Juventus
Technikerschule HF
www.technikerschule.ch

Embedded Systems

Calculating PI

Von Robin Bühler (5. Semester)

Inhaltsverzeichnis:

1 Aufgabenstellung

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.

Dabei soll folgendes stets gegeben sein:

- Der aktuelle Wert soll stets gezeigt werden. Update alle 500ms
- Der Algorithmus wird mit einem Tastendruck gestartet und mit einem anderen Tastendruck gestoppt.
- Mit einer dritten Taste kann der Algorithmus zurückgesetzt werden.
- Mit der vierten Taste kann der Algorithmus umgestellt werden. (zwischen Leibniz und dem zweiten Algorithmus)
- Die Kommunikation zwischen den Tasks kann entweder mit EventBits oder über TaskNotifications stattfinden.
- Folgende Event-Bits könnte man beispielsweise verwenden:
  - EventBit zum Starten des Algorithmus
  - EventBit zum Stoppen des Algorithmus
  - EventBit zum Zurücksetzen des Algorithmus
  - EventBit für den Zustand des Kalkulationstask als Mitteilung für den Anzeige-Task
- Mindestens drei Tasks müssen existieren.
  - Interface-Task für Buttonhandling und Display-Beschreiben
  - Kalkulations-Task für Berechnung von PI mit Leibniz Reihe
  - Kalkulations-Task für Berechnung von PI mit anderer Methode
- Erweitere das Programm mit einem Zeitmess-Hardware-Timer (wie in der Alarm-Clock Übung) und messe die Zeit, bis PI auf 5 Stellen hinter dem Komma stimmt. (Zeit auf dem Display mitlaufen lassen und bei Erreichen der Genauigkeit den Timer anhalten. Die Berechnung von PI soll weitergehen.)

Abbildung 1 Aufgabenstellung

2 Algorithmus

2.1 Chudnovsky

Der Chudnovsky Algorithmus zur Berechnung von Pi ist eine schnelle Berechnungsmethode welche auf der Formel von Ramanujan basiert. Es wurde 1988 von den Chudnovsky Brüdern im Jahr 1988 veröffentlicht und wurde bereits mehrfach verwendet, um grosse Mengen an Kommastellen von Pi zu berechnen. So wurden bereits Weltrekorde mit diesem Algorithmus aufgestellt

Da es sehr grosse Zahlenwerte zum Berechnen benötigt, wurde die avr_f64 Library eingebunden, mit welcher es möglich ist 64-Bit float Werte zu erstellen und damit zu rechnen.

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

Abbildung 2 Chudnovsky Formel

Der Algorithmus hat jedes Mal bei der Division freeRTOS zum Absturz gebracht. Ich nehme an, es hat mit den grossen Zahlen zu tun und der Division mit der 64Bit Library. Auch macht das Verwenden der Library den Code sehr unübersichtlich, da jede mathematische Funktion nun mit einer Funktion programmiert werden muss.

Beispiel der Schreibweise:

$$a + a * 2 \;\; = \;\; f\_add(\,a, f\_mult(a, 2))$$

Da der Chudnovsky Algorithmus eine Weiterentwicklung von dem Ramanjan Algorithmus ist, könnte es mit diesem vielleicht möglich sein, PI zu berechnen. Die Zahlenwerte wären auf jeden Fall viel kleiner.

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{99^2}\sum_{k=0}^{\infty}\frac{(4k)!}{k!^4}\frac{26390k+1103}{396^{4k}}$$

Abbildung 3 Ramanujan-Sato Formel

2.2 Euler

Leonhard Euler konnte bereits im Jahre 1748 148 Stellen von Pi mit der von ihm entdeckten Formel berechnen.

$$\zeta(2) = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots = \frac{\pi^2}{6}$$
$$\zeta(4) = \frac{\pi^4}{90}, \quad \zeta(6) = \frac{\pi^6}{945}, \quad \ldots$$
$$\frac{\pi^2}{8} = \frac{1}{1^2} + \frac{1}{3^2} + \frac{1}{5^2} + \frac{1}{7^2} + \frac{1}{9^2} + \cdots$$

Abbildung 4 Formel von Leonhard Euler

Mithilfe von der Riemannsche Zeta-Funktion konnte im 18. Jarhundert von Euler das Basler Problem gelöst werden.

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \cdots.$$

Abbildung 5 Basler Problem

$$\frac{\pi^2}{6} \approx 1{,}644934$$

Abbildung 6 Reihenwert

Für den Algorithmus verwende entschied ich mich dafür, diese Formel zu verwenden, welche unter dem Basler Problem bekannt war im 18. Jahrhundert. Ich nehme an die andere Form, welche unten jeweils in zweier Schritten inkrementiert wäre ein besserer Algorithmus für den Microcontroller, da bei diesem eine Kommazahlen wahrscheinlich schneller generiert werden könnte.

Dies wäre noch etwas zum Ausprobieren, ob es einen Unterschied gibt auf dem Microcontroller zwischen diesen zwei F

3 Task

Für die Berechnung von PI werden drei verschiedene Task aufgesetzt.

```
xTaskCreate( vControllerTask,   (const char *) "control_tsk",
xTaskCreate( vLeibnizTask,      (const char *) "leibniz_tsk",
xTaskCreate( vEuler_PI,         (const char *) "Euler_PI_tsk",
```

Im Controller Task wird die Tastenabfrage, Steuerungslogik und die Displayausgabe berechnet. Der Leibniz-Task berechnet π mit dem Leibniz Algorithmus. Der Euler Task berechnet π nach dem Euler Algorithmus.

3.1 State Machine

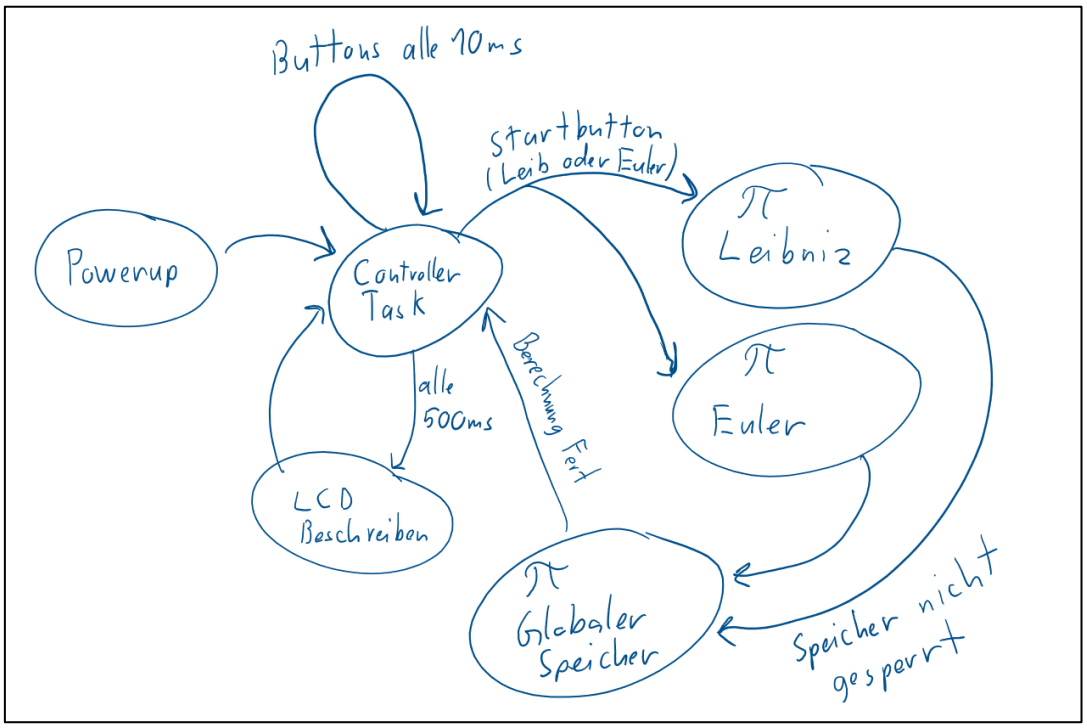

Abbildung 7 State Machine Code

Der wichtigste Task ist der Controller-Task, welcher die beiden anderen Task steuert. Der Task wird ca. alle 10 ms ausgeführt, was genügt, damit die Tastenabfrage funktioniert. Was ich noch ausprobieren wollte, ist das vTaskDelay mit einem vTaskDelayUnitl zu ersetzen, damit die 10 ms genauer eingehalten werden. Da das Programm mit dem vTaskDelay funktioniert, ist es geblieben.

Wichtige Bestandteile von vom Controller Task.

- Buttonhandler (Funktion Tastenabfrage)
- Software-Timer ca. alle 500ms
    - Menu Steuerung mit Tasten
    - Switch mit Grundzuständen
        - Start Menue
            - LCD
        - Berechnung Start
            - Kommunikation mit Berechnungstask
            - Zeitmessung
            - LCD
        - Berechnung Stopp
            - LCD

LeibnizTask:

- Berechnung PI
- Kommunikation / Steuerung mit Controller Task

Euler PI:

- Berechnung PI

4 Task Notification

Damit die Tasks untereinander kommunizieren können, wurde eine Eventgroup aufgesetzt. Dabei sind die Flags in vier verschiedene Kategorien unterteilt:

1. TASK → Die Task defines sind dazu da, um Informationen zwischen dem Berechnungs- und dem Steuertask zu übertragen.
2. DATA → Mit den DATA defines werden die Passiven und Aktivenvariabeln Geschütz.
3. RESET → Mit dem Reset kann die Berechnung sowie die Buttons zurückgesetzt werden.
4. BUTTON → Eventgroups zum Auswerten, ob eine Taste gedrückt wurde.

```
EventGroupHandle_t egPI_Calc;

#define TASK_CHUDNOVSKY             1 << 0
#define TASK_LEIBNITZ               1 << 1
#define TASK_EULER                  1 << 2

#define DATA_READ_REQUEST_LOCK      1 << 3
#define DATA_CALCULATION_READY      1 << 4
#define DATA_READ_LOCK_CLEARED      1 << 5

#define RESET_PI                    1 << 6

#define BUTTON_START                1 << 7
#define BUTTON_STOP                 1 << 8
#define BUTTON_RESET                1 << 9
#define BUTTON_SWITCH               1 << 10

#define TASK_START_STOP             1 << 11
#define TASK_TIME_FINISHED          1 << 12

#define RESET_EG_BUTTONS            0x780 //Reset all Buttons
```

Abbildung 8 Event Group

4.1 Ressourcenschutz

Um die Globale PI Variable zu schützen, werden die DATA Flags aus der Eventgroup verwendet. Die P-Ressourcen sind dabei die globalen PI-Variabeln und die A-Ressource die Event Flags. Im Anzeigetask wird die ganze Logik gesteuert. Da dieser die globale PI-Variable alle 500 ms auslesen soll, wird dieser Ablauf mit der Eventgroup geschützt. Es ist jeweils nur ein Berechnungstask aktiv, deshalb reicht ein Eventgroupsatz für die Kommunikation.

1. Ein neuer Wert soll auf dem LCD angezeigt werden. → Setzen vom Lock Flag.
    a. Jetzt wird darauf gewartet, dass der Berechnungstask bereit ist
2. Ist das Lock Flag gesetzt, setzt der Berechnungstask das Calculation-finished Flag.
    a. Jetzt wartet der Berechnungstask darauf, dass das Lock wieder geöffnet wird.
3. Wenn das Calculation-finished Flag erkannt wurde, wird PI aus der globalen Variable herausgelesen und gespeichert.
    a. Das Lock wird nach Abspeichern von dem globalen PI wieder geöffnet
4. Lock ist jetzt geöffnet und der Berechnungstask kann wieder in die globale Variable hineinschreiben.

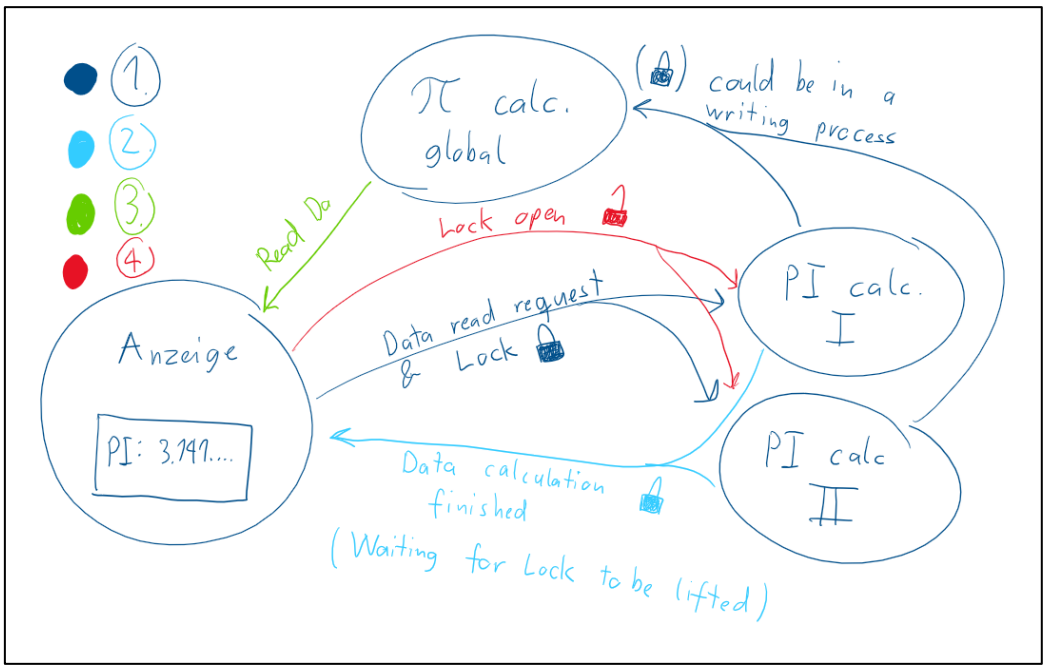

Abbildung 9 Ressourcenschutz Event Group

5 Geschwindigkeit

Die Geschwindigkeit konnte mit der internen TickTime von Scheduler gemessen werden. Dazu kann die xTask-GEt-TickCount Funktion von freeRTOS verwendet werden.

```
xTaskTimeStop = xTaskGetTickCount();
Calculation_Time_ms = (xTaskTimeStop - xTaskTimeStart);
Calculation_Time_s = Calculation_Time_ms / 1000;
```

Abbildung 10 Berechnung der Zeit

Die Leibnitz-Variante brauch ca. 52 bis 53 Sekunden bis 5 Kommastellen berechnet sind. Die Euler-Variante ist sehr langsam. Mit 32-Bit float Variablen ist es nicht möglich, 5 Kommastellen zu berechnen. Auch mit 64-Bit float Variabeln geht es sehr langsam und es war noch nicht möglich, innerhalb von ca. 2 min genug Kommastellen zu berechnen.

6 Rechenleistung

Für die Rechnung wurde die Zeit von dem Softwaretimer genommen, weshalb die Berechneten werden, nur etwa stimmen. Um eine Aussage über die Rechenleistung zu tätigen, reicht dies jedoch. Der Euler-Algorythmus konnte nicht berechnet werden, da dieser nicht 5 Kommastellen berechnen kann, da die Zahlen zu schnell zu gross werden.

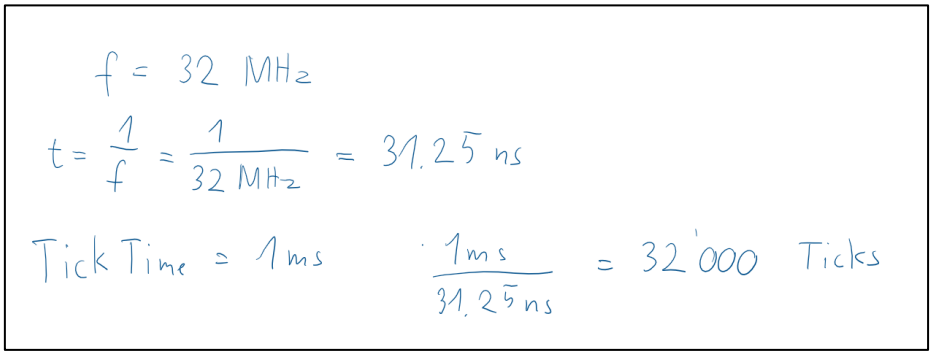

Abbildung 11 Berechnung Scheduler Time und Periodendauer

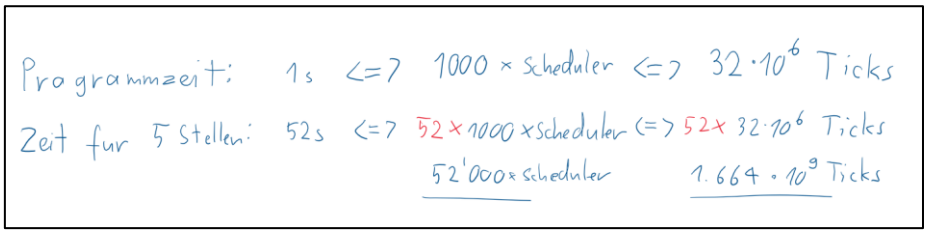

Abbildung 12 Berechnung Anzahl Ticks und Scheduler-Ticks für 5 Kommastellen

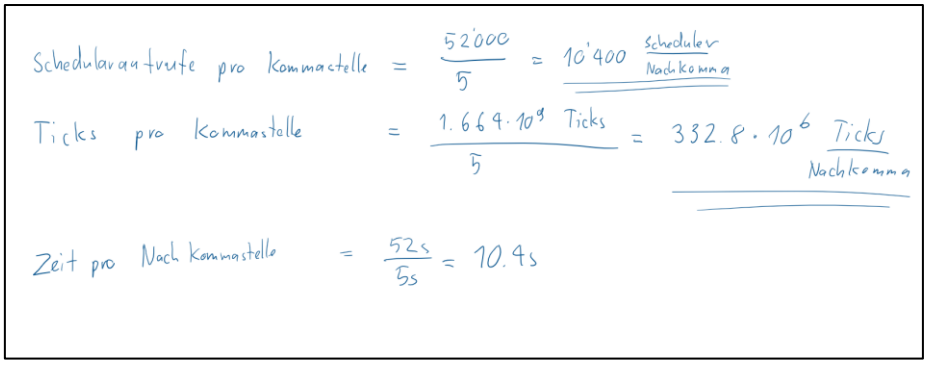

Abbildung 13 Berechnung Scheduler aufrufe, Ticks und Zeit pro Kommastelle

Der Microkontroller benötigt ca. 332 Millionen Taktzyklen zur Berechnung von einer Kommastelle von PI mit meiner Firmware. Um den Leibnitz Algorithmus zu werten und wie effizient dieser ist, muss noch die Scheduler TickTime mit-einbezogen werden. Es wird der Preamtive-Scheduler-Modus verwendet, welcher automatisch alle TickTime den Fokus neu verteilt. Soweit ich verstehe, falls alle Tasks innerhalb der 1ms abgearbeitet werden, ist der Microcontroller im Idle Zustand. Das würde bedeutet das der Algorithmus ca. 10400 Durchläufe benötigt für eine Nachkommastelle. Diese Anzahl Durchlaufe / Nachkommastelle würde ich als Vergleichswert zwischen den Algorithmen verwenden.

Auch wurde angenommen, dass die Berechnungszeit pro Kommastelle Linear ist. Diese Theorie funktioniert nur, wenn der Scheduler bei Abschluss aller Tasks in 1ms nicht nochmals in den PI-Task geht.

7 Fazit

7.1 Abweichungen

Die Steuerung zwischen den beiden verschiedenen Task ist noch nicht fertig implementiert. Die Steuerungslogik ist bereits im Controllertask vorhanden, wird jedoch noch nicht verwendet. Deshalb ist die Zeitmessung nur von dem Leibnitz Task möglich. Da der Euler-Task es jedoch nicht schafft 5 Kommastellen zu berechnen, was ich jeweils mit 32 und 64 bit float ausprobierte, habe ich diese noch nicht implementiert, sowie auch aus Zeitgründen.

7.2 Erkenntnisse

Das Vergleichen von den Variablen war schwierig zu implementieren, da in manuell das Ende des Strings bei der 6. Kommazahl hinzufügen musste, damit es funktioniert. Mithilfe von ausprobieren und Google funktionierte es irgendwann.

```
sprintf(&pi_compare[0], "%f", lokal_leib_pi);
pi_compare[7] = '\0';

if ( strcmp(pi_comp_loesung, pi_compare ) == 0)
{

    if (lokal_task_flag == pdFALSE )
    {
        xEventGroupSetBits(egPI_Calc, TASK_TIME_FINISHED);
        lokal_task_flag = pdTRUE;

    }
}
```

Abbildung 14 Überprüfung von PI

Teilweises funktioniert die PI-Überprüfung nicht, da der String nicht richtig abgespeichert wird. Der erste char geht verloren und wird auf 0 gesetzt, obwohl dieser 3 sein sollte. Ich konnte dieses Problem während dem Debuggen beheben, indem ich den Ort, wo die Variable erstellt wird, verschoben habe. Dies hat jedoch nicht immer funktioniert. Der erste char wird während der Laufzeit irgendwann auf «0» gesetzt.

```
154     char pi_compare[8];                 //Überprüfung 5 Kommastellen
155     char pi_comp_loesung[] = "3.14159";
156
157     while(1)
158     {
```

117 %

Watch 1

Name	Value
pi_comp_loesung	0x2d63
[0]	0
[1]	46
[2]	49
[3]	52
[4]	49
[5]	53
[6]	57
[7]	0

Die Berechnung mit dem Chudonovsky Algorithmus war sehr herausfordernd, zusammen mit der 64-Bit float Library. Das schwierigste, was das es sehr unübersichtlich wurde mit den mathematischen Operationen sowie das Handling der verschiednen Datentypen. Dies führte dazu, dass ich sehr viel Zeit damit verbrachte zu versuchen, diesen Algorithmus zu programmieren

```
f_chud_helpA = f_div( f_sd( pow( -1, count_pi) *  rCalcFakultaet(6 * count_pi)), f_sd(  rCalcFakultaet(3 * count_pi) * rCalcFakultaet(count_pi) * pow(640320, (3 * count_pi))) );
f_chud_helpB = f_mult( f_sd(count_pi) , f_chud_helpA);
f_Chudnov_PI = f_div(   (f_sd( 426880 * f_pow( f_sd(10005), f_sd(0.5))))  ,  (   f_add( f_mult(f_sd(13591409), f_chud_helpA )  ,   f_mult( f_sd(545140134) , f_chud_helpB)) )  );
```

Abbildung 15 Chudonvosky Algorithmus versuch

Leider haben meine Berechnungen ab einem bestimmten Punkt immer das Betriebssystem zum Absturz gebracht. Nach meiner Vermutung liegt dies entweder an dem begrenzten RAM von dem Microcontroller oder an einem anderen Fehler im Algorithmus.

8 Anhang

8.1 Quellen

Chudnovsky:

https://www.craig-wood.com/nick/articles/pi-chudnovsky/
https://en.wikipedia.org/wiki/Chudnovsky_algorithm
https://en.wikipedia.org/wiki/Srinivasa_Ramanujan

Euler PI:

https://3.14159265358979323846264338327950288419716939937510582097494459 2.eu/pi-berechnen-formeln-und-algorithmen/
https://de.wikipedia.org/wiki/Kreiszahl#Produktformeln_von_Leonhard_Euler
https://de.wikipedia.org/wiki/Basler_Problem

8.2 Dateien:

8.3 Bildverzeichnis