


CALCULATING PI MIT ATXMEGA128A3U

Praktische Vertiefungsarbeit – Youssef Horriche
Studiengang: HF Elektrotechnik / Elektronik
Juventus Technikerschule HF Zürich
Klasse 1902 - 5. Semester
Fach: Embedded Systems
Dozent: Martin Burger
Abgabetermin: 30. Mär. 2021



Inhaltsverzeichnis

1.	<u>EINLEITUNG</u>	<u>2</u>
2.	<u>ERKLÄRUNG DES ALGORITHMUS.....</u>	<u>3</u>
2.1	LEIBNIZ REIHE ALGORITHMUS	3
2.2	MONTE-CARLO ALGORITHMUS	3
2.3	BEDIENUNGSANLEITUNG	4
3.	<u>BESCHREIBUNG DER TASKS</u>	<u>4</u>
3.1	VINTERFACE TASK	4
3.2	VBUTTON TASK	5
3.3	VLEIBNIZ TASK.....	5
3.4	VMONTECARLO TASK	6
4.	<u>EVENTGROUPS / BITS.....</u>	<u>7</u>
5.	<u>ZEITMESSUNG.....</u>	<u>7</u>
6.	<u>QUELLENVERZEICHNIS</u>	<u>8</u>
7.	<u>ABBILDUNGSVERZEICHNIS</u>	<u>8</u>

1. Einleitung

Im Rahmen des Unterrichts Embedded Systems führen wir eine praktische Vertiefungsarbeit (im Folgenden mit PVA abgekürzt) durch.

Aufgabe war es eine Berechnung von PI mit zwei verschiedenen Algorithmen zu realisieren (1. Leibniz-Reihen / 2. Algorithmus aus dem Internet), auf dem EduBoard v1.0 zu zeigen und die verschiedenen Elemente / Komponenten und Funktionen mit einer Dokumentation darzustellen.

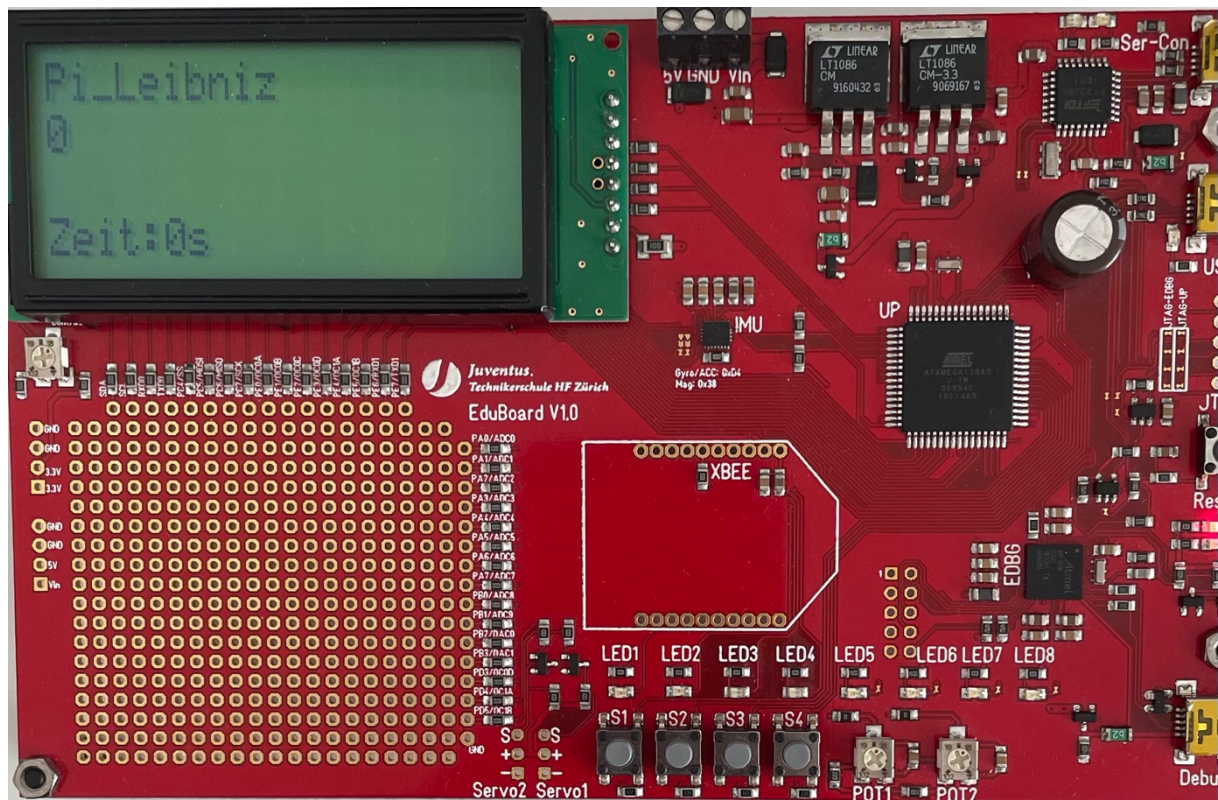


Abbildung 1 – EduBoard

2. Erklärung des Algorithmus

2.1 Leibniz Reihe Algorithmus

Mit die Leibniz-Reihe lässt sich die Kreiszahl Pi einfach und übersichtlich berechnen.

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

Den Algorithmus konvergiert immer weiter zu Pi-Viertel, und zwar indem abwechselungsweise ein Teil weg und wieder dazu gerechnet wird. Diese Teile werden immer kleiner und so nähert sich das Ergebnis immer weiter an Pi-Viertel. Der Nachteil dieses Algorithmus sind die vielen Iterationen welche nötig sind, um eine tiefe Genauigkeit von Pi zu erhalten.

2.2 Monte-Carlo Algorithmus

Mit dem Namen Monte-Carlo bzw. Monte-Carlo-Simulation verbindet man die Lösung von mathematischen Problemstellungen mit Hilfe von Zufallszahlen. für die Berechnung der Kreisfläche oder auch der Zahl Pi beginnen wir mit einem Quadrat der Fläche 1. Dieses Quadrat hat die Kantenlänge 1. In dieses Quadrat zeichnen wir einen Viertelkreisbogen mit dem Radius 1 ein.

Wir erzeugen mit einem Zufallsgenerator beliebige Punkte innerhalb des Quadrats. Das bedeutet,

dass die Punkte innerhalb des Quadrats jeweils x- und y-Werte im Bereich von 0 bis 1 haben. Bei mehreren tausenden solcher Punkte füllt sich das Quadrat mehr oder weniger gleichmäßig mit diesen Punkten.

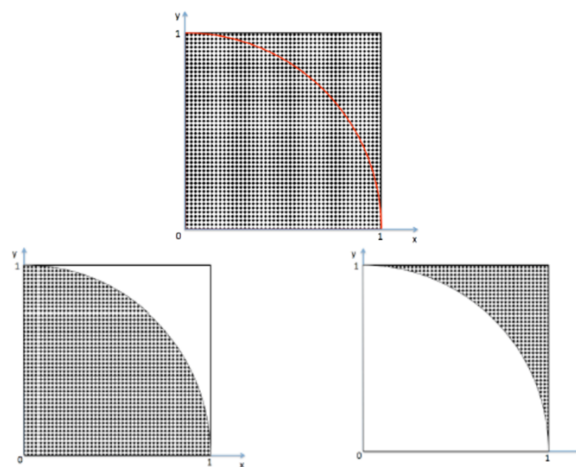


Abbildung 2 – Monte-Carlo Methode

2.3 Bedienungsanleitung

Button	Funktion	Beschreibung
S1 LONG	Start	Start der Leibniz Berechnung
S1 SHORT	Stop	Stop der Leibniz Berechnung
S2 LONG	Start	Start der Monte-Carlo Berechnung
S2 SHORT	Stop	Stop der Monte-Carlo Berechnung
S4	Programm Resetieren	Alles wird zurückgesetzt

3. Beschreibung der Tasks

3.1 vInterfaceTask

In diesem Task wird die Displayausgabe und Buttonhandling beschrieben.

Ebenfalls in diesem Task, ist der Vergleich, ob die geforderte Genauigkeit erreicht ist. Falls dies der Fall ist wird die Zeit angehalten.

```
void vInterfaceTask(void *pvParameters) {
    (void) pvParameters;
    float64_t floatpii=0;
    float64_t float0=0;
    float64_t float00001=0;
    int8_t hilf1=0;
    int8_t hilf2=0;
    int8_t hilf3=0;
    int8_t hilf4=0;

    for(;;)
    {
        updateButtons();

        floatpii=f_sd(3.141595);
        float0=f_sd(0);
        float00001=f_sd(0.000005);
        hilf1=f_compare(f_sub(floatpii, Pi),float0);
        hilf2=f_compare(float00001,f_sub(floatpii, Pi));
        hilf3=f_compare(f_sub(Pi,floatpii),float0);
        hilf4=f_compare(float00001,f_sub(Pi,floatpii));
        if((hilf3>0&&hilf4>0)||((hilf1>0&&hilf2>0)){
            zeitlauft=0;
        }
        char* tempResultString = f_to_string(Pi, 16, 16);
        sprintf(Pistring, "%s", tempResultString);
        vDisplayClear();
        vDisplayWriteStringAtPos(0,0,"Pi_Leibniz");
        vDisplayWriteStringAtPos(1,0,"%s", Pistring);
        vDisplayWriteStringAtPos(3,0,"Zeit:%ds", zeit);

        vTaskDelay(pdMS_TO_TICKS(500));
        //vTaskDelay(500 / portTICK_RATE_MS);
    }
}
```

Abbildung 3 – vInterfaceTask

3.2 vButtonTask

Hier werden alle Knöpfe abgefragt und die jeweiligen EventBits gesetzt oder gelöscht.

```
void vButtonTask(void *pvParameters) {
    (void) pvParameters;

    initButtons();

    while (1) {
        updateButtons();

        if(getButtonPress(BUTTON1) == LONG_PRESSED) {
            zeitlaufft=1;
            xEventGroupSetBits(xLeibnizeventgroup, StartStop);
        }
        if(getButtonPress(BUTTON1) == SHORT_PRESSED) {
            zeitlaufft=0;
            xEventGroupClearBits(xLeibnizeventgroup, StartStop);
        }
        if(getButtonPress(BUTTON2) == LONG_PRESSED) {
            zeitlaufft=1;
            xEventGroupSetBits(xMonteCarloeventgroup, StartStop);
        }
        if(getButtonPress(BUTTON2) == SHORT_PRESSED) {
            zeitlaufft=0;
            xEventGroupClearBits(xMonteCarloeventgroup, StartStop);
        }

        if(getButtonPress(BUTTON4) == SHORT_PRESSED) {
            xEventGroupSetBits(xLeibnizeventgroup, Reset);
        }
        if(getButtonPress(BUTTON4) == SHORT_PRESSED) {
            xEventGroupSetBits(xMonteCarloeventgroup, Reset);
        }

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

Abbildung 4 – vButtonTask

3.3 vLeibnizTask

In diesem Task ist die Berechnung von Pi nach der Leibniz Formel zu sehen.

Da Float64 hier benutzt wurde, sind die Operationen sehr umständlich und langsam. Das Setzen und Löschen der Status EventBits, ist ebenfalls hier beschrieben. Um bessere Übersicht zu erhalten habe ich diverse Zwischenspeicher (float64_t PiFloatx=0) Variablen erstellt.

```
void vLeibnizTask(void *pvParameters) {
    (void) pvParameters;

    float64_t PiFloat1=0;
    float64_t PiFloat2=0;
    float64_t PiFloat3=0;
    float64_t PiFloat4=0;
    float64_t PiFloat5=0;
    float64_t PiFloat6=0;
    float64_t PiFloat7=0;
    for(;;)
    {
        if((xEventGroupGetBits(xLeibnizeventgroup)&1)==1)
        {
            if(counter%2)
            {
                PiFloat1=f_mult(PiFloat2, f_sd(2));
                PiFloat3=f_add(PiFloat1, f_sd(1));
                PiFloat4=f_div(f_sd(4.0), PiFloat3);
                Pi=f_sub(Pi, PiFloat4);
                PiFloat2=f_add(PiFloat2, f_sd(1));
                counter++;
            }
            else
            {
                PiFloat5=f_mult(PiFloat2, f_sd(2));
                PiFloat6=f_add(PiFloat5, f_sd(1));
                PiFloat7=f_div(f_sd(4.0), PiFloat6);
                Pi=f_add(Pi, PiFloat7);
                PiFloat2=f_add(PiFloat2, f_sd(1));
                counter++;
            }
            xEventGroupSetBits(xLeibnizeventgroup, Status);
        }
        else
        {
            xEventGroupClearBits(xLeibnizeventgroup, Status);
        }
        if((xEventGroupGetBits(xLeibnizeventgroup)&2)==2)
    }
}
```

Abbildung 5 – vLeibnizTask

3.4 vMonteCarloTask

Hier wird Pi nach der Monte-Carlo Methode berechnet. EventGroups werden auch eingesetzt Und dienen der Steuerung des Task.

```
void vMonteCarloTask(void *pvParameters) {
    (void) pvParameters;

    srand( SEED );
    int i, count, n;
    double x,y,z,pi;
    double qn(int n, int max)
    counter = 0;

    while(1)
    {
        if((xEventGroupGetBits(xMonteCarloeventgroup)&1)==1)
        counter = 0;
        for(i = 0; i < n; ++i)
        {
            x = (double)rand() / RAND_MAX;
            y = (double)rand() / RAND_MAX;
            z = x * x + y * y;
            if( z <= 1 ) count++;
        }

        pi = (double) count / n * 4;
        double qn(int n, int max)
        {
            if (n==0)
            return 1.+1./qn(1,max);
            else if (n < max)
            return 2.+(2.*n+1.)*(2.*n+1.)/qn(n+1,max);
            else
            return 2.+2.*sqrt(n*n+n+1);
            double pi;
            {
                if (n==0) break;
                pi = 4.0/qn(0,n);
            }
            xEventGroupSetBits(xMonteCarloeventgroup, Status);
        }
        else
        {
            xEventGroupClearBits(xMonteCarloeventgroup, Status);
        }
    }
}
```

Abbildung 6 – vMonteCarloTask

4. EventGroups / Bits

EventGroup	Beschreibung
xInterfaceeventgoup	Dient der kommunikation mit der InterfaceTask
xButtoneventgroup	Dient der Kommunikation mit der ButtonTask
xLeibnizeventgroup	Dient der Kommunikation mit der LeibnisTask
xMonteCarloeventgroup	Dient der Kommunikation mit der MonteCarloTask

EventBit	Beschreibung
StartStop	Starten und Stoppen der Berechnung
Status	repräsentiert den aktuellen Status der Berechnung
Reset	Alles wird gelöscht

5. Zeitmessung

Die Berechnung mit dem Leibniz-Reihe ist langsam. Es werden ca. 158 Sekunden benötigt, um Pi auf fünf Stellen nach dem Komma zu berechnen. Anders gesagt es braucht 136120 Durchläufe mit der Leibnetzreihe, um die geforderte Genauigkeit zu erreichen. Mit anderen Worten, es werden 860 Durchläufe pro Sekunden berechnet. Damit braucht ein Durchlauf 38000 Taktzyklen, was nicht gerade effizient ist.

6. Quellenverzeichnis

FreeRTOS10_Template_V1 der Juventus Technikerschule (als Projektvorlage)

<https://www.spektrum.de/lexikon/mathematik/arcustangensreihen-fuer/260>

<https://www.freertos.org>

<https://github.com>

7. Abbildungsverzeichnis

Abbildung1: EduBoard v1.0

Abbildung2: Monte-Carlo Methode

Abbildung3: vInterfaceTask

Abbildung4: vButtonTask

Abbildung5: vLeibnizTask

Abbildung6: vMonteCarloTask