

Proseminar OS Internals: Ausarbeitung

TinyOS: NesC

Simon E. Silva Lauinger

25. März 2011



Dieses Werk ist unter einem Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Germany Lizenzvertrag lizenziert. Um die Lizenz anzusehen, klicken Sie auf den [Link](#) oder schicken Sie einen Brief an Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

Einleitung

Ein Teil der Evolutionstheorie besagt, dass sich Lebewesen an neue Umweltbedingungen anpassen, um zu überleben. Dieser Prozess lässt sich auch auf die Entwicklung von Software übertragen, die auf einer speziellen Hardware laufen soll. Die Art der Softwareentwicklung muss den neuen Hardwaregegebenheiten angepasst werden, damit ein optimales Ergebnis erzielt werden kann. Vor allem im Bereich der Betriebssysteme ist dies unabdingbar, damit die Entwicklung guter Software überhaupt möglich wird. Das Thema, mit dem sich diese Ausarbeitung befasst, kommt aus dem Bereich der Sensornetze. Ein solches Netzwerk besteht aus einzelnen Knoten die, meist über Funk, miteinander kommunizieren können. Ein Beispiel dafür ist ein Multi-Hop-Netzwerk (Abbildung 1). Jeder Knoten kann nur mit einem Knoten in seiner Funkreichweite kommunizieren. Um eine Nachricht von einem Randknoten zur Basis zu übermitteln wird diese solange von Knoten zu Knoten weitergereicht bis sie am Ziel ankommt.

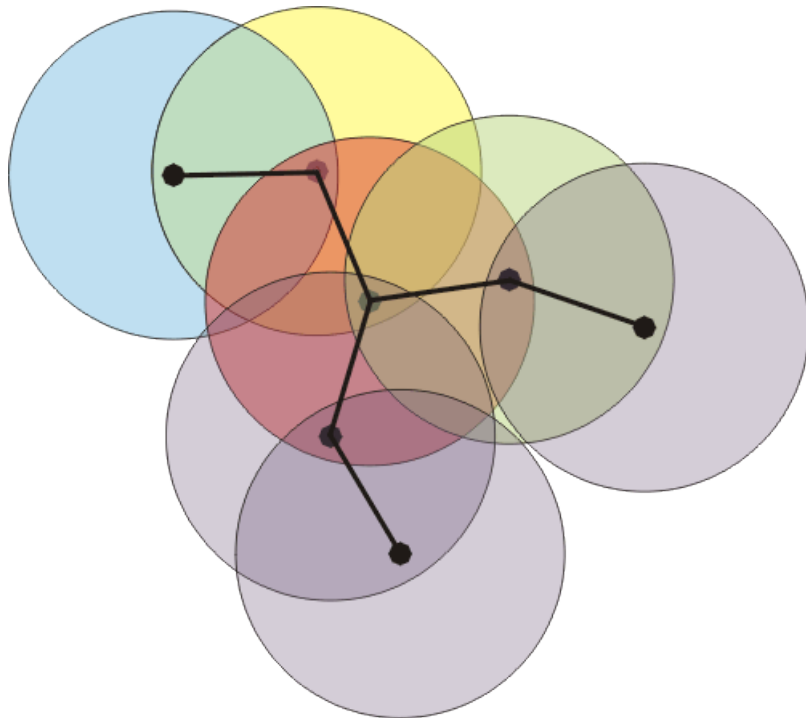


Abbildung 1: Ein über Funk kommunizierendes Multi-Hop-Netzwerk
http://upload.wikimedia.org/wikipedia/de/b/be/Adhoc_multihop.gif

Die einzelnen Knoten eines solchen Netzwerkes bestehen meist nur aus einem Sensor und einem Chip, auf dem ein Mikroprozessor, ein kleiner Speicher und die Funknetzwerkverbindung untergebracht sind. Da im Allgemeinen keine Controller vorhanden sind, muss jede Aktivität vom Mikroprozessor selbst durchgeführt werden. Auf

Grund der geringen Speicherkapazität muss ein Betriebssystem möglichst klein gehalten werden, damit eine Anwendung überhaupt untergebracht werden kann. Durch die geringe Größe der Knoten und damit auch der Batterie ist ein effizientes Energiemanagement keinesfalls zu vernachlässigen. Durch diese speziellen Anforderungen werden herkömmliche Betriebssysteme als Softwareplattform unbrauchbar. Ein vielversprechender Ansatz diesen Anforderungen gerecht zu werden, ist das Betriebssystem TinyOS[[tinyoshp](#)], mit dessen eigens dafür entwickelten Programmiersprache NesC[[neschp](#)] sich diese Ausarbeitung beschäftigt.

1 TinyOS & NesC

TinyOS ist ein Betriebssystem, das speziell für die Knoten von Sensornetzwerkssystemen entwickelt wurde. Es wird unter der von der OSI anerkannten BSD Lizenz von der Berkley University of California seit dem Jahr 2000 entwickelt. Durch seine Modularität und seine geringe Komplexität ist es optimal für Mikroprozessoren mit geringem Hauptspeicher geeignet. Die Programmiersprache NesC, in der TinyOS geschrieben ist, macht es auch für Softwareentwickler einfach, das Modularitätsprinzip bei den eigenen Anwendungsprogrammen umzusetzen. Das Programmiermodell von NesC ist sowohl einfach zu verstehen wie auch effektiv und bringt dabei die Mächtigkeit von C mit, auf dessen Basis es aufgesetzt ist. Im Gegensatz zu objektorientierten Programmiersprachen fällt bei NesC der Speicherplatz, den Vererbung und Polymorphie mit sich bringen, weg. Dadurch können Kosten eingespart werden, die ansonsten für schnellere Mikroprozessoren mit mehr Arbeitsspeicher anfallen würden. Auch der benötigte Stromverbrauch wird damit auf ein Minimum reduziert.

TinyOS und NesC ergänzen sich hervorragend, da die Prinzipien, mit denen TinyOS entwickelt wurde, auch für die Programme, die auf diesem Betriebssystem laufen, durch NesC gewahrt werden.

2 Nebenläufigkeit in TinyOS

Die Nebenläufigkeit in TinyOS ist sehr simpel gehalten. Es ist ein ereignisgetriebenes Betriebssystem, welches keine parallele Ausführung von Code kennt. Es wird lediglich zwischen synchronem (Tasks) und asynchronem Code (Routinen für die Hardwareereignisbehandlung) unterschieden. Tasks sind dabei Funktionen, die zeitversetzt ausgeführt werden. Einmal gestartet, läuft ein Task immer bis zum Ende und kann von keinem anderen Task unterbrochen werden. Routinen für die Hardwareereignisbehandlung können laufende Tasks unterbrechen. Dies sind *commands* oder *events*, die mit dem *async* Schlüsselwort gekennzeichnet sind und nur von Hardwareinterrupts gestartet werden können. Auch sie laufen bis zum Ende ab und können nur durch den Start anderer Hardwarebehandlungsroutinen durch Hardwareinterrupts unterbrochen werden.

TinyOS besitzt einen einfachen nicht-preemptiven Scheduler. Er ist als eine FIFO-Liste von Tasks implementiert, das heißt die Tasks werden nach der Reihenfolge ihres Eintreffens in der Liste ausgeführt (siehe Abbildung 2). Enthält der Scheduler keine weiteren Tasks mehr, wird das Betriebssystem in einen Stromsparmmodus überführt, bis es durch einen Hardwareinterrupt aufgeweckt wird.

Wurde ein Hardwareinterrupt ausgelöst, wird der aktuell laufende Task unterbrochen und das asynchrone Event aufgerufen welches als Behandlungsroutine für den Interrupt registriert ist. Nach Ablauf dieser Routine wird der Task an der unterbrochenen Stelle fortgesetzt (siehe Abbildung 3). Um die Unterbrechung des laufenden Tasks so kurz wie möglich zu halten ist es zudem ratsam, in der Behandlungsroutine möglichst wenig Code auszuführen und stattdessen einen neuen Task zu erzeugen (zu *posten*). Zusätzlich ist es möglich aber nicht empfehlenswert, ein asynchrones *command* aufzu-

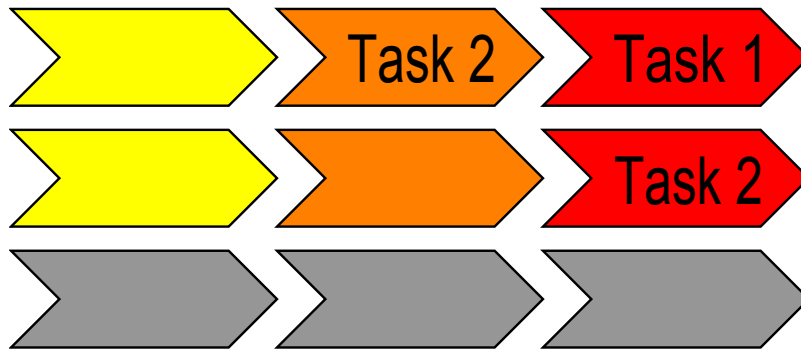


Abbildung 2: Zwei Tasks werden hintereinander ausgeführt. Nachdem sie beendet wurden, fährt das Betriebssystem in den Stromsparmodus.

rufen.

Auf Grund der Nichtunterbrechbarkeit von Tasks sind diese untereinander vor Wett-

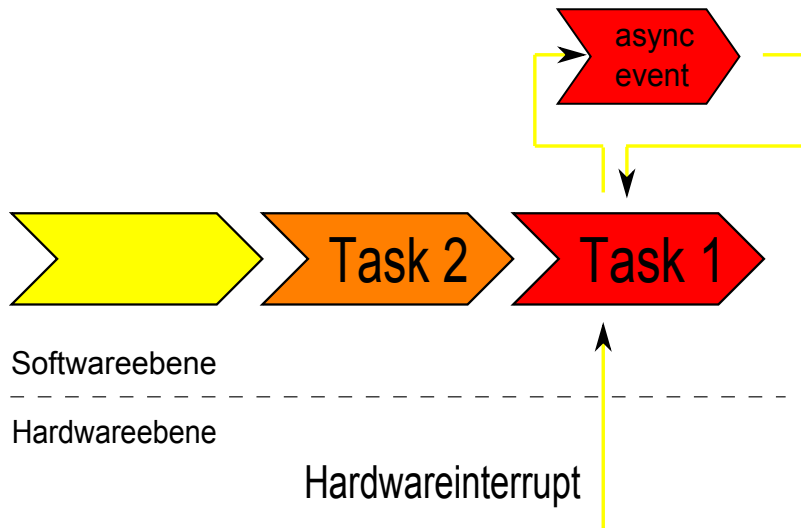


Abbildung 3: Ein laufender Task wird durch einen Hardwareinterrupt unterbrochen, die Routine für die Hardwareereignisbehandlung wird abgearbeitet und der unterbrochene Task wird fortgesetzt.

laufsituationen geschützt. Jedoch kann es zwischen asynchronem und synchronem sowie zwischen asynchronem und asynchronem Code immer noch zu Wettlaufsituationen kommen. Allerdings können sie nur beim Zugriff auf gemeinsamen Speicher auftreten. Darum wird NesC das Prinzip angewandt, dass nur synchroner Code in den gemeinsamen Speicher schreiben darf. Soll in gemeinsamen Speicher geschrieben werden, sollten

alle Zugriffe als atomar gekennzeichnet werden. Der Compiler warnt vor vergessener Kennzeichnung.

Der TinyOS Compiler kann Wettlaufsituationen aufspüren. Er gibt beim Kompilieren eine Warnung aus, sollte eine solche Situation auftreten. Da es möglich ist, dass der Compiler fälschlicherweise eine Wettlaufsituation erkennt, kann diese Warnung durch die Kennzeichnung der entsprechenden Variable mit dem Schlüsselwort *norace* unterdrückt werden.

3 NesC Programmmodell

Ein Applikation in NesC ist ein Graph aus Interfaces und Komponenten. Interfaces spezifizieren *commands* und *events*. Eine Komponente implementiert ein oder mehrere Interfaces. In dieser Ausarbeitung betrachten wir das Beispielprogramm „Blink“, anhand dessen die einzelnen Elemente des Programmgraphen vorgestellt werden. Blink ist ein einfaches Programm das jede Sekunde eine LED aufleuchten bzw. abblenden lässt.

Die Benennung der einzelnen Elemente des Programmgraphen folgen einer bestimmten Namenskonvention. Dabei sind Elemente, die mit „C“ enden, Konfigurationen und Elemente, die auf „M“ enden, Module. Interfaces haben keine spezielle Endung.

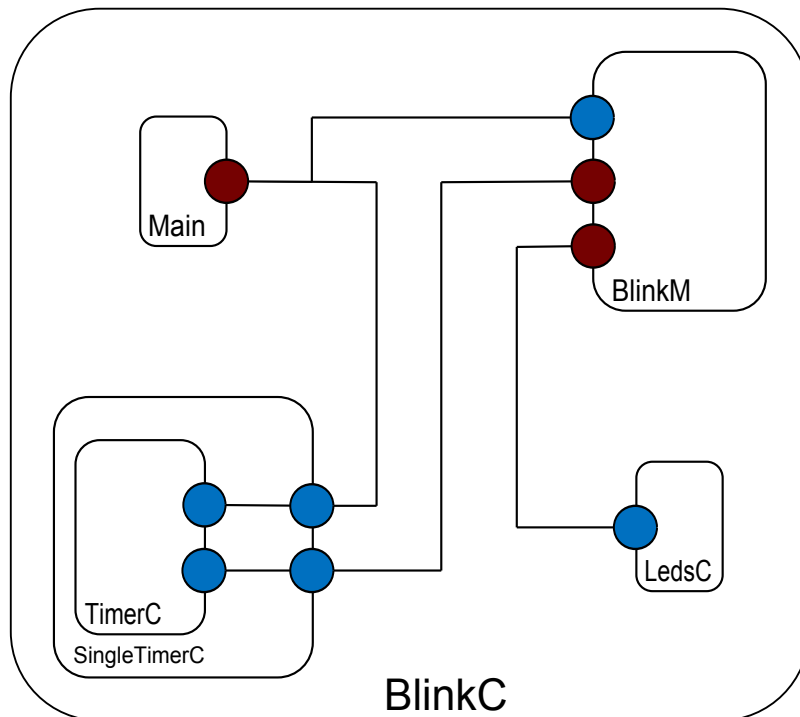


Abbildung 4: Struktur des Beispielprogramms „Blink“. Die ausgefüllten Kreise stellen Interfaces dar (ein blauer Kreis stellt ein Interface bereit, ein roter ist ein genutztes). Abgerundete Rechtecke sind Komponenten. Die Linien zwischen Interfaces repräsentieren eine Konfiguration innerhalb einer Komponente.

4 Interfaces

Interfaces spezifizieren die bidirektionale Kommunikation zwischen zwei Komponenten, einem Anbieter und einem Nutzer. Sie definieren benannte Funktionen die entweder vom Anbieter (*commands*) oder vom Nutzer (*events*) des Interfaces implementiert werden müssen. Ein Interface hat folgende Struktur:

```
interface: interface interfacename { funktionsliste }
funktionsliste: ((command | event) funktionssignatur))*
```

Die beiden Interfaces in unserem Beispielprogramm sind „Timer“ und „StdControl“. „Timer“ wird von der Konfiguration „SingleTimerC“ bereitgestellt und von Modul „BlinkM“ verwendet (siehe Abbildung 5). Die Deklarationen des Interfaces sind in Listing 1 zu sehen.

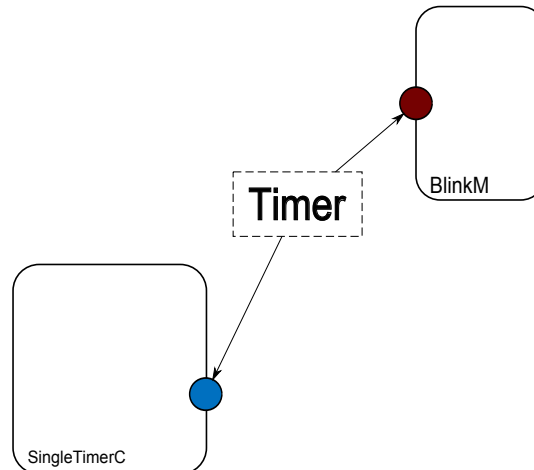


Abbildung 5: Das Interface „Timer“ wird von „SingleTimerC“ bereitgestellt und von „BlinkM“ verwendet.

Listing 1: Das Interface „Timer“. `start(...)` startet einen Timer (entweder wiederholend oder einmalig) der nach *interval* das event `fired()` auslöst. Mit `stop()` kann er vorzeitig gestoppt werden.

```

1 interface Timer {
2     command result_t start(char type, uint32_t interval);
3     command result_t stop();
4     event result_t fired();
5 }

```

Die Funktionen **command** `result_t start(char type, uint32_t interval)` und **command** `result_t stop()` werden also vom Bereitsteller des Interfaces (hier „SingleTimerC“) implementiert. Die Funktion **event** `result_t fired()` muss von „BlinkM“ implementiert werden.

Ein sehr spezielles Interface ist „StdControl“. Es wird verwendet, um Komponenten zu initialisieren und zu starten. In Listing 2 sind die Funktionen von „StdControl“ gezeigt.

Listing 2: Die Funktionen des Interfaces „StdControl“.

```

1 interface StdControl {
2     command result_t init();
3     command result_t start();
4     command result_t stop();
5 }

```

Jedes NesC Programm verwendet dieses Interface mindestens einmal. Zum Starten eines Programms ist die Konfiguration „Main“ notwendig. Sie nutzt „StdControl“. Der erste Funktionsaufruf eines Programms ist „Main.StdControl.init()“. Um das Pro-

programm nach der Initialisierung zu starten und zu beenden, sind *start()* und *stop()* zu verwenden. Um sonstige Komponenten zu initialisieren, müssen die Interfaces miteinander verbunden werden (mehr dazu in Abschnitt 6).

5 Module

Module stellen den Code bereit, der ein Interface implementiert. Ein Modul kann mehrere Interfaces benutzen oder bereitstellen. In „Blink“ ist das einzige Modul „Blink“ das in Abbildung 6 gezeigt und dessen Programmcode in Listing 3 zu sehen ist.

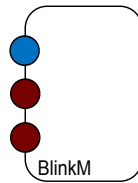


Abbildung 6: Das Modul „BlinkM“ stellt das Interface „StdControl“ bereit und nutzt „Timer“ und „Leds“.

Listing 3: Der Programmcode des Moduls „BlinkM“.

```

1 module BlinkM {
2     provides interface StdControl;
3     uses {
4         interface Timer;
5         interface Leds;
6     }
7 }
8
9 implementation {
10    command result_t Std.init() {
11        call Leds.init();
12        return SUCCESS;
13    }
14    command result_t Std.start(){
15        return call Timer.start(TIMER_REPEAT, 1000);
16    }
17    command result_t Std.stop() {
18        return call Timer.stop()
19    }
20    event result_t Timer.fired() {
21        call Leds.redToggle();
22        return SUCCESS;
23    }

```

Ein Modul hat zwei Abschnitte. Der erste definiert, welche Interfaces verwendet und bereitgestellt werden. Er wird mit dem Schlüsselwort *module* eingeleitet, dem der Name des Moduls folgt. Mit *provides* werden die bereitgestellten Interfaces gekennzeichnet, mit *uses* die verwendeten. Mehrere verwendete oder bereitgestellte Interfaces können in geschweiften Klammern zusammengefasst werden, vor denen dann das Schlüsselwort nur einmal vorkommen muss (siehe Listing 3).

Der zweite Abschnitt ist die tatsächliche Implementierung der Interfaces und wird mit dem Schlüsselwort *implementation* eingeleitet. Im Beispiel sind die *commands* die Funktionen, die durch das Interface „StdControl“ bereitgestellt werden. *init()* initialisiert die Leds (auch „LedsC“ implementiert „StdControl“). Die Funktion *start()* startet den Timer als einen sich wiederholenden Timer, der alle 1000 Millisekunden das Event *fired()* auslöst. *stop()* hält den Timer schließlich an.

Da das Interface „Timer“ verwendet wird, muss auch das Event *fired()* implementiert werden. Hier soll es die rote LED an- beziehungsweise abschalten.

6 Konfigurationen

Eine Konfiguration verbindet Interfaces von Komponenten. Die miteinander verbundenen Komponenten bilden eine neue Komponente. Es können allerdings nur gleiche Elemente miteinander verbunden werden (also Interface mit Interface, *command* mit *command* und *event* mit *event*). Außerdem müssen *commands* und *events*, sollen sie miteinander verbunden werden die gleiche Funktionssignatur aufweisen, Interfaces müssen den gleichen Typ haben. Zudem ist es möglich auch nur einzelne *commands* bzw. *events* miteinander zu verbinden.

Elemente können auf verschiedene Arten miteinander verbunden werden. Zwei Elemente können gleich gesetzt (mit „=“) oder gebunden (mit „→“ oder „←“) werden. Zwei Elemente gleichzusetzen bedeutet sie äquivalent zu machen. Gebunden werden können nur zwei Elemente von denen das eine bereitgestellt und das andere verwendet wird. Es wird immer ein genutztes mit einem bereitgestellten verbunden (entweder *nutzendes Element* → *genutztes Element* oder *genutztes Element* ← *nutzendes Element*). Die beiden Konfigurationen in unserem Beispiel sind „SingleTimerC“ und das eigentliche Programm „BlinkC“, zu sehen in Abbildung 8 und 4 (Seite 7). Die Konfiguration wird dort mit den Verbindungslinien zwischen den Komponenten dargestellt.

Wie schon bei den Modulen werden auch Konfigurationen in zwei Abschnitte aufgeteilt. Der erste Abschnitt wird mit *configuration* eingeleitet und definiert die bereitgestellten Interfaces einer Konfiguration. Der zweite Abschnitt (eingeleitet mit *implementation*) verbindet die genutzten Komponenten miteinander. Innerhalb werden dabei nach dem Schlüsselwort *components* die Namen der verwendeten Komponenten angegeben. Daraufhin werden sie mit „→“, „←“ und „=“ miteinander verbunden. Listing 4 und 5 zeigen die Implementierung der Komponenten „SingleTimerC“ und „BlinkC“.

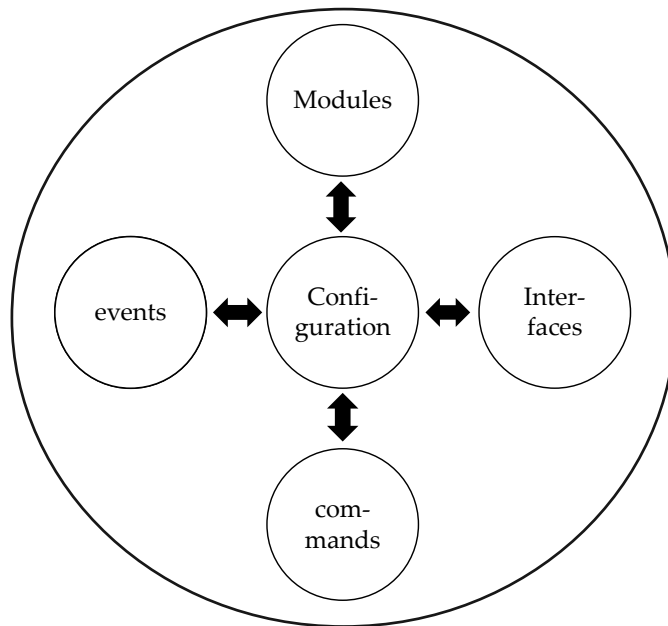


Abbildung 7: Eine Konfiguration verbindet einzelne Elemente zu einer Komponente.

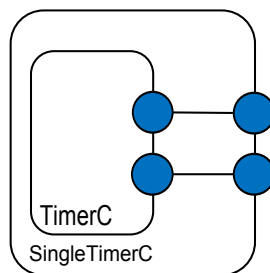


Abbildung 8: Die Konfiguration „SingleTimerC“.

Listing 4: Die Implementierung der Konfiguration „SingleTimerC“.

```

1 configuration SingleTimerC {
2   provides interface Timer;
3   provides interface StdControl;
4 }
5 implementation {
6   components TimerC;
7   Timer = TimerC.Timer[unique("Timer")];
8   StdControl = TimerC;
9 }

```

„SingleTimerC“ ist nur ein Stellvertreter für „TimerC“. Es beinhaltet einen einzelnen, eindeutigen Timer. Da es ein Stellvertreter ist, stellt es die gleichen Interfaces bereit wie „TimerC“, mit dem Unterschied, dass „TimerC“ das Interface „Timer“ mehrmals bereitstellt. In Zeile 7 wird ein eindeutiges (gewahrt durch das Compilermakro *unique(...)*) Interface „Timer“ mit dem bereitgestellten Interface „Timer“ gleichgesetzt. Das heißt ein Aufruf von „SingleTimerC.Timer.*“ wird immer an `TimerC.Timer[unique("Timer")]` weitergeleitet. Auch die Aufrufe von „SingleTimerC.StdControl“ werden an „TimerC“ weitergeleitet.

Listing 5: Die Implementierung der Konfiguration „BlinkC“.

```

1 configuration BlinkC { }
2 implementation {
3     components Main, BlinkM, SingleTimerC, LedsC;
4     Main.StdControl -> SingleTimerC.StdControl;
5     Main.StdControl -> BlinkM.StdControl;
6     BlinkM.Timer -> SingleTimerC.Timer;
7     BlinkM.Leds -> LedsC;
8 }

```

Das Herzstück des Programms „Blink“ liegt in „BlinkC“. Hier werden wirklich alle verwendeten Komponenten miteinander verbunden. Die bereitgestellten Interfaces „StdControl“ von „SingleTimer“ und „BlinkM“ werden mit dem von „Main“ verknüpft. Damit werden, wenn zum Beispiel „Main.StdControl.init()“ aufgerufen wird, die *init()* Funktionen von „SingleTimerC“ und „BlinkM“ mit aufgerufen. Das gleiche gilt natürlich auch für *start()* und *stop()* von „StdControl“.

Weiterhin wird das Interface „Timer“, bereitgestellt von „BlinkM“, mit „SingleTimerC.Timer“ und „BlinkM.Leds“ mit „LedsC“ verbunden.

7 Kompilieren

Die Programmdateien eines NesC Programms sollten immer mit der Dateiendung „.nc“ gekennzeichnet werden. Um ein NesC Programm zu übersetzen, werden die „*.nc“ Dateien vom NesC-Compiler in „.c“ Dateien konvertiert, um dann mit einem C-Compiler wie gcc in den Binärcode der entsprechenden Architektur überführt zu werden.

Literatur

- [tinyoshp] TinyOS Webseite, August 2010, <http://www.tinyos.net/>
- [tinyostutorial] TinyOS Tutorial, 23.September 2003, <http://www.tinyos.net/tinyos-1.x/doc/tutorial/>
- [neschp] NesC Webseite, August 2010, <http://nesc.sourceforge.net/>
- [nescreferenz] David Gay, Philip Levis, David Culler, Eric Brewer, *nesC 1.1 Language Reference Manual*, Mai 2003, <http://nesc.sourceforge.net/papers/nesc-ref.pdf>
- [nesctutorialshort] J.Hill, R.Szewczyk, A.Woo, S.Hollar, D.Culler, K.Pister, *System Architecture Directions for Networked Sensors*, http://www.ece.uah.edu/~jovanov/CPE621/notes/NesC_Tutorial_short.ppt
- [paperrenner] Christian Renner, Betriebssysteme für Sensornetze, http://www.net2.uni-tuebingen.de/fileadmin/RI/teaching/seminar_mobil/ss04/papers/paper-renner.pdf