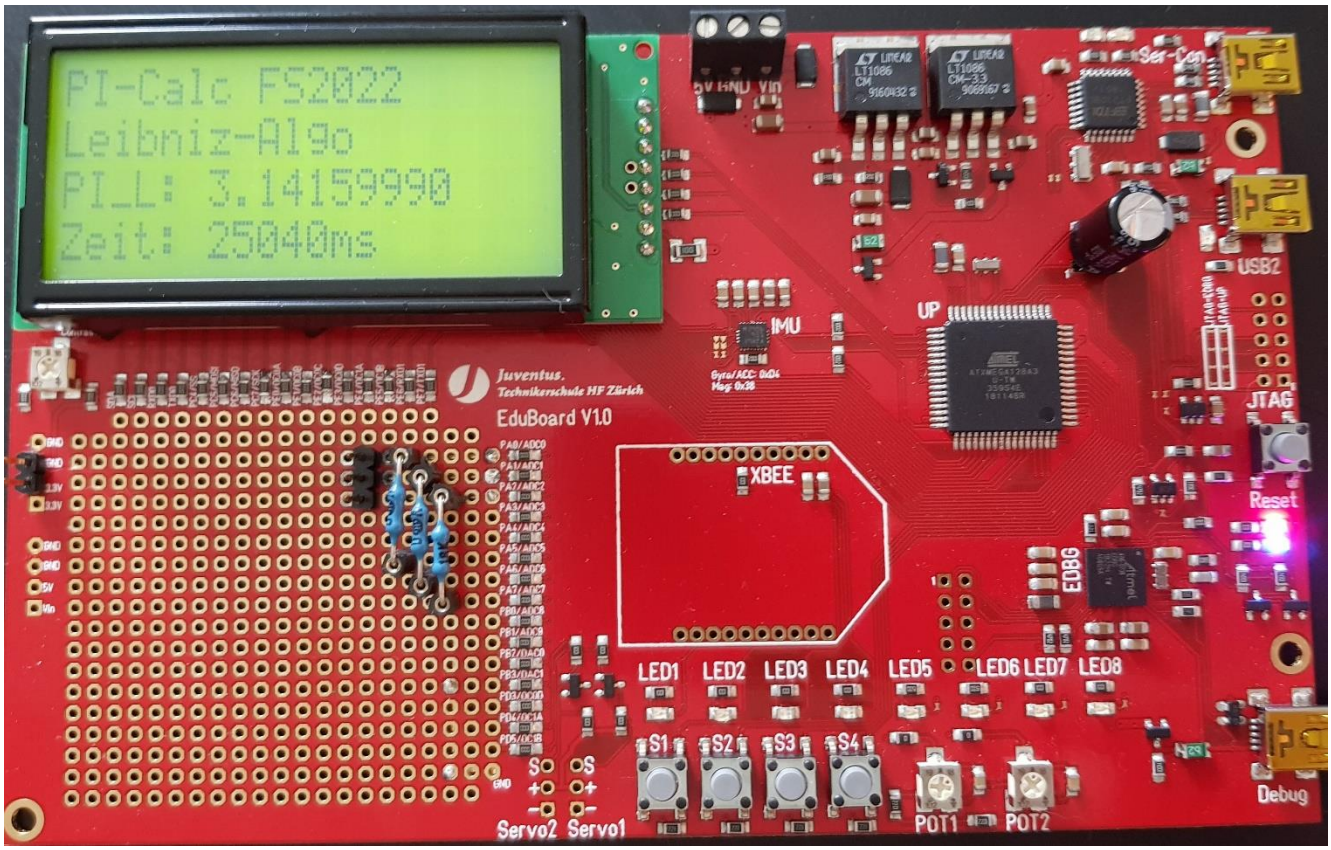
Juventus Technikerschule HF
www.technikerschule.ch

Pi-Calculator



Lisa-Marie Gent
Embedded Systems

Inhaltsverzeichnis:

Aufgabenstellung

Juventus
Technikerschule HF

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.)

1 Algorithmus

1.1 Leibniz-Reihe

Als ersten Algorithmus wurde uns die Leibniz-Reihe vorgeschlagen. Mit diesem Algorithmus erhält man π/4. Damit man Pi berechnen kann, muss das Gesamtergebnis also noch mit 4 multipliziert werden. Die Leibniz-Reihe wird wie folgt beschrieben:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

Abbildung 1: Leibniz-Reihe

Wie alle Reihen geht auch die Leibniz-Reihe unendlich weiter und nähert sich Pi nur an, weil Pi als irrationale Zahl nicht abschliessend berechnet werden kann. Es ist zu erkennen, dass die Addition und Subtraktion der Brüche sich abwechseln und der Nenner immer um zwei erhöht wird, wobei der Zähler bei 1 bleibt. Diese Art der Berechnung ist auf einen Sonderfall der Arcustangens Reihe zurück zu führen, mittlerweile gibt es aber deutlich bessere Varianten der Arcustangens Reihe zur Berechnung von Pi.

1.2 Nilakantha-Reihe

Für den zweiten, frei wählbaren Algorithmus habe ich mich ebenfalls für eine Zahlenreihe entschieden, für die Nilakantha-Reihe. Es gibt zwei unterschiedliche Varianten von dieser Berechnungsform, welche in Abbildung 2 und Abbildung 3 zu erkennen sind. Weil es für mich von der Umsetzung her einfacher war, habe ich mich für die zweite Variante entschieden.

$$\pi = 3 + \frac{4}{2 \cdot 3 \cdot 4} - \frac{4}{4 \cdot 5 \cdot 6} + \frac{4}{6 \cdot 7 \cdot 8} - \frac{4}{8 \cdot 9 \cdot 10} + \cdots$$

Abbildung 2: Nilakantha-Reihe 1

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

Abbildung 3: Nilakantha-Reihe 2

Dabei ist zu erkennen, dass die 3 vor dem Komma bereits vorgegeben ist und durch die Brüche die Nachkommastellen berechnet werden. Im Gegensatz zur Leibniz-Reihe muss zur Errechnung von Pi nicht mehr mit einer Zahl multipliziert werden.

Auch bei der Nilakantha-Reihe ist eine Regelmässigkeit festzustellen. Wie bei der Leibniz-Reihe wechseln sich Addition und Subtraktion ab. Der jeweilige Bruch ist aber etwas komplizierter. Im Zähler bleibt die 4 und im Nenner wir ebenfalls immer um zwei erhöht, allerdings wird der Nenner erst ^3 gerechnet und anschliessen nochmal mit der jeweiligen Nenner-Zahl subtrahiert.

Diese Art der Berechnung von Pi ist zwar deutlich schneller als die Berechnung mit der Leibniz-Reihe, aber sie gehört dennoch zu den eher langsamen Berechnungen. Eine noch deutlich schnellere Berechnungsart für Pi ist die Berechnung nach Ramanujan, wobei sich bei jedem Durchgang 18 weitere richtige Nachkommastellen berechnet lassen.

2 Tasks

Damit das Programm den Anforderungen entsprechend funktioniert, werden mehrere Tasks verwendet. Für mein Programm habe ich drei Tasks erstellt.

```
xTaskCreate( controllerTask, (const char *) "control_tsk", configMINIMAL_STACK_SIZE+150, NULL, 3, NULL);
xTaskCreate( leibniz_task, (const char *) "leibniz_tsk", configMINIMAL_STACK_SIZE+150, NULL, 1, NULL);
xTaskCreate( Nilakantha_task, (const char *) "Nilakantha_tsk", configMINIMAL_STACK_SIZE+150, NULL, 1, NULL);
```

Abbildung 4: Tasks

2.1 Leibniz-Task

Im Leibniz-Task wird Pi nach der Leibniz-Reihe berechnet.

```
void leibniz_task(void* pvParameters){
    pi4 = 1;
    uint32_t zaehler = 3;
    pi_nachkomma = 3.14160;
 for (;;) {
    if((xEventGroupGetBits(EventGroupPiCalc)&ResetTask)){
        pi4 = 0;
        uint32_t zaehler = 3;
        pi_l = 0;
    }
    if((xEventGroupGetBits(EventGroupPiCalc)&StartTask)){
            pi4 = pi4-(1.0/zaehler);
            zaehler += 2;
            pi4 = pi4 +(1.0/zaehler);
            zaehler += 2;
            pi_l = pi4*4;
        if (pi_l < pi_nachkomma){
            Zeit_End = xTaskGetTickCount();
            Zeit_Difference = Zeit_End - Zeit_Start;
            }
        if((xEventGroupGetBits(EventGroupPiCalc) & UebertragungsReset) == UebertragungsReset) {
            xEventGroupClearBits(EventGroupPiCalc, UebertragungsReset);
            pi_x = 0;
            }
        EventBits_t bits = xEventGroupGetBits(EventGroupPiCalc);
        if((bits & Datensperren) != 0) {
                xEventGroupSetBits(EventGroupPiCalc, daten_bereit);
                xEventGroupWaitBits(EventGroupPiCalc, daten_cleared, pdTRUE, pdFALSE, portMAX_DELAY);
            }
         datenuebertragung = pi_x;
        }
    }
 }
```

Abbildung 5: Leibniz-Task

Dazu werden zuerst die Werte für pi4 und zaehler definiert. In der Forever-Schleife des Tasks wird dann eine Abfrage der Buttons bzw. der EventBits gemacht. Sobald die Taste für den Reset gedrückt wird, werden alle Werte wieder zurückgesetzt, damit die Berechnung bei erneutem Starten von vorne anfängt. Wird die Taste für den Start gedrückt, so beginnt die Berechnung von Pi nach dem Muster der Leibniz-Reihe. Das heisst, dass die Addition und Subtraktion sich abwechseln und der zaehler bei jedem Wechsel um zwei erhöht wird. In dem Berechnungs-Task ist auch eine Abfrage zur Zeit-Messung. Die Abfrage startet, sobald die Taste zum Starten gedrückt wird und endet, sobald Pi kleiner als 3.14160 ist. Weil hier zwei Floats verglichen werden sollen, habe ich mich bewusst für ein „kleiner als" endschieden, weil die zwei Floats mit sehr grosser Wahrscheinlichkeit nie genau gleich sein werden. Somit werden die Ticks gemessen, wie lange es vom Start bis zum Erreichen der fünften Nachkommastelle von Pi dauert. Ausserdem wird die Datenübertragung mithilfe von Event Groups überwacht. Die Variable datenuebertragung ist dabei die P-Ressource und die Event Group die A-Ressource, weil die Event Group aktiv die Variable mit den Daten schützt. Der Leibniz-Task hat am Ende des Tasks kein vTaskDelay, weil der Task durchgehend ausgeführt werden soll.

2.2 Nilakantha -Task

Der zweite Task ist die Berechnung von Pi nach der Nilakantha-Reihe.

```
void Nilakantha_task(void* pvParameters){
    piN = 3;
    uint32_t zaehler_s = 3;
    pi_nachkomma = 3.14158;
 for (;;) {
     if((xEventGroupGetBits(EventGroupPiCalc)&ResetTask)){
        piN = 0;
        uint32_t zaehler_s = 3;
        pi_n = 0;
        }
     if((xEventGroupGetBits(EventGroupPiCalc)&StartTask)){
         piN = piN+(4.0/(pow(zaehler_s,3)-zaehler_s));
         zaehler_s += 2;
         piN = piN-(4.0/(pow(zaehler_s,3)-zaehler_s));
         zaehler_s += 2;
         pi_n = piN;
         if (pi_l > pi_nachkomma){
            Zeit_End_n = xTaskGetTickCount();
            Zeit_Difference_n = Zeit_End_n - Zeit_Start_n;
            }
            if((xEventGroupGetBits(EventGroupPiCalc) & UebertragungsReset) == UebertragungsReset) {
                xEventGroupClearBits(EventGroupPiCalc, UebertragungsReset);
                pi_x = 0;
            }
            EventBits_t bits = xEventGroupGetBits(EventGroupPiCalc);
            if((bits & Datensperren) != 0) {
                xEventGroupSetBits(EventGroupPiCalc, daten_bereit);
                xEventGroupWaitBits(EventGroupPiCalc, daten_cleared, pdTRUE, pdFALSE, portMAX_DELAY);
            }
            datenuebertragung = pi_x;
        }
     }
}
```

Abbildung 6: NikiLauda-Task

Die Vorgehensweise beim Nilakantha-Task ist sehr ähnlich zum Leibniz-Task. Zuerst werden die Werte für piN sowie zaehler_s definiert. Anschliessend wird in der Forever-Schleife ebenfalls eine Abfrage der EventBits durchgeführt. Wenn die Taste für den Reset gedrückt wird, werden alle Werte wieder auf ihren ursprünglichen Wert zurückgesetzt, damit bei einem erneuten Starten des Tasks die Berechnung von vorne beginnt. Wird die Taste für den Start gedrückt, wird Pi nach der Nilakantha-Reihe berechnet. Ähnlich wie bei dem Leibniz-Task wechseln sich die Subtraktion und die Addition ab und der zaehler_s wird bei jedem Wechsel um zwei erhöht, allerdings ist in der Berechnung noch ein Exponent vorhanden, welcher durch die Funktion „pow" berechnet wird. In diesem Task gibt es ebenfalls eine Zeit-Messung, die startet, sobald die Start-Taste gedrückt wird und endet, wenn der berechnete Wert von Pi grösser als 3.14158 ist. Auch hier wird die Zeit mithilfe der Funktion xTaskGetTickCount durchgeführt und die Subtraktion der beiden Tick-Werte ergibt die Zeit in ms.

Um die Datenübertragung zu schützen gibt es in diesem Task ebenfalls eine Absicherung durch die A-Ressource mithilfe einer Event Group. Auch hier ist die P-Ressource die Variable datenuebertragung. Durch die Sicherung der Datenübertragung wird wie im Leibniz-Task verhindert, dass die Daten gleichzeitig berechnet und gelesen werden und es so zu fehlerhaften Daten kommen könnte. Auch der Nilakantha -Task hat am Ende des Tasks kein vTaskDelay, weil der Task ebenfalls durchgehend ausgeführt werden soll.

2.3 Controller –Task

Der dritte Task in meinem Programm ist der Controller-Task. Hier passiert der wichtigste Teil des Programmes, weshalb die Priorität von diesem Task im Vergleich zu den anderen beiden auch bei 3 und nicht bei 1 ist.

```
void controllerTask(void* pvParameters) {
    char pistring[12];
    char pistring1[12];
    char pistring2[12];
    uint32_t fivehundertmillisecondscounter = 0;
    initButtons();
    for(;;) {
        updateButtons();
        if (fivehundertmillisecondscounter == 0){
            xEventGroupSetBits(EventGroupPiCalc, Datensperren);
            xEventGroupWaitBits(EventGroupPiCalc, daten_bereit, pdTRUE, pdFALSE, portMAX_DELAY);
            pi_x = datenuebertragung;
            xEventGroupClearBits(EventGroupPiCalc, Datensperren);
            xEventGroupSetBits(EventGroupPiCalc, daten_cleared);
            if((xEventGroupGetBits(EventGroupPiCalc)&WechsleLeibniz)){
                vDisplayClear();
                vDisplayWriteStringAtPos(0,0,"PI-Calc FS2022");
                vDisplayWriteStringAtPos(1,0,"Leibniz-Algo");
                sprintf(&pistring[0], "PI_L: %.8f", pi_l);
                vDisplayWriteStringAtPos(2,0, "%s", pistring);
                sprintf(&pistring1[0], "Zeit: %lu ms", Zeit_Difference);
                vDisplayWriteStringAtPos(3,0, "%s", pistring1);
                } else {
                    vDisplayClear();
                    vDisplayWriteStringAtPos(0,0,"PI-Calc FS2022");
                    vDisplayWriteStringAtPos(1,0,"Nilakantha-Algo");
                    sprintf(&pistring[0], "PI_N: %.8f", pi_n);
                    vDisplayWriteStringAtPos(2,0, "%s", pistring);
                    sprintf(&pistring2[0], "Zeit: %lu ms", Zeit_Difference_n);
                    vDisplayWriteStringAtPos(3,0, "%s", pistring2);
                }
                fivehundertmillisecondscounter = 50;
             } else {
                fivehundertmillisecondscounter --;
            }
```

Abbildung 7: Controller-Task 1

Im ersten Teil des Controller-Tasks sind die Ausgabe über das Display, welches mithilfe eines Counters und eines Delays alle 500ms aktualisiert wird, sowie der Gegenpart zur Datenübertragung. Hier werden die Daten auf dem Display angezeigt, aber nur dann, wenn die Berechnung unterbrochen ist und es zu keiner falschen Datenübertragung kommen kann.

Auf dem Display wird je nach Task-Auswahl die Berechnung von Pi nach Leibniz oder nach Nilakantha angezeigt. Ausserdem wird auf dem Display auch die Zeit angezeigt, wie lange es bei dem jeweiligen Berechnungs-Task dauert, bis die fünfte Nachkommastelle von Pi berechnet ist. Eigentlich werden mit der Funktion xTaskGetTickCount die Ticks seit Anfang des Schedulers gezählt. Aber durch die Subtraktion des Start-Wertes von End-Wert wird die Differenz der Ticks gezählt. Weil die TickTime in unserem Fall bei 1ms liegt, wird die Zeit bzw. die Ticks auf dem Display in ms angezeigt.

Im zweiten Teil des Controller-Tasks ist die Button-Abfrage.

```
    if(getButtonPress(BUTTON1) == SHORT_PRESSED) {
        xEventGroupSetBits(EventGroupPiCalc, StartTask);
        xEventGroupClearBits(EventGroupPiCalc, ResetTask);
        xEventGroupClearBits(EventGroupPiCalc, UebertragungsReset);
        Zeit_Start = xTaskGetTickCount();
        Zeit_Start_n = xTaskGetTickCount();
    }
    if(getButtonPress(BUTTON2) == SHORT_PRESSED) {
        xEventGroupSetBits(EventGroupPiCalc, StopTask);
        xEventGroupClearBits(EventGroupPiCalc, StartTask);
    }

    if(getButtonPress(BUTTON3) == SHORT_PRESSED) {
        xEventGroupSetBits(EventGroupPiCalc, ResetTask);
        xEventGroupClearBits(EventGroupPiCalc, StopTask);
        xEventGroupClearBits(EventGroupPiCalc, StartTask);
        xEventGroupClearBits(EventGroupPiCalc, WechsleLeibniz);
        xEventGroupClearBits(EventGroupPiCalc, WechsleNilakantha);
    }

    if(getButtonPress(BUTTON4) == SHORT_PRESSED) {
        uint8_t algostatus = (xEventGroupGetBits(EventGroupPiCalc) & 0x0018 ) >> 3;
        if(algostatus == 0x01) {
            xEventGroupClearBits(EventGroupPiCalc, WechsleLeibniz);
            xEventGroupSetBits(EventGroupPiCalc, WechsleNilakantha);
        } else {
            xEventGroupClearBits(EventGroupPiCalc, WechsleNilakantha);
            xEventGroupSetBits(EventGroupPiCalc, WechsleLeibniz);
        }
    }
    if(getButtonPress(BUTTON1) == LONG_PRESSED) {

    }
    if(getButtonPress(BUTTON2) == LONG_PRESSED) {

    }
    if(getButtonPress(BUTTON3) == LONG_PRESSED) {

    }
    if(getButtonPress(BUTTON4) == LONG_PRESSED) {

    }
    vTaskDelay(10/portTICK_RATE_MS);
    }
}
```

Abbildung 8: Controller-Task 2

Für meinen Code habe ich nur die vier Tasten in kurzer Form verwendet. Somit stehen noch weitere Optionen offen, wenn man die lang gedrückten Tasten auch mit rein nehmen möchte.

Wird der Button 1 gedrückt, starten die Berechnungs-Tasks und der Reset wird zurückgesetzt. Ausserdem wird die Zeitmessung gestartet. Bei Taste zwei werden die Berechnungs-Tasks gestoppt und das Start-Bit wird zurückgesetzt. Taste drei ist für den Reset zuständig. Wird diese Taste gedrückt, wird das Reset-Bit gesetzt und alle anderen wieder zurückgesetzt. Mit der vierten Taste kann die Anzeige auf dem Display zwischen den beiden Berechnungs-Tasks hin und her wechseln.

Am Ende des Tasks ist ein Delay von 10 Ticks bzw. ms eingeführt, wodurch die Tasten alles 10ms abgefragt werden. Zusammen mit der Variablen fivehundertmillisecondscounter wird das Display somit alle 500ms aktualisiert.

3 Event Group

```
//Eventgroup erstellen
EventGroupHandle_t EventGroupPiCalc;
#define StartTask           1<<0
#define StopTask            1<<1
#define ResetTask           1<<2
#define WechsleLeibniz      1<<3
#define WechsleNilakantha   1<<4
#define UebertragungsReset  1<<5
#define Datensperren        1<<6
#define daten_bereit        1<<7
#define daten_cleared       1<<8
```

Abbildung 9: EventBits

Damit die einzelnen Tasks in der richtigen Reihenfolge abgerufen werden können, werden EventBits benutzt, welche in der Event Group definiert werden. Die Bits werden je nach Task oder Aktion gesetzt oder gelöscht. In den einzelnen Task gibt es Befehle, welche die EventBits setzen oder löschen. Wird eine Abfrage erfolgreich durchgeführt, wird durch die EventBits eine Befehlskette ausgelöst. War die Abfrage nicht erfolgreich, wird gewartet oder es wird keine Aktion ausgeführt.

Für meinen Code habe ich die folgenden neun EventBits verwendet:

- StartTask: Wird durch Button 1 gesetzt und ist im Leibniz- bzw. Nilakantha-Task dafür zuständig, dass die Tasks gestartet werden. Durch das Betätigen von Button 2 oder 3 wird das Bit wieder zurückgesetzt.
- StopTask: Wird durch die Betätigung von Button 2 gesetzt und stoppt die Berechnung des Leibniz- und Nilakantha-Tasks. Wird durch Button 3 wieder zurückgesetzt.
- ResetTask: Wird durch die Betätigung von Button 3 gesetzt und setzt die Berechnung des Leibniz- und Nilakantha-Tasks zurück. Wird durch Button 1 wieder zurückgesetzt.
- WechsleLeibniz: Wird durch Button 4 gesetzt oder gelöscht, je nachdem welcher von den Berechnungs-Tasks im Moment aktiv ist. Wird durch Button 3 wieder zurückgesetzt. Je nachdem ob dieses EventBit gesetzt oder gelöscht ist, werden die unterschiedlichen Berechnungs-Task auf dem Display angezeigt.
- WechsleNilakantha: Wird durch Button 4 gesetzt oder gelöscht, je nachdem welcher von den Berechnungs-Tasks im Moment aktiv ist. Wird durch Button 3 wieder zurückgesetzt.
- UebertragungsReset: Wird durch Betätigen von Button 1 gesetzt und wird benötigt, damit eine fehlerfreie Übertragung der berechneten Werte für Pi durchgeführt werden kann. Dadurch wird verhindert, dass die Werte gleichzeitig geschrieben und gelesen werden und es somit zu fehlerhaften Daten kommt.
- Datensperren: Durch dieses EventBit werden die Daten gesperrt, welche später ausgelesen werden sollen.
- daten_bereit: Wird im Berechnungs-Task gesetzt und im Controller-Task durch ein xEventGroupWait wieder zurückgesetzt. Die Daten werden erst ausgelesen, wenn das EventBit gesetzt ist.
- daten_cleared: Wird im Controller-Task gesetzt und im Berechnungs-Task wieder zurückgesetzt.

Mit den Funktionen xEventGroupSetBits werden die einzelnen Bits gesetzt und durch die Funktion xEventGroupClearBits werden sie zurückgesetzt. Durch die Funktion xEventGroupWaitBits können Abfragen realisiert werden, bei welchen nach der Ausführung das gesetzte Bit wieder zurückgesetzt wird. Die letzten vier EventBits in meinem Code dienen alle zur Sicherung der Datenübertragung.

4 Vergleich Zeit-Messung

In der Zeitmessung der unterschiedlichen Berechnungs-Task gibt es einen deutlichen Unterschied. Der Leibniz-Task benötigt für die Berechnung bis auf die fünfte Nachkommastelle ca. 21 Sekunden, während der Nilakantha-Task weniger als eine Sekunde benötigt. Bei meinem Code hat die Zeitmessung leider nicht richtig funktioniert. Im Code selbst war beim Debuggen zwar ersichtlich, dass die Zeile mit der Zeitmessung erreicht wird, aber das Ergebnis wird nicht auf dem Display angezeigt. Deswegen habe ich extern mit einer Stoppuhr die Zeit gemessen. Dadurch wird das Ergebnis sehr ungenau, aber der deutliche Unterschied zwischen den zwei Berechnungs-Task ist trotzdem zu erkennen.

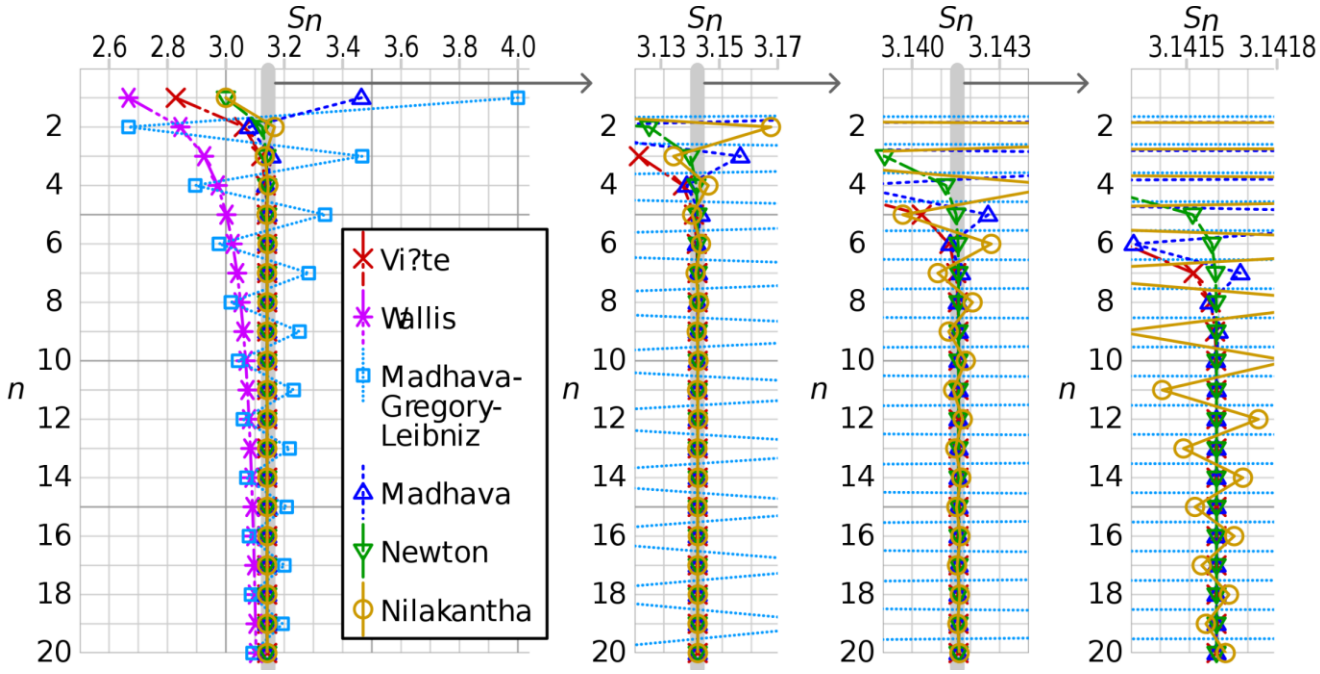


Abbildung 10: Vergleich Leibniz und Nilakantha

Wie in Abbildung 10 zu erkennen ist, näher sich die Nilakantha-Reihe deutlich schneller an Pi an als die Leibniz-Reihe. Bei der Leibniz-Reihe müssen ca. 1000000 Durchläufe durchgeführt werden, bis die fünfte Nachkommastelle berechnet ist. Bei der Nilakantha-Reihe müssen nur ca. 50 Durchläufe durchgeführt werden, damit die fünfte Nachkommastelle berechnet wird. Für unsere Aufgabenstellen mit der fünften Nachkommastelle bedeutet dies also, dass die Nilakantha-Reihe ca. 20000mal schneller ist als die Leibniz-Reihe. Dies liegt mitunter auch daran, dass bereits der erste berechnete Wert der Nilakantha-Reihe mit 3 deutlich näher an Pi ist als bei der Leibniz-Reihe mit 4.

5 Rückschluss zur Prozessorleistung

Obwohl die Leibniz-Reihe deutlich länger für die Berechnung von Pi benötigt, denke ich, dass die Nilakantha-Reihe deutlich mehr Prozessorleistung benötigt, als die Leibniz-Reihe. Bei der Leibniz-Reihe kann deutlich auf dem Display nachvollzogen werden, wie die Berechnung stattfindet. Es ist ein einfacher Algorithmus der sich nicht sonderlich schnell an Pi annähert. Wohingegen man bei der Nilakantha-Reihe bereits bei der Display-Anzeige merkt, dass die Berechnung schnell an ihre Grenzen kommt. Bereits in einem Bruchteil einer Sekunde ist Pi bis auf die fünfte Nachkommastelle berechnet und damit die Berechnung noch genauer durchgeführt werden kann, müsste in diesem Beispiel für pi_n ein float64 eingeführt werden.

Abbildungsverzeichnis

Quellen:

Abbildung 10: https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80

Ramanujan Genauigkeit: https://fichtnergasse.at/wp-content/uploads/2021/04/VWA_Pantscharowitsch.pdf Seite 30

Allgemein: https://www.freertos.org/a00106.html