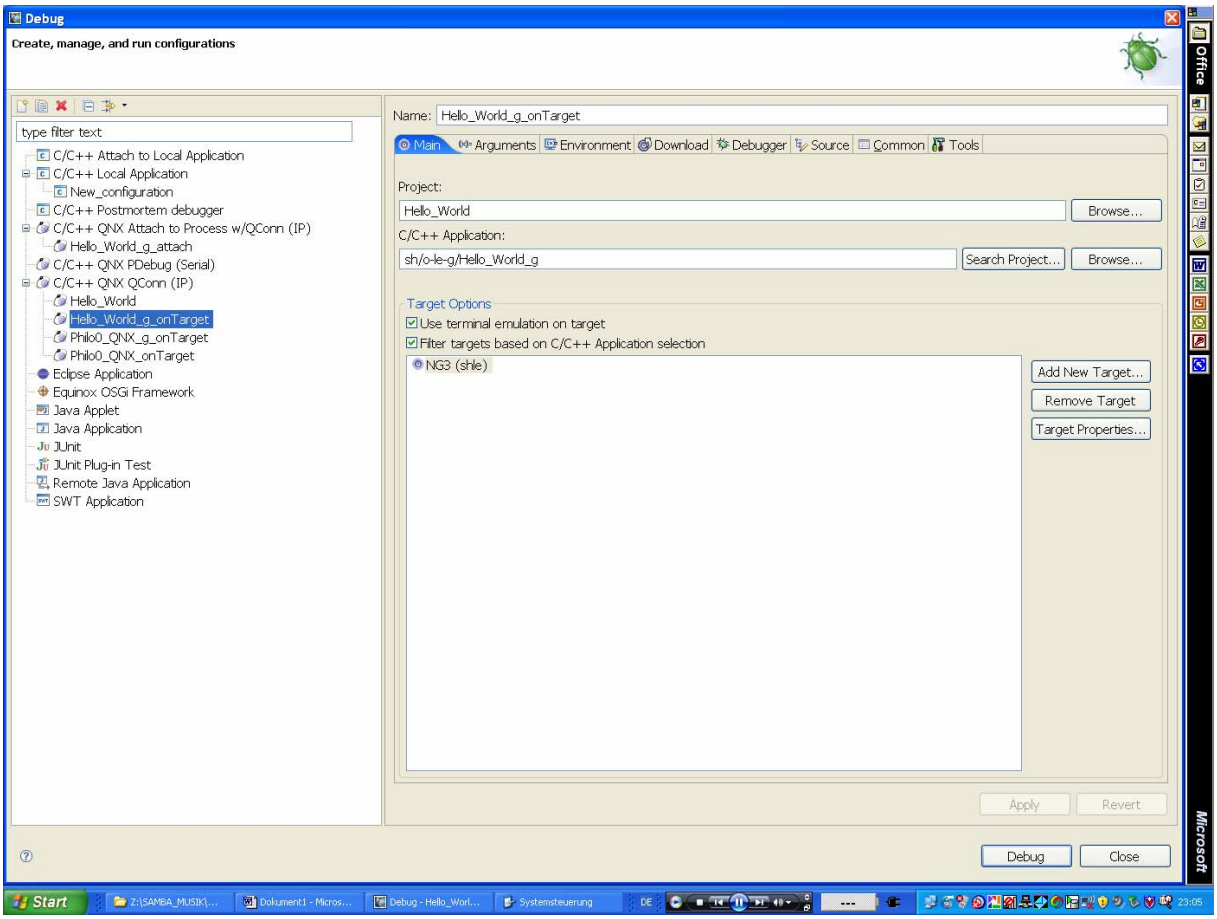
# Target-Debugging am Beispiel ’Hello World’

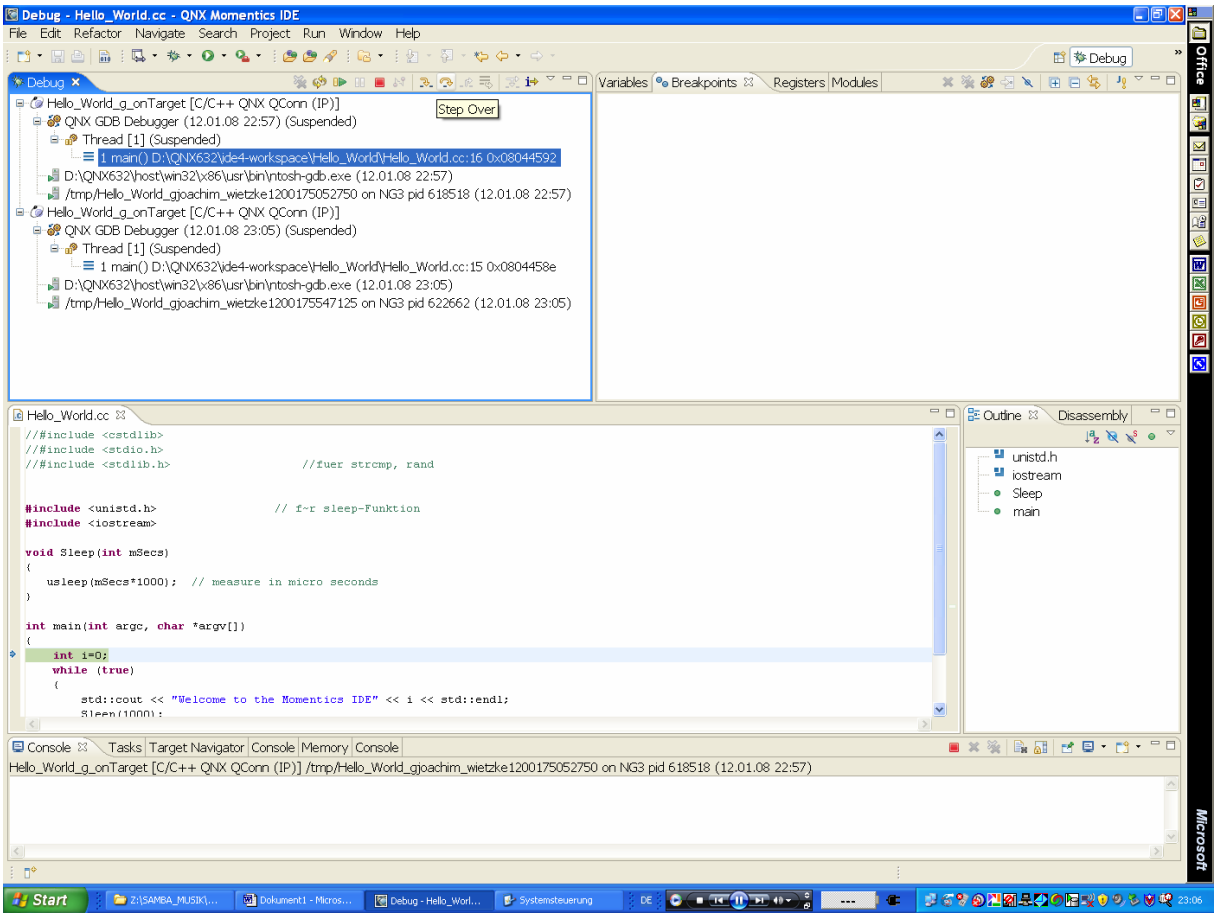
```
#include <unistd.h>                         // fuer sleep-Funktion
#include <iostream>

void Sleep(int mSecs)
{
   usleep(mSecs*1000);  // measure in micro seconds
}

int main(int argc, char *argv[])
{
      int i=0;
      while (true)
      {
            std::cout << "Welcome to the Momentics IDE" << i << std::endl;
            Sleep(100);
            i++;
      }
}
```

Hello World wird in einer Dauerschleife ausgegeben. Mit einer eingebauten Pause von 0,1 sec. Die usleep-Funktion nutzt QNX-nanosleep, wofür keine CPU-Last gebraucht wird.

Nutzen wir das Programm nur einmal, kann es mit normalen Debugg-Funktionen auf dem Target durchlaufen werden. Dazu wird über qconn Verbindung mit dem Target aufgenommen. Sollte qconn nicht als Standard auf dem Target schon laufen, muss es über Konsole manuell gestartet werden ( /usr/sbin/qconn).



Zum Beispiel mit der Step-Funktion kann das Programm untersucht werden, es kennen Breakpoints gesetzt werden der mit Watchpoints Daten überwacht werden.




Auch das Disassemblieren des C-Codes ist möglich.

Im nächsten Schritt stoppen wir das Programm mit dem Debugger und starten es erneut, diesmal im Debugger freilaufend (resume) oder per Konsole.
Nun können wir uns mit der View QNX System Information Details zur Last, zum Speicherverbrauch und andere Daten anschauen.
Es ist auch möglich, Kernel-Traces zu ziehen und sich Kernel-Events und Kommunikation zwischen Threads in verschiedenen Darstellungen anzuschauen.

# Debuggen am Philosophenproblem

Es sitzen fünf Philosophen an einem runden Tisch, auf dem fünf Teller randvoll gefüllt mit Spaghetti gedeckt sind, zwischen denen sich fünf einzelne Gabeln befinden, so gibt es ein Problem: ein Philosophenproblem! Philosophen können nur mit zwei Gabeln Spaghetti essen. Daraus folgt, dass nie alle Philosophen gleichzeitig essen können. Zum Essen braucht jeder Philosoph die Gabeln zu seiner linken und rechten Seite, hat er diese genommen, können seine beiden Tischnachbarn nicht essen. Den größten Teil des Tages beschäftigen sich Philosophen damit, tiefgründige Gedanken zu wälzen. So denken sie, nachdem sie sich zu Tisch gesetzt haben, erst einmal über Sinn und Unsinn der Nahrungsaufnahme nach. Jeder Philosoph braucht in der Regel unterschiedlich viel Zeit, wenn er sich mit dieser Frage zu beschäftigt. Kommt es jedoch einmal vor, dass alle Philosophen gleich

lange nachgedacht haben und anschließend alle gleichzeitig ihre linke Gabeln aufnehmen, dann kann keiner essen, weil ihm seine rechte Gabel fehlt. Diesen Zustand nennt man "deadlock". Die Philosophen würden in dieser Situation verhungern, weil keiner von ihnen auf die Idee käme, seine linke Gabel wieder hinzulegen.

Die Philosophen werden in 5 einzelnen Prozessen realisiert, in denen die CP#_Component.run Methoden die Hauptschleifen implementieren.
Die 5 Prozesse werden von einem Main-Prozess geforkt und anschließend über Run-Variablen freigegeben. Dieser Main-Prozess könnte anschließend die Watchdog-Überwachung der anderen Prozesse übernehmen. In unserem Fall schaltet er die Programme nach einer Zeit wieder aus.

Die Gabeln sind Mutexe im Shared Memory.
Zusätzlich gibt es für jeden Prozess eine Run-Variable, die für das Debugging hilfreich ist.

Jeder Prozess hat einen Kontext im Shared Memory und eine Kontext-Beschreibung, mit deren Hilfe das Shared Memory angelegt wird.

Zu beachten ist, dass das Shared Memory am Ende wieder geschlossen und freigegeben werden sollte, oder noch vor dem Erzeugen sicherheitshalber gelöscht wird.
Tut man dies nicht, bleiben das SHM und die Mutexe erhalten. Im nächsten Lauf könnte dann mit bereits gelockten Mutexen gestartet werden, was gelegentlich schwer zu finden ist.

Das Beispielprogramm ist auf meiner Homepage zu finden. Es ist mit vielen Testausgaben und mit User-Events gespickt, zur Demonstration während der Tafelübung, wie sich die Philosophen verklemmen können.

Eindrucksvoll ist, dass es bis zur Verklemmung einige Stunden dauern kann, der Beweis dafür, dass man viele Fehler nicht durch Testen finden kann. Man kann sie nur durch kompetente SW-Ingenieure vermeiden.

Zum Demo-Debuggen starten wir zwei Binaries_g per Konsole auf dem Targetsystem und attachen uns mit der QNX-IDE. Anschließend können wir den gewählten Prozess im Single-Step debuggen. Zu beachten ist, dass wir vor dem Single-Step Mode mit dem Debugger einmal auf einen Breakpoint laufen müssen, ein Fehler in der IDE oder im Zusammenspiel zwischen dem GDB und dem Prozessor.