Juventus
Technikerschule HF
www.technikerschule.ch

Projektarbeit

Calculating PI

Embedded Systems

Inhaltsverzeichnis:

1 Aufgabenstellung

Im Fach Embedded Systems wurde eine Aufgabe als Projektarbeit erteilt, in welcher es darum ging, mit Hilfe eines Mikro-Controllers in Verbindung mit dem Betriebssystem Free RTOS, die Zahl PI auf 5 Stellen nach dem Komma zu berechnen. Hierzu sollten zwei Algorithmen verwendet werden, wobei die Leibniz-Reihe als erste Variante vorgegeben wurde. Die beiden Algorithmen sollen als Tasks implementiert und über eine Statemachine gesteuert werden, ebenfalls soll das Display zur Anzeige eingebunden werden.

Zu dieser Aufgabenstellung entstand folgende Zielsetzung:

- Ausgeben des aktuellen Wertes auf dem Display, Update alle 500ms
- Starten und Stoppen des Algorithmus durch Tastendruck
- Wechseln des Algorithmus durch Tastendruck
- Zurücksetzen des Algorithmus durch Tastendruck
- Kommunikation zwischen den Tasks mit EventBits oder via TaskNotifications
- Mindestens 3 Tasks implementieren
- Erweitern des Programmes durch Zeitmessung, messen der Zeit bis PI auf 5 Stellen genau berechnet wurde
- Verfassen eines Berichtes

Die Arbeit wird nach folgenden Kriterien bewertet:

- Vollständigkeit gem. Aufgabenstellung
- Einhalten der Vorgaben gem. Aufgabenstellung
- Realisierung der zwei Algorithmen
- Realisierung des Steuertasks
- Realisierung der Zeitmess-Funktion
- Vollständigkeit der Dokumentation
- Korrekte Benutzung von Git

2 Vorstudie

In den nachfolgenden Kapiteln wird das nötige Vorwissen zu dem in der Arbeit vorgestellten Thema erläutert. Diese enthalten Informationen zu der angewandten Software, sowie zu den gewählten Berechnungsalgorithmen.

2.1 Free RTOS

Bei dem Betriebssystem Free RTOS handelt es sich um ein Real Time Operating Systems, welches zur Echtzeitsteuerung von kleineren Architekturen verwendet werden kann. Echtzeit Systeme zeichnen sich vor allem dadurch aus, dass sie lediglich CPU, Speicher, Interrupt Handler sowie Prozess-Timer verwalten. Hierdurch wird der Footprint bzw. die benötigten Ressourcen klein gehalten. Um die CPU zu entlasten, wurden die eingesetzten Funktionen so weit als möglich optimiert und verursachen somit keine Verzögerungen. Um noch mehr Perfomance zu erreichen werden keine geschichteten Treiber verwendet, sondern Hardware-Nah programmiert. Dank der geringen RAM Anforderung dieses Betriebssystems, kann es auf fast jedem Mikro-Controller eingesetzt werden.

2.2 GIT-Hub

Um einen Überblick über das Projekt zu generieren, wurde das Versionenverwaltungssystem GIT eingesetzt. GIT basiert auf nicht-linearer Softwareentwicklung, welche an keinen zentralen Server gebunden ist. Jegliche Änderungen oder Bewegungen in bzw. an einem Projekt werden registriert und aufgezeichnet. Somit ist jederzeit der aktuelle Stand eines Projekts feststellbar. Ebenfalls kann in grösseren Gruppen mit jeweils einem Master zusammengearbeitet werden. Die Teilnehmer können dann je nach Bedarf Anfragen an den Master stellen und somit dessen Projekt nutzen und gegebenenfalls anpassen. Diese Anpassungen können wiederum vom Master eingesehen werden, welcher dann entscheiden kann, ob der diese für sein Projekt einsetzen möchte.

2.3 Task Management

Der Aufgabenstellung entsprechend sollen die Hauptfunktionen des Projekts als Tasks implementiert werden, welche mit Hilfe von Free RTOS verwaltet werden. Hierbei werden ca. 50% des Codes innerhalb von Free RTOS für das Task Handling aufgewendet. Darin werden die Tasks sowie deren Stacks und Prioritäten verwaltet. Als Funktionen werden tasks.c sowie tasks.h genutzt.

Unweigerlich ab dem zweiten Task kommt die notwendige Kommunikation dazu, was wiederum 40% des Codes beansprucht. Dieser Vorgang wird durch das Queues Handling vollzogen und wird als queue.c und queue.h verwendet.

Für die Verwaltung der Tasks werden 4 Linked Lists für 4 verschiedene Zustände eingesetzt:

- Running: laufende Tasks, max. 1 Task in der Liste
- Ready to run: Tasks, welche bereit sind zu laufen
- Suspended: Tasks warden nicht vom System reaktieviert, vTaskResume muss eingesetzt werden
- Blocked: Tasks, welche auf einen Event bzw. ablaufen eines Delays warten

Mit Hilfe von Zustands Automaten, Finite State Machines (FSM), können komplexe Abläufe vereinfacht und realisiert werden. Hierdurch können verschiedene Zustände generiert werden, welche zuvor definiert werden müssen. Diese Variante wurde in der Projektarbeit dazu genutzt, um die zwei Algorithmen voneinander zu trennen und die jeweiligen Funktionen ihnen zuordnen zu können. Ebenfalls wurde ein Anfangszustand erstellt, der eine Menu Auswahl anzeigen soll, um den entsprechenden Algorithmus auszuwählen.

Um die einzelnen Tasks zu klassifizieren bezüglich ihrer Wichtigkeit muss eine Prioritätsverteilung vorgenommen werden. Diese Einteilung soll wie folgt vorgenommen werden:

- Höchste Anforderung an Rechenleistung = niedrigste Priorität
- Nichtdeterministisch < deterministisch
- Prioritätsänderungen können die Prozessorlast optimieren
- Tasks mit ähnlichen Aufgaben sollen ähnliche Prioritäten haben

2.4 Berechnungsalgorithmen

Zur Berechnung von PI mussten zwei Algorithmen definiert werden, welche im Anschluss als Task realisiert werden können. Als erster Algorithmus wurde die Berechnung mit Hilfe der Leibniz Reihe vorgegeben, als zweite Variante wurde das wallisische Produkt gewählt. Nachfolgend werden beide Algorithmen genauer beschrieben.

2.4.1 Leibniz Reihe

Mithilfe der Leibniz Reihe kann eine Annäherung an die Kreiszahl PI getroffen werden. Sie wurde 1674 von Gottfried Leibniz abgeleitet und als arithmetische Kreisquadratur bezeichnet, da der Reihenwert das Verhältnis der Flächeninhalte des Kreises und des umschriebenen Quadrats ist. Dies galt als die erste Reihendarstellung von PI.

Die Grundform der Formel sieht wie folgt aus:

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

Wie diese Formel im Projekt als Code umgesetzt wurde, wird in dem entsprechenden Kapitel genauer beschrieben.

2.4.2 Wallisisches Produkt

Als zweiter Algorithmus zur Berechnung von PI wurde das Wallisische Produkt gewählt, beim welchem es sich um ein Produkt aus unendlich vielen Faktoren handelt, deren Grenzwert PI ist. Dieses Verfahren wurde 1655 vom englischen Mathematiker John Wallis entdeckt.

Die Grundform der Formel sieht wie folgt aus:

$$\frac{\pi}{2} = \frac{2}{1} * \frac{2}{3} * \frac{4}{3} * \frac{4}{5} * \frac{6}{5} * \frac{6}{7} * \dots$$

Wie diese Formel im Projekt als Code umgesetzt wurde, wird in dem entsprechenden Kapitel genauer beschrieben.

3 Haupteil

Anhand der bisher vorgestellten Techniken konnte nun das Programm zur Berechnung von PI entworfen werden. Nachfolgend werden die einzelnen Teilabschnitte genauer beschrieben und anhand von Codeausschnitten aufgezeigt.

3.1 Erstellen des Programms

Als Programmierumgebung für das Projekt wurde das Programm Microchip Studio gewählt, dies aufgrund bisheriger Erfahrungen bei anderen Projekten und Kompatibilität mit der angewandten Hardware.

3.1.1 Verfassen der Tasks

Es wurde entschieden die Funktionen auf vier verschiedene Tasks aufzuteilen, um so die Abläufe aufzutrennen und trotzdem nicht zu viel Leistung zu brauchen. Ebenfalls konnte durch die Priorisierung der einzelnen Tasks das Programm optimiert werden.

Die Aufteilung erfolgte wie folgt:

- Task 1 für die allgemeine Steuerung der Abläufe, sowie der Ausgabe auf dem Bildschirm (controllerTask)
- Task 2 für das Erstellen der Berechnung anhand der Leibniz-Reihe (leibnizTask)
- Task 3 für das Erstellen der Berechnung anhand des Wallisischen Produkts (KetteTask)
- Task 4 für das Einlesen der verschiedenen Buttons sowie deren Funktionen (ButtonTask)

Die Allgemeine Kommunikation zwischen und innerhalb der Tasks erfolgte durch die Verwendung der diversen «Event-Group» Funktionen. Hierbei wurden folgenden Varianten verwendet:

- xEventGroupGetBits -> gibt die aktuellen Werte einer Event Group aus
- xEventGroupClearBits -> setzt die Werte innerhalb einer Event Group zurück
- xEventGroupSetBits -> setzt bestimmte Werte innerhalb einer Event Group
- xEventGroupWaitBits -> abwarten bestimmter Werte aus einer Event Group, Blocked State

Diese einzelnen Funktionen wurden innerhalb des Programms mit den Modi aus dem Controllertask, sowie den beiden Berechnungstasks und dem Buttontask verbunden. Somit konnten Bitabfragen taskübergreifend erfolgen.

```
// Tasks definieren
xTaskCreate( controllerTask, (const char *) "control_tsk", configMINIMAL_STACK_SIZE+150, NULL, 2, NULL);
xTaskCreate( leibniztask, (const char *) "leibniz_tsk", configMINIMAL_STACK_SIZE+150, NULL, 1, NULL);
xTaskCreate( KetteTask, (const char*) "Kette_tsk", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
xTaskCreate( ButtonTask, (const char *) "Button_tsk", configMINIMAL_STACK_SIZE, NULL, 3, NULL);
```

Abbildung 1: Task Definition

In Abbildung 1 sind die zuvor erklärten Tasks ersichtlich. Ebenso lässt sich die Priorisierung gegen Ende der Zeilen anhand der Nummerierung 1-3 erkennen. Dem Controller Task wurde die Priorität 2 zugewiesen, da dieser wichtig ist für die Steuerung der Tasks und daher höher als diese sein muss. Dem Button Task wurde die höchste Priorität zugewiesen, da die Buttons jederzeit erkennt werden müssen. Bei den Berechnungstask reichte die niedrigste Priorität aus, da diese nur für die Berechnung zuständig sind und alles andere vom Controller Task übernommen wird.

Der Controller Task wurde mit Hilfe einer Finite State Machine entworfen, welche die Zustände «MODE_IDLE, MODE_PI, MODE_Display» besitzt. Der Mode Idle wurde als Grundeinstellung definiert, aus welchem in die anderen Modi gewechselt werden kann. Um das Display wie gewünscht alle 500ms zu aktualisieren, wurde im Mode Idle ein Timer initialisiert, welcher diese Zeit generiert und mit den anderen zwei Modi verknüpft werden konnte. Im Mode Pi wurden die Berechnungsalgorithmen eingebunden und untereinander abgestimmt, wie sie zu reagieren haben. Nachdem ein Wert für PI generiert wurde, musste dieser an das Display ausgegeben werden, dies erfolgte durch den Mode Display. In diesem wurden zwei Displaybeschreibungen für die einzelnen Algorithmen entworfen, somit wechselt das Display je nach gewählter Berechnung die Anzeige. Ebenso werden die Befehle «calculate cal, stop stp, reset rst, switsch sw» angezeigt, mit welchen die Bedienung ersichtlich gemacht wurde.

3.1.2 Einbinden der Algorithmen

Wie zuvor erwähnt, wurden auch die beiden Algorithmen als Tasks verfasst und im Programm eingebunden. Bei beiden Berechnungen mussten als Erstes Anfangswerte definiert werden, welche dann nach einem Durchlauf, je nach Algorithmus, weiterverarbeitet und als neue Anfangswerte genutzt wurden. Dies geschah jeweils zu Beginn der Tasks.

```
void leibniztask(void *pvParameters) {
    float piviertel = 1.0;
    if (xEventGroupGetBits(egButtonEvents) & Start_Leibniz){
        for(;;){
            uint32_t n = 3;
            piviertel = piviertel -(1.0/n) + (1.0/(n+2));
            n = n+4;
            pi_calc = piviertel * 4;
        }
    }
    else{
        xEventGroupSetBits(egButtonEvents, PI_Ready);
        xEventGroupWaitBits(egButtonEvents, Start_Leibniz, false, true, portMAX_DELAY);
    }
}
```

Abbildung 2: Berechnung mit der Leibnizmethode

In Abbildung 2 ist die Berechnung mit der Leibnizmethode aufgezeigt. Wie anhand der Grundformel erklärt wurde, geht man von dem Wert $\frac{\pi}{4}$ aus, welcher hier als «piviertel» definiert wurde. Weiter wurde ein Wert «n» benötigt welcher in jedem Durchgang durch «n=n+4» um 4 erhöht wurde. Als Resultat wurde der Wert «piviertel» mit 4 multipliziert und dem Ausgabewert «pi_calc» zugewiesen.

```
void KetteTask(void *pvParameters){
    float piKette = 2.0;
    float piRechnen = 1.0;
    if (xEventGroupGetBits(egButtonEvents) & Start_Kette){
        for(;;){
            piRechnen = piRechnen * (piKette/(piKette - 1.0));
            piRechnen = piRechnen * (piKette/(piRechnen + 1.0));
            piKette = piKette + 2;
            pi_calc = piRechnen * 2;
        }
    }
    else{
        xEventGroupSetBits(egButtonEvents, PI_Ready);
        xEventGroupWaitBits(egButtonEvents, Start_Kette, false, true, portMAX_DELAY);
    }
```

Abbildung 3: Berechnung mit dem Wallisischen Produkt

In Abbildung 3 ist nach ähnlichem Schema wie zuvor erklärt, die Berechnung mit Hilfe des Wallisischen Produkts aufgezeigt. Hier wurden zur Berechnung die zwei Variablen «piKette» und «piRechnen» definiert, durch welche die fortlaufenden Brüche dargestellt werden konnten. Die Variablen wurden wie gehabt bei jedem Durchgang angepasst und zum Schluss als Ergebnis dem Anzeigewert «pi_calc» zugewiesen.

4 Fazit

4.1 Abschluss des Projektes

Das Projekt konnte nicht wie geplant abgeschlossen werden, die Berechnung ist unvollständig. Das Programm wurde durch Debuggen versucht zu erweitern, sowie die Fehler ausfindig zu machen. Dies gelang nicht und es blieb bei der Grundanzeige auf dem Display, die Algorithmen konnten nicht ausgewählt werden. Aufgrund der ausstehenden Berechnung liess sich auch keine Zeitmessung erstellen, daher wurde dieser Punkt in der Dokumentation nicht verfasst. Während der Fehlersuche stellte sich heraus, dass einige EventGroup Funktionen den Ablauf des Programms blockieren. Als diese auskommentiert wurden konnte ein Teil durchlaufen werden, jedoch zeigten sich hierbei noch weitere Probleme. Die zielbringende Lösung konnte nicht erarbeitet werden. Ebenfalls entstand der Gedanke, dass eine weitere Aufteilung mit einem zusätzlichen Task sinnvoll gewesen wäre. Dies zum Beispiel in Form eines Display Tasks, welcher alle Funktionen, die das Display betreffen aus dem Controller Taks entfernt hätten. Somit könnte sich die Komplexität verringern, sowie die Fehlersuche vereinfachen.

4.2 Persönliches Fazit

Wie bis anhin gestaltete sich diese Programmierübung für mich als sehr schwierig. Da kaum Vorkenntnisse bestehen, konnte ich bisher die angewandten Techniken der C Programmierung nicht festigen und daher entstanden gewisse Lücken. Dies zeigt sich vor allem dann, wenn es darum geht, selbst von Grund auf ein Programm zu entwerfen. Trotzdem versuchte ich mir einen Aufbau zu generieren, welcher zumindest grob die gewünschten Funktionen vereinte. Um diese Grundideen in einen einigermassen nachvollziehbaren Code zu verfassen, war ich wiederum auf Hilfe von Kameraden angewiesen. So versuchte ich anhand von anderen Beispielen mein Programm Stück für Stück zu erweitern. Dies gelang auch zu einem gewissen Teil, jedoch ist die Fehlersuche bei mangelnden Kenntnissen sehr schwierig. Somit fehlten zuletzt doch noch einige Dinge bis zum fertigen Projekt. Trotz aller dieser Punkte konnte ich erneut viel dazu lernen in diesem Fachbereich. Auch das Anwenden eines Betriebssystems mit dessen Vorteilen wurde für mich verständlicher. Daher sehe ich dies nicht als Misserfolg an, sondern als erneuter Schritt im Bereich C-Programmierung.

5 Anhang

5.1 Abbildungsverzeichnis

5.2 Literaturverzeichnis

Sämtliche verwendeten Bilder in dieser Dokumentation stammen aus dem Programm Microchip Studio und zeigen den für das Projekt entworfenen Code.