts eines Arrays oder Bytearrays ist es nicht. Das liegt daran, dass Bytes oder Wörter als eine einzige Maschinencodeanweisung geschrieben werden, die nicht unterbrochen werden kann: Im Sprachgebrauch der Echtzeitprogrammierung ist das Schreiben atomar. Ein benutzerdefiniertes Objekt kann einen Integer, ein Array oder ein Bytearray instanziieren. Es ist sowohl für die Hauptschleife als auch für die ISR zulässig, den Inhalt dieser Objekte zu ändern.

MicroPython unterstützt ganze Zahlen mit beliebiger Genauigkeit. Werte zwischen 2**30 -1 und -2**30 werden in einem einzigen Maschinenwort gespeichert. Größere Werte werden als Python-Objekte gespeichert. Folglich können Änderungen an langen Ganzzahlen nicht als atomar angesehen werden. Die Verwendung von langen Ganzzahlen in ISRs ist unsicher, da eine Speicherzuweisung versucht werden kann, wenn sich der Wert der Variablen ändert.

Überwindung der Float-Beschränkung¶

Im Allgemeinen ist es am besten, die Verwendung von Fließkommazahlen in ISR-Code zu vermeiden: Hardwaregeräte verarbeiten normalerweise Ganzzahlen, und die Umwandlung in Fließkommazahlen erfolgt normalerweise in der Hauptschleife. Es gibt jedoch einige DSP-Algorithmen, die Fließkommazahlen benötigen. Auf Plattformen mit Hardware-Fließkomma (wie dem Pyboard) kann der Inline-ARM-Thumb-Assembler verwendet werden, um diese Einschränkung zu umgehen. Dies liegt daran, dass der Prozessor Fließkommawerte in einem Maschinenwort speichert; Werte können zwischen der ISR und dem Hauptprogrammcode über ein Array von Fließkommawerten ausgetauscht werden.

Verwendung von micropython.schedule¶

Mit dieser Funktion kann ein ISR einen Rückruf zur Ausführung "sehr bald" einplanen. Der Callback steht in der Warteschlange für die Ausführung, die zu einem Zeitpunkt erfolgt, zu dem der Heap nicht gesperrt ist. Daher kann er Python-Objekte erstellen und Fließkommazahlen verwenden. Außerdem wird der Callback garantiert zu einem Zeitpunkt ausgeführt, zu dem das Hauptprogramm alle Aktualisierungen von Python-Objekten abgeschlossen hat, so dass der Callback nicht auf teilweise aktualisierte Objekte trifft.

Typische Anwendung ist die Handhabung von Sensor-Hardware. Der ISR holt Daten von der Hardware ab und ermöglicht ihr, einen weiteren Interrupt auszulösen. Anschließend wird ein Rückruf zur Verarbeitung der Daten eingeplant.

Geplante Rückrufe sollten den unten beschriebenen Grundsätzen für den Entwurf von Unterbrechungsbehandlungsprogrammen entsprechen. Damit sollen Probleme vermieden werden, die sich aus der E/A-Aktivität und der Änderung gemeinsam genutzter Daten ergeben und die in jedem Code auftreten können, der der Hauptprogrammschleife vorgreift.

Die Ausführungszeit muss im Verhältnis zu der Häufigkeit, mit der Unterbrechungen auftreten können, betrachtet werden. Wenn eine Unterbrechung auftritt, während der vorherige Rückruf ausgeführt wird, wird eine weitere Instanz des Rückrufs zur Ausführung in die Warteschlange gestellt; diese wird ausgeführt, nachdem die aktuelle Instanz abgeschlossen wurde. Eine anhaltend hohe Unterbrechungshäufigkeit birgt daher das Risiko eines unkontrollierten Anstiegs der Warteschlange und eines möglichen Scheiterns mit einem `RuntimeError`.

Wenn es sich bei dem an `schedule()` zu übergebenden Callback um eine gebundene Methode handelt, beachten Sie den Hinweis unter "Erstellung von Python-Objekten".

Ausnahmen¶

Wenn ein ISR eine Ausnahme auslöst, wird diese nicht an die Hauptschleife weitergegeben. Die Unterbrechung wird deaktiviert, es sei denn, die Ausnahme wird durch den ISR-Code behandelt.

Allgemeine Themen¶

Dies ist nur eine kurze Einführung in die Thematik der Echtzeitprogrammierung. Anfänger sollten beachten, dass Designfehler in Echtzeitprogrammen zu Fehlern führen können, die besonders schwer zu diagnostizieren sind. Dies liegt daran, dass sie selten und in eher zufälligen Abständen auftreten können. Es ist entscheidend, dass der ursprüngliche Entwurf richtig ist und dass man Probleme vorhersieht, bevor sie auftreten. Sowohl die Interrupt-Handler als auch das Hauptprogramm müssen unter Berücksichtigung der folgenden Punkte entworfen werden.

Interrupt Handler Design¶

Wie bereits erwähnt, sollten ISRs so einfach wie möglich gestaltet sein. Sie sollten immer in einer kurzen, vorhersehbaren Zeitspanne zurückkehren. Dies ist wichtig, denn wenn die ISR läuft, läuft die Hauptschleife nicht: Die Hauptschleife macht zwangsläufig an zufälligen Stellen im Code Pausen in ihrer Ausführung. Solche Pausen können eine Quelle für schwer zu diagnostizierende Fehler sein, insbesondere wenn ihre Dauer lang oder variabel ist. Um die Auswirkungen der ISR-Laufzeit zu verstehen, ist ein grundlegendes Verständnis der Unterbrechungsprioritäten erforderlich.

Unterbrechungen sind nach einem Prioritätsschema organisiert. ISR-Code kann selbst durch einen Interrupt höherer Priorität unterbrochen werden. Dies hat Auswirkungen, wenn die beiden Unterbrechungen Daten gemeinsam nutzen (siehe unten "Kritische Abschnitte"). Wenn eine solche Unterbrechung auftritt, wird eine Verzögerung in den ISR-Code eingefügt. Tritt eine Unterbrechung niedrigerer Priorität auf, während die ISR läuft, wird sie verzögert, bis die ISR abgeschlossen ist: Ist die Verzögerung zu lang, kann die Unterbrechung niedrigerer Priorität fehlschlagen. Ein weiteres Problem

bei langsamen ISRs ist der Fall, dass während ihrer Ausführung ein zweiter Interrupt desselben Typs auftritt. Der zweite Interrupt wird nach Beendigung des ersten bearbeitet. Wenn jedoch die Anzahl der eingehenden Unterbrechungen ständig die Kapazität des ISR übersteigt, sie zu bearbeiten, ist das Ergebnis nicht erfreulich.

Folglich sollten Schleifenkonstruktionen vermieden oder minimiert werden. E/A an andere Geräte als das unterbrechende Gerät sollte normalerweise vermieden werden: E/A wie Festplattenzugriff, Druckanweisungen und UART

Zugriff ist relativ langsam, und seine Dauer kann variieren. Ein weiteres Problem ist, dass Dateisystemfunktionen nicht reentrant sind: Die Verwendung von Dateisystem-E/A in einer ISR und dem Hauptprogramm wäre gefährlich. Entscheidend ist, dass ISR-Code nicht auf ein Ereignis warten sollte. E/A ist akzeptabel, wenn garantiert werden kann, dass der Code in `einem` vorhersehbaren Zeitraum zurückkehrt, z. B. durch Umschalten eines Pins oder einer LED. Der Zugriff auf das unterbrechende Gerät über I2C oder SPI kann notwendig sein, aber die für solche Zugriffe benötigte Zeit sollte berechnet oder gemessen und ihre Auswirkungen auf die Anwendung bewertet werden.

In der Regel besteht die Notwendigkeit, Daten zwischen der ISR und der Hauptschleife auszutauschen. Dies kann entweder über globale Variablen oder über Klassen- oder Instanzvariablen geschehen. Bei den Variablen handelt es sich in der Regel um Integer- oder Boolean-Typen oder um Integer- oder Byte-Arrays (ein vorab zugewiesenes Integer-Array bietet schnelleren Zugriff als eine Liste). Wenn mehrere Werte durch den ISR geändert werden, muss der Fall berücksichtigt werden, dass die Unterbrechung zu einem Zeitpunkt erfolgt, zu dem das Hauptprogramm auf einige, aber nicht auf alle Werte zugegriffen hat. Dies kann zu Inkonsistenzen führen.

Betrachten Sie den folgenden Entwurf. Ein ISR speichert eingehende Daten in einem Bytearray und addiert dann die Anzahl der empfangenen Bytes zu einer Ganzzahl, die die Gesamtzahl der zur Verarbeitung bereiten Bytes darstellt. Das Hauptprogramm liest die Anzahl der Bytes, verarbeitet die Bytes und löscht dann die Anzahl der bereitstehenden Bytes. Dies funktioniert so lange, bis eine Unterbrechung eintritt, kurz nachdem das Hauptprogramm die Anzahl der Bytes gelesen hat. Der ISR fügt die hinzugefügten Daten in den Puffer ein und aktualisiert die empfangene Anzahl, aber das Hauptprogramm hat die Anzahl bereits gelesen und verarbeitet daher die ursprünglich empfangenen Daten. Die neu eingetroffenen Bytes gehen verloren.

Es gibt verschiedene Möglichkeiten, diese Gefahr zu vermeiden, wobei die einfachste darin besteht, einen runden Puffer zu verwenden. Wenn es nicht möglich ist, eine Struktur mit inhärenter Fadensicherheit zu verwenden, werden im Folgenden andere Möglichkeiten beschrieben.

Wiedereintrittsstellen¶

Eine potenzielle Gefahr kann auftreten, wenn eine Funktion oder Methode zwischen dem Hauptprogramm und einer oder mehreren ISRs oder zwischen mehreren ISRs gemeinsam genutzt wird. Das Problem dabei ist, dass die Funktion selbst unterbrochen werden kann und eine weitere Instanz dieser Funktion ausgeführt wird. Wenn dies geschehen soll, muss die Funktion so gestaltet werden, dass sie reentrant ist. Wie man das macht, ist ein fortgeschrittenes Thema, das den Rahmen dieses Tutorials sprengen würde.

Kritische Abschnitte¶

Ein Beispiel für einen kritischen Codeabschnitt ist ein Code, der auf mehr als eine Variable zugreift, die von einem ISR betroffen sein kann. Tritt die Unterbrechung zufällig zwischen den Zugriffen auf die einzelnen Variablen auf, so sind deren Werte inkonsistent. Dies ist ein Beispiel für eine Gefahr, die als Race Condition bekannt ist: die ISR und die Hauptprogrammschleife wetteifern um die Änderung der Variablen. Um Inkonsistenzen zu vermeiden, muss sichergestellt werden, dass die ISR die Werte für die Dauer des kritischen Abschnitts nicht verändert. Eine Möglichkeit, dies zu erreichen, besteht darin, `pyb.disable_irq()` vor dem Beginn des Abschnitts und `pyb.enable_irq()` am Ende auszuführen. Hier ist ein Beispiel für diesen Ansatz:

```
pyb, micropython, array importieren
micropython.alloc_emergency_exception_buf(100)

class BoundsException(Exception):
    pass
```

```
ARRAYSIZE = const(20)
Index = 0
Daten = array.array('i', 0 for x in range(ARRAYSIZE))

def callback1(t):
    global data, index
    for x in range(5):
        data[index] = pyb.rng() # Eingabe simulieren
        index += 1
        wenn index >= ARRAYSIZE:
            raise BoundsException('Array-Grenzen

überschritten') tim4 = pyb.Timer(4, freq=100,

callback=callback1)

for loop in range(1000):
    if index > 0:
        irq_state = pyb.disable_irq() # Beginn des kritischen
        Abschnitts for x in range(index):
            print(daten[x])
        index = 0
        pyb.enable_irq(irq_state) # Ende des kritischen
        Abschnitts print('loop {}'.format(loop))
    pyb.delay(1)

tim4.callback(Keine)
```

Ein kritischer Abschnitt kann aus einer einzigen Codezeile und einer einzigen Variablen bestehen. Betrachten Sie das folgende Codefragment.

```
Anzahl = 0
def cb(): # Ein Interrupt-Callback
    Zählung +=1
def main():
    # Code zum Einrichten des Interrupt-Callbacks
    entfällt while True:
        Anzahl += 1
```

Dieses Beispiel veranschaulicht eine subtile Quelle von Fehlern. Die Zeile `count += 1` in der Hauptschleife birgt die Gefahr einer speziellen Race Condition, die als read-modify-write bekannt ist. Dies ist eine klassische Ursache von Fehlern in Echtzeitsystemen. In der Hauptschleife liest MicroPython den Wert von `t.counter`, addiert 1 dazu und schreibt ihn zurück. In seltenen Fällen tritt der Interrupt nach dem Lesen und vor dem Schreiben auf. Die Unterbrechung ändert `t.counter`, aber die Änderung wird von der Hauptschleife überschrieben, wenn die ISR zurückkehrt. In einem realen System könnte dies zu seltenen, unvorhersehbaren Fehlern führen.

Wie bereits erwähnt, ist Vorsicht geboten, wenn eine Instanz eines in Python eingebauten Typs im Hauptcode geändert wird und auf diese Instanz in einer ISR zugegriffen wird. Der Code, der die Änderung vornimmt, sollte als kritischer Abschnitt betrachtet werden, um sicherzustellen, dass sich die Instanz in einem gültigen Zustand befindet, wenn die ISR ausgeführt wird.

Besondere Vorsicht ist geboten, wenn ein Datensatz von verschiedenen ISRs gemeinsam genutzt wird. Hier besteht die Gefahr, dass die Unterbrechung höherer Priorität eintritt, wenn die Unterbrechung niedrigerer Priorität die gemeinsam genutzten Daten teilweise aktualisiert hat. Der Umgang mit dieser Situation ist ein fortgeschrittenes Thema, das den Rahmen dieser Einführung sprengen würde, mit der Ausnahme, dass die unten beschriebenen Mutex-Objekte manchmal verwendet werden können.

Die Deaktivierung von Unterbrechungen für die Dauer eines kritischen Abschnitts ist die übliche und einfachste Vorgehensweise, aber sie deaktiviert alle Unterbrechungen und nicht nur diejenige, die zu Problemen führen könnte. Es ist im Allgemeinen nicht wünschenswert, einen Interrupt für längere Zeit zu deaktivieren. Im Falle von Timer-Interrupts führt dies zu einer Variabilität des Zeitpunkts, zu dem ein Rückruf erfolgt. Im Falle von Geräte-Interrupts kann dies dazu führen, dass das Gerät zu spät bedient wird, was zu Datenverlusten oder Überlauffehlern in der Gerätehardware führen kann. Wie ISRs sollte auch ein kritischer Abschnitt im Hauptcode eine kurze, vorhersehbare Dauer haben.

Ein Ansatz zum Umgang mit kritischen Abschnitten, der die Zeit, in der Unterbrechungen deaktiviert sind, radikal reduziert, ist die Verwendung eines Objekts, das als Mutex bezeichnet wird (der Name leitet sich vom Begriff des gegenseitigen Ausschlusses ab). Das Hauptprogramm sperrt den Mutex vor der Ausführung des kritischen Abschnitts und gibt ihn am Ende wieder frei. Der ISR prüft, ob der Mutex gesperrt ist. Ist dies der Fall, umgeht er den kritischen Abschnitt und kehrt zurück. Die Herausforderung beim Entwurf besteht darin, zu definieren, was die ISR tun soll, wenn der Zugriff auf die kritischen Variablen verweigert wird. Ein einfaches Beispiel für einen Mutex finden Sie hier. Beachten Sie, dass der Mutex-Code zwar Unterbrechungen deaktiviert, aber nur für die Dauer von acht Maschinenbefehlen: Der Vorteil dieses Ansatzes ist, dass andere Unterbrechungen praktisch nicht betroffen sind.

Unterbrechungen und die REPL¶

Unterbrechungsbehandlungsprogramme, wie z. B. solche, die mit Zeitgebern verbunden sind, können nach Beendigung eines Programms weiterlaufen. Dies kann zu unerwarteten Ergebnissen führen, wenn man erwartet hätte, dass das Objekt, das den Rückruf auslöst, den Anwendungsbereich verlassen hat. Zum Beispiel auf dem Pyboard:

```
def bar():
    foo = pyb.Timer(2, freq=4, callback=lambda t: print('.', end=''))

bar()
```

Dieser läuft so lange, bis der Timer explizit deaktiviert oder die Karte mit `ctrl D` zurückgesetzt wird.