Berechnungszeiten von PI
unter Verwendung von FreeRTOS und ESP32-S3

Nathanael Gubler

12. Oktober 2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Aufgabenstellung

Zur Berechnung der Kreiszahl π sollten zwei Methoden unabhängig voneinander gestartet und deren Berechnungszeit für 5 Nachkommastellen verglichen werden können. Dies soll in C auf einem ESP32-S3 realisiert werden, unter Verwendung des FreeRtos Betriebssystems.

Aufgabe:

- Realisiere die Leibniz-Reihen-Berechnung in einem Task.
- Wähle einen weiteren Algorithmus aus dem Internet
- Realisiere den Algorithmus in einem weiteren Task.
- Schreibe einen Steuertask, der die zwei erstellten Tasks kontrolliert

Zudem waren das Verhalten bei Drücken von bestimmten Tasten sowie die Update Time des Displays vorgegeben.

Kapitel 2

Berechnungsmethoden

2.1 Methode A: Madhava/Leibniz

Diese Berechnungsmethode ist eine der rudimentären Methoden zur Berechnungsmethoden zur Approximation von Pi. Geläufig "Leibniz-Methode"genannt, wurde sie allerdings schon im 14. Jahrhundert vom indischen gelehrten Madhava von Sangamagrama (c. 1340 – c. 1425) beschrieben. Die kann als infinite Serie in folgender Form beschrieben werden:

$$\frac{\pi}{4} = \frac{1}{3} - \frac{1}{5} + \frac{1}{7} - \frac{1}{9} + \dots \tag{2.1}$$

Durch den Wechsel zwischen Addition und Subtraktion entsteht ein hin- und her-*˝hüpfen˝* um die Kreiszahl π herum.

2.2 Methode B: Chudnovsky

Eine alternative zu 2.1 stellt die vom den Brüdern David Volfovich Chudnovsk und Gregory Volfovich Chudnovsky entwickelte Methode dar. Diese ist eine Erweiterung einer vom indischen Mathematikers Srinivasa Ramanujan entwickelten Methode. Sie zeichnet sich durch eine sehr schnelle Konvergenz zu π aus, mit jeweils mehreren Nachkommastellen pro Iteration:

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(545140134 * k + 13591409)}{(3k)!(k!)^3 640320^{3k+\frac{3}{2}}} \tag{2.2}$$

Diese Formel kann umgeformt und vereinfacht weiter werden, sodass sie einfacher auf einem Prozessor zu realisieren ist. Im Internet existierte dazu bereits ein fertiger Code in Python, der allerdings rekursive Funktionsaufrufe beinhaltete:

```python
import decimal


def binary_split(a, b):
    if b == a + 1:
        Pab = -(6*a - 5)*(2*a - 1)*(6*a - 1)
        Qab = 10939058860032000 * a**3
        Rab = Pab * (545140134*a + 13591409)
    else:
        m = (a + b) // 2
        Pam, Qam, Ram = binary_split(a, m)
        Pmb, Qmb, Rmb = binary_split(m, b)

        Pab = Pam * Pmb
        Qab = Qam * Qmb
        Rab = Qmb * Ram + Pam * Rmb
    return Pab, Qab, Rab


def chudnovsky(n):
    """Chudnovsky algorithm."""
    P1n, Q1n, R1n = binary_split(1, n)
    return (426880 * decimal.Decimal(10005).sqrt() * Q1n) / (13591409*Q1n + R1n)


print(f"1 = {chudnovsky(2)}")  # 3.14159265358979323846264338 4

decimal.getcontext().prec = 100 # number of digits of decimal precision
for n in range(2,10):
    print(f"{n} = {chudnovsky(n)}")  # 3.14159265358979323846264338...
```

Im Laufe der Programmierarbeit führte der rekursive Ansatz jedoch zu instabilitäten auf dem ESP32. Daher wurde später ein Zwischenschritt in der Umformung verwendet:

$$\pi = \frac{426880\sqrt{10005}}{13591409 + (545140134k + 13591409) * \sum_{k=1}^{\infty} \prod_{1}^{k} \frac{-(6k-1)(2j-1)(6j-5)}{10939058860032000k^3}} \tag{2.3}$$

Diese Umformung führte zu mehr Stabilität. Jedoch entstanden extrem grossen Zahlen und darum auch extrem kleine Zwischenprodukte. Deswegen war der ESP32 bereits nach der zweiten Iteration überlastet. Das führte dazu, dass ein Schutzmechanismus eingebaut werden musste, der den Rechenprozess abbrach sobald zu kleine Zahlen entstanden.

Kapitel 3

Softwareaufbau

Der Programmcode instanziert fünf verschiedene Task und vier Eventgroups, welche zur Synchronisation zwischen den Task dienen. Zudem waren globale Variablen zur Speicherung der aktuellen Daten sowie des Resultats nötig. Diese sind als Timestamps, kurz *ts*, umgesetzt und beinhalten neben dem aktuellen Wert auch Anzahl Iterationen, start- und end-tick, sowie ob die Genauigkeit erreicht wurde.

3.1 Eventgroups

- **Btn_Eventgroup**
  Die Tasteneingaben werden in diese Eventgroup geschrieben. Dabei wird zwischen kurzen und langen Tastendrücken unterschieden. Nach der Verarbeitung wird die Eventgroup wieder auf 0x00 gesetzt.
- **Calc_Eventgroup_A/B**
  Steuert die Zustände des jeweiligen Berechnungstasks (siehe 3.3).
- **MethodInfo_Eventgroup**
  Zeigt, welche Berechnungsmethode vom Benutzer derzeit ausgewählt ist.

3.2 Taskübersicht

- **ButtonTask**
  - Überprüft Tastenfunktion auf Änderungen und schreibt diese in Btn_Eventgroup.
  - Priorität: **10**
  - RAM: **2*2048**
  - Updatetime: ***ca.* 50ms**
  - Schreibt Eventgroup: **Btn_Eventgroup**
- **LogicTask**
  - Wartet auf Änderungen von Btn_Eventgroup und kontrolliert die Berechnungs-Eventgroups.
  - Priorität: **5**
  - RAM: **2*2048**
  - Updatetime: **undefiniert**
  - Liest Eventgroup: **Btn_Eventgroup, MethodInfo_Eventgroup**
  - Schreibt Eventgroup: **Calc_Eventgroup_A, Calc_Eventgroup_B, MethodInfo_Eventgroup**
- **DisplayTask**
  - Zeigt Text gemäss CalcTask Zuständen und globalen Daten an.
  - Priorität: **4**
  - RAM: **2*2048**
  - Updatetime: ***ca.* 500ms**
  - Liest Eventgroup: **MethodInfo_Eventgroup, Calc_Eventgroup_A, Calc_Eventgroup_B,**
  - Liest globale Variablen: **g_running_ts_A, g_running_ts_B, g_calc_result_A, g_calc_result_B**
  - Schreibt Eventgroup: **Calc_Eventgroup_A, Calc_Eventgroup_B**
- **CalcTaskA**
  - Berechnet iterativ gemäss der Madhava/Leibniz Methode (siehe 2.1) und überprüft das Resultat auf die gegebenen Grenzwerte.
  - Priorität: **2**
  - RAM: **8*2048**
  - Updatetime: ***ca.* 1ms**
  - Liest Eventgroup: **Calc_Eventgroup_A**
  - Schreibt Eventgroup: **Calc_Eventgroup_A**
  - Schreibt globale Variablen: **g_running_ts_A, g_calc_result_A**
- **CalcTaskB**
  - Berechnet iterativ gemässe der Chudnovsky Methode (siehe 2.2) und überprüft das Resultat auf die gegebenen Grenzwerte.
  - Priorität: **2**

- RAM: **8*2048**
- Updatetime: ***ca.* 1ms**
- Liest Eventgroup: **Calc_Eventgroup_B**
- Schreibt Eventgroup: **Calc_Eventgroup_B**
- Schreibt globale Variablen: **g_running_ts_B, g_calc_result_B**

Der DisplayTask nutzt die Funktion *GetCurrTimestamp* um den jeweiligen Berechnungstask anzuhalten, die aktuellen Daten abzugreifen und dann wieder zu starten. Für die Anzeige der Resultate werden die Daten gelesen sobald die Genauigkeit erreicht wurde und der jeweilige Berechnungstask nicht das Resultat timestamp beschreibt. Abbildung 3.1 zeigt die Schnittstellen zwischen den Task sowie die Struktur des Timestamp structs.

3.3 Task states

Die Berechnungstasks *CalcTaskA* und *CalcTaskB* prüfen in jedem loop, in welchem Zustand sie sich befinden. Dieser Zustand ist in der jeweiligen *Calc_Eventgroup* festgelegt. Jeder Berechnungs-Task kann daher unabhängig voneinander die folgenden Zustände einnehmen:

- **STOPPING**: Setzt die *Calc_Eventgroup* auf **STOPPED** und blockiert dann bis ein anderer Zustand in die Eventgroup gesetzt wird.
- **RESETTING**: Setzt alle Werte des aktuellen sowie des Resultat-Timestamps zurück und setzt die *Calc_Eventgroup* auf **STOPPING**.
- **STARTING**: Setzt den start tick count des aktuellen Timestamps zurück falls die Berechnung neu gestartet wurde. Setzt dann die *Calc_Eventgroup* auf **RUNNING**.
- **RUNNGING**: Berechnet die nächste Iteration der jeweiligen Methode und überprüft dann fortlaufend, ob sich der aktuelle Wert innerhalb der gegebenen Grenzwerte befindet.

Zusätzlich gibt es noch den temporären Zustand **WRITING_RESULT**, welcher allerdings nur zur Synchronisation dient. Abbildung 3.2 zeigt die Zustände und wie sie erreicht werden können. Der LogicTask steuert den Wechsel zwischen den vom Benutzer ausgewählten Berechnungsmethode und leitet die Tasteneingaben jeweils um. Abbildung 3.3 zeigt diese Zustände.

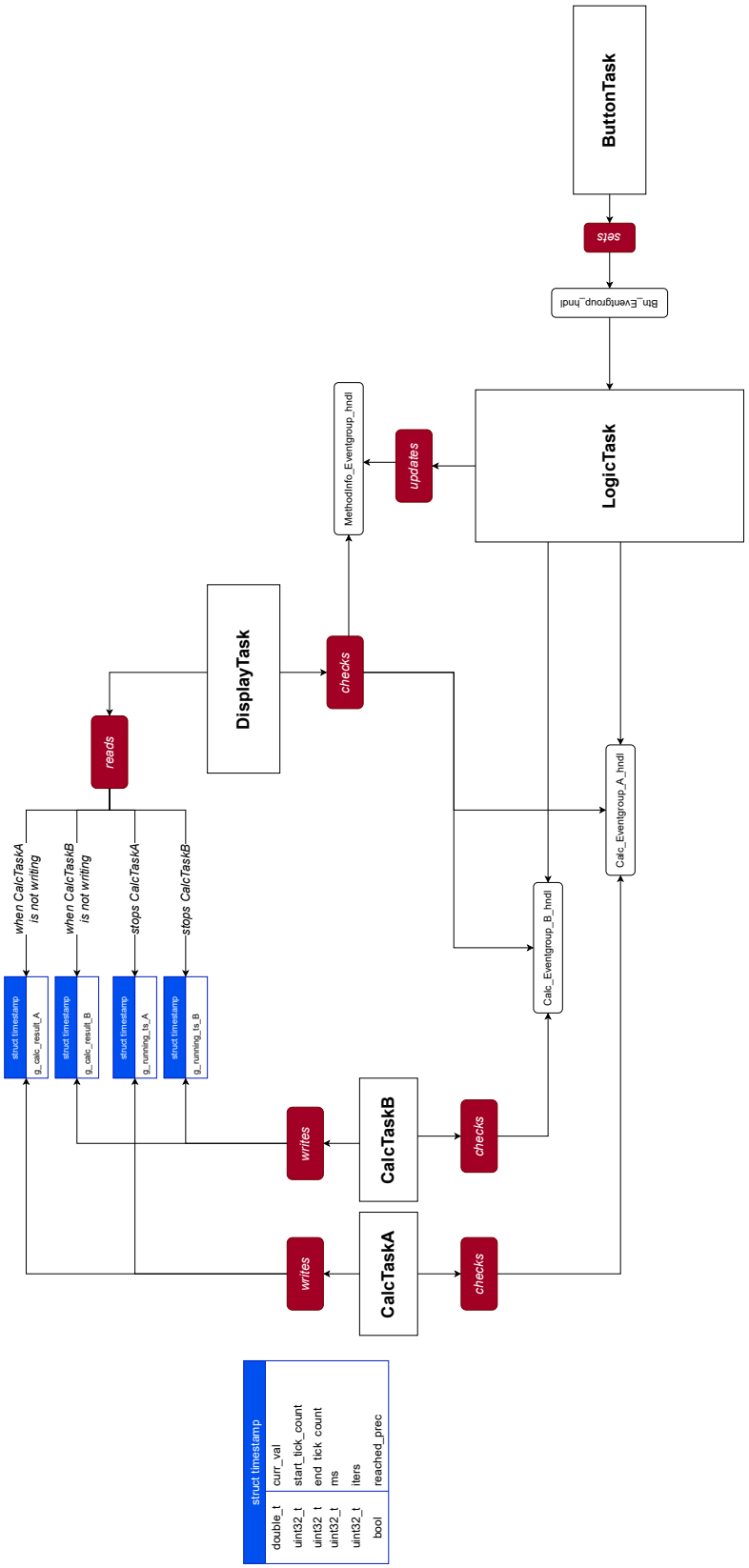

Abbildung 3.1: Die Tasks kommunizieren jeweils über Eventgroups oder globale Variabeln.

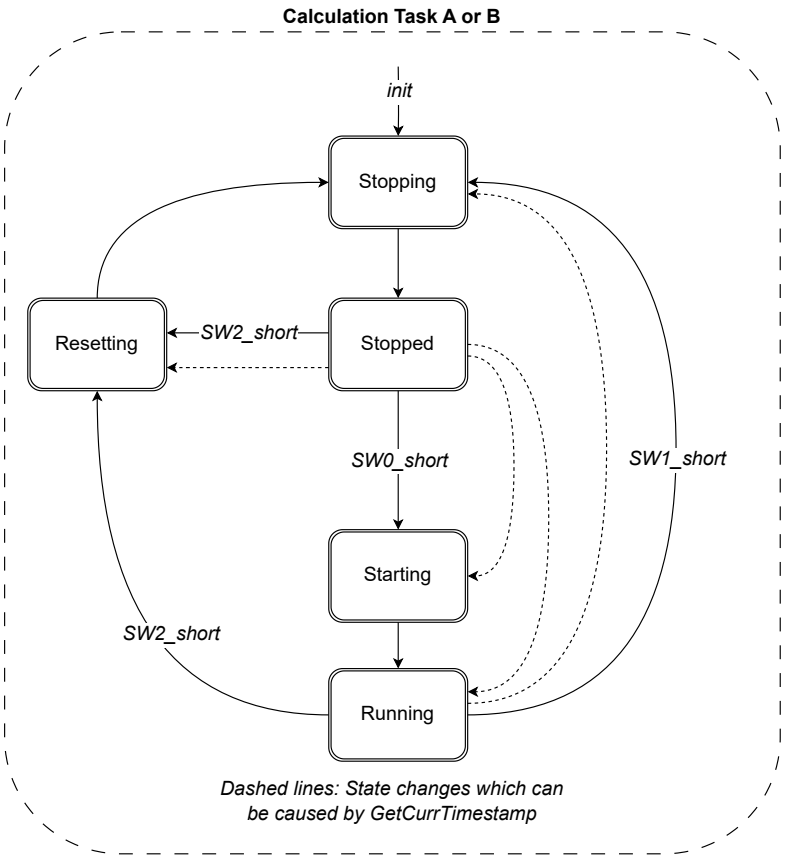

Abbildung 3.2: State-Event Diagramm der Berechnungstasks. Übergänge ohne Text laufen automatisch ab.

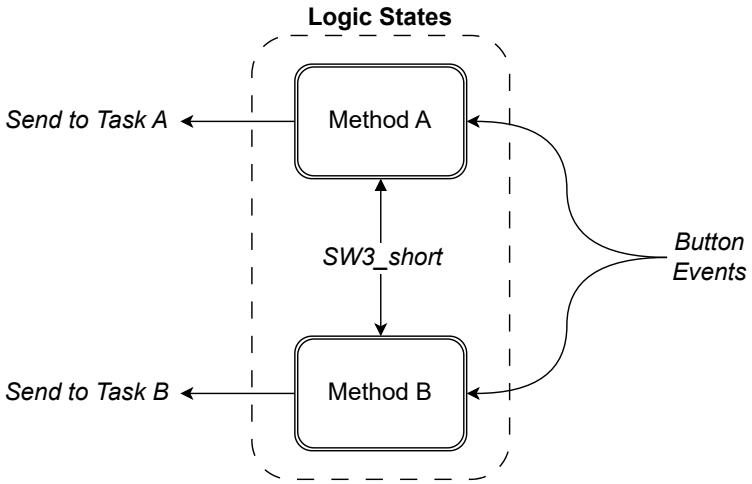

Abbildung 3.3: Die Zustände der Calc_Eventgroups werden jeweils gemäss aktuell ausgewähltem Task und Tasteneingaben gesetzt.

Kapitel 4

Resultate

4.1 Direktvergleich

Es wurden mit beiden Methoden jeweils 10 Durchläufe gestartet und die Zeit bis die Methode fünf Nachkommastellen erreicht hatte gemessen. Tabelle 4.1 zeigt die Resultate für Madhava-Leibniz. CalcTaskA, also die Madhava Methode, erreichte in Testläufen eine Genauigkeit von fünf

	Methode A		**Methode B**
Lauf	**Zeit [ms]**	**Lauf**	**Zeit [ms]**
1	**1740**	1	**0**
2	**1742**	2	**0**
3	**1733**	3	**0**
4	**1742**	4	**0**
5	**1743**	5	**0**
6	**1738**	6	**0**
7	**1735**	7	**0**
8	**1750**	8	**0**
9	**1742**	9	**0**
10	**1732**	10	**0**

Tabelle 4.1: Die Messresultate zeigen deutliche Unterschiede zwischen den Methoden.

Nachkommastellen nach **1741ms**$\pm$**9ms**. CalcTaskB, also die Chundnovsky Methode, erreichte in

allen Testläufen bereits nach der ersten Iteration eine Genauigkeit von 12 Nachkommastellen und konnte daher gleich im ersten Tick erreicht werden. Dies führte dazu, dass für die Zeitberechnung eine Tick-Differenz von 0 entstand und daher auch ein Zeitunterschied von *˝0ms˝*.

4.2 Interpretation der Messungen

Die Ergebnisse in 4.1 zeigen zum einen die enorme Effizienzunterschiede zwischen den beiden Methoden: Während die Madhava Methode mit jeder Iteration länger um eine Ziffer hin- und her springt, hantiert die Chudnovsky Methode bereits mit der ersten Iteration mit sehr grossen Zahlen. Somit hat die letztere Methode den Vorteil, mit jeder Iteration mehrere Dutzend Stellen sehr effizient zu berechnen.
Zudem lässt sich aus dem Geschwindigkeitsunterschied ableiten, dass ein Prozessor wie der ESP32 so aufgebaut ist, dass grosse Zahlen schneller verarbeitet werden als viele kleinere.