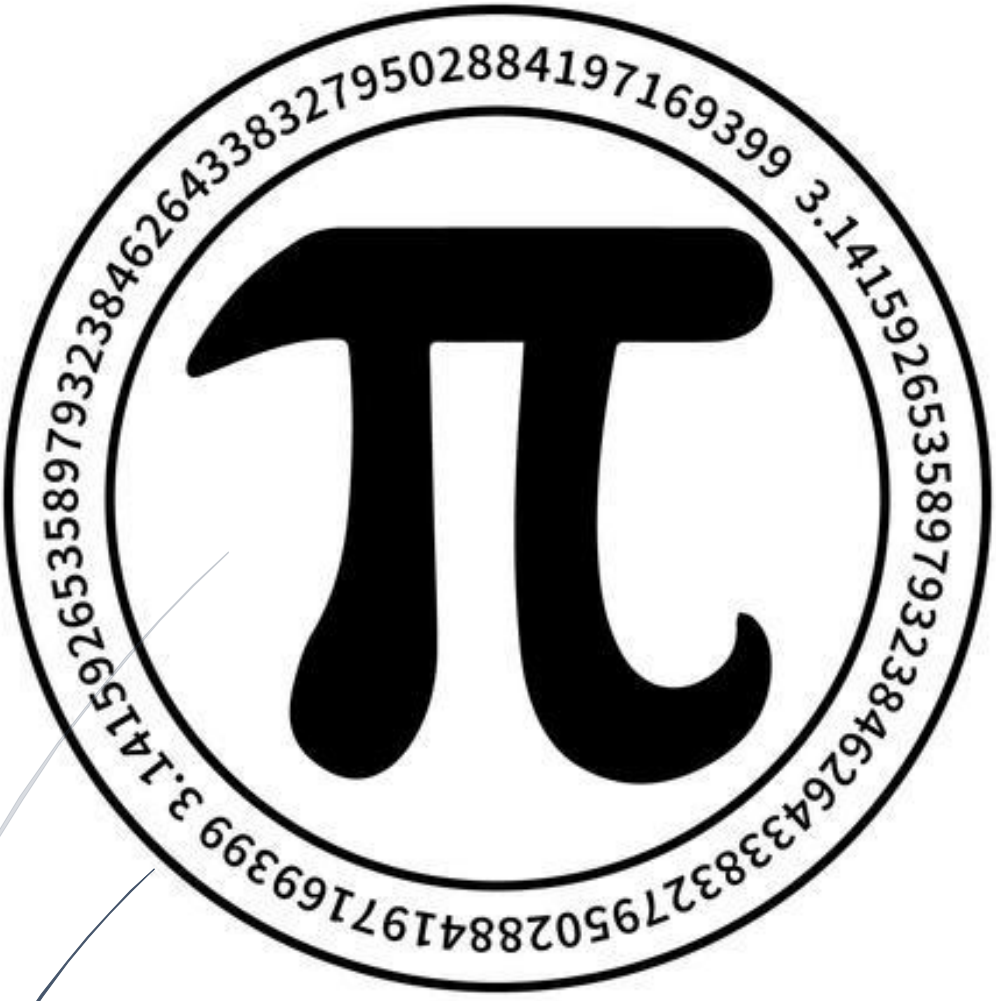
28.9.2020

Pi Calculator



Marco Widmer

JUVENTUS SCHULEN

Inhaltsverzeichnis

Aufgabenstellung

Die Aufgabe besteht daraus Pi zu berechnen. Dazu wurden im Laufe der Zeit mehrere Algorithmen gefunden welche eine Annäherung an Pi möglich machen. Es sollen Zwei verschieden Algorithmen ausgewählt werden. Diese sollen anschliessend miteinander verglichen werden. Der erste Algorithmus (Leibniz-Reihe) wurde jedoch vorgegeben.
Es soll möglich sein zwischen den beiden Verfahren auszuwählen. Dabei soll der aktuell berechnete Wert von Pi mit einer Update Rate von 500ms auf dem Display aktualisiert werden. Die Algorithmen sollen gestartet, gestoppt und zurückgesetzt werden können. Da das Ganze mithilfe von Free-RTOS erstellt wird, soll die Funktion der TaskNotification bzw. Event-Bits verwendet werden. Es soll beiläufig ein Timer mitlaufen, welcher die Zeit misst, welche die Tasks benötigen um Pi auf 5 Nachkommastellen genau zu berechnen.

Leibniz-Reihe

Der erste vorgegebene Algorithmus ist die Leibniz-Reihe. Dieser sieht folgendermassen aus.[1]

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} .... = \frac{\Pi}{4}$$

Im C Syntax könnte der Algorithmus in etwa so aussehen:

```
while (1){
	pi=pi-(float)(1.0/n);
	n+=2;
	pi=pi+(float)(1.0/n);
	n+=2;
	GlPi = (pi*4);
	}
```

Kelallur Nilakantha Somayaji-Reihe

Als Zweites entschied ich mich dazu den Alghorythmus des indischen Mathematiker Kelallur Nilakantha Somayaji zu verwenden.
Dieser sieht wie folgt aus. [2]

$$\Pi = 3 + \frac{4}{3^3 - 3} - \frac{4}{5^3 - 5} + \frac{4}{7^3 - 7} - \frac{4}{9^3 - 9} + ...$$

Im C Syntax könnte der Algorithmus in etwa so aussehen:

```
while (1){
	pi=pi+(float)(4.0/(pow(n,3)-n));
	n=n+2;
	pi=pi-(float)(4.0/(pow(n,3)-n));
	n=n+2;

	}
```

[1] https://de.wikipedia.org/wiki/Leibniz-Reihe

[2] https://3.14159265358979323846264338327950288419716939937510582097494459 2.eu/pi-berechnen-formeln-und-algorithmen/

Beschreibung der Tasks

ButtonTask

Der Button-Task wird dazu verwendet die Taster auf dem Edu-Board auszulesen. Es wird unterschieden ob die einzelnen Buttons lange oder kurz gedrückt werden. Da diese Funktion aus dem Beispielprogramm entnommen wurde, nahm ich die Funktion einfach wie sie ist und beschäftigte mich nicht vertieft mit ihrer Arbeitsweise.

UserInterface

Der UserInterface Task ist zum einen der Task indem die Displayausgabe abgehandelt wird und zum anderen der Steuere-Task. Ich habe die Eventbit's verwendet um die Zwei verschiedenen Algorithmen zu starten, stoppen und zurücksetzten. Um zwischen den Beiden Algorithmus Tasks zu wechseln verwendete ich die Hilfsvariablen (bool) `LeibinzRunning` und `SomayajiRunning`. Der Steuertask sendet dem Algorithmus-Task ein Stopp Befehl. Anschliessend wartet der Steuertask bis er die Bestätigung des Algorithmus Tasks erhält, dass dieser fertig gerechnet hat. Sobald diese Bestätigung erhalten wurde, kann auf die Globalen Variablen zugegriffen werden. Sobald die Displayausgabe abgehandelt wurde, gibt der Steuertask die Freigabe damit der Algorithmus Task fortgesetzt werden kann.

Algorithmus Tasks

Das zurücksetzen der Algorithmen wird in dem Algorithmus-Task abgehandelt. Dort wird in jedem Durchgang abgefragt ob das entsprechende Event-Bit gesetzt ist. Falls dies der Fall ist, werden die zum rechnen benötigten Variablen auf ihren Initials Zustand gesetzt. Ich habe darauf geachtet, dass der Algorithmus zu jedem Zeitpunkt zurückgesetzt werden kann. Es ist egal ob der Task läuft, pausiert oder fertig ist. Anschliessend werden noch die oben beschriebenen Algorithmen eingefügt. Bis auf die beiden Algorithmen sind die Tasks grösstenteils identisch.

HeartBeat

Im Heartbeat Task wird ein Counter erhöht sobald ein Zeitgesteuerter Interrupt ausgelöst wird. Die Millisekunden Variabel steht Global zur Verfügung.

EventBits/TaskNotification

Ich entschied mich dafür hauptsächlich die Funktion der Eventgroups zu benutzen. Indem ich die einzelnen Bits setzte (=1) oder wieder löschte (=0) konnte ich Zustände oder Informationen global zur Verfügung stellen.
Ich benötigte die TaskNotification lediglich um die Zeitmessung zu realisieren.
Ich definierte folgende Masken für die Eventbits:

```
#define STOPALGO         0x01
#define TASKSTOPPED      0x02
#define TASKRESUME       0x04
#define TASKDONE         0x08
#define STARTALGO        0x10
#define STOPALGOBUTTON   0x20
#define RESETALGO        0x40
#define CHANGEALGO       0x80
```

Dies hat den Vorteil das die einzelnen Bit's einen selbsterklärenden Namen bekommen. Dies macht es sehr viel einfacher den Code zu schreiben. Flüchtigkeits- / Schreibfehler kann so ebenfalls vorgebeugt werden. Falls im Verlauf des Projektes ein anderes Bit verwendet werden soll muss dies nur an einer Stelle geändert werden.

Ich benötigte hauptsächlich folgende Funktionen:
xEventGroupGetBits:
Diese Funktion erlaubt es die gesetzten Bits in abzufragen und in einer Zwischenvariabel zu speichern.

xEventGroupSetBits:
Mit dieser Funktion kann man einzelne Bits setzen.

xEventGroupClearBits:
Diese Funktion wird benötigt um Bits zu «löschen» (Wert auf 0 setzen).

xEventGroupWaitBits.:
Diese Funktion versetzt den Task, in welcher die Funktion aufgerufen wird, in den Zustand «Suspended» bis das angegebene Bit auf «1» gesetzt wird.

Dies ist eine grobe Funktionsbeschreibung. Genauere Infos können auf der FreeRTOS Seite entnommen werden.[3]

[3] https://www.freertos.org/

Zeitmessung

Die Zeitmessung wurde mit dem oben beschriebenen Task "Heartbeat" realisiert. Dieser misst die vergangenen Millisekunden. Diese Variabel wird nur erhöht falls der Berechnungs-Task läuft. Falls dieser gestoppt wird oder die Berechnung abgeschlossen wurde, wird die Erhöhung des Counters blockiert indem der Task *Suspended* wird. Falls die Berechnung zurückgesetzt wird oder zwischen den beiden Tasks gewechselt wird, wird der Counter zurückgesetzt.

Geschwindigkeitsvergleich

Um einen Geschwindigkeitsvergleich der beiden Methoden aufzustellen habe ich die Tasks jeweils 10-mal hintereinander laufen lassen und die benötigte Zeit, sowie die Anzahl der Rechenschritte notiert. Dabei entstand folgendes Ergebnis:

Leibniz		
Messung	**Steps**	**ms**
1	142317	13081
2	142317	12990
3	142317	13004
4	142317	13006
5	142318	12952
6	142317	13004
7	142318	13940
8	142318	12929
9	142317	13004
10	142318	12978
Ø	142317	13089

Somayaji		
Messung	**Steps**	**ms**
1	21	8
2	21	8
3	21	8
4	21	22
5	21	23
6	21	8
7	21	8
8	21	8
9	21	23
10	21	23
Ø	21	13.9

Um Pi auf Fünf Nachkommastellen mithilfe der Leibniz-Reihe zu berechnen, benötigte ich durchschnittlich **142'317 Rechenschritte** und **13’089 Millisekunden**.
Für die gleiche Aufgabe, jedoch mit der Somayaji-Reihe berechnet, benötigte ich durchschnittlich **21 Rechenschritte** und **13.9 Millisekunden**.
Obwohl die Zeitunterschiede von Mal zu Mal etwas zu gross sind um sie als einfache Messungenauigkeit abzustempeln, konnte ich den Fehler leider nicht eruieren. Aus Zeitgründen habe ich mich dazu entschieden die Werte über 10 Messungen zu mitteln, um trotzdem annehmbare Daten für einen Aussage zu machen.

Rückschluss bezüglich der gemessenen Rechenleistung

Man kann einen Rückschluss auf die benötigte Rechenleistung des Algorithmus machen, indem man berechnet wie viele Rechenschritt der Mikroprozessor pro Millisekunde abarbeiten kann.

Leibniz	
Step/ms	10.8732

Somayaji	
Step/ms	1.5108

Mann kann deutlich erkennen das der Rechentask mit dem Algorithmus von Somayaji mehr Rechenleistung erfordert. Mit der Leibnizreihe kann in der gleichen Zeit etwa Sieben Mal öfters gerechnet werden. Dies war eindeutig zu erwarten, da bei der Somayajireihe jeweils hoch 3 (*pow(n,3)*) gerechnet werden muss. Dies ist bekannterweise eine sehr zeitaufwendige Operation. Trotzdem ist man mit diesem Alghgoritmus beinahe 950mal schneller. Dies liegt daran das der Alghorithmus die Annäherung an Pi auf 5 Stellen mit sehr viel weniger Rechenschritten bewerkstelligt.

Um einen Rückschluss auf die Rechenleistung des Prozessors zu machen habe ich ermittelt wie viele Taktzyklen der Prozessor durchläuft, um die Aufgabe abzuschliessen.

Leibniz	
Tackt Zyklen:	418'841'600
Zyklen/ms:	32'000
Zyklen/Step:	2'943

Somayaji	
Tackt Zyklen:	444'800
Zyklen/ms:	32'000
Zyklen/Step:	21'181

Da der Leibniz Tasks mehr Zeit benötigte, benötigt er natürlich auch mehr Tackt-Zyklen. Auch die Zyklen-pro-Millisekunde Berechnung ergab keine Überraschung, da die 32´000 der Taktfrequenz von 32MHz entspricht. Interessant wird es bei der letzten Berechnung. Es wurde festgestellt das beinahe Dreitausend Tackt Zyklen benötigt werden, um einen Rechenschritt der Leibniz Reihe abzuhandeln und etwa Einundzwanzigtausend Tackt Zyklen bei der Somayaji Reihe.

GIT

Für die Versionierung und Sicherung des Projektes verwendete ich *GitHub*. *GitHub* funktioniert mithilfe von einem *Repository*. Dies ist eine Kopie von einem lokal gespeichertem Ordner. Sobald lokale Änderungen vorgenommen werden wird dies erkannt. Anschliessend kann man diese *committen*. Das bedeutet es werden alle Änderungen zwischengespeichert. Bei diesem Schritt kann/muss einen Kommentar beifügt werden, auf dem ersichtlich sein sollte was sich mit diesem *committ* verändert hat (z.B. HMI integriert oder Datenbank Anbindung implementiert).

Zu einem späteren Zeitpunkt kann man das ganze *pushen*. Bei diesem Schritt werden alle lokalen *committs* mit dem *Repository* auf dem Server zusammengeführt (*merge*).

Solange man *Git* alleine benützt ist damit die Funktionalität mehr oder weniger ausgeschöpft. Falls aber mehrerer Projektmitglieder am selben File arbeiten wollen, bietet *Git* noch viele andere Möglichkeiten. Z.B kann das *Repository* nochmal kopiert werde. Der eine Mitarbeiter arbeitet an der einen Kopie und der andere an der anderen. Sobald einer der beiden ein Feature abgeschlossen hat, kann er diese mit dem Ursprünglichen Code Zusammenführen. Auch kann man später genau nachvollziehen welches Projektmitglied welche Zeile Code geschrieben hat. Es ist nicht so, dass man diese Funktionen alleine nicht nutzen kann, sie haben aber meistens keinen grossen nutzen.