School of Engineering
InES Institute of Embedded Systems

MC1 Praktikum

Real Time Operating System / RTOS

1 Einleitung

Sie werden in diesem Praktikum die Features eines Real Time Betriebssystems kennenlernen. Ein Betriebssystem kann Ihnen viele Aufgaben abnehmen, wie z.B. das Scheduling von verschiedenen Threads, die quasi-parallel ausgeführt werden können, oder Zugriffe auf gemeinsame Ressourcen mittels Mutexes zu synchronisieren.

- Informationen zum CMSIS-RTOS finden Sie unter https://www.keil.com/cmsis/rtos.
- Verwenden Sie für die Ansteuerung der GPIOs (LEDs) die Funktionen aus dem HAL (Hardware Abstraction Layer), also z.B. `hal_gpio_bit_set(GPIOG, LED_GREEN);` und `hal_gpio_bit_reset(GPIOG, LED_GREEN);`

2 Lernziele

- Sie kennen die verschiedenen Scheduling Techniken und können sie in eigenen Programmen anwenden.
- Sie können mit Hilfe von Signals und Mutexes den Programmablauf synchronisieren.

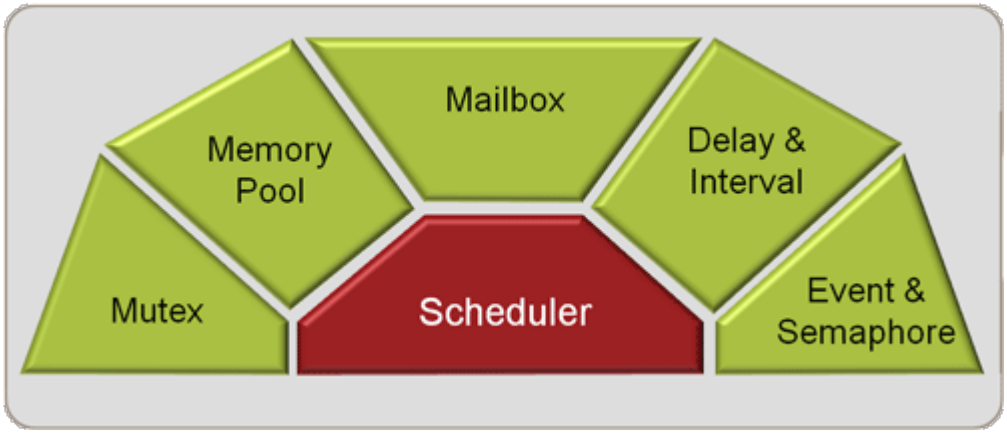


Abbildung 1: CMSIS RTOS Features

3 Multiprojekt

In Keil kann ein sogenanntes Multiprojekt erstellt werden. Ein Multiprojekt ist eine Zusammenfassung mehrerer Keil Projekte, im Idealfall logisch zusammengehörend.

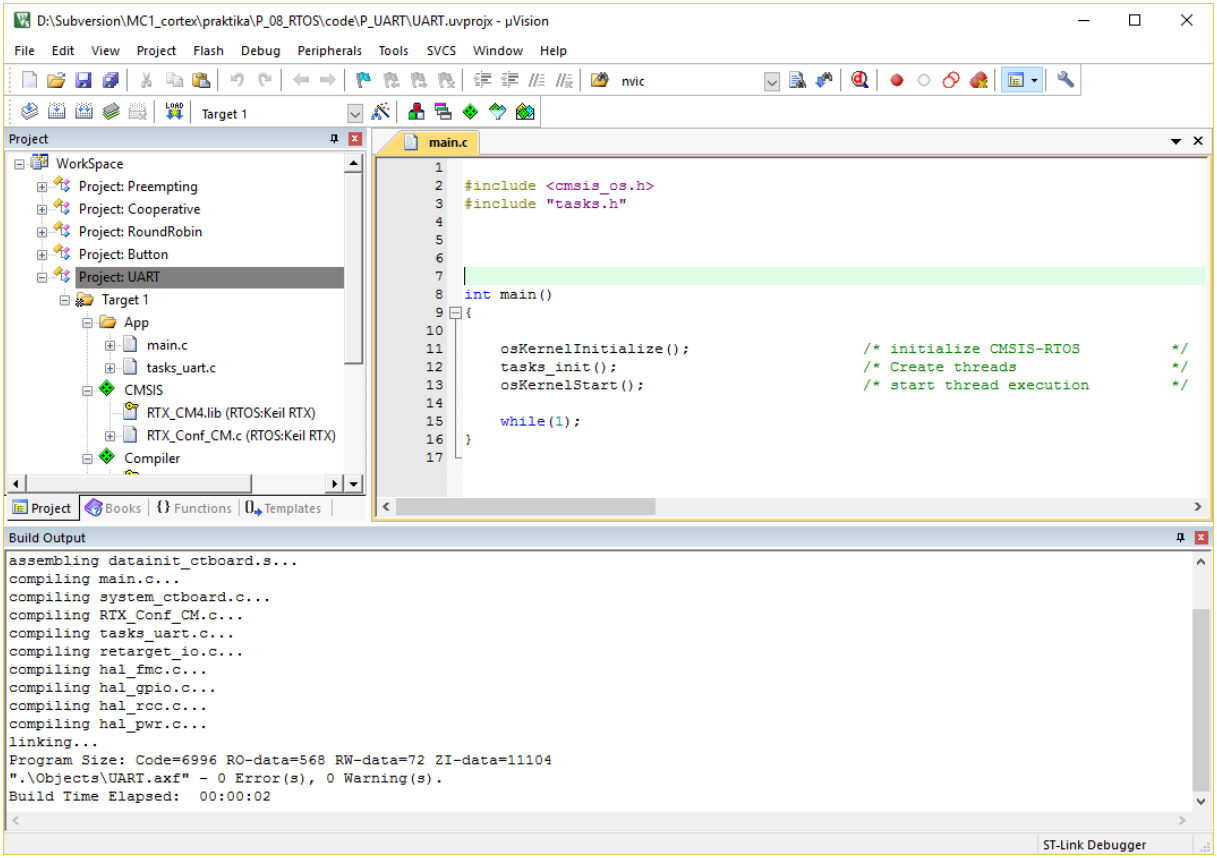


Abbildung 2: Keil Multiprojekt

Innerhalb eines Multiprojekts kann immer nur ein Projekt aktiv sein. Um das aktive Projekt zu wechseln, muss im Projektfenster, auf der linken Seite, das gewünschte Projekt markiert werden und mit einem rechtsklick „*Set as Active Project*“ ausgewählt werden.

Die einzelnen Projekte sind komplett unabhängig, jedes Projekt muss separat eingestellt und konfiguriert werden.

- Öffnen Sie für dieses Praktikum das vorhandene Multiprojekt *.\code\Lab.uvmpw*.

4 Umgebung

Hardware

Verwenden Sie für **alle** Aufgaben in diesem Praktikum das Filesystem Board. (Siehe Abbildung 3).

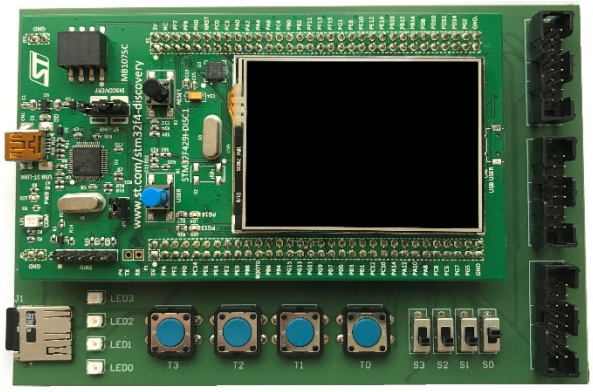


Abbildung 3: Filesystem Board

CMSIS_RTOS RTX Einstellungen

Die Einstellungen des Real Time Betriebssystems finden Sie jeweils hier:

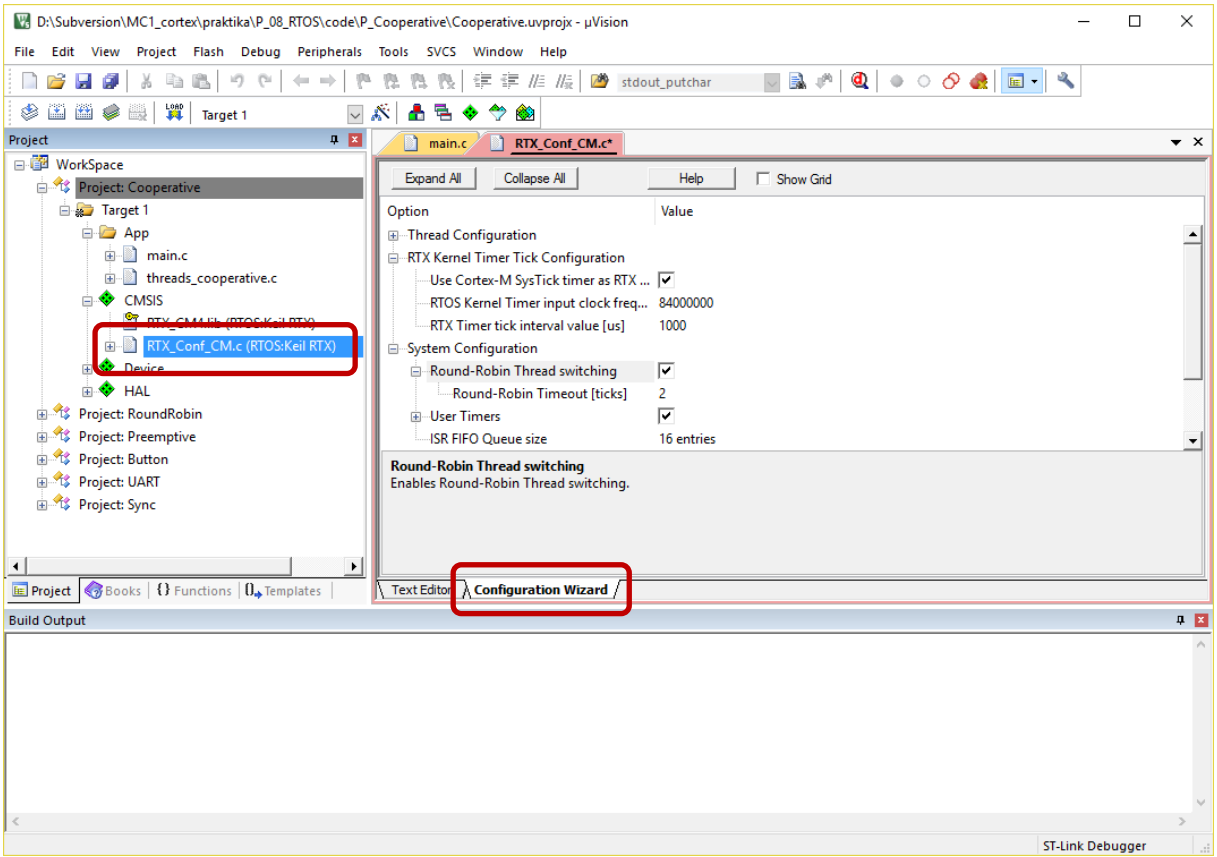


Abbildung 4: RTOS konfigurieren

Event Viewer

Das Scheduling-Verhalten können Sie im Keil Debugger mit dem Event Viewer untersuchen (*Debug -> OS Support -> Event Viewer*). Weitere Informationen zum Event Viewer finden Sie unter http://www.keil.com/support/man/docs/uv4/uv4_db_dbg_event_viewer.htm.

Standardausgabe

Die Standardausgabe wird auf den UART1 Port gelegt (siehe Anhang 9). Mittels **`printf(const char * format, ...)`** können Sie beliebige Strings an den UART1 Port senden. Wie Sie den UART1 Port mit dem PC verbinden ist in Anhang 10 beschrieben.

5 Aufgabe – Scheduling

Das CMSIS_RTOS RTX ist **preemptive**. D.h. wenn bestimmte Bedingungen erfüllt sind, unterbricht der Scheduler einen laufenden Thread und setzt ihn zu Gunsten eines anderen Threads aus. Im Folgenden werden Sie mit verschiedenen Einstellungen des Schedulers arbeiten.

5.1 Fall 1: Equal Priority – No Round-Robin

Der Scheduler wird in diesem Fall keinen der beiden Threads unterbrechen (d.h. es tritt keine Preemption auf), da beide Threads die gleiche Priorität haben und kein Round-Robin aktiviert ist. Die Threads müssen sich daher kooperativ verhalten und die Kontrolle über die CPU und ihre Ressourcen eigenständig wieder abgeben

Project	Cooperative
LED Grün (LD3)	GPIO Port G.13
LED Rot (LD4)	GPIO Port G.14

- Erstellen Sie in **`threads_init()`** zwei Threads, beide mit gleicher Priorität.
- Prüfen Sie für jeden der beiden Threads, ob das Erstellen erfolgreich war. Geben Sie entweder eine entsprechende Erfolgs- oder Misserfolgsmeldung mit **`printf()`** auf der UART aus.
- Thread 1 schaltet die grüne LED ein, wartet ungefähr 0.5s, schaltet die LED wieder aus und wartet wieder ungefähr 0.5s. Danach gibt er sich frei (yield).
- Thread 2 macht dasselbe mit der roten LED.
- Verwenden Sie für das Warten die gegebene Funktion **`wait_blocking(HALF_SECOND);`**
- Fügen Sie im Anhang einen Screenshot des Event Viewers ein. Beschreiben Sie das Verhalten.
- Experimentieren Sie mit den Prioritäten der Threads. Was stellen Sie fest? Wie ändert sich das Verhalten der Schaltung?

5.2 Fall 2: Equal Priority – With Round-Robin

Beim Round Robin Scheduling erhält der laufende Thread eine konfigurierbare Anzahl Zeitschlitze zugeteilt. Ein Thread läuft so lange, bis entweder (a) ein Thread mit höherer Priorität bereit ist, (b) der Thread blockiert wird, (c) er sich selbst frei gibt oder (d) seine Zeitschlitze abgelaufen sind und ein Thread mit gleicher Priorität bereitsteht.

Project	Cooperative

Untersuchen Sie die Unterschiede zum Fall 1.

- Verwenden Sie dasselbe Projekt wie in Aufgabe 5.1., d.h. die beiden Threads mit gleicher Priorität.
- Schalten Sie Round Robin Scheduling ein.
- Fügen Sie im Anhang einen Screenshot ein.
- Erklären Sie die Unterschiede zum Fall 1.

5.3 Messung Round Robin Timeout

Das Round Robin Timeout lässt sich einstellen. Sie können einerseits das SysTick Intervall selbst (*RTX timer tick intervall [us]*, Siehe Abbildung 4) verändern sowie die Anzahl Ticks bis ein anderer Thread ausgeführt wird (*Round Robin timeout [ticks]*, Siehe Abbildung 4).

Project	RoundRobin
LED Grün (LD3)	GPIO Port G.13

Untersuchen Sie das Round Robin Timeout Verhalten mit Hilfe von zwei Threads. Messen Sie mit dem Oszilloskop das Round Robin Timeout. Verändern Sie das Round Robin Timeout und beweisen sie die Änderungen mit einer Messung.

- Erstellen Sie zwei Threads, beide mit gleicher Priorität.
- Thread 1 schaltet die grüne LED ein und wartet dann für ca. 0.5s.
- Thread 2 schaltet die grüne LED aus und wartet dann für ca. 0.5s.
- Verwenden Sie für das Warten die gegebene Funktion:
  `wait_blocking(HALF_SECOND);`
- Bestimmen Sie die Round Robin Timeouts anhand der SysTick Timer und Round Robin Timeout Einstellungen.
- Messen Sie mit dem Oszilloskop das Round Robin Timeout.
- Verändern Sie die Timeout Zeiten und verifizieren Sie damit Ihre Theorie.
- Fügen Sie im Anhang einen Screenshot des Oszilloskops ein, in dem die Timeout Zeit ersichtlich ist.

Falls kein Oszilloskop zur Verfügung steht, dokumentieren Sie Ihre Versuche mit Event Viewer Plots und beschreiben, was Sie auf der LED feststellen (genügend lange Timout Einstellungen verwenden).

5.4 Fall 3: Different Priorities with osDelay()

Beim Preemptive Scheduling kann ein Thread mit einer höheren Priorität einen Thread mit niedrigerer Priorität unterbrechen.

Project	Preemptive
LED Grün (LD3)	GPIO Port G.13
LED Rot (LD4)	GPIO Port G.14

Schreiben Sie Threads, mit welchen Sie das Preemptive Scheduling testen können. Zeigen Sie, dass Thread 1 erst aktiv wird, wenn Thread 2 im Zustand *waiting* ist.

- Erstellen Sie zwei Threads, Thread 2 soll eine höhere Priorität haben.
- Thread 1 schaltet die grüne LED ein, wartet ungefähr 0.5s, schaltet die LED wieder aus und wartet wieder ungefähr 0.5s (**`wait_blocking(HALF_SECOND)`**).
- Thread 2 macht dasselbe mit der roten LED.
- Fügen Sie am Ende des while-Loop von Thread 2 ein **`osDelay(40u)`** ein, damit der höher priorisierte Thread 2 für 40ms schlafen gelegt wird.
- Fügen Sie im Anhang einen Screenshot des Event Viewers und eine Erklärung ein.

6 Aufgabe – Signals

Signals werden verwendet um einem anderen Thread zu signalisieren, dass ein bestimmter Event aufgetreten ist. Threads können auf Signals warten, welche auslösen oder löschen.

Project	Button
User Button	GPIO Port A.0
LED Grün (LD3)	GPIO Port G.13

Auf dem Discovery Board befindet sich ein User Button. Dieser Button soll mit Hilfe von Signals die grüne LED ein- bzw. ausschalten.

- Erstellen Sie zwei Threads, beide mit gleicher Priorität.
- Thread 1 pollt den User Button und setzt, falls er gedrückt wurde, ein Signal ab. Implementieren Sie eine geeignete Flankendetektion.
- Thread 2 wartet auf das Signal von Thread 1 und schaltet, wenn es empfangen wurde, die grüne LED ein bzw. aus.

Hinweis: Die Aufgabe soll zwingend mit Signals des RTOS gelöst werden!

7 Aufgabe – Mutexes

Wenn mehrere Threads auf eine gemeinsame Ressource, z.B. eine Variable, zugreifen möchten, kann es zu Problemen kommen. Der Zugriff auf eine Ressource ist selten atomar (Variable lesen, verändern und wieder zurückschreiben). Damit es nicht zu Inkonsistenzen kommt, z.B. aufgrund eines Thread Switches, muss der Zugriff auf diese Ressource gesteuert werden. Dies geschieht mit Hilfe von Mutexes. Ein Mutex wird vor dem eigentlichen Zugriff angefordert und danach wieder freigegeben. Ein anderer Thread, der ebenfalls auf diese Ressource zugreifen möchte, wird blockiert. Somit ist sichergestellt, dass ein Thread den Zugriff vollständig abschliessen kann.

Project	UART

Setzen Sie das Programm zuerst ohne Mutex um, damit Sie den Effekt sehen. Lösen Sie das Problem nachher mit einem Mutex.

- Erstellen Sie zwei Threads, beide mit gleicher Priorität.
- Beide Threads sollen jeweils die vorgegebene statische Variable **`count`** hochzählen und diese mittels **`printf()`** ausgeben.
- Verwenden Sie nach der Ausgabe die Funktion **`wait_blocking(HALF_SECOND)`** um die Ausgabe am Bildschirm verfolgen zu können.
- Mit **`printf()`** können Sie die Ausgabe noch formatieren. Weitere Informationen finden Sie unter https://www.tutorialspoint.com/c_standard_library/c_function_printf.htm.
- Für einen Zeilenumbruch müssen Sie **`\n\r`** verwenden, statt nur **`\n`**.
- Sorgen Sie dafür, dass bei jeder Ausgabe ersichtlich ist, von welchem Thread sie stammt.
- Zeigen Sie beide Versionen, mit und ohne Mutex.

8 Bewertung

Das Praktikum wird mit maximal 3 Punkten bewertet:

- Aufgabe – Scheduling 1 Punkt
  Funktionalität inclusive Plots und Erklärungen
- Aufgabe – Signals 1 Punkt
- Aufgabe – Mutexes 1 Punkt

Punkte werden nur gutgeschrieben, wenn die folgenden Bedingungen erfüllt sind:

- Der Code muss sauber strukturiert und kommentiert sein.
- Das Programm ist softwaretechnisch sauber aufgebaut.
- Die Funktion des Programmes wird erfolgreich vorgeführt.
- Der/die Studierende muss den Code erklären und zugehörige Fragen beantworten können.
- Der Zusatzpunkt wird nur vergeben, wenn alle anderen Aufgaben gelöst sind.

9 Anhang – STDOUT über UART

Um die Standardausgabe auf den UART1 Port zu legen, öffnen Sie *Project → Manage → Runtime Environment…*. Aktivieren Sie *Compiler → I/O → STDOUT* und wählen Sie im dazugehörenden Dropdown Feld *USER* aus (Siehe Abbildung 5).

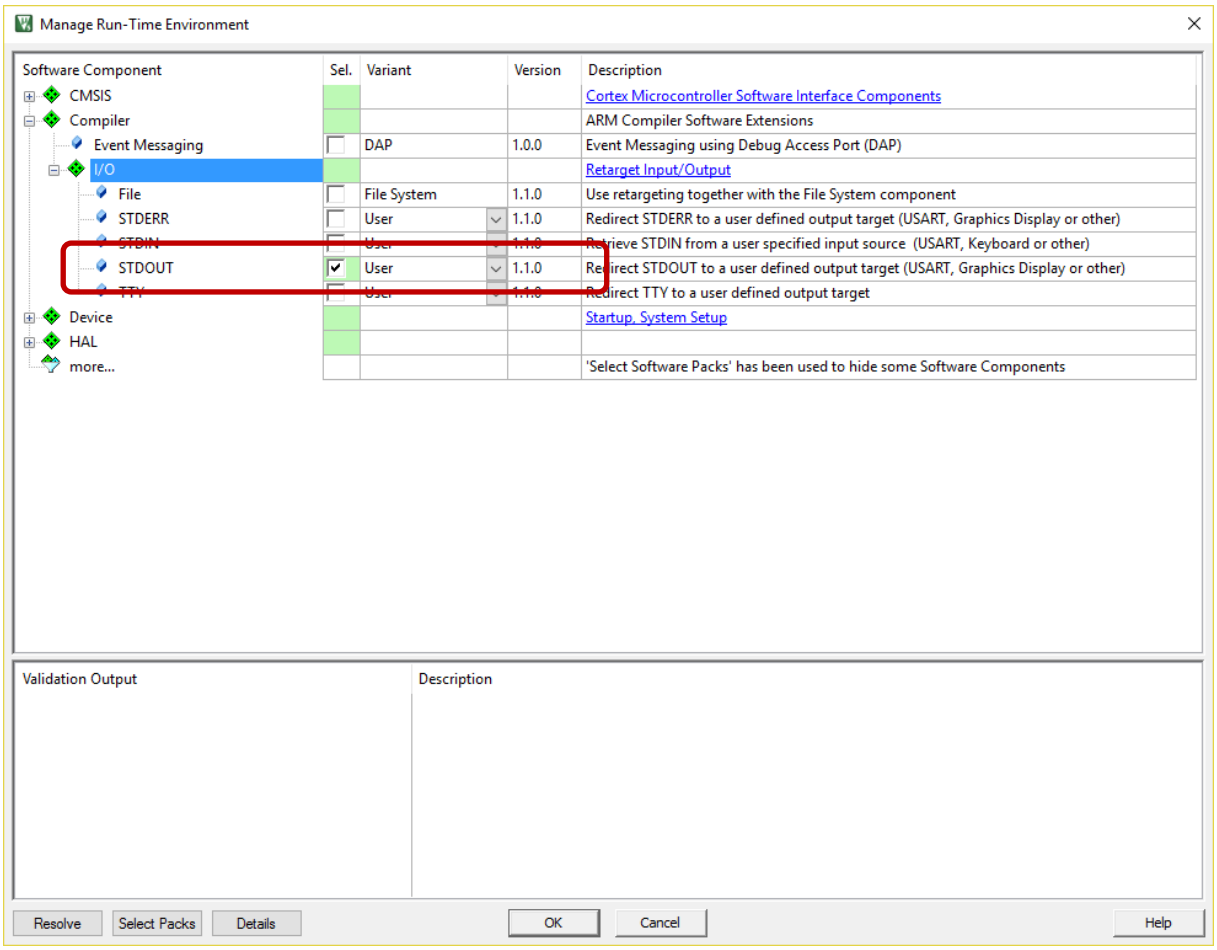


Abbildung 5: STDOUT aktivieren

Anschliessend müssen Sie den entsprechenden UART konfigurieren und eine spezielle Funktion `int stdout_putchar(int ch)` erstellen in welcher jeweils ein Zeichen ausgegeben wird. In diesem Projekt sind aber beide Punkte im Modul *uart.c* bereits implementiert.

10 Anhang – UART an PC anschliessen

Beim Discovery Version 1 (DISC1) ist UART1 über den ST-Link Debugger mit dem PC verbunden. Verwenden Sie den ST-Link Virtual COM Port (im Gerätemanager suchen, siehe Abbildung 6).

Öffnen Sie mit PuTTY eine Verbindung mit 115‘200 bps über den entsprechenden COM Port.

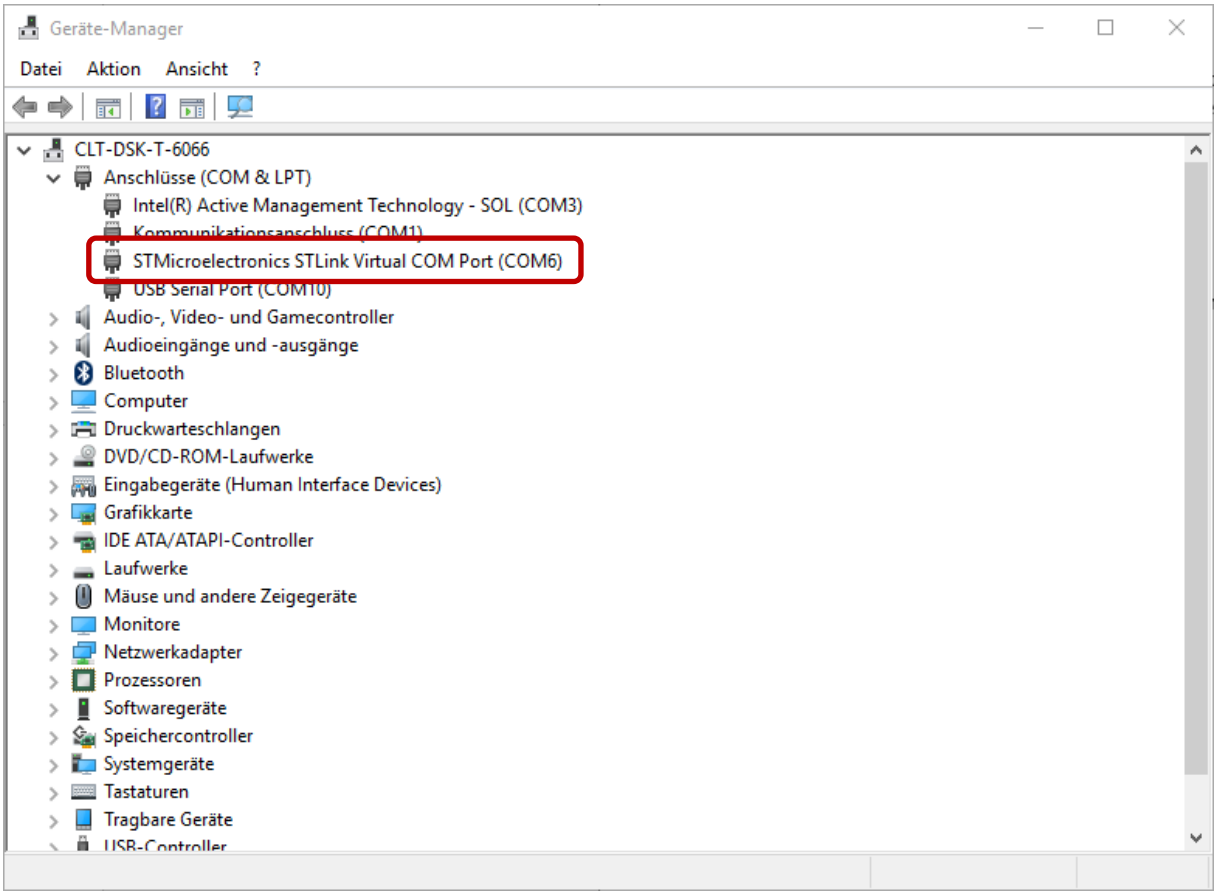


Abbildung 6: Geräte-Manager

Beim Discovery Board wurde zusätzlich die Lötbrücke SB9 verbunden, damit die Trace Funktionalität (u.a. für den Event Viewer benötigt) genutzt werden kann.

11 Anhang – Screenshots zu Aufgaben

11.1 Fall 1: Equal Priority – No Round-Robin

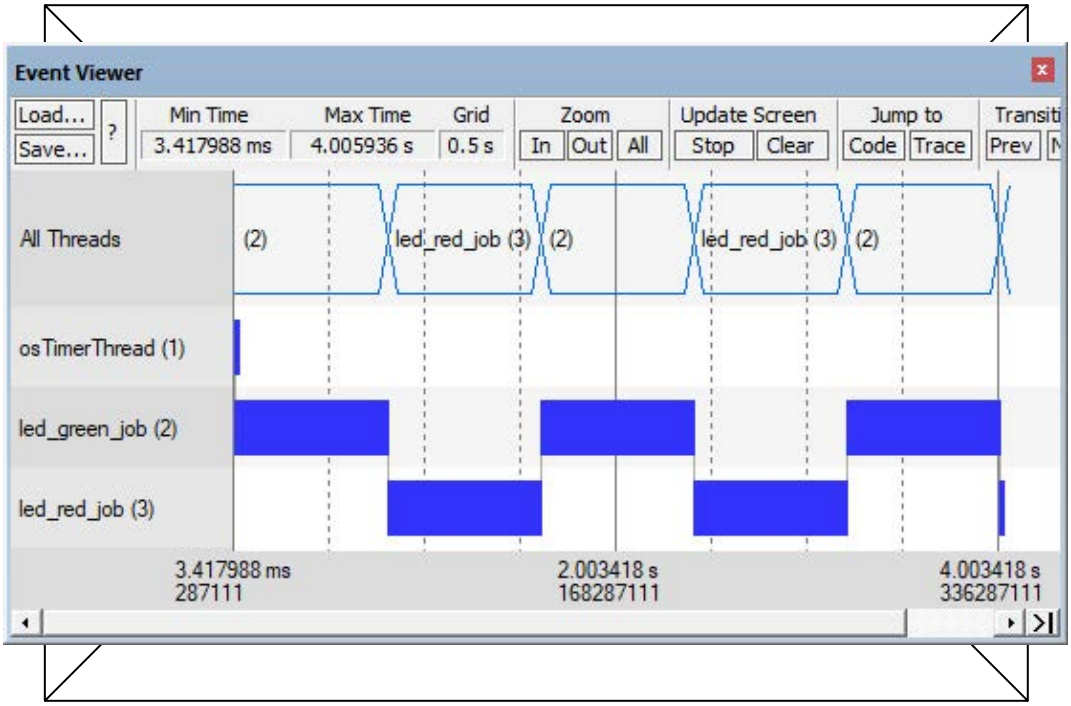


Abbildung 7: Fall 1: Equal Priority – No Round-Robin

Beschreibung des Verhaltens

Die Threads werden nacheinander ausgeführt.

Wie ändert das Verhalten bei unterschiedlichen Prioritäten der beiden Threads?

Nur der Thread mit der höchsten Priorität wird ausgeführt.

11.2 Fall 2: Equal Priority – With Round-Robin

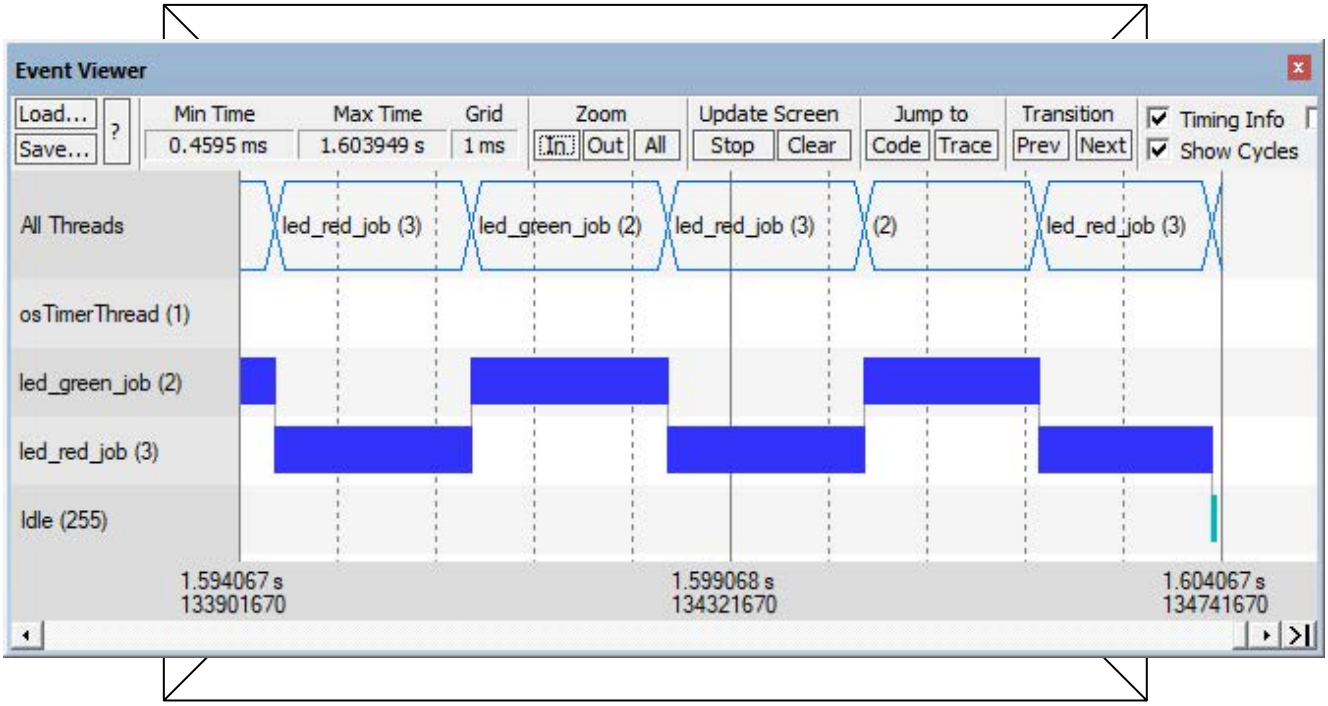
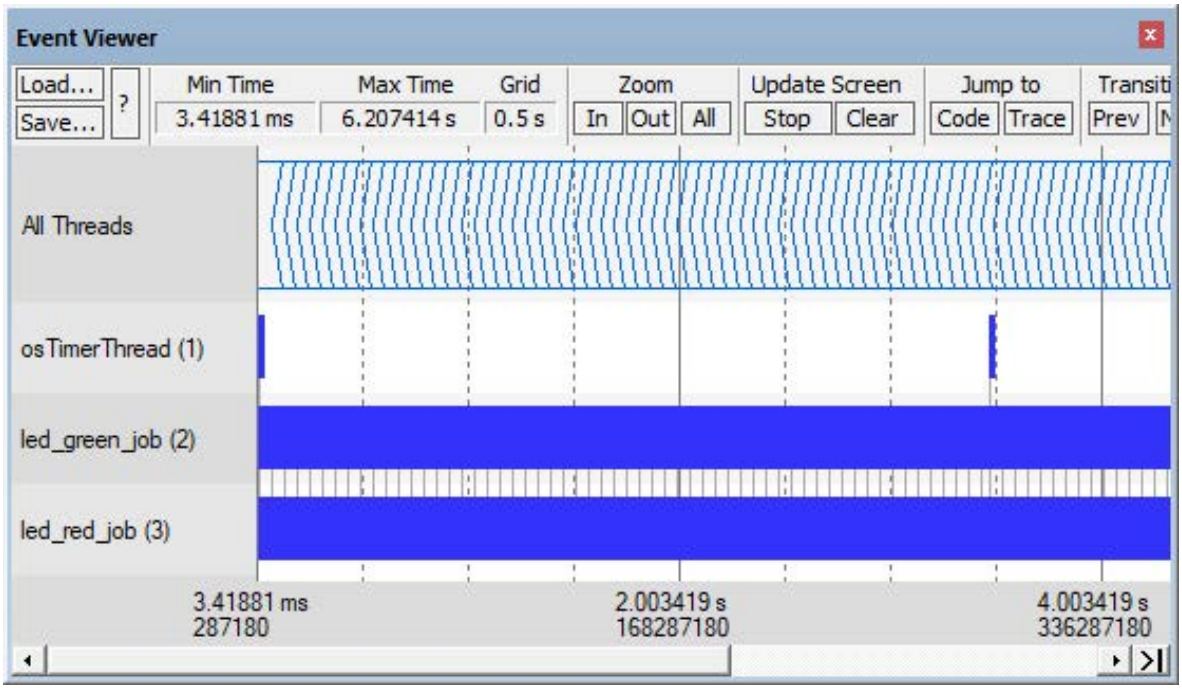


Abbildung 8: Equal Priority – With Round-Robin

Erklären Sie die Unterschiede zum Fall 1:

Die beiden Threads werden abwechslungsweise ausgeführt.



11.3 Messung Round Robin Timeout

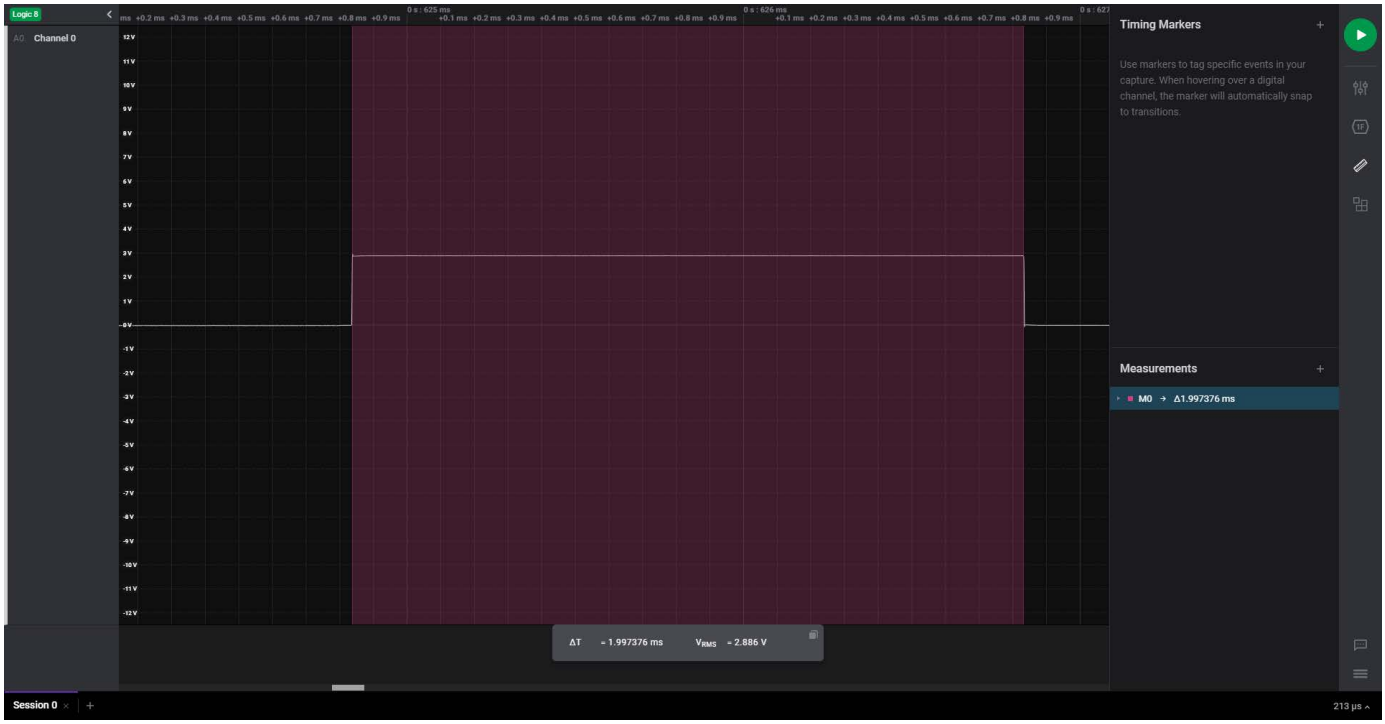


Abbildung 9: Output grüne LED (Port G.13)

Erklärungen:

Der Round-Robin Timeout ist 2 Ticks lang. Ein Tick ist 1ms. Somit dauert es 2ms, bis vom Einschalt-Thread zum Ausschalt-Thread gewechselt wird.

11.4 Fall 3: Different Priorities with osDelay()

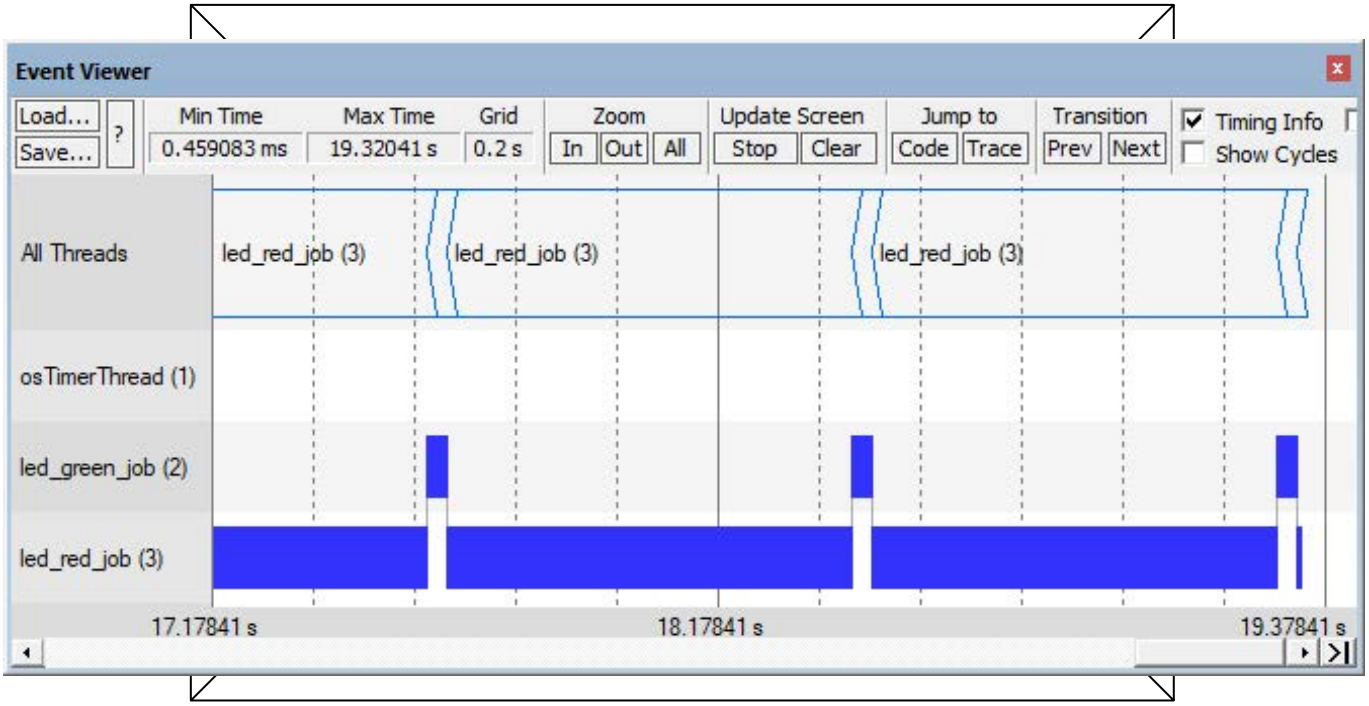


Abbildung 10: Fall 3: Different Priorities with osDelay()

Erklärungen:

Während des osDelays wird der niedrigerer priorisierte Thread ausgeführt. Sobald die osDelay vorüber ist, wird der höhere priorisierte Thread erneut ausgeführt.