π berechnen

Embedded Systems

HS 2021

Yves Savoy

Inhaltsverzeichnis

Wallis-Reihe

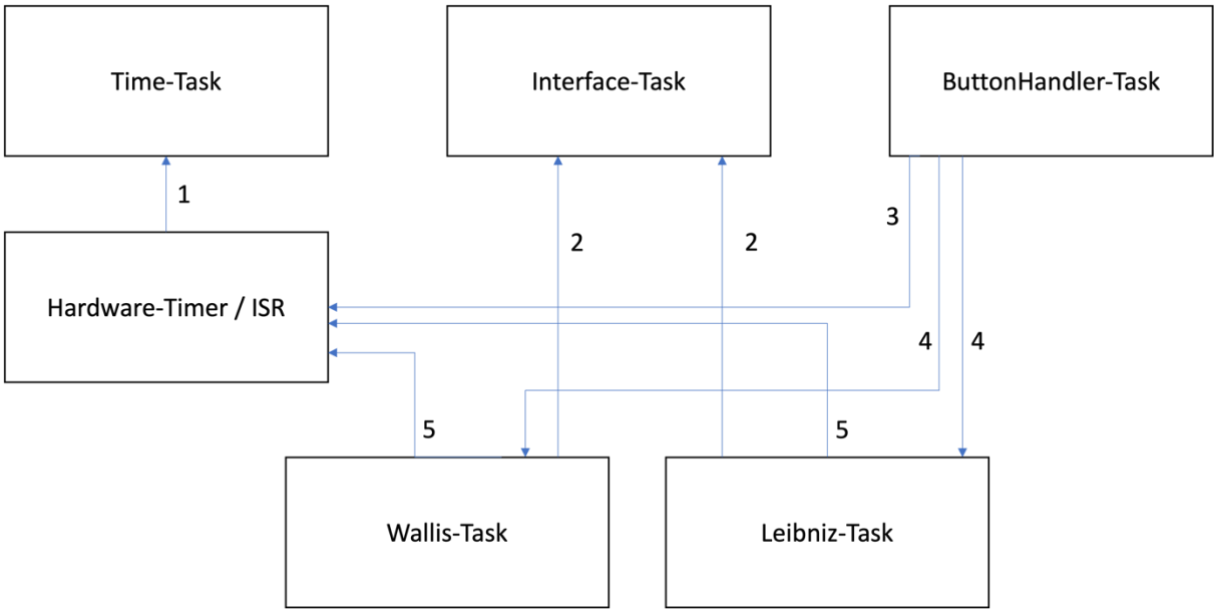
Als zweiten Algorithmus wurde die Wallis-Reihe gewählt. Sie funktioniert wie folgt:

$$\pi = 4 * \frac{2}{3} * \frac{4}{3} * \frac{4}{5} * \frac{6}{5} * \frac{6}{7} * \frac{8}{7} * \dots$$

Mit jeder zweiten Iteration wird der Nenner des Bruchs um 2 erhöht (3, 5, 7, 9, etc.). Der Zähler ist abhängig vom Nenner. In der ersten Iteration wird der Zähler als Nenner – 1 berechnet, in der zweiten Iteration wird der Zähler als Nenner + 1 berechnet.

Tasks

Systemübersicht



Nummer	Beschreibung
1	Task-Notification an Time-Task um die Zeit weiterlaufen zu lassen
2	Auf Freigabe des Locks warten; Wenn Lock frei ist, PI lesen
3	Hardware-Timer starten
4	Kalkulationstask starten
5	Hardware-Timer stoppen

Interface-Task

Der Interface-Task ist für das User-Interface und den darauf angezeigten Daten zuständig. Der Task ist mit einer einfachen Finite-State Maschine realisiert. Es gibt den Gestartet- und Gestoppt-State, je nachdem ob einer der beiden Algorithmen aktuell gerade am Rechnen ist oder nicht. Aufgrund der Einfachheit der State-Maschine wird bewusst auf ein State-Diagramm verzichtet.

Damit auf dem Display immer ein gültiger Float-Wert angezeigt werden kann wurde ein einfaches Locking-System ohne Mutex realisiert. Der Interface-Task wartet auf ein Event-Bit welches vom aktuell laufenden Kalkulationstask gesetzt wird sobald das eigentliche Rechnen von PI beendet ist. Während die Kalkulation prüft ob der berechnete Wert genau genug ist

oder ob es neue Task-Notifications gibt hat der Interface-Task Zeit PI zu lesen und anzuzeigen. Während der Kalkulationstask mit dem Rechnen beschäftigt ist wartet der Interface-Task auf das entsprechende Event-Bit. Da die Update-Rate des Displays 500ms betragen muss wird vTaskDelayUntil() anstelle von vTaskDelay() genutzt. Mit dieser Variante wird das Warten auf das Event-Bit ebenfalls berücksichtigt und der Task läuft näher an der 500ms-Rate als ohne vTaskDelayUntil().

Buttonhandler-Task

Dieser Task ist für die Eingaben des Benutzers und deren Aktionen zuständig. Es wird eine externe Bibliothek verwendet welche Aufgaben wie die Entprellung und das Aufsetzen der Register übernimmt. Folgende Funktionen können über die Knöpfe gesteuert werden:

- S1: Starten des aktuell ausgewählten Algorithmus
- S2: Stoppen des aktuell laufenden Algorithmus
- S4: Wechseln zwischen dem Leibniz- und dem Willis Algorithmus

Wird der ausgewählte Algorithmus über die Taste S1 gestartet schickt dieser Task eine Notification an den entsprechenden Algorithmus-Task mit der Meldung das dieser zuerst einen Reset machen und danach starten kann. Zudem wird der Hardware-Timer gestartet, der State des Interface angepasst und die Zeit zurück auf 0 gesetzt.

Wird der aktuelle Algorithmus über die Taste S2 gestoppt wird der Hardware-Timer ausgeschaltet, der State des Interface entsprechend in den Stopp-State versetzt und eine Notification mit dem Stopp-Signal an den entsprechenden Task gesendet.

Timehandler-Task

Dieser Task reagiert auf die Interrupts des Hardware-Timers. Die ISR sendet dem Timehandle-Task pro Interrupt eine Notification. Der Task ist, bis eine Notification kommt, im Blocking-State. Sobald eine Notification ‘pending’ ist läuft der Task, zählt die Millisekunden entsprechend nach oben und geht wieder in den Blocking-State über bis die nächste Notification ankommt.

Kalkulations-Tasks

Pro Algorithmus gibt es einen separaten Task, jedoch funktionieren beide vom Kontrollfluss her gleich, daher werden beide Tasks in diesem Kapitel beschrieben.

Der Task wird beim Programmstart gestartet und wartet auf eine entsprechende Start-Nachricht. Sobald diese verfügbar ist beginnt der Algorithmus mit der Annäherung an Pi. Bevor die Pi-Variable mutiert wird, wird ein Event Bit gelöscht welches andere Tasks daran hindert Pi zu diesem Zeitpunkt zu lesen. Sobald die Mutation beendet ist, wird das Event Bit gesetzt und gibt so das Lesen von Pi frei. Das Display kann zu diesem Zeitpunkt Pi fehlerfrei lesen.

Innerhalb des Tasks wird in jeder Iteration auf neue Nachrichten geprüft. Sobald eine Stopp-Nachricht den Task erreicht, wird dieser Unterbrochen, geht in den Blocking-State über und wartet auf eine erneute Start-Nachricht.

Ist die geforderte Genauigkeit von Pi erreicht wird der Hardware-Timer gestoppt, die Berechnung von Pi läuft jedoch weiter bis der Task durch eine Stopp-Nachricht unterbrochen wird.

Task Notifications

Timer-Notifications

Der Zeit-Task kann mit 2 Task-Notifications gesteuert werden. Die erste, Time Tick, wird verwendet, um die angezeigte Zeit weiterlaufen zu lassen. Die zweite, Time Reset, wird verwendet, um die angezeigte Zeit zurück auf 0 zu setzen.

Notification	**Sender**	**Empfänger**	**Bit**
Time Tick	Timer-Interrupt	vTimeHandler	N_TIME_TICK
Time Reset	vButtonHandler	vTimeHandler	N_TIME_RST

Kalkulations-Notifications

Die beiden Algorithmen werden am Anfang des Programms gestartet und warten dann auf eine Nachricht, welche sie aus dem Blocking in den Running-State versetzt. Wird ein Algorithmus von neuem gestartet wird in der Notification zudem ein Reset verlangt damit die Variablen wieder korrekt initialisiert werden.

Notification	**Sender**	**Empfänger**	**Bit**
Kalkulation starten	vButtonHandler	vCalculateLeibniz vCalculateWallis	N_CALC_START
Kalkulation stoppen	vButtonHandler	vCalculateLeibniz vCalculateWallis	N_CALC_STOP
Kalkulation zurücksetzen	vButtonHandler	vCalculateLeibniz vCalculateWallis	N_CALC_RST

Event Groups und Event Bits

Das Programm benutzt eine globale EventGroup und ein EventBit. Über diese Gruppe wird gesteuert, wann es für Tasks sicher ist Pi zu lesen, zum Beispiel für eine Anzeige, und wann der Algorithmus Pi gerade berechnet und nicht garantiert ist das Pi einen validen float-Wert enthält. Da Event Groups Thread-Save sind wird diese anstelle eines globalen Flags bevorzugt.

Man könnte an dieser Stelle anstatt der EventGroup auch einen Mutex oder eine ähnliche Datenstruktur verwenden um sicherzustellen, dass die Variable nicht gleichzeitig gelesen und geschrieben wird.

Event Bit	Gesetzt von	Gelöscht von	Gelesen von	Zweck
EG_CALC_RELEASED	vCalculateLeibniz vCalculateWallis	vCalculateLeibniz vCalculateWallis	vInterface	Korrektheit von Pi gewährleisten

Zeitmessung

Um einen Vergleich der beiden Algorithmen zu machen wird während 5 Durchläufen mit verschiedenen Compiler-Optimierungen die Zeit gemessen bis PI eine Genauigkeit von 5 Dezimalstellen erreicht.

Leibniz

Debug-Optimierung (Option -Og)

Lauf	Zeit
1	11318ms
2	11294ms
3	11292ms
4	11329ms
5	11292ms

Grösste Optimierung (Option -O3)

Lauf	Zeit
1	11133ms
2	11132ms
3	11165ms
4	11133ms
5	11133ms

Wallis

Debug-Optimierung (Option -Og)

Lauf	Zeit
1	2677ms
2	2640ms
3	2677ms
4	2642ms
5	2653ms

Grösste Optimierung (Option -O3)

Lauf	Zeit
1	2597ms
2	2595ms
3	2595ms
4	2596ms
5	2595ms

Schlussfolgerung

Die aufgeführten Tests zeigen im Wesentlichen die folgenden 3 Punkte.

Die Wallis-Reihe erreicht die geforderte Genauigkeit zirka 4-5x schneller als die Leibniz-Reihe.

Alle Durchläufe sind immer gleich schnell, mit einer Differenz von +/- 50ms. Das System erzeugt also vorhersehbare Ergebnisse. Die Differenzen sind auf den Scheduler und die Tasks, welche dieselbe Priorität haben, zurückzuschliessen. Da 3 von 5 Tasks mit derselben Priorität laufen kann man nicht vorhersagen welcher der 3 vom Scheduler als nächstes geplant wird. Jedoch macht, wie in den Messergebnissen zu sehen ist, diese Unsicherheit in der Planung nur einen minimen Unterschied auf die Resultate.

Die Compiler-Optimierung macht die Berechnung der Wallis-Reihe zirka 50-80ms schneller. Die Leibniz-Reihe wird durch die Optimierung zirka 120-180ms schneller.

Optimierung

Momentan wird jede Millisekunde ein Interrupt des Timers generiert. Daraus folgt eine Task-Notification an den Zeitmesstask. Durch die höhere Priorisierung des Timer-Task wird der aktuelle Kalkulationstask mindestens jede Millisekunde unterbrochen. Um dem entgegenzuwirken können weniger Timer-Interrupts generiert werden in der Hoffnung die Berechnung schneller zu machen. Im Gegenzug wird jedoch die Zeitmessung ungenauer, da mit dieser Optimierung die Zeit in 5ms oder 10-Takten läuft und nicht mehr im Millisekunden-Takt. Die Firmware wird für die folgenden Tests mit der grössten Optimierung (-O3) kompiliert.

Versuch 1 (5ms)

Als erster Test wird anstatt jeder Millisekunde nur alle 5 Millisekunden ein Interrupt generiert.

Folgende Zeiten wurden für die Leibniz-Reihe gemessen:

Lauf	**Zeit**
1	10645ms
2	10680ms
3	10675ms

Folgende Zeiten wurden für die Wallis-Reihe gemessen:

Lauf	**Zeit**
1	2500ms
2	2495ms
3	2470ms

Versuch 2 (10ms)

In diesem Test werden die Interrupts erneut halbiert und es wird nur alle 10 Millisekunden ein Interrupt generiert.

Folgende Zeiten wurden für die Leibniz-Reihe gemessen:

Lauf	**Zeit**
1	10590ms
2	10600ms
3	10600ms

Folgende Zeiten wurden für die Wallis-Reihe gemessen:

Lauf	**Zeit**
1	2480ms
2	2450ms
3	2490ms

Fazit

Im ersten Versuch wurde die Leibniz-Reihe zirka 500ms schneller, die Wallis-Reihe 90ms-100ms schneller. Da jedoch weniger Interrupts generiert werden ist die Zeit nun ungenauer als vorher. Eine Zeit von 2500ms muss also als ein Zeitfenster von 2500ms-2504ms beachtet werden da zwischen dem letzten Interrupt und dem Erreichen der geforderten Genauigkeit weiter 4ms vergangen sein könnten.

Bei der erneuten Teilung der Interrupts auf einen Interrupt alle 10ms wurde die Leibniz-Reihe, gegenüber der Messung mit einem Interrupt alle 5ms, zirka 60ms schneller, die Wallis-Reihe 20ms schneller. Die Zeitmessung ist jetzt auf 10ms ungenau. Trotz der erhöhten Ungenauigkeit der Zeit wurde die Berechnung in beiden Fällen nicht signifikant schneller, daher scheint die Option mit einem Interrupt alle 5ms die beste Lösung zu sein.

Die Leibniz-Reihe zeigt deutlich mehr Zeitgewinn als die Wallis-Reihe nach diesen Optimierungen. Das liegt daran, dass die Leibniz-Reihe mehr Iteration durchlaufen muss um das gleiche Resultat wie die Wallis-Reihe zu erzeugen. Durch diese zusätzlichen Iterationen profitiert die Leibniz-Reihe mehr von der Optimierung als die Wallis-Reihe.

Gegenüberstellung von Rechenleistung und Prozessorleistung

Anhand der Leibniz-Kalkulation soll herausgefunden werden, wie oft die Kalkulation im Verhältnis zu den Ticks läuft. Der Tick wird in diesem Fall mit einem 1kHz-Timer aufgerufen, also einem Intervall von 1ms.

Für diese Analyse wurden 3 globale Variablen definiert: Ein Zähler, welcher pro Iteration im Kalkulationstask erhöht wird, ein Zähler, welcher pro Iteration eines anderen Tasks erhöht wird und eine Tick-Variable, welche die Anzahl Ticks enthält, welche zwischen dem Start der Kalkulation und dem Erreichen der Genauigkeit vergangen sind.

Die folgende 3 Messungen wurden für die Leibniz-Reihe gemacht:

Zeit	**Ticks**	**Iterationen der Kalkulation**	**Iterationen anderer Tasks**
10’395ms	10’807	125’202	3484
10’465ms	10’808	125’202	3528
10’325ms	10’808	125’202	3542

Wie ersichtlich ist braucht die Kalkulation immer dieselbe Anzahl Iteration, um auf das gewünschte Resultat zu kommen. Dies ist nachvollziehbar da ein Algorithmus deterministisch ist und, unabhängig der Zeit, in immer gleich vielen Iterationen zum gleichen Ergebnis kommt.

Die Unterschiede in der Zeit, die das System braucht, um die Genauigkeit von PI zu erreichen, lässt sich durch die unterschiedliche Anzahl von Iterationen der anderen Tasks (Time-Handler, Interface, Button-Handler) erklären. Im ersten Versuch wurden die anderen 3 Tasks 44 Iterationen weniger ausgeführt als im zweiten Versuch. Dies bedeutet wiederum, dass der Kalkulationstask mehr Zeit und Iterationen bekommen hat und daher im ersten Versuch schneller fertig war als im zweiten Versuch.

Aus den 3 getätigten Messungen lässt sich nun ableiten, dass der Kalkulationstask trotz niedrigster Priorität ungefähr 97% der gesamten Rechenleistung erhält während die anderen Tasks sich die restlichen 3% teilen. Diese einfache Rechnung beachtet Takte, welcher der Scheduler komplett für sich braucht oder den ISR-Overhead, nicht und gibt daher nur eine grobe Idee der Verteilung der Rechenleistung.