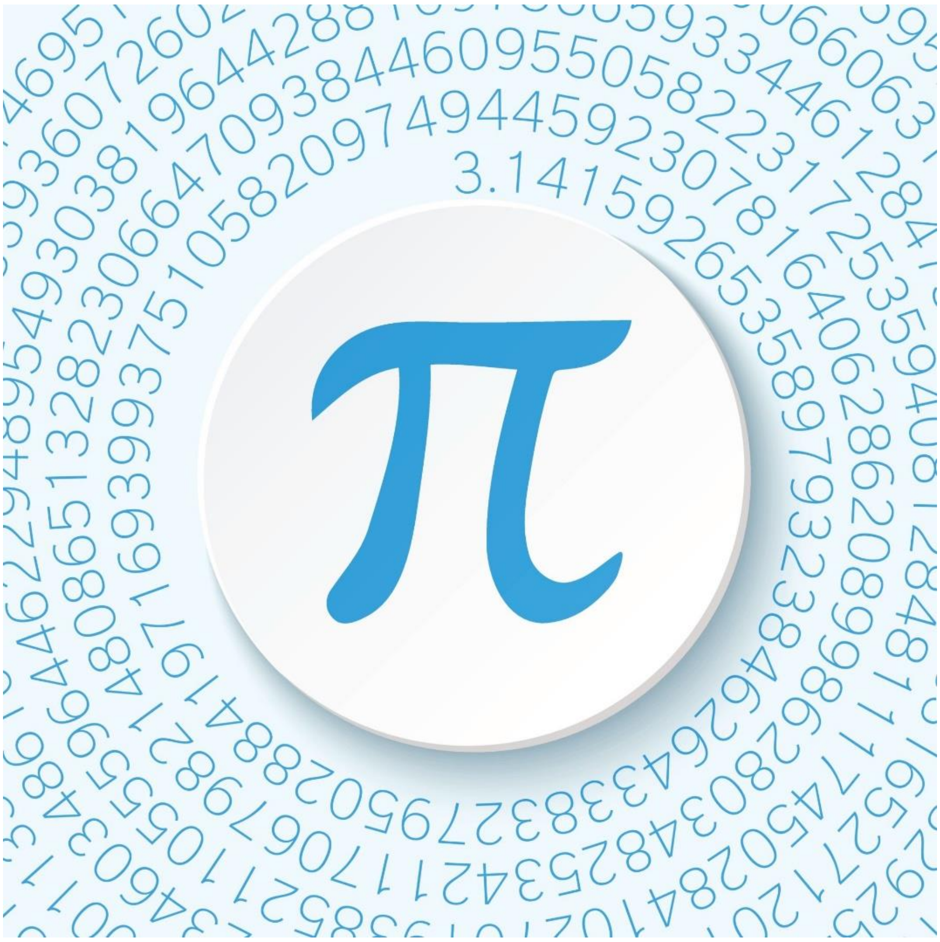
Juventus Technikerschule HF
www.technikerschule.ch

Calculating Pi Projekt

Projektarbeit im Fach Embedded Systems



Inhaltsverzeichnis:

1 Einleitung 3
1.1 Aufgabenstellung 3
1.2 Pi Basics 4
1.3 Berechnungsmethoden 4
1.3.1 Archimedes 4
1.3.2 Leibniz-Reihe 4
1.3.3 Nilikantha-Reihe 5
1.3.4 Euler-Reihe 5
1.3.5 Entscheid 5
1.4 Abschliessen Entscheidung 5
2 Software 6
2.1 Einleitung 6
2.2 EventGroup 6
2.3 Programmablauf 7
2.3.1 Start-Up 7
2.3.2 ControllerTask 7
2.3.3 Berechnung Madhava-Leibniz-Reihe 10
2.3.4 Berechnung Nilikantha-Reihe 11
2.4 Vergleich der Berechnungsalgorithmen 12
2.5 CPU-Leistung 12
3 Fazit 13
3.1 Technisches Fazit 13
3.2 Persönliches Fazit 13
4 Anhang 14
4.1 Abbildungsverzeichnis 14
4.2 Bilderquelle 14
4.3 Tabellenverzeichnis 14
4.4 Code 14
4.5 Literaturverzeichnis 14

1 Einleitung

1.1 Aufgabenstellung

Martin Burger hat uns mit folgender Aufgabenstellung konfrontiert:
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:
o EventBit zum Starten des Algorithmus
o EventBit zum Stoppen des Algorithmus
o EventBit zum Zurücksetzen des Algorithmus
o EventBit für den Zustand des Kalkulationstask als Mitteilung für den Anzeige-Task

- Mindestens drei Tasks müssen existieren.
o Interface-Task für Buttonhandling und Display-Beschreiben
o Kalkulations-Task für Berechnung von PI mit Leibniz Reihe
o Kalkulations-Task für Berechnung von PI mit anderer Methode

- Erweitere das Programm mit einer Zeitmess-Funktion (verwende xTaskGetTickCount) und messe die Zeit, bis PI auf 5 Stellen hinter dem Komma stimmt. (Zeit auf dem Display mitlaufen lassen und beim Erreichen der Genauigkeit die Zeit berechnen. Die Berechnung von PI soll weitergehen.)

1.2 Pi Basics

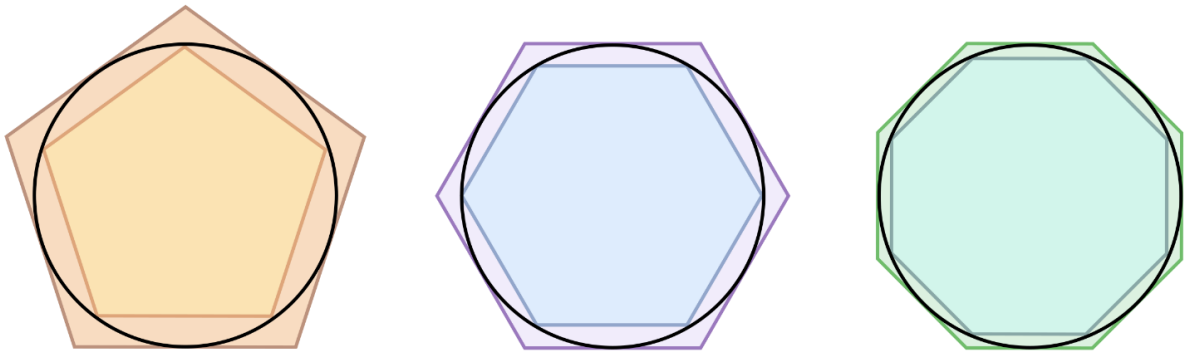
Am Anfang musste geklärt werden, was Pi ist und wie man es berechnet. Pi auch bekannt als die Kreiszahl gehört zu den wichtigsten Konstanten der Mathematik. Pi ist eine irrationale Zahl, somit kann sie weder als ganze Zahl noch als Bruch dargestellt werden. Pi wurde bereits von den Babyloniern benutzt. Diese hatten die Zahl auf 3.125 geschätzt. Diese Schätzung ist nicht genau.

Pi lässt sich leider nicht mit einer einfachen Formel berechnen, sondern muss über Annäherungsversuche genauer beschrieben werden. Diese Annäherungsversuche werden wir uns im nächsten Kapitel genauer anschauen.

1.3 Berechnungsmethoden

1.3.1 Archimedes

Als eine der ersten Berechnungsmethode für Pi war die die Methode von Archimedes. Archimedes lebte zwischen 287 v. Chr. – 212 v. Chr. Archimedes wollte Pi nicht direkt bestimmen, sondern eine obere und untere Grenze festlegen. Man konnte zwar nicht den Umfang eines Kreises genau bestimmen, aber den Umfang eines Vielecks berechnen. In der Abbildung 1 kann man sehen, wie die Vielecke immer mehr sich dem Kreis annähern. Mit dieser Methode konnte Archimedes leider nur die Grenzen festlegen, in welchen sich Pi befindet.



[1]

Abbildung 1: Archimedes Methode

1.3.2 Leibniz-Reihe

Gottfried Willhelm Leibniz wollte Pi genauer bestimmen. Er entwickelte eine neue Methode wie man Pi genauer bestimmen kann. Er hat zwischen 1673 – 1676 eine neue Methode entwickelt. Seinen Ansatz basierte auf die bereits bekannte Arkustangens-Reihe. Siehe Abbildung 2. Aus der Arkustangens-Reihe konnte Leibniz nun seine Formel Herleiten. Dieser Formel war jedoch bereits vom Mathematiker Madhava entdeckt. Deswegen heisst die Formel auch Madhava-Leibniz-Reihe, welche in der Abbildung 3 ersichtlich ist. Die Madhava-Leibniz-Reihe konvergiert leider sehr langsam. Nach 32 Brüchen hat man eine Genauigkeit von 3 Nachkommastellen. Aus diesem Grund habe ich mich auf die Suche gemacht nach einem neuen Algorithmus, welcher schneller konvergiert.

$$= \int_0^1 \frac{1}{x^2+1}\,\mathrm{d}x = \Big[\arctan(x)\Big]_{x=0}^{x=1} = \arctan(1) = \frac{\pi}{4}$$ [2]

Abbildung 2: Arkustangens-Reihe

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \mp \cdots$$ [3]

Abbildung 3: Madhava-Leibniz-Reihe

[1] (Kreiszahl, 2023)
[2] (Leibniz-Reihe, 2023)
[3] (Kreiszahl, 2023)

1.3.3 Nilikantha-Reihe

Bei meiner Internetrecherche bin ich über die Nilikantha-Reihe gestolpert. Kelallur Nilakantha Somayaji (1444 - 1544) war ein indischer Astrologe und Mathematiker. Die Formel für die Entwicklung sieht man in der Abbildung 4.

$$\pi = 3 + \frac{4}{3^3 - 3} - \frac{4}{5^3 - 5} + \frac{4}{7^3 - 7} - \frac{4}{9^3 - 9} + \cdots$$ [4]

Abbildung 4: Nilikantha-Reihe

1.3.4 Euler-Reihe

Ich habe mir auch die Euler-Reihe genauer angeschaut. Der Schweizer Mathematiker Leonard Euler (1707 - 1783) hat mithilfe einer komplexen Mathematischen Funktion (Riemannsche Zeta-Funktion) einen Weg gefunden, wie man Pi ausrechnet. Die Formel für die Reihenentwicklung sieht man in der Abbildung 5.

$$\zeta(2) = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots = \frac{\pi^2}{6}$$ [5]

Abbildung 5: Euler-Reihe Pi

1.3.5 Entscheid

Der Leibniz Algorithmus wurde von der Aufgabenstellung vorgegeben. Der zweite Ansatz, welchen ich gerne verfolgen möchte, ist der Ansatz von Nilikantha. Die Euler-Reihe konvergiert zwar um einiges schneller aber benötigt viel mehr Rechenleistung vom EduBoard.

1.4 Abschliessen Entscheidung

Ich werde im Rahmen von meinem Projekt folgendes vorgehen wählen. Als Entwicklungsboard werde ich das EduBoard nehmen. Ich werde mein Projekt auf der Basis von 32 Bit Float Zahlen bauen. Die Variante mit den 64 Bit Float Zahlen wird aus Zeit und Wissens Gründen nicht weiterverfolgt. Ein weiterer Grund, was gegen die 64 Bit zahl spricht, ist das verbaute Display auf dem EduBoard. Ich habe zu wenig Platz für einen sinnvollen Nutzen von 64 Bit Float Zahlen.

Mit diesem Kapitel ist der Theoretische Teil von dieser Arbeit abgeschlossen und ich starte mit der Planung und Umsetzung von meinem PiCalculator.

[4] (Steffens, 2023)
[5] (Kreiszahl, 2023)

2 Software

2.1 Einleitung

Für eine saubere Versionsverwaltung werde ich GitHub einsetzten. Mit GitHub ist es mir möglich meinen Programm Code ortsunabhängig zu bearbeiten und abzuspeichern. Es ist auch möglich jede Änderung zwischenzuspeichern. Dies ist sehr Vorteilhaft, wenn man einen Fehler gemacht hat, kann man die Änderung wieder rückgängig machen. Für mein Projekt werde ich folgende Tasks erstellen und programmieren müssen:

1. Control Task
2. Nilikantha-Reihe
3. Madhava-Leibniz-Reihe

Für die Kommunikation zwischen den Task werde ich eine EventGroup erstellen. Mithilfe von dieser EventGroup kann ich die möglichen Zustände meines Programmes darstellen und den Tasks kommunizieren.

2.2 EventGroup

Ich habe mir eine EventGroup mit dem Namen evButtonState erstellt. Diese EventGroup ist 32 Bit gross. In der Tabelle 1 sieht man wie das erste Byte aufgeteilt wurde.

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Start_State	Stop_State	RST_State	LEIBNIZ	NILIK	NOT USED	NOT USED	NOT USED

Tabelle 1: Erstes Byte von EventGroup

Die restlichen drei Bytes von der EventGroup werden nicht benötigt. Somit habe ich sie hier nicht aufgelistet. Um eine saubere Datenstruktur zu gewährleisten habe ich beim Auslesen der EventGroup die Bits maskiert. Maskieren beschreibt eine Technik, um zu verhindern, dass man ein falsches Bit manipuliert. Ich habe am Anfang mit defines vorgefertigte Bitmuster für die EventGroup erstellt. Meine defines werden in der Abbildung 6 gezeigt.

```
#define EVSYSTEM_START  0x1
#define EVSYSTEM_STOP   0x2
#define EVSYSTEM_RESET  0x4
#define LEIBNIZ_STATUS  0x8
#define NILA_STATUS     0x10
#define EVSTATUS_MASK   0x1F
```

Abbildung 6: Defines für EventGroup

2.3 Programmablauf

2.3.1 Start-Up

Bei Starten vom Board werden alles Tasks, Eventgroups und Variablen initialisiert. Ausserdem soll der Nilikantha-Task mit der Funktion vTaskSuspend pausiert werden. Die einzigen Tasks, welche beim Starten laufen dürfen, ist die Controllertask und die LeibnizTask. Somit ist mein Standardalgorithmus die Leibniz-Reihe.

2.3.2 ControllerTask

Die controllerTask soll das Herzstück sein für meinen Code. Ich möchte, dass jeder Prozess am Schluss wieder auf die controllerTask zurückfällt. Ich werde einfachheitshalber User Interface mit UI abkürzen. Die TaskPriority wird bei diesem Task auf 3 festgelegt. Die controllerTask soll die höchste Priorität haben. Ich möchte das zu jedem Zeitpunkt die Eingaben des Users verarbeitet werden können. In der Tabelle 2 kann man sehen, was auf dem Display alles angezeigt werden soll. In der Abbildung 7 sieht man das UI, wenn der Leibniz Algorithmus ausgewählt ist und in der Abbildung 8 ist das UI aufgezeigt, wenn der Nilikantha Algorithmus angewählt ist.

Pos	*Value*
0	Titel vom Projekt
1	Gewählter Algorithmus
2	Der aktuelle wert PI & die Zeit für die Berechnung
3	Menüführung

Tabelle 2: Display Layout UI

Abbildung 7: UI Leibniz

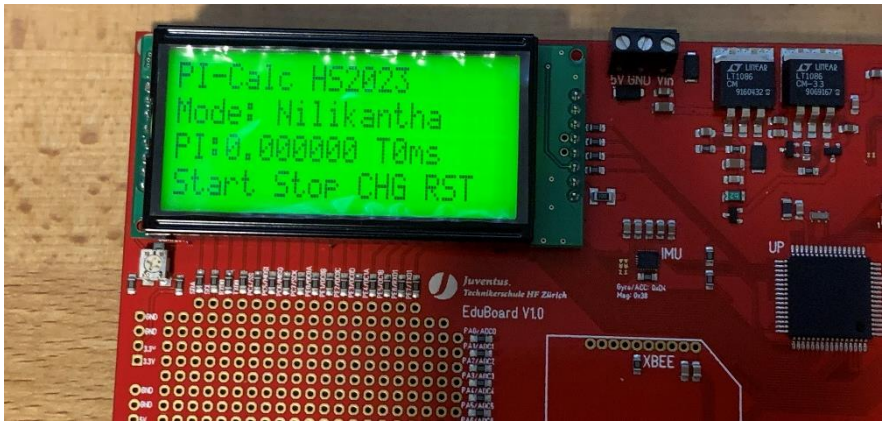


Abbildung 8: UI Nilikantha

Das Display soll alle 500ms aktualisiert werden. Aus diesem Grund habe ich eine Variabel namens displaycounter erstellt. Wenn der Task ausgeführt wird, wird als erstes überprüft, ob diese Variabel gleich null ist. Falls die Variabel Null ist, soll das Display aktualisiert werden. Wenn nicht dann wird der Wert um 1 reduziert. Die Schleife ist in der Abbildung 9 abgebildet.

```
switch(displaycounter){
    case 0:
        vDisplayClear();
        vDisplayWriteStringAtPos(0,0,"PI-Calc HS2023");
        switch(Calc_Mode){
            case 0:
            vDisplayWriteStringAtPos(1,0,"Mode: Leibniz" );
            break;
            case 1:
            vDisplayWriteStringAtPos(1,0,"Mode: Nilikantha");
            break;
            default:
            vDisplayWriteStringAtPos(1,0,"Model: NOT SET");
            break;
        }
        char time_Calc [12];
        char pistring[12];
        sprintf(&pistring[0], "%.6f", pi);
        sprintf(&time_Calc[0], "%lu", time);
        vDisplayWriteStringAtPos(2,0,"PI:%s T:%sms", pistring, time_Calc);
        vDisplayWriteStringAtPos(3,0,"Start Stop CHG RST");
        displaycounter = 50;
        break;
    default:
        displaycounter--;
        break;
}
```

Abbildung 9: Switch Loop Display

2.3.2..1.1 Button Auslesen

Die controllerTask muss das Drücken eines Buttons Erkennen und dieses Event speichern. Als Taskvorlage habe ich die Vorlage von Buttonhandler.c und Buttonhandler.h benutzt. Je nach Tastendruck wird ein Bit in der EventGroup gesetzt und gelöscht. Die EventGroup kann dann von den Berechnungstasks ausgelesen werden. Beim Start Zeitpunkt laufen noch keine Berechnungen im Hintergrund. Mit dem Button 3 kann man zwischen den beiden Algorithmen wechseln. Wenn man mit der Taste 1 den Start Befehl schickt, soll der entsprechende Task, für den ausgewählten Algorithmus anfangen zu rechnen. Wenn der Button 2 gedrückt wird, soll der Task des dementsprechenden Algorithmus wieder pausiert werden. Wenn die gewünschte Genauigkeit von Pi erreicht wird, soll die Berechnung von sich aus pausieren und die benötigte Zeit auf dem Display anzeigen. In der Abbildung 10 ist der Programmcode für das Auslesen der Buttons abgebildet.

```
        if(getButtonPress(BUTTON1) == SHORT_PRESSED) {

            xEventGroupClearBits(evButtonState, EVSTATUS_MASK);
            xEventGroupSetBits(evButtonState, EVSYSTEM_START);
        }
        if(getButtonPress(BUTTON2) == SHORT_PRESSED) {
            xEventGroupClearBits(evButtonState, EVSTATUS_MASK);
            xEventGroupSetBits(evButtonState, EVSYSTEM_STOP);
        }
        if(getButtonPress(BUTTON3) == SHORT_PRESSED) {
            switch(Calc_Mode){
                case 1:
                    vTaskSuspend(NilaCalc);
                    vTaskResume(LeibnizCalc);
                    xEventGroupClearBits(evButtonState, EVSTATUS_MASK);
                    xEventGroupSetBits(evButtonState, EVSYSTEM_RESET);
                    Calc_Mode = 0;
                    break;
                case 0:
                    vTaskSuspend(LeibnizCalc);
                    vTaskResume(NilaCalc);
                    xEventGroupClearBits(evButtonState, EVSTATUS_MASK);
                    xEventGroupSetBits(evButtonState, EVSYSTEM_RESET);
                    Calc_Mode = 1;
                    break;
            }
        }
        if(getButtonPress(BUTTON4) == SHORT_PRESSED) {

            xEventGroupClearBits(evButtonState, EVSTATUS_MASK);
            xEventGroupSetBits(evButtonState, EVSYSTEM_RESET);
        }
        vTaskDelay(10/portTICK_RATE_MS);
    }
```

Abbildung 10: Auslesen der Buttons

2.3.3 Berechnung Madhava-Leibniz-Reihe

Der Leibniz-Berechnungstask ist in zwei Teile aufgeteilt. Der Erste Teil ist ein Endlicher Automat, welche die Werte aus der EventGroup ausliesst und dementsprechend auswertet. Der zweite Teil ist der Berechnungsalgorithmus selbst. Am Anfang werden die zwei Variablen pi4 und n erstellt. Die Variabel n soll anzeigen in welcher Iteration man sich gerade befindet. Ausserdem wird n gebraucht damit die Nenner eine Variabel haben. Bei der Madhava-Leibniz-Reihe nähert man sich ja nicht direkt an Pi, sondern an $\frac{\pi}{4}$. Wegen diesem Vorgehen musste ich den Wert, welcher mir die Reihenentwicklung ausgibt, zwischenspeichern. Die Variabel pi4 ist für mich der Puffer damit ich Pi ausrechnen kann. In diesem Task ist kein vTaskDelay vorhanden. Dies aus dem Grund, dass die Berechnung so schnell wie möglich ausgeführt werden soll. Weil die Berechnungstasks kein vTaskDelay haben wurde die TaskPriority auf 1 gesetzt. Die Begründung hierfür ist die folgende Grundregel: Leistungshungrige Tasks sollten immer eine tiefe Priorität haben.

Danach wird die EventGroup ausgelesen. Der Wert wird in der Variabel systemstate zwischengespeichert. Mein Endlicher Automat habe ich mit einem switch case realisiert. Ich habe mich bewusst für eine switch Schleife entschieden. Dies aus dem Grund für einen möglichst laufzeitoptimierten Code. Als Switch Faktor habe ich die systemstate variabel genommen. Beim case Reset werden die Variablen wieder auf ihren Anfangswert zurückgesetzt. In der EventGroup werden alle Bits gelöscht damit man wieder vom Startpunkt anfangen kann. Beim Stop Case wird die EventGroup geleert. Der Start Case macht als erstes eine Zeitmessung mit xTaskGetTickCount für den Start der Berechnung. xTaskGetTickCount zählt die Anzahl Ticks, welche seit dem Start vom vTaskStartScheduler, verstrichen sind. Danach wird das Bit LEIBNIZ_STATUS in der EventGroup gesetzt. Ausserdem wird das Startbit aus der EventGroup gelöscht. Die Abbildung 11 zeigt die Implementierung von meinem Endlichen Automaten an.

```
float pi4 = 1;
uint32_t n = 3;
for (;;)
{
    systemstate = (xEventGroupGetBits(evButtonState) & EVSTATUS_MASK);

    switch(systemstate){
        case EVSYSTEM_RESET:
            pi4 = 1;
            n = 3;
            pi = 0;
            time = 0;
            xEventGroupClearBits(evButtonState, EVSTATUS_MASK);
            break;
        case EVSYSTEM_STOP:
            xEventGroupClearBits(evButtonState, EVSTATUS_MASK);
            break;
        case EVSYSTEM_START:
            starttime = xTaskGetTickCount();
            xEventGroupSetBits(evButtonState, LEIBNIZ_STATUS);
            xEventGroupClearBits (evButtonState, EVSYSTEM_START);
            break;
```

Abbildung 11: Endlicher Automat Leibniz

Der letzte Case führt die Berechnungen für die Reihenentwicklung durch. In diesem Case befindet sich auch die Überprüfung, ob Pi bereits genau genug berechnet wurde. Wenn die gewünschte Genauigkeit erreicht ist, wird wieder mit xTaskGetTickCount die Anzahl Ticks ausgelesen. Ich habe dann die Differenz zwischen den beiden Tick Werten berechnet und hatte nun die Anzahl Ticks, welche benötig wurden für die Berechnung von Pi. Bei unserem System entspricht 1 Tick genau 1ms. Diesen Zeitwert wird in der Variabel time gespeichert. Diesen Ablauf sieht man in der Abbildung 12.

```
        case LEIBNIZ_STATUS:
            pi4 = pi4 - (1.0/n) + (1.0/(n+2));
            n += 4;
            pi = pi4*4;
            time = xTaskGetTickCount() - starttime;
            if (pi > 3.141598 && pi < 3.141599)
            {
                xEventGroupClearBits(evButtonState, LEIBNIZ_STATUS);
            }
            break;
```

Abbildung 12: Implementierung Leibniz

2.3.4 Berechnung Nilikantha-Reihe

Der Task, welcher für die Berechnung der Nilikantha-Reihe verantwortlich ist, wurde genau gleich aufgebaut wie der Task von der Leibniz-Reihe. Der einzige Unterschied liegt in dem Start Case und der Berechnung selbst. Bei der Nilikantha-Reihe wird beim Start Case nicht das LEIBNIZ_STATUS Bit gesetzt, sondern das NILA_STATUS Bit. In der Abbildung 13 ist der Endliche Automat von der Nilikantha-Reihe dargestellt. Abbildung 14 zeigt die Implementierung des Algorithmus in C. In der Abbildung 15 ist ein Ablaufdiagram von meinem endlichen Automaten dargestellt.

```
systemstate = (xEventGroupGetBits(evButtonState) & EVSTATUS_MASK);

switch(systemstate){
    case EVSYSTEM_RESET:
    k = 3;
    sign = 1;
    pi = 0;
    time = 0;
    xEventGroupClearBits(evButtonState, EVSTATUS_MASK);
    break;
    case EVSYSTEM_STOP:
    xEventGroupClearBits(evButtonState, EVSTATUS_MASK);
    break;
    case EVSYSTEM_START:
    starttime = xTaskGetTickCount();
    xEventGroupSetBits(evButtonState, NILA_STATUS);
    xEventGroupClearBits (evButtonState, EVSYSTEM_START);
    pi = 3;
    break;
```

Abbildung 13: Endlicher Automat Nilikantha

```
        case NILA_STATUS:
        pi = pi + (sign * (4/(pow(k,3) - k)));
        sign = sign *(-1);
        k = k+2;
        time = xTaskGetTickCount() - starttime;
        if (pi > 3.141598 && pi < 3.141599)
        {
            xEventGroupClearBits(evButtonState, NILA_STATUS);
        }
        break;
```

Abbildung 14: Implementierung Nilikantha

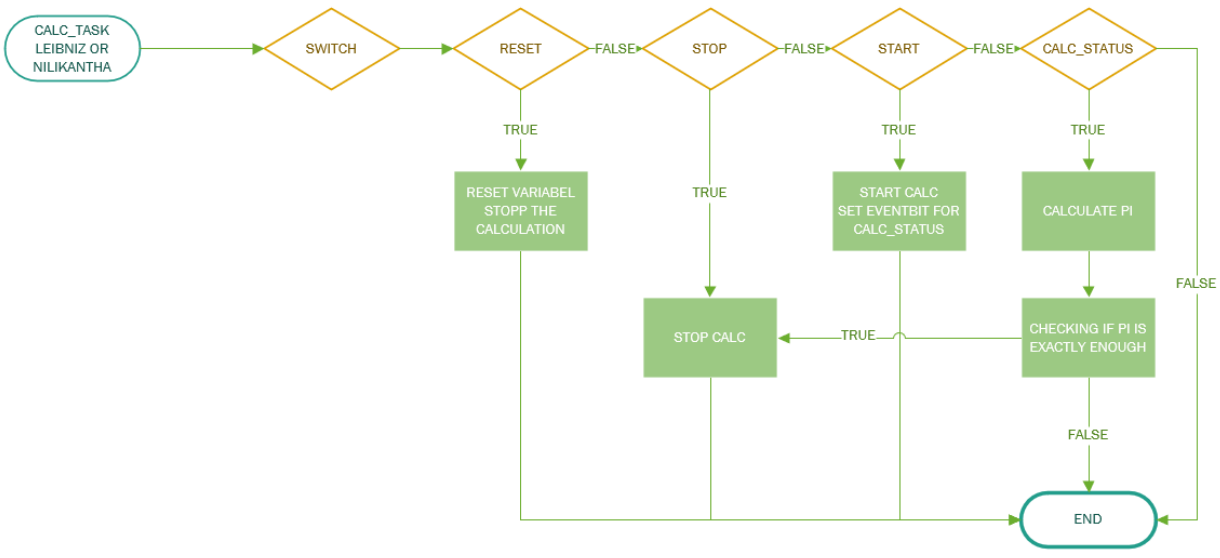


Abbildung 15: Endlicher Automat Calctasks

2.4 Vergleich der Berechnungsalgorithmen

Die Nilikantha-Reihe benötigt 10ms für die Berechnung von Pi. Die Leibniz-Reihe braucht im Durchschnitt 11s. Das finde ich doch sehr erstaunlich. Beide Implementierungen brauchen gleich viele Zeilen Code (3). In der Nilikantha-Reihe wird sogar mit Potenzen gearbeitet (*pow*). Die beiden Algorithmen arbeiten auch sehr ähnlich. Der grosse Unterschied liegt darin, dass bei der Leibniz-Reihe Pi komplett entwickelt wird. Das heisst bei der Leibniz-Reihe fängt man mit dem Startwert 1 an. Bei der Nilikantha-Reihe entwickelt man nur die Nachkommastellen von Pi, weil der Startwert bei 3 liegt. Ich denke, dass dieser Punkt Matchentscheidend ist für die schnellere Entwicklung von Pi. In der Abbildung 16 und in der Abbildung 17 sieht man die gemessen Zeit für die jeweilige Berechnung von Pi.

Abbildung 16: Zeit Nilikantha

Abbildung 17: Zeit Leibniz

2.5 CPU-Leistung

Von meinem jetzigen Wissensstand aus kann ich leider nicht viel bezüglich der CPU-Auslastung sagen. Ich habe zwar eine Formel auf Wikipedia gefunden, welche man in der Abbildung 18 sieht. Aber mit dieser kann ich leider nicht viel anfangen. Ich denke aber nicht, dass die CPU stark ausgelastet ist.

$$\eta = \frac{\sum_{k=1}^{n} t_k}{t_{\text{gesamt}}}$$ [6]

Abbildung 18: Formel CPU Auslastung

[6] (Prozessorauslastung, 2023)

3 Fazit

3.1 Technisches Fazit

Als technisches Fazit kann ich sagen, der Kalkulator läuft. Die Anforderungen, welchen vom Dozenten gegeben worden sind, wurden alle umgesetzt. Das Menü wird alle 500ms aktualisiert. Die Tastenführung funktioniert, soweit ich das beurteilen kann, einwandfrei. Die Algorithmen wurden auch implementiert. Die Kommunikation zwischen den Tasks wird durch eine EventGroup gelöst. Der Programmcode wurde sauber und möglichst kompakt implementiert. Aus Technischer Sicht ist dieses Projekt für mich ein Erfolg.

3.2 Persönliches Fazit

Als persönliches Fazit kann ich sagen, dass mir die Arbeit sehr geholfen hat. Mit dieser Aufgabe konnte ich neue Erfahrungen sammeln. Die grösste Erfahrung war das Einbetten von Mathematischen Algorithmen in C. Ich hatte am Anfang Probleme damit die Formeln in C Code umzuschreiben. Nach einer gewissen Zeit hatte ich das richtige Denkmuster gefunden. Mit diesem Denkmuster konnte ich die Formeln in C Code umschreiben und auch im Programm einbringen. Zudem konnte ich auch meine frischen FreeRTOS Kenntnisse anwenden und vertiefen. Ein weiteres Thema, welches für mich sehr interessant war, war die Menü Funktion. Ich habe noch nicht viele Menüführungen programmiert. Beim Programmieren ist mir aber aufgefallen, dass es mir immer leichter fällt und ich immer schneller werde. Ausserdem habe ich gelernt Code möglichst effizient zu schreiben. Auf unserer Hardware hat man leider nicht viel Rechenleistung zur Verfügung. Ich wollte trotzdem, dass die Berechnung schnell ablaufen. Als abschliessendes Fazit kann ich definitiv sagen, dass ich durch dieses Projekt persönlich gewachsen bin. Zu diesem Wachstum dazu habe ich mir neues Wissen angeeignet und direkt anwenden können. Also ist dieses Projekt auch aus persönlicher Sicht ein Erfolg.

4 Anhang

4.1 Abbildungsverzeichnis

Abbildung 1: Archimedes Methode 4
Abbildung 2: Arkustangens-Reihe 4
Abbildung 3: Madhava-Leibniz-Reihe 4
Abbildung 4: Nilikantha-Reihe 5
Abbildung 5: Euler-Reihe Pi 5
Abbildung 6: Defines für EventGroup 6
Abbildung 7: UI Leibniz 7
Abbildung 8: UI Nilikantha 7
Abbildung 9: Switch Loop Display 8
Abbildung 10: Auslesen der Buttons 9
Abbildung 11: Endlicher Automat Leibniz 10
Abbildung 12: Implementierung Leibniz 10
Abbildung 13: Endlicher Automat Nilikantha 11
Abbildung 14: Implementierung Nilikantha 11
Abbildung 15: Endlicher Automat Calctasks 11
Abbildung 16: Zeit Nilikantha 12
Abbildung 17: Zeit Leibniz 12
Abbildung 18: Formel CPU Auslastung 12

4.2 Bilderquelle

Titelbild: https://t0.gstatic.com/licensed-image?q=tbn:ANd9GcTUbOTXTc2tQEcbj_Ue1ua9B60x0X_PnIcGc2sWI6kOPR2HJbRQ0ME5PsXUxdQpleNh

Abbildung 1: https://de.wikipedia.org/w/index.php?title=Kreiszahl&oldid=237839131

Abbildung 2: https://de.wikipedia.org/w/index.php?title=Leibniz-Reihe&oldid=233489532

Abbildung 3: https://de.wikipedia.org/w/index.php?title=Kreiszahl&oldid=237839131

Abbildung 4: https://3.14159265358979323846264338327950288419716939937510582097494459 2.eu/pi-berechnen-formeln-und-algorithmen/

Abbildung 5: https://de.wikipedia.org/w/index.php?title=Kreiszahl&oldid=237839131

Abbildung 18: https://de.wikipedia.org/w/index.php?title=Prozessorauslastung&oldid=208508548

4.3 Tabellenverzeichnis

Tabelle 1: Erstes Byte von EventGroup 6
Tabelle 2: Display Layout UI 7

4.4 Code

https://github.com/Twister10000/U_PiCalc_HS2023

4.5 Literaturverzeichnis

Kreiszahl. (6. Oktober 2023). Von Wikipedia – Die freie Enzyklopädie.: https://de.wikipedia.org/w/index.php?title=Kreiszahl&oldid=237839131 abgerufen

Leibniz-Reihe. (6. Oktober 2023). Von Wikipedia – Die freie Enzyklopädie.: https://de.wikipedia.org/w/index.php?title=Leibniz-Reihe&oldid=233489532 abgerufen

Prozessorauslastung. (15. Oktober 2023). Von Wikipedia – Die freie Enzyklopädie.: https://de.wikipedia.org/w/index.php?title=Prozessorauslastung&oldid=208508548 abgerufen

Steffens, G. (6. Oktober 2023). *https://3.14159265358979323846264338327950288419716939937510582097494459 2.eu/*. Von https://3.14159265358979323846264338327950288419716939937510582097494459 2.eu/pi-berechnen-formeln-und-algorithmen/ abgerufen