TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

ambient INTELLIGENCE
Kaiserslautern

Berichte des Forschungsschwerpunkts
Ambient Intelligence

Nr. 3

Laufzeitmessungen in MICA2-Systemen

Thomas Kriz
Oliver Gabel

Inhalt

1. Übersicht

Dieser Bericht soll einen grundlegenden Überblick über das MICA2-System von Crossbow bieten.
Als Quellen dienen dabei hauptsächlich die von Crossbow ausgelieferten Komponenten.
Für ein problemloses Arbeiten werden folgende Komponenten benötigt:

- mehrere MICA2-Motes (Batterien im Lieferumfang enthalten)
- 1x MIB500CA Programming-Board
- 1x Parallelportkabel (Druckerkabel)
- 1x Serielles Kabel (NICHT Nullmodem)
- Stabilisiertes Netzteil (5-15 VDC)
- evtl. Internetzugang während der Installation
-

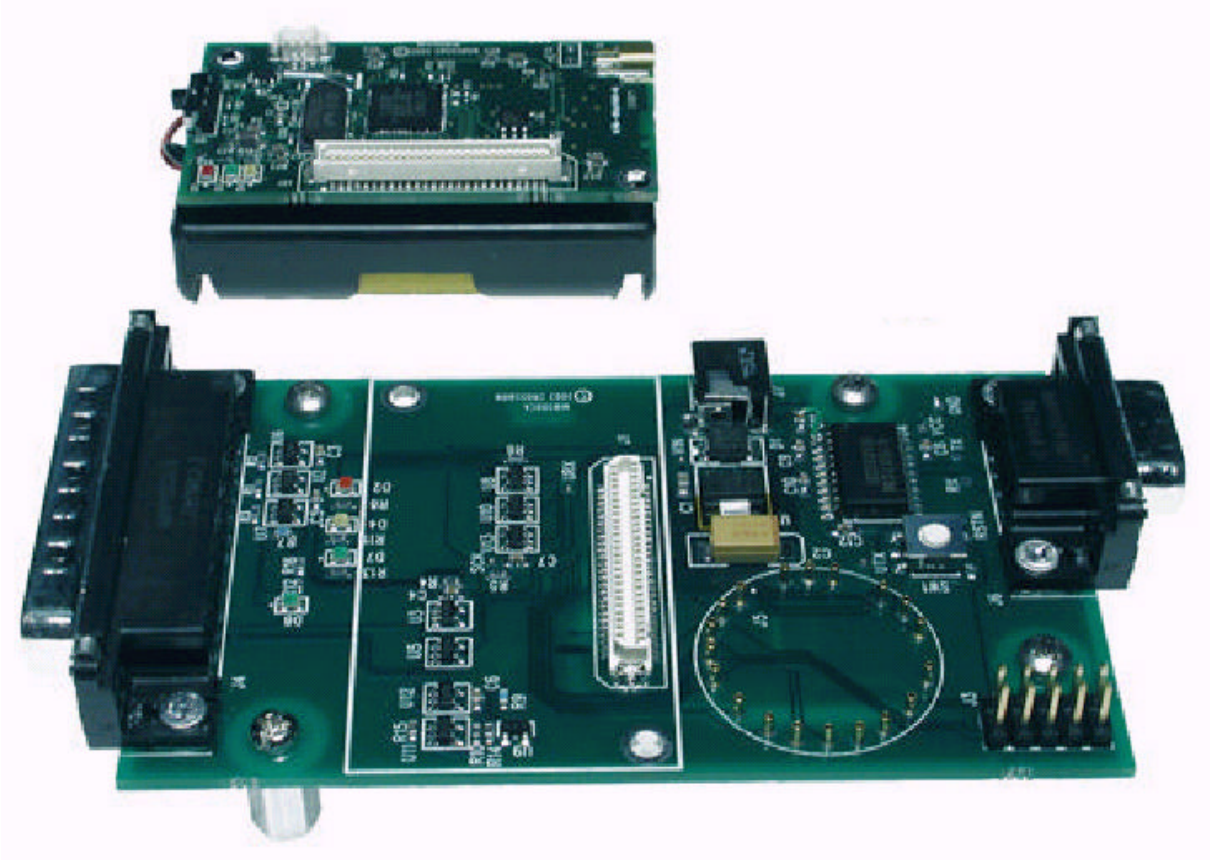

Abb. 1 (MICA2 oben; MIB500CA unten)

Es wird zunächst auf den Installationsvorgang eingegangen, dann kurz die Grundidee des Sensornetzwerk-Betriebssystems TinyOS beschrieben und anschließend eine Einführung in die Programmierung eines TinyOS-Systems gegeben.
Im letzten Teil befinden sich Versuchsbeschreibungen von selbstdurchgeführten Experimenten bezüglich verschiedener Transitionszeiten in MICA2-Netzwerken.
Dabei werden auch das Übertragungsprotokoll und die zugehörigen Algorithmen vorgestellt.

2. Installation

2.1 Auslieferungsumfang

Die Crossbow-CD bietet sämtliche benötigten Ressourcen um eine vollständige, lauffähige Entwicklungsumgebung für das MICA2-System zu schaffen.
Einzige Voraussetzung ist ein Internetanschluss während der Installation.
Auf den Installationsvorgang selbst wird weiter unten eingegangen.
Im Lieferumfang der Crossbow-CD enthalten sind

- das Betriebssystem TinyOS
- der avrgcc-Compiler (GNU)
- Java-Applikationen (u.a. Parallelport-Treiber)
- Crossbow Firmware (u.a. Hardware-Test-Software)
- Komplettsystemlösungen für verschiedene Anwendungen (Software)
- Tutorials und Dokumentationen

2.2 Installationsvorgang

Auf der CD befindet sich ein Ordner „Manuals und Docs“, und dort eine Datei „Readme1st.pdf“, die zwingend beachtet werden sollte. Sie enthält eine StepByStep Anleitung für die Installation. Hiermit werden sämtliche mitgelieferten Software-Komponenten, sowie die nötigen Updates in der korrekten Reihenfolge installiert.

Achtung: Es ist zwingend notwendig, volle Administrationsrechte auf dem PC, auf dem die Installation vorgenommen werden soll, zu haben. Ist dies nicht der Fall, können keine Registraturen angelegt werden, und die Installation des Parallelport-Treibers schlägt fehl. Es kann in solchen Fällen notwendig sein, das gesamte teilinstallierte System wieder zu deinstallieren.

Es empfielt sich, stets die vorgeschlagenen Installationspfade zu verwenden, da sonst evtl. Pfadeintragungen per Hand gemacht werden müssen.

Achtung: In Schritt 10 der Anleitung wird das Avrgcc-Update installiert. Der vom InstallWizard vorgeschlagene Pfad lautet ‚C:\winavr\’. Dies ist jedoch **nicht** der Pfad, der von den vorherigen Schritten des Tutorials angelegt wurde. Es ist notwendig, den vorgeschlagenen Pfad in ‚C:\avrgcc\’ abzuändern um ein ordnungsgemäßes Funktionieren der Software sicher zu stellen!
Des Weiteren wird während des Tutorials (das übrigens diesem Bericht beiliegt) die Funktion des Parallelport-Treibers überprüft. Sollte in Schritt 16 bei Eingabe der folgenden Zeilen

```
uisp - - version
uisp – dprog=dapa - -rd_fuses
```

eine Fehlermeldung auftreten, so ist bei jedem Aufruf des Parallelporttreibers der Suffix ‘-dlpt=x’, wobei x standardmäßig 1 gesetzt ist, jedoch bei manchen PC-Konfigurationen 2 oder 3 sein muss.
Der entsprechende Aufruf sähe dann folgendermaßen aus:

```
uisp – dprog=dapa -dlpt=3 - -rd_fuses
```

Achtung: Sollte der Standardaufruf des Parallelport-Treibers nicht funktionieren, so ist es notwendig, das Suffix '-dlpt=x' in die Datei ‚Makerules' in ‚C:\tinyos\apps\' einzutragen. Diese Datei wird vor jedem Kompilieren der selbsterstellten Anwendungen im ‚Apps'-Ordner ausgeführt. Sie enthält u.a. Suffixe für den Compiler, Aufruf des Parallelport-Treibers, etc. Die im ‚Readme1st.pdf'-Tutorial über den Aufruf von
'uisp – dprog=dapa -dlpt=x - -rd_fuses'
ermittelte Zahl x wird nun in Zeile 23 von 'Makerules' wie folgt abgeändert:
‚PROGRAMMER_EXTRA_FLAGS := -dlpt=x'
Hierdurch ist gewährleistet, dass die erstellte Anwendungssoftware auf die MICA2-Motes aufgespielt werden kann.

Zu beachten ist die Einstellung für die Transmitting-Frequenz. Die Datei ‚C:\tinyos-1.x\tos\platform\mica2\CC1000Const.h' sollte ab Zeile 201 folgendermaßen lauten:

```
#define CC1K_433_002_MHZ           0x00
#define CC1K_916MHZ                0x01
#define CC1K_434_845_MHZ           0x02
#define CC1K_914_007_              0x03
#define CC1K_315_MHZ               0x05

//#define CC1K_DEFAULT_FREQ (CC1K_916MHZ)

#ifndef CC1K_DEFAULT_FREQ
#define CC1K_DEFAULT_FREQ    (CC1K_433_002_MHZ)
#endif
```

Nach dem Tutorial müssen noch die Xbow-Firmenkomponenten installiert werden: im ‚Crossbow-Software'-Ordner im Ordner ‚Mote-Test' die Installation durchführen. Dann die 6 Ordner in ‚Mica2 Firmware' nach ‚C:\tinyos-1.x\apps\' kopieren (dieser Pfad sollte während der Installation angelegt worden sein).

Damit ist die Installation abgeschlossen.

2.3 Deinstallation [4]

Sollte die Installation unvollständig erfolgen oder das System nicht lauffähig sein, empfiehlt es sich unter Umständen, eine Neuinstallation vorzunehmen.
Je nach Beschädigungsgrad der Installation ist der InstallWizard nicht mehr in der Lage, die Software vom System zu entfernen. Dies muss dann von Hand vorgenommen werden, wobei man sich an folgendes Schema halten sollte:
Manuell entfernen (über die ‚Uninstall'-Funktion) bzw. durch Löschen der Ordner:

- Java 2 Runtime EnvironmentStandard Edition
- Cygwin
- Avrgcc
- TinyOS

Alle zugehörigen Einträge im Startmenü.
Folgende Registry löschen (mit RegEdit):
‚HKEY_LOCAL_MACHINE\SoftWare\Cygnus Solutions\'
Cygwin in exakt das Verzeichnis neuinstallieren, in dem es vorher angelegen war (normalerweise ‚C:\Cygwin\'). Anschließend die Windows-Uninstall-Funktion verwenden, die nun funktionieren sollte; dann das gesamte System entsprechend ‚Readme1st.pdf' reinstallieren.

2.4 Einarbeitung

Im Lieferumfang enthalten sind etliche Readme-Files, die eine gute Einführung
In TinyOS bieten, z.B. ein GettingStarted-Tutorial für TinyOS in ‚tinyos-1.x\doc\',
oder eine Dokumentation über den Compiler in ‚C:\avrgcc\doc\'.
Alle Manuals liegen im .pdf-Format vor.

Bemerkungen:
Beim Nacharbeiten der Tutorials kann evtl. ein Problem beim Aufruf des SerialForwarders auftreten. Der SerialForwarder ist ein Treiber, der den Datenverkehr über die serielle Schnittstelle verwaltet.
Standardmäßig sind 19.2 kBaud als Transferrate eingestellt. MICA2-Nodes lassen sich jedoch nur mit 57.6 kBaud auslesen. Daher muss der Kommandozeilenaufruf folgendermaßen lauten:
‚java net.tinyos.sf.SerialForward -baud 57600'
Es ist übrigens nötig, eine zweite Cygwin-Konsole zu starten, sobald der SerialForwarder aktiviert ist, da er die aktuelle Konsole blockiert.

3. Arbeiten mit TinyOS

Grundsätzlich wird mit einem Texteditor (z.B. dem mitgelieferten ‚Programmer's Notepad') gearbeitet, mit dem die Programme erstellt werden. Die Synthax folgt weitestgehend der Standard-C-Notierung und ist somit in einem C-Editor am leichtesten darzustellen.
Um ein Programm zu kompilieren und anschließend auf einen MICA-Node aufzuspielen, muss zunächst die Cygwin-Konsole geöffnet (die sich nach der Installation im Startmenü befinden sollte) und anschließend ins TinyOS-Verzeichnis gewechselt werden.
Dies geschieht mit

```
cdtinyos
```

Im folgenden beziehen sich sämtliche Pfadangaben auf eine Standardinstallation, so dass evtl. variierende Eingaben nötig sind.
Eigene Applikationen befinden sich immer im ‚Apps'-Verzeichnis, in dem auch eine Datei ‚Makerules' abgelegt ist. In dieser können lokale Variablen wie Gruppen-Nummer und Ähnliches eingetragen werden. Des Weiteren werden hier die Aufruf-Parameter des avrgcc-Compilers festgelegt.
Eine Applikation besteht für gewöhnlich aus zwei Dateien:
‚Applikation.nc' und ‚ApplikationM.nc'
Die Nomenklatur mit der Schreibweise des additionalen ‚M' ist nicht zwingend notwendig, erleichtert jedoch das Arbeiten. Hinter den beiden Dateien steht die grundsätzliche Struktur des Betriebssystems. Auf sie wird weiter unten eingegangen. Jene zwei Dateien befinden sich in einem Unterordner des ‚C:\tinyos-1.x\apps\'-Verzeichnisses idealerweise ebenfalls ‚Applikation' genannt.
Dazu kommt noch eine Datei ‚Makefile', ebenfalls im ‚Applikation'-Ordner. Als Beispiel kann der werksmäßig beiliegende Ordner ‚Blink' dienen. Die Dateien in ihm lauten: ‚Blink.nc', ‚BlinkM.nc' und ‚Makefile'.
Darüberhinaus befinden sich etliche Dateien ähnlicher Nomenklatur im Ordner ‚C:\tinyos-1.x\tos\lib\'. Es handelt sich hierbei um Library-Dateien, die universell genutzt werden können, unabhängig der Applikation. Sie werden quasi von Applikationen implementiert.
Eine Applikation wird kompiliert, indem man eine Cygwin-Konsole öffnet und ins Applikationsverzeichnis wechselt.

```
cd tinyos
cd apps
cd blink
```

Hier wieder am Beispiel der 'Blink'-Applikation dokumentiert:
Der Befehl

```
make mica2
```

startet den Kompilierungsvorgang, in dessen Verlauf ein Unterordner in ‚Blink' angelegt wird (‚Build'), der den erzeugten Assemblercode für die gewünschte Hardwarebasis enthält (hier: MICA2).
Eventuelle Programmierfehler in der Applikation werden vom Compiler angezeigt, so dass unter Umständen ein Debugging stattfinden muss. Die ‚Blink'-Applikation sollte sich jedoch als ohne Fehler erweisen.

```
make mica2 install
```

kompiliert den Applikations-Code und schreibt diesen anschließend auf den MICA2-Mote im Programming-Board MIB500CA.

Achtung:
Das MIB500CA muss mit einem stabilisierten Netzteil (5-15 VDC) versorgt und die Batterien im eingelegten MICA2 entfernt bzw. abgeschaltet sein! Der Mote verfügt **nicht** über Freilaufdioden, so dass er zerstört werden kann!

Sollten sich während der Kompilierung Fehler ergeben, bricht der Vorgang ab, ein Schreiben findet nicht statt.
Sollte das MIB500CA nicht mit Energie versorgt sein, bricht der Schreibvorgang ab, nicht jedoch der Kompilierungsvorgang.
Soll ein bereits generierter Code im ‚Build'-Ordner der jeweiligen Applikation installiert werden **ohne** ihn zuvor erneut mittels der Applikationsprogramme zu rekompilieren, so ist
`make mica2 reinstall`
zu verwenden.

Für verschiedene Anwendungen im Sensornetzwerk ist es nötig, den MICA2 eine ID zuzuteilen, die u.a. ein gezieltes Verschicken von Paketen an Motes ermöglicht.
Dies geschieht, indem dem Mote diese ID während des Installationsvorgangs manuell zugeteilt wird:
`make mica2 install.x`
wobei x der gewünschten ID entspricht. Hier sei erneut darauf verwiesen, dass in der Datei ‚C:\tinyos-1.x\apps\makerules' eine Gruppennummer definiert werden kann, so dass Netzwerkverbände mit unterschiedlichen Gruppennummern überlagert werden können.

4. TinyOS – Struktur

4.1. Allgemeine Struktur

Das Sensornetzwerk-Betriebssystem TinyOS ist ein eventbasiertes Multitaskingsystem, das auf voneinander unabhängigen Komponenten, den ‚components' basiert.
Es gibt grundsätzlich drei Arten von Komponenten: die Module, die Interfaces und die Konfigurationen.
Die Module enthalten den Source-Code für die spezifische Applikation. Sie werden in eigenen Ordnern im ‚Apps'-Verzeichnis abgelegt, und werden ‚ApplicationM.nc' mit ‚M' für ‚Modul' benannt.
Ein **Modul** stellt zwei Arten von Funktionen zur Verfügung: Commands und Events. Grundsätzlich stellen Commands Funktionsaufrufe an untere Softwareschichten dar, also in Richtung Hardware. Im Gegensatz dazu bilden die Events eine Upwards gerichtete Funktionskette.
Eine **Konfiguration**, die im selben Ordner wie das Modul abgelegt wird, jedoch ‚Applikation.nc' benannt wird, stellt eine Verknüpfung zwischen verschiedenen Modulen her.
Ein **Interface** ist eine abstrakte Beschreibung der Kopplungsstelle zwischen zwei Modulen. Hier stehen die Commands, die der ‚Provider' eines Interfaces implementiert haben, sowie die Events, die der ‚Requirer' implementiert haben muss.
Wenn ein Command in einer Applikation ausgeführt wird, geschieht dies über den Befehl

```
call Command_Name()
```

(evtl. auch mit Parametern). Die Funktion, die dieses Command auslöst, ist in einem Modul, das tiefer in der Software-Hierarchie liegt, definiert.
Dieses Modul empfängt das Command, arbeitet eine dort definierte Routine ab, und meldet schließlich den Empfang des Commands an den Caller (hier die Applikation) zurück. Dies geschieht über einen Return-Wert.
Die umgekehrte Variante – ein Funktionsaufruf an der Hierarchie hinauf – erfolt mittels der Events.
Die Reaktion auf ein Event kann beliebig in der Applikation festgelegt werden.
z.B. stellt der Befehl zum Versenden einer Nachricht

```
call send(...)
```

ein Command dar, dessen Implementierung in einer anderen Applikation steht, die diesen Befehl bereitstellt. Im Gegensatz dazu muss die Applikation die Implementierung für ein Event

```
event result_t sendDone(...) { ...}
```

bereitstellen, wobei in {...} ein beliebiger Code stehen kann, der als Reaktion auf das hereinkommende Event ausgeführt wird.
Welche Commands also welche Events voraussetzen, um die Kommunikation gewährleisten zu können, steht in dem zugehörigen Interface.
Und schließlich sorgt die Konfiguration dafür, dass z.B. zwei Module vom selben Modul in einer tieferen Ebene Events empfangen bzw. dort Commands auslösen können.
Die Struktur eines Interfaces sieht also folgendermaßen aus:

```
TOS_MODUL interface_1;
ACCEPTS {
   commands;      // Liste der Commands, die das nutzende Modul
                  // implementieren muss,
                  // da sie von höheren Ebenen gesendet werden
                  // können
   };
HANDLES {
   events;        // Liste der Events, die von tieferen Ebenen
                  // kommend verarbeitet
                  // werden, also implementiert sein müssen
   }
USES {
   commands;      // Liste der Commands, die vom nutzenden Modul
                  // aufgerufen werden
                  // dürfen, da sie von tieferen
                  // Ebenen bereitgestellt sind
   }
SIGNALS {
   events;        // Liste der Events, die an höhere Ebenen
                  // gesendet werden, als
                  // Reaktion auf empfangene Commands von dort
   }
```

4.2.Beispiel ‚Blink'

Um das eben Aufgeführte zu untermalen soll hier, ebenso wie in den meisten Fallbeispielen der Tutorials von Crossbow, auf die Applikation ‚Blink' verwiesen werden. Wie alle Applikationen befindet sie sich im Verzeichnis ‚C:\tinyos-1.x\apps\' im gleichnamigen Ordner.
Sie besteht aus den Dateien ‚Blink.nc', ‚BlinkM.nc' und ‚Makefile'.
Letztere Datei enthält nur folgendes:

```
COMPONENT=Blink   // Hiermit wird festgelegt, dass ‚Blink' die
                  // Anwendungs-Applikation, also die höchste Ebene
                  // darstellt. Blink selbst gibt also keine Events mehr
                  // an noch höhere Instanzen zurück
include ../Makerules    // Die schon oben angesprochene Datei, in der z.B.
                        // Gruppennummer oder Compilersuffixe gesetzt
                        // werden können
```

Zunächst ein Blick in ‚Blink.nc'. Es handelt sich um eine Komponente vom Typ Konfiguration. Sie verbindet also verschiedene Module zu einer funktionsfähigen Applikation. Die von Crossbow mitgelieferten Tutorials beschäftigen sich ebenfalls mit diesem Fallbeispiel [3]:

```
configuration Blink { // Komponenten-Typ

implementation {
  components Main, BlinkM, ClockC, LedsC;
                  // Dies sind die miteinander verbundenen Komponenten.
                  // ‚Main' ist eine globale übergeordnete Komponente, die
                  // standardmäßig eingebunden wird. Sie enthält
                  // Initialisierungs- und Powermanagementfunktionen
   Main.StdControl -> BlinkM.StdControl;
                  // Das Interface StdControl der Komponente
                  // ‚Main' wird mit dem des Moduls ‚BlinkM'
                  // verknüpft. Die Pfeilrichtung gibt die Hierar-
                  // chie an. ‚Main' ist die Top-Komponente,
                  // ‚BlinkM' liegt tiefer
  BlinkM.Clock -> ClockC;
                  // Das Interface 'Clock' der Komponente
                  // 'BlinkM' wird mit dem der tiefer liegenden
                  // Komponente ‚ClockC' verknüpft.
  BlinkM.Leds -> LedsC;
                  // Das Interface ‚Leds' der Komponente
                  // ‚BlinkM' wird mit der tiefer liegenden
                  // Komponente ‚LedsC' verknüpft.
}
```

Dies führt zu folgender Hierarchie:

<------ höhere Software-Ebene Hardware-Nähere Ebene -------->

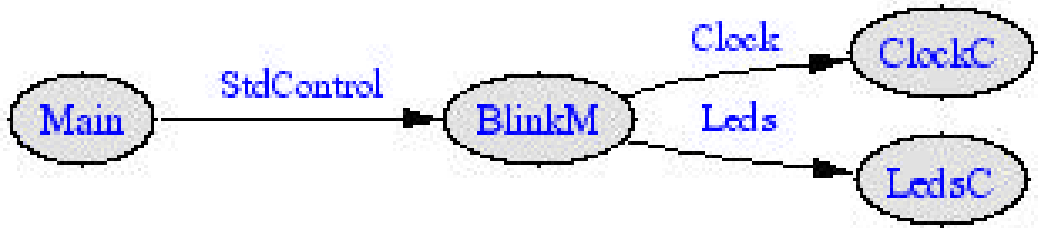

Abb. 2 (Strukturbild der ‚Blink'-Applikation)

Derartige Grafiken können vom Compiler direkt erzeugt werden. Dazu ist in einer Cygwin-Konsole folgendes einzugeben (hier am Beispiel ‚Blink'):

```
make mica2 docs
```

Die erzeugten Dateien sind dann im Verzeichnis ‚C:\tinyos-1.x\doc\nesdoc\' zu finden.
In den Grafiken stehen die Ovale für Module und die Pfeile für Interfaces, die die Module aneinanderkoppeln. Die jeweiligen Namen der Module bzw. Interfaces stehen anbei.

In ‚BlinkM.nc' steht folgendes:

```
module BlinkM { // Komponenten-Typ
  provides {
    interface StdControl;     // Die Synthax besagt, dass die in einem
                              // 'Provide'-Interface stehenden Funktionen
                              // Commands oder Events, sind, die sich auf
                              // eine übergeordnete Komponente beziehen,
                              // (hier: ‚Main').
  }
  uses {
    interface Clock;          // 'Use'-Interfaces enthalten Commands oder
                              // Events, die
    interface Leds;           // die Kommunikation mit niederen Komponenten
                              // betreffen.
  }
}
```

Um zu sehen, wieso die Verknüpfungen gerade auf diese Weise erfolgen, folgen hier die Interfaces ‚StdControl' und ‚ClockC':

```
interface StdControl
{
  command result_t init();    // Die höhere Komponente 'Main' nutzt ein
  command result_t start();   // Interface, durch das 3 verschiedene
  command result_t stop();    // Commands in 'BlinkM' ausgelöst werden
                              // können.
                              // Diese 3 Commands müssen also als Command-
                              // Funktionen implementiert sein
                              // (‚What-to-do-if-start()-is-called')
                              // Der Sinn: Da das StdControl-Interface in
                              // nahezu jede Applikation eingebunden wird,
                              // kann somit ein einmaliges Aufrufen des
                              // Commands ‚init()' in ‚Main' sämtliche
                              // zugehörigen Routinen in allen an ‚Main'
                              // ge'wire'ten Komponenten wie ‚BlinkM'
                              // auslösen.
}

interface Clock
{
  command result_t setRate (char interval, char scale);
  event result_t fire();
}
```

Aus diesem Interface geht hervor, dass einerseits die Komponente ClockC die Implementierung des Commands ‚setRate(...)' bereitstellen muss, die unsere Applikation ‚BlinkM' aufrufen kann, andererseits aber ‚BlinkM' eine Implementierung für den Event ‚fire()' besitzen muss.
Im Gegensatz zum ‚StdControl'-Interface bezieht sich das ‚command' nun **nicht** darauf, dass ‚BlinkM' die Implementierung bereitstellen muss. Der Grund liegt in der Konfigurierung, wo festgelegt wurde, dass zwischen ‚Main' und ‚BlinkM' erstere auf der höheren Ebene liegt, und somit Commands der niederigeren Ebenen **aufrufen** darf, die dazu dort implementiert sein müssen.

Bei ‚BlinkM' und ‚ClockC' ist ‚BlinkM' auf der höheren Ebene, woraus folgt, dass die Command-Implementierung bei ‚ClockC' liegt.
Und da Events immer bei der höhergelegenen Komponente implementiert sein müssen, aber von der niedrigeren aufgerufen werden, ist klar, dass das Modul ‚BlinkM' eine Funktionsdeklaration für das Event ‚fire()' haben muss.

Um dies zu verifizieren, hier der Rest von ‚BlinkM.nc':

```
implementation {                    // Liste der nötigen Implementierungen
                                    // von Commands und Events
  command result_t StdControl.init() {   // Über das Interface StdControl
                                         // kann von der Komponente ‚Main'
                                         // das Command'init()' kommen, das
                                         // hier behandelt werden muss.

    call Leds.init();                    // Hier wird in der niedrigeren
                                         // Komponente 'LedsC' das Command
                                         // 'init()' aufgerufen.
                                         // Dabei wird das Interface 'Leds'
                                         // genutzt.
    return SUCCESS;
}

  command result_t StdControl.start() {  // Wie 'init()'
    return call Clock.setRate(TOS_I2PS, TOS_S2PS);
                                         // hier wird in der niedrigeren
                                         // Komponente 'ClockC' das
                                         // Command 'setRate(...)
                                         // aufgerufen. Dabei wird das
                                         // Interface ‚Clock' benutzt.
  }

  command result_t StdControl.stop() {   // Wie 'init()'
    return call Clock.setRate(TOS_I0PS, TOS_S0PS);
                                         // s.o.
  }

  event result_t Clock.fire() {     // Von der tiefer gelegenen Komponente
                                    // 'ClockC' kann ein Eventaufruf kommen,
                                    // der hier behandelt wird. Das dabei
                                    // benutzte Interface heißt ‚Clock'

    call Leds.redToggle();          // In der tiefer gelegenen Komponente
                                    // ‚LedsC' wird das Command ‚redToggle'
                                    // ausgeführt.
    return SUCCESS;
  }
```

Hier nicht angeführt ist das Interface 'Leds.nc'. Es ist aber selbstverständlich, und dem voranstehenden Code entnehmbar, dass dort die Commands ‚redToggle()' und ‚init()' deklariert sein müssen, deren Implementierung sich demzufolge in ‚LedsC.nc', dem zugehörigen Modul befindet. Das Interface ‚Leds' besitzt im Gegensatz zu ‚Clock' keine (genutzten) Events.
Es ist jederzeit möglich, ein Modul, das 50 implementierte Commands anbietet, über ein Interface so zu beschneiden, dass nur 4 davon von der höheren Komponente genutzt werden können. Es brauchen also nicht **alle** Implementierungen genutzt werden.

Es gilt aber:
Tauchen in einem Interface Commands auf, so sind diese in der niedrigeren Komponente zu implementieren.
Tauchen in einem Interface Events auf, so sind diese in der höheren Komponente zu implementieren.

Für tiefergehendes Verständnis wird auf auf diverse Internetquellen, wie z.B. die avrgcc-Referenz oder weiterführende Literatur zu TinyOS verwiesen.

Da die Komponenten ‚LedsC.nc' und ‚ClockC.nc' allgemeine und sinnvolle Commands bzw. Events anbieten, sind sie nicht als Applikation abgelegt, sondern in ‚C:\tinyos-1.x\tos\lib\' als Library-Files zu finden.
Die Suchpfade in den ‚lib'-Ordner sind bei fehlerfreier Installation ordnungsgemäß gesetzt worden, so dass keine Kompilationsfehler auftreten sollten.

Um einen Überblick über die hier nicht aufgeführten Interfaces der Komponenten ‚ClockC' und ‚LedsC' zu geben, sei mit nachstehender Grafik ein Gesamtüberblick über die Struktur der ‚Blink'-Applikation gegeben. In Abbildung 3 sind zusätzlich zu den direkt an ‚BlinkM' gekoppelten Modulen, auch die tieferliegenden Module samt der kommunizierenden Interfaces aufgeführt.
Hier wird der Vorteil des modularen Aufbaus des Systems sichtbar:
Der Anwender muss für die Funktionsfähigkeit des Systems nur die Interfaces ‚StdControl', ‚Clock' und ‚Leds' beachten.

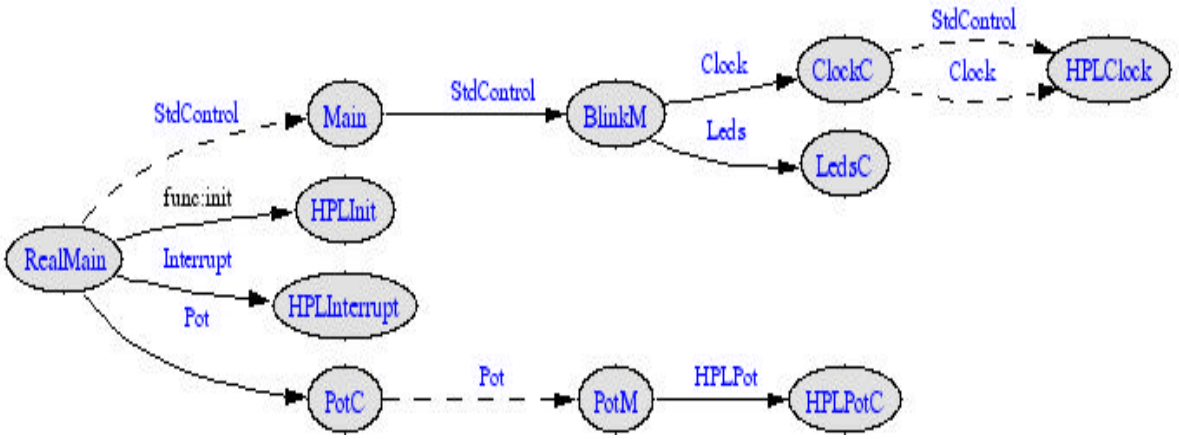

Abb. 3 (Gesamtstruktur ‚Blink')

5. Software-Grundlagen zu den Versuchen

Um Laufzeitmessungen in MICA2-Systemen durchführen zu können, wird stets auf einen Standard-Programmrumpf zurückgegriffen, der – je nach Art der durchzuführenden Messung - mehr oder weniger modifiziert wird.
Hier seien nur grundsätzlich die jeweils benutzten Module und ihre Funktion aufgeführt, um dann bei den konkreten Versuchen nur noch auf die spezifischen Änderungen gegenüber dem Basisprogramm verweisen zu müssen.

Als Beispiel ist hier nicht der (programmmäßig) einfachste Code gewählt, damit für die folgenden Versuche eine Reduktion des Programms vorgenommen werden kann, statt dieses jeweils umständlich erweitern zu müssen.

5.1. Die Konfiguration

```
includes HopMsg;
configuration BaseStation_1 {
}
implementation {
  components Main, Counter, BasisstationM, TimerC, GenericComm, LedsC;

  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> BasisstationM.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Counter.Timer -> TimerC.Timer[unique("Timer")];
  Counter.IntOutput -> BasisstationM.IntOutput;
  BasisstationM.Leds -> LedsC;
  BasisstationM.ReceiveIntMsg -> GenericComm.ReceiveMsg[AM_INTMSG];
  BasisstationM.Send -> GenericComm.SendMsg[AM_INTMSG];
  BasisstationM.CommControl -> GenericComm;
}
```

Die Include-Datei enthält die Definition einer Struct, in die die zu übertragenden Werte geschrieben werden können. ‚HopMsg.h' (im Ordner ‚Lib') definiert eine Struct mit nur 2 Int-Werten für eine ‚Source' und einen ‚Value', die natürlich beliebig belegt werden können.
Das zum Senden einer Nachricht verwendete Modul ‚AM' (‚Active Messaging') lässt einen maximalen Payload von 29 Byte zu. Dieser Wert kann über die Datei ‚C:\tinyos-1.x\tos\system\AM.h' editiert werden (dort in Zeile 88). Bei Werten über 29 Byte wird allerdings ein größerer programmiertechnischer Aufwand nötig, da andere am Sendeprozess beteiligte Module der unteren Schichten von 29 Byte als Maximum-Payload ausgehen.
Aus dem Rest der Konfiguration geht hervor, dass die Initialisierungs-Commands der Main-Komponente an gleich drei andere Module (‚Counter','BasisstationM' und ‚TimerC') gekoppelt sind. Wie oben erwähnt, müssen in jenen Komponenten also Implementierungen für die Commands ‚Start', ‚Stop' und ‚Init' vorhanden sein. Diese Commands nutzen allesamt das ‚StdControl'-Interface.
Die Komponente ‚Counter' ist in der Lage einen Timer (‚TimerC') zu starten. Über das Interface ‚IntOutput' gibt ‚Counter' den aktuellen Timer-Wert (Char-Typ) an die Hauptkomponente ‚BasisstationM' weiter.
Des Weiteren kann ‚BasisstationM' die Leds über das gleichnamige Interface steuern und Nachrichten über das Interface ‚Send' versenden bzw. über ‚ReceiveMsg' ebensolche empfangen. Die niederhierarchische Komponente für diese Ereignisse ist ‚GenericComm', die wiederum in ihren tieferen Schichten mit dem zentralen Modul zum Message-Verarbeiten ‚AM' kommuniziert.

Beide Interfaces, ‚send' und ‚ReceiveMsg', bestehen aus Commands **und** Events, was eine bidirektionale Kommunikation von ‚GenericComm' mit ‚BasisstationM' ermöglicht.
Die Commands stellen z.B. den Befehl ‚send(...)' dar, der als Parameter die zu versendende Message enthält, während dazu ein Event ‚SendDone(...)' gehört, das rückmeldet, dass der Sendepuffer wieder freigegeben ist.

Zusammenfassend ist also folgendes festzustellen:
Das Hauptmodul ‚BasisstationM' ist in der Lage

- Nachrichten zu versenden
- Nachrichten zu empfangen
- Die Leds des MICA2 zu steuern
- Timer-Events mit dem aktuellen Timer-Wert zu empfangen

5.2. Die Implementierung

Das Modul ‚BasisstationM' selbst, weist folgende Struktur auf:

```
module BasisstationM {
  provides
      {
        interface StdControl;
                        // -> dieses Modul stellt die Implementierung der
                        // Commands des Interface ‚StdControl' für die
                        // höhere Komponente ‚Main' bereit -> ‚Main' kann
                        // also ‚Start' etc. hier aufrufen.
        interface IntOutput;  // -> dieses Modul stellt die Implementierung
                              // der Commands und Events des Interface
                              //‚IntOutput' für die höhere Komponente
                              //‚Counter' bereit.
      }
      uses
      {
        interface ReceiveMsg as ReceiveIntMsg;
                                    // -> dieses Modul muss die Events des
                                    // Interface implementiert haben, die
                                    // eine niederere Komponente, hier
                                    // ‚GenericComm' aufrufen kann.
        interface StdControl as CommControl;
        interface SendMsg as Send;  // -> das Command dieses Interfaces
                                    // muss in einer tieferen Komponente
                                    // implementiert sein. Das Event
                                    // ‚SendDone' dieses Interface muss
                                    // dagegen hier implementiert sein.
        interface Leds;             // Das bekannte Interface aus 'Blink'
      }
}
```

```
implementation {  // Beginn des Hauptprogrammteils -> hier müssen sämtliche
                  // Implementierungen für alle Commands und Events
                  // geschehen.
      bool busy;
      bool able_to_send = TRUE;
      struct TOS_Msg data;

// COMMANDS
      command result_t StdControl.init()
      {
            return call CommControl.init();
            call Leds.init();
      }

      command result_t StdControl.start()
      {
            return call CommControl.start();
      }

      command result_t StdControl.stop()
      {
            return call CommControl.stop();
      }
      command result_t IntOutput.output(uint16_t value) // (1)
      {
            IntMsg *message = (IntMsg *)data.data; (3)
            call Leds.redOn();
            if (!busy && able_to_send)
            {
                  busy = TRUE;
                  message->val = value;
                  atomic {
                         message->src = TOS_LOCAL_ADDRESS;
            }
                  if (call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg),
                  &data)) // (4)
                    return SUCCESS;
                  busy = FALSE;
            }
            return FAIL;
      }

// EVENTS
      event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m)
      {
          call Leds.redOff();
            able_to_send = TRUE;
            return m;
      }

      event result_t Send.sendDone(TOS_MsgPtr msg, result_t success)
      {
            if (busy && msg == &data)
            {
                  busy = FALSE;
                  able_to_send = FALSE;
                  signal IntOutput.outputComplete(success);  // (2)
                  }
            return SUCCESS;
      }
}
```

Die Provides-Interfaces zu Beginn des Moduls beziehen sich auf die Komponenten der Applikation, die höher in der Hierarchie stehen, als ‘BasisstationM’. Die Uses-Interfaces beziehen sich auf die niedrigeren Komponenten.
In all diesen Interfaces stehen nun Commands und Events, die implementiert werden müssen bzw. genutzt werden können.
Nach wie vor gilt:

Provided Interface: Commands müssen implementiert werden,
Events können genutzt werden.
Used Interface: Commands können genutzt werden,
Events müssen implementiert werden.

Dies führt zu der oben aufgelisteten Struktur.
Während die ‚StdControl’-Commands lediglich zum Starten bzw. Initialisieren der Gesamtapplikation dienen, ist das ‚otput’-Command (1) die erste ‚arbeitende’ Funktion.
Sie kann vom ‚Counter’-Modul aufgerufen werden und wird immer dann ausgelöst, wenn der Timer um eins erhöht wurde **und** der ‚Counter’ eine Bereitmeldung über das Event ‚outputComplete’ (2) erhalten hat. In diesem Fall ruft ‚Counter’ das Command auf, woraufhin eine Nachricht generiert wird. Dabei wird eine Variable erzeugt (3), die ein Pointer auf eine Struct des eingebundenen Typs (die Include-Datei) ist. Die Felder der Struct werden mit dem aktuellen Timer-Wert bzw. der lokalen Adresse des Senders gefüllt, und die Struct dann als Übergabeparameter im Command-Aufruf an ‚GenericComm’ verwendet (4).
Da das Command hiermit abgearbeitet ist, verfällt das Modul in einen Wartemodus, bis von der niedrigeren Ebene die Rückmeldung ‚SendDone’ über ein Event kommt.
In der zugehörigen Implementierung des Events wird das ‚busy’-Flag rückgesetzt (es dient der Identifikation, ob das Radio-Modul gerade am Senden ist) und ‚able_to_send’ auf FALSE gesetzt. Zudem wird an ‚Clock’ gemeldet, dass der nächste auftretende Timer-Tick wieder an ‚BasisstationM’ übergeben werden kann (2).
‚able_to_send’ dient dazu, zu verhindern, dass mit jedem Clock-Tick eine Message gesendet wird. Da die Clock-Rate bei 4Hz liegt, wäre also zum jeweils nächsten Tick die aktuelle Übertragung abgeschlossen, und es könnte eine neue gestartet werden (die reine Sendezeit für eine Nachricht liegt bei ca. 35ms).
Daher wird ‚able_to_send’ nur zurückgesetzt (TRUE), wenn das Event ‚receive’ ausgelöst wurde. In diesem Fall ist nämlich eine Message empfangen worden, die von einem anderen MICA2-Node stammen muss.

Das Verhalten kann also vereinfacht so beschreiben werden:
‚Basisstation’ sendet eine Nachricht. Eine weitere Nachricht wird erst gesendet, wenn vorher eine Nachricht empfangen wurde. Alle Sendevorgänge beginnen stets mit dem nächsten Clock-Tick (4Hz-Clock-Rate, also alle 250ms).
Wie beim Beispiel ‚Blink’ soll ein Strukturbild hier eine Übersicht bieten. Das vereinfachte Strukturbild spiegelt die voranstehenden Erläuterungen wieder.

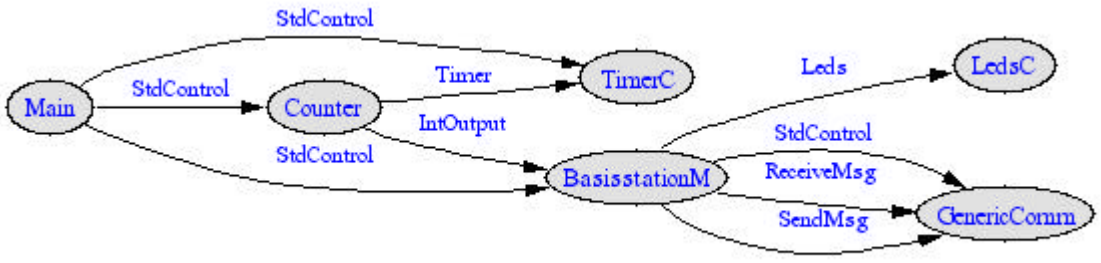

Abb. 4 (Strukturbild ‚BasisstationM’)

Hier die vollständige Struktur mit sämtlichen genutzten Komponenten:

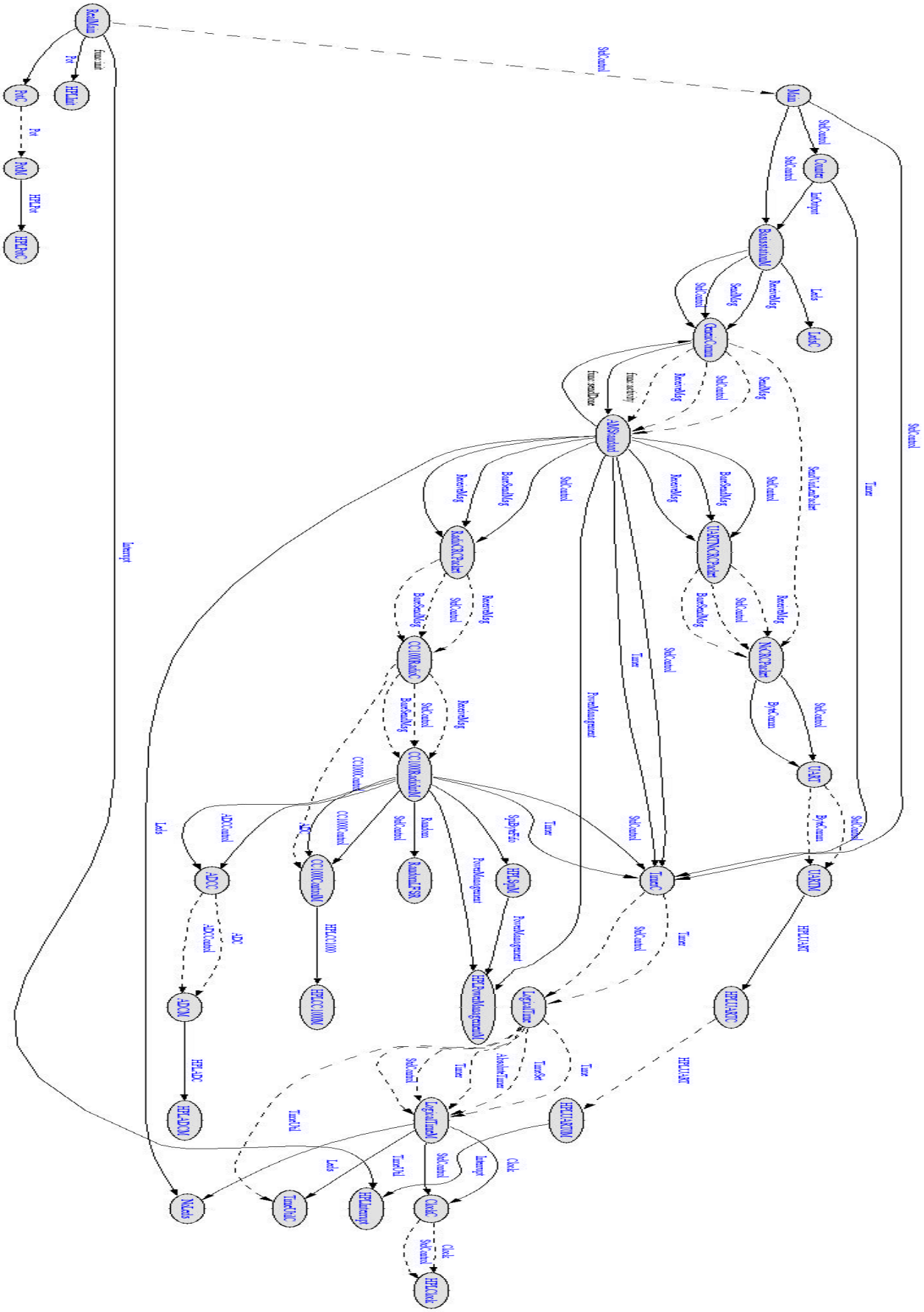

Abb. 5 (Gesamtstruktur ‚Basisstation_M')

5.3. Der Radio-Stack

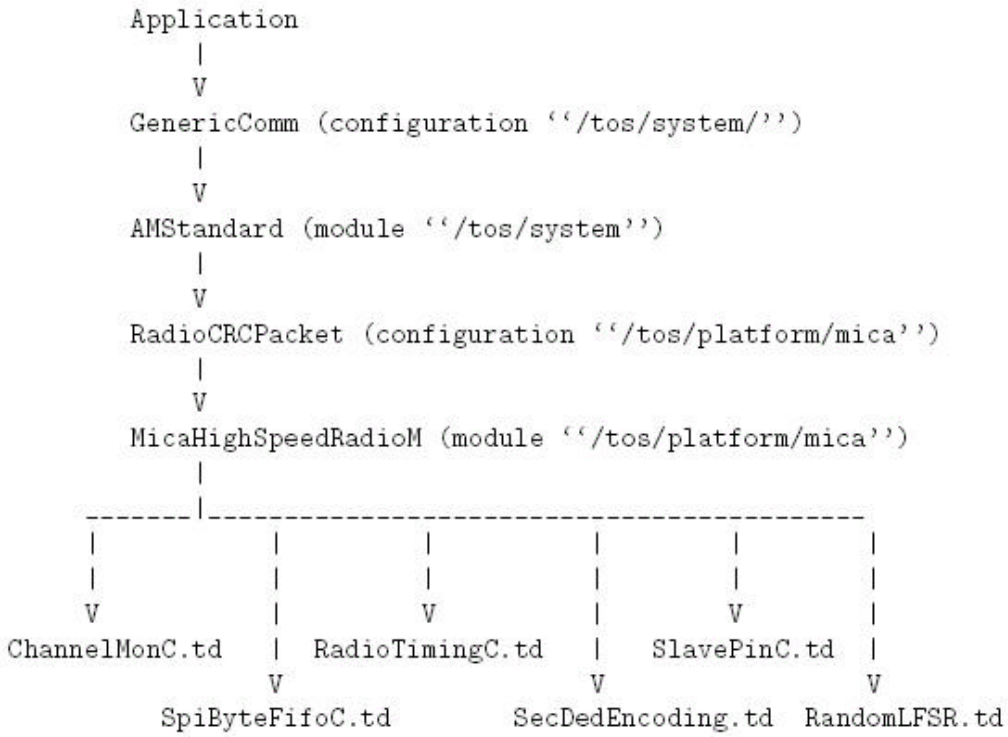

Abb. 6 (‚RadioStack-Struktur’ [1])

Diese Grafik zeigt schematisch die Top-Down-Struktur des MICA-Radio-Stack. Die Struktur der MICA2-Serie ist entsprechend gestaltet.
Die von der erstellten Applikation angekoppelte Komponente ‚GenericComm’ erhält als Übergabeparameter eine Struct, die in das 29-Byte lange Data-Feld der zu übertragenden Message eingefügt wird.
Nur die niedrigsten Ebenen der Struktur haben direkten Zugriff auf die Hardware (‚SpiByteFifoC.td’).
Die gesamte Nachricht wird in einzelne Bits zerlegt, CRC berechnet und schließlich das ‚SendDone’-Event in der Hierarchie nach oben weitergeleitet.

Eine exakte Beschreibung der Abläufe zwischen diesen Komponenten findet sich in [1].

Eine vollständige TinyOS-Nachricht enthält folgende Datenfelder:

```
typedef struct TOS_Msg
{
        uint16_t        addr;   // ID des Ziel-MICA2 oder allg. Broadcast-Addr.
                                // Wenn empfangene addr weder eigene ID, noch
                                // allg. Broadcast-Addr. -> Drop
        uint8_t         type ;  // spezifiziert Handler-ID, mit der AM die Msg.
                                // verarbeitet
        uint8_t         group ; // Gruppen-ID. Wenn eigene Gruppe != group -> Drop
        uint8_t         length ;
        uint8_t         data[TOSH_DATA_LENGTH] ;      // Von der Applikation erstellter
                                                      // Datenteil -> Payload
        uint16_t        crc;            // Fehlerpolynom
        uint16_t        strength ;      // nicht genutzt
        uint8_t         ack ;
        uint16_t        time ;
}
```

Die wichtigsten Parameter für die Datenübertragung mit TinyOS finden sich in folgenden Dateien:

- ‚C:\tinyos-1.x\tos\platform\mica2\CC1000RadioM.nc' (z.B. Preambel-Länge)
- ‚C:\tinyos-1.x\tos\platform\\mica2\CC1000Const.h'
- ‚C:\tinyos-1.x\tos\system\AMStandard.nc'
- ‚C:\tinyos-1.x\tos\system\AM.h'

In [1] findet sich folgende Grafik, die ein Timing-Diagramm darstellt, aus dem abzulesen ist, wie lange ein TinyOS-Message-Send-Vorgang dauert.
Die 4MHz-Clock bezieht sich allerdings auf einen MICA-Node. MICA2 arbeitet mit 7.2MHz, so dass die Werte entsprechend umgerechnet werden müssen.

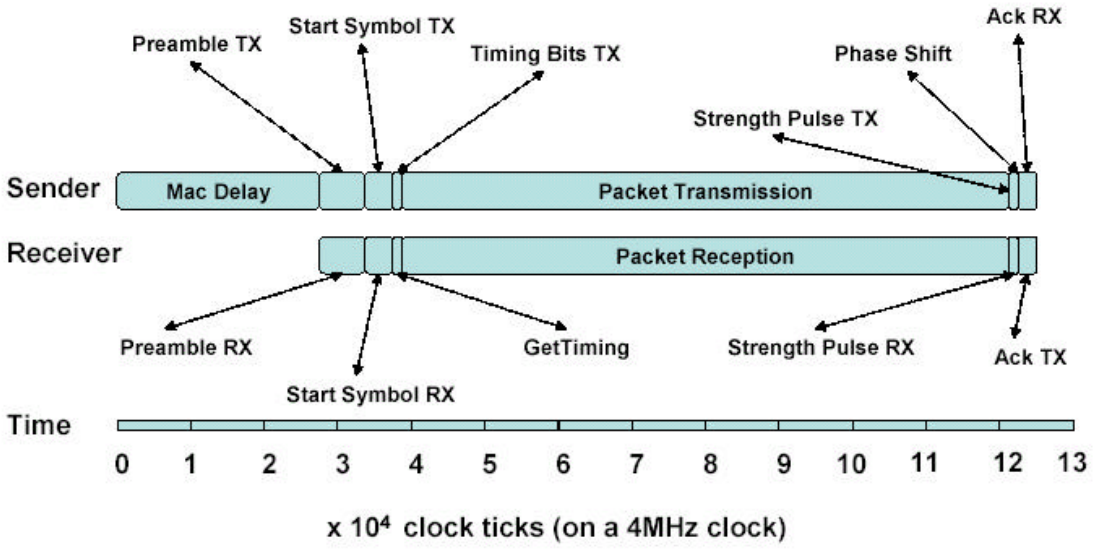

Abb. 7 ('Timing-Protokoll' [1])

6. Messmethodik

6.1. Messung

Um eine Aussage über Laufzeiten in MICA2-Systemen machen zu können, muss eine Methode gefunden werden, diese messen zu können.
Für diee Messungen wird ein Oszilloskop
TEKTRONIX ‚TDS-210'
mit integriertem Kommunikationsmodul
TDS-2CM
verwendet. Die Software ‚WaveStar' kann auf der Homepage von Tektronix in einer 30-Tage Testversion kostenlos heruntergeladen werden.

Die Intention ist, einen messbaren Impuls zu erzeugen, der zu beliebigen Zeitpunkten im Programmablauf gesetzt bzw. gelöscht werden kann. Dieser soll dann über das Oszilloskop erfasst, und anschließend in die PC-Software transferiert werden zwecks Auswertung durch MATLAB.

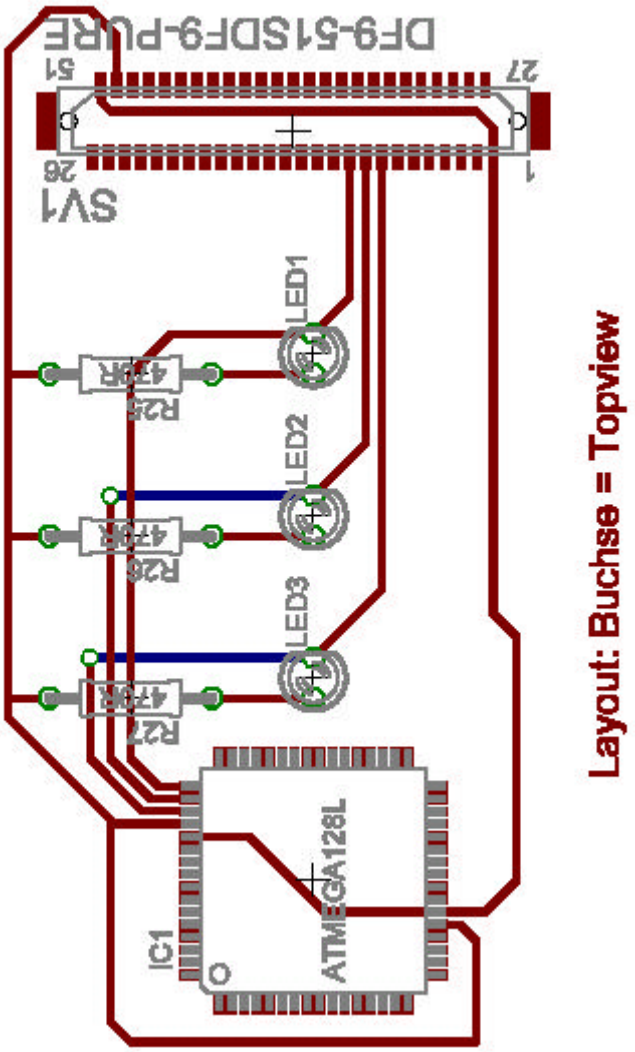

Abb. 8 ('MICA2-Layout' [2])

Einen derartigen Impuls erhält man, indem man die LED-Spannung des MICA2 an der Buchse der Programmierschnittstelle abgreift und die LEDs an gewünschten Stellen des Programms ein- bzw. ausschaltet.
Somit ist ein gezieltes Messen bestimmter Zeitspannen möglich, die durch das abstrakte Messen der LED-Spannung über die Messungsvielfalt hinausgeht, die ein Abgreifen an den Pins mit Aktivität bei Sendevorgängen geboten hätte. Selbstverständlich tritt durch das Aktivieren der LEDs eine Verzögerung im Programmablauf auf, die jedoch im Gegensatz zum Sendevorgang als klein eingestuft werden kann und im Folgenden vernachlässigt wird. Die Messungen dienen in erster Linie einer grundsätzlichen Abschätzung der gewünschten Größen.

Gemäß nebenstehender Grafik kann an den Pins 8, 9 und 10 der Buchse gegen Masse auf Pin 51 die LED-Spannung gemessen werden. Diese ist allerdings invertiert, was bei der Auswertung berücksichtigt werden muss. Auf keinen Fall darf gegen die Versorgungsspannung auf Pin 50 gemessen werden.

Der mit diesem Verfahren erzeugte Impuls ist nun auf dem Oszilloskop-Schirm sicht- und nachregelbar. Die gewählten Auflösungen des Displays betragen jeweils 250-500ms, so dass stets ca. 5 Impulse gleichzeitig sichtbar sind.
Das Kommunikationsmodul des TDS-210 wird mittels eines Nullmodemkabels mit dem COM-Port des PCs verbunden, auf dem ‚WaveStar' installiert ist.
Die Software ist in der Lage, Snapshots des Oszilloskop-Displays in tabellarischer Form darzustellen. Dabei werden jeweils 2500 Abtastwerte übertragen.
Diese schlüsseln sich auf zu:

Zeit [s]	gemessene Spannung [V]
.	.
.	.
.	.

Die Auflösung der Zeitbasis beträgt 400µs. Für jede Messung wurden 40 derartige Tabellen erstellt. Diese Tabellen wurden im ‚.csv'-Format gespeichert und anschließend in eine einzige große Datei zusammenkopiert um diese mit MATLAB auswerten zu können.
Die Gesamtzahl an Messwerten beträgt also 40*5, bereinigt um unbrauchbare Randwerte.
Wichtig bei der Messung ist evtl. die Kenntnis über die Clockrate des Timers, damit evtl. der Trigger des Oszilloskops justiert werden kann:
Die Clock-Frequenz beträgt 4Hz und kann für Variationen in der Datei ‚C:\tinyos-1.x\tos\lib\counter.nc' modifiziert werden.

6.2. Auswertung in MATLAB

Hier der MATLAB-Algorithmus zum Herausfiltern und Darstellen der interessierenden Werte:

```
% DATA IMPORT
[buffer] = importdata('1.csv'); % imports whole data packet from file
% VARIABLES AND CONSTANTS DECLARATION
[main] = zeros(255,1);
counter = 1; % index pointer of the planned matrix
len = max(size(buffer));
led_off_tolerance = 2.5;
led_on_tolerance = 0.5;
noise_level = -5;
candidate = 0;
searching_4_last_zero = 1;
searching_4_last_value = 0;
noise_found = 0;
average = 0;
% GENERATING WORK-MATRIX
for X = 1:(len-1);
        % Check whether noise is occuring and set noise_found = 1 if so.
    if buffer(X,2) < noise_level
       noise_found = 1;
    end;

    % If there was noise - wait until it's over and then look for the next
    % period of waiting_4_value
    if noise_found == 1
       if buffer(X,2) > noise_level
           noise_found = 0;
           searching_4_last_zero = 1;
           searching_4_last_value = 0;
       end;
    end;
```

```
    % Only search for anything when there is no actual noise occuring!
    if noise_found == 0

      % At first - search for a value that is above the LED_OFF_Tolerance
      % to make sure that we are looking at a non-trigger-signal.
      if searching_4_last_zero == 1
        if buffer(X,2) > led_off_tolerance % last zero before switch_on
           if buffer(X+1,2) < led_on_tolerance % first value of switch_on
              if buffer(X+1,2) > noise_level % make sure that it's no noise
                  % Breakpoint is found - put value into candidate-state
                  candidate = buffer(X+1,1);
                  searching_4_last_zero = 0;
                  searching_4_last_value = 1;
              end;
           end;
        end;
      end;

      % Then - have a look for the last value while LED_ON when the next
step
      % switches it off
      if searching_4_last_value == 1
         if buffer(X,2) > led_on_tolerance % last value before switch_off
           if buffer(X+1,2) > noise_level % make sure that it's no noise
              if buffer(X+1,2) > led_off_tolerance % first switch_off-Val
                  % Endpoint of interval is found -> calculate length
                  main(counter) = (buffer(X,1) - candidate);
                   average = average + main(counter);
                  counter = counter+1;
                  searching_4_last_zero = 1;
                  searching_4_last_value = 0;
              end;
           end;
        end;
     end;

    end; % of if noise_found = 0

end; % of for

average = average / counter;
[edges] =
[0.02,0.022,0.024,0.026,0.028,0.03,0.032,0.034,0.036,0.038,0.04,0.042,0.044
,0.046,0.048,0.05,0.052,0.054,0.056,0.058,0.06,0.062,0.064,0.066,0.068,0.07
,0.072,0.074,0.076,0.078,0.08];
 result = histc(main,edges); % calcule the histogram
 BAR(edges,result,'histc'); % plot the histogram
 xlabel('transmit time (s)');
 ylabel('counts');
 title('MsgSend - Histogram');
```

Der größte Teil des Algorithmus dient dazu, die Tabelle jeweils nach einem Sprung in den Messwerten zu durchsuchen. Dies ist in diesem Fall der Sprung von X>2.5V auf 0V<X<0.5V. Wie oben erwähnt, entspricht der High-Pegel dem LED-Off, und umgekehrt.
Zum Erkennen des ersten Auftretens eines On-Pegels werden Schwellenspannungen eingeführt, die über- bzw. unterschritten werden müssen, damit ein zugehöriger Zeitpunkt als Beginn eines Impulses gewertet werden kann.
Die Schwellenspannungen sind notwendig, um das leichte Rauschen des Signals auszublenden.

Des Weiteren muss darauf geachtet werden, dass in jeder Tabelle eine extrem verrauschte Stelle auftritt, die jeweils an der Stelle liegt, an der gerade das Display des Oszilloskops im Refresh-Zustand war. Diese Stellen sind anhand von negativen Messwerten <-10V identifizierbar.
Sollte eine solche Störung auftreten, während nach Auffinden eines Impulsstarts das zugehörige Ende gesucht wird, verfällt der Impulsstartwert, da sein Ende nicht mehr gesichert auffindbar ist. Die Gesamtanzahl Messwerte, die aus der Tabelle gewonnen werden kann, reduziert sich daher um maximal 1 Messwert pro Tabelle, in der sich, da jeder Snapshot mit ca. 5 Impulsen pro Display getätigt wurde, ebenfalls ca. 5 Impuls-Start- bzw. Endwerte befinden.
Ein Messwert errechnet sich also aus
Impuls-Startzeit – Impuls-Endzeit = Brenndauer LED
Die Gesamtzahl an verwertbaren Werten beträgt bei jeder Messung ca. 130.
Der letzte Teil des Algorithmus schließlich dient der Darstellung der Ergebnisse, indem eine 1-dimensionale Matrix ‚edges' erzeugt wird, die (hier in Klartext) die die Intervalle angeben, in die die etwa 130 Messwerte eingeordnet werden.
Die ‚Histogram'-Funktion erzeugt dann eine Verteilung der Werte auf die vorgegebenen Intervalle mittels der <- oder > -Prüfung.
Die ‚BAR'-Funktion erzeugt eine Grafik auf Basis der Histogram-Funktion, um das Ergebnis zu visualisieren.
Die Edges für die Intervallängen wurden aufgrund mehrerer Versuche mit unterschiedlichen Auflösungen ermittelt. Grundsätzlich sind sie frei wählbar, jedoch ist hier diejenige Auflösung gewählt worden, die die Verteilung der Messwerte auf das maximale Intervall, optimal darstellt. Eine weitere Verfeinerung der Intervallängen auf <2ms bringt grafisch keine neuen Erkenntnisse.

7. Messung: Hop-And-Back-Time (V1)

7.1.Versuchsziel

Ziel dieses Versuchs ist die Ermittlung der Zeitspanne, die vergeht zwischen:
Aufrufen des ‚Send'-Commands und Empfangen des ‚ReceiveMsg'-Events bei dem MICA2, der als Basisstation deklariert ist.
Dazu wird in die in Abschnitt 6 erläuterte Methode verwendet.
Das Command ‚redOn' zum Einschalten der roten LED wird vor ‚Send' aufgerufen, und ‚redOff' ist erstes Command in der Implementierung des ‚ReceiveMsg'-Events.

7.2. Versuchsvorbereitungen

Benötigtes Material:

- 2 MICA2-Nodes
- TDS-210 mit TDS-2CM
- Nullmodemkabel
- modifizierter Stecker zum Anschluss des MICA2 an TDS-210
- Software ‚WaveStar'
- Dateien: ‚BasisstationM.nc', ‚ZielstationM.nc'
- Ordner: ‚Basisstation_1', ‚Zielstation_1' mit Inhalt

Die beiden genannten Ordner sind die Applikationen und müssen nach ‚C:\tinyos-1.x\apps\' kopiert werden. Sie enthalten die Makefiles und die Konfigurationen.
Letztere entsprechen exakt den schon oben erläuterten Konfigurationen und sind für sämtliche Versuche unverändert (bis auf die jwls. unterschiedliche Anwendungsdatei, z.B. ‚BasisstationM.nc'). Dies geschieht, damit eine gewisse Vergleichbarkeit der Ergebnisse untereinander gewährleistet bleibt, und nicht etwa ein verkürzter Code zu extremen Laufzeitverfälschungen führt.
Die beiden Applikationsdateien müssen in ‚C:\tinyos-1.x\tos\lib\' angelegt werden.
Der Code in ‚BasisstationM.nc' entspricht exakt dem in Abschnitt 5 diskutierten Code.
Der Code in ‚ZielstationM.nc' weicht nur in folgenden Punkten von ersterem ab:

```
command result_t IntOutput.output(uint16_t value)
      {

      }
```

Die Hereingabe von Timer-Werten über die ‚Counter'-Komponente hat hier keinerlei Reaktion zur Folge.

```
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m) {
            IntMsg *message = (IntMsg *)data.data;
            if (!busy)
            {
                  busy = TRUE;
                  message->val = 1;
                  atomic {
                        message->src = TOS_LOCAL_ADDRESS;
            }
                  call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &data);
            }
            return m;
      }

      event result_t Send.sendDone(TOS_MsgPtr msg, result_t success)
      {
            if (busy && msg == &data)
            {
                  busy = FALSE;
        }
            return SUCCESS;
      }
```

Der gesamte Sende- und Empfangsteil wird bei der Zielstation über Events abgehandelt. Da die Zielstation nicht an einen Timer gebunden sein soll, sondern eine hereinkommende Nachricht **sofort** zurücksenden soll, ist das Sende-Command direkt in das ‚Receive'-Event eingebunden. Der Code zwischen ‚if(!busy) {...}' entspricht aber exakt der Senderoutine in ‚BasisstationM.nc'. Die Bereitschaft zum erneuten Senden/Empfangen wird wieder wie bei der Basis über das ‚SendDone'-Event gehandhabt.

Das Verhalten des Gesamtsystems lässt sich also wie folgt beschreiben:
‚Basisstation' sendet eine Message. ‚Zielstation' empfängt diese und sendet unverzüglich eine Message zurück. ‚Basisstation' empfängt diese und sendet mit dem nächsten hereinkommenden Timer-Tick eine weitere Nachricht...

Ein MICA2-Node wird also mit ‚Basisstation_1' programmiert, ein weiterer mit ‚Zielstation_1'. Es seien hier nur die Namen der zugehörigen Applikationsordner angegeben. Dies bedeutet jedoch, dass man eine Cygwin-Konsole öffnen, ins ‚Apps'-Verzeichnis wechseln, und dann die Eingabe ‚make mica2 install' tätigen muss.

7.3. Versuchsdurchführung

Der mit angelöteten Anschlüssen für das TDS-210 versehene Stecker wird auf die Basisstation gesteckt, das TDS-210 eingeschaltet.
Dann wird zunächst die Zielstation, danach die Basisstation eingeschaltet.
Man beobachtet jetzt ein Flackern der roten LED.
Am TDS-210 werden jetzt solange die Einstellungen für die Horizontalachse verstellt, bis etwa 4-5 Impulse der LED auf dem Display zu sehen sind.
Dabei ist nach wie vor zu beachten, dass die Impulse invertiert sind, das heisst, die LED-On-Bereiche sind Ausschläge des Signals nach **unten**!
Ist der Signalverlauf eindeutig identifiziert und erkennbar, wird mit der Software ‚WaveStar' ein Snapshot getätigt und die erzeugte Tabelle ins ‚.cvs'-Format exportiert.
Dies wird ausreichend oft durchgeführt (hier: 40 mal).

Anschließend werden alle Tabellen zusammenkopiert und in einer Datei ‚1.cvs' abgelegt. Bei variierendem Dateinamen muss die Matlabdatei ‚Histogram3.h' angeändert werden (der Importdatei-Name).

7.4.Ergebnisse

Ohne die Auswertung durch MATLAB lässt sich bereits folgendes feststellen:
Die Laufzeit, die über die Brenndauer der LED gemessen wird, schwankt um ca. 110ms. Dabei sind teilweise deutliche Abweichungen zu bemerken, so dass auch Werte um 180ms bzw. unter 70ms auftreten.
Teils treten diese ‚Ausreißer' scheinbar rein zufällig auf, teils ist ein ‚Aufschaukeln' zu Bemerken, dass sich über Zeiträume von etwa 6 Impulse erstreckt, und in etwa derselben Zeitspanne wieder abklingt. Dieses ‚Aufschaukeln' tritt sowohl nach oben, als auch nach unten auf.
Eine derartige Korrelation der Messwerte über der Zeit ist in den Auswertungen durch MATLAB natürlich nicht feststellbar.
Im Gegensatz zu der anfänglichen Annahme ist es daher nicht möglich, die Laufzeiten in Form einer Verteilungsfunktion (gaußkurvenförmig) darzustellen, da die Werte nicht um einen Mittelwert mit abnehmender Häufigkeit verteilt sind.
Die Resultate aus MATLAB sehen visualisiert folgendermaßen aus (erzeugt mit ‚Histogram3.m'):

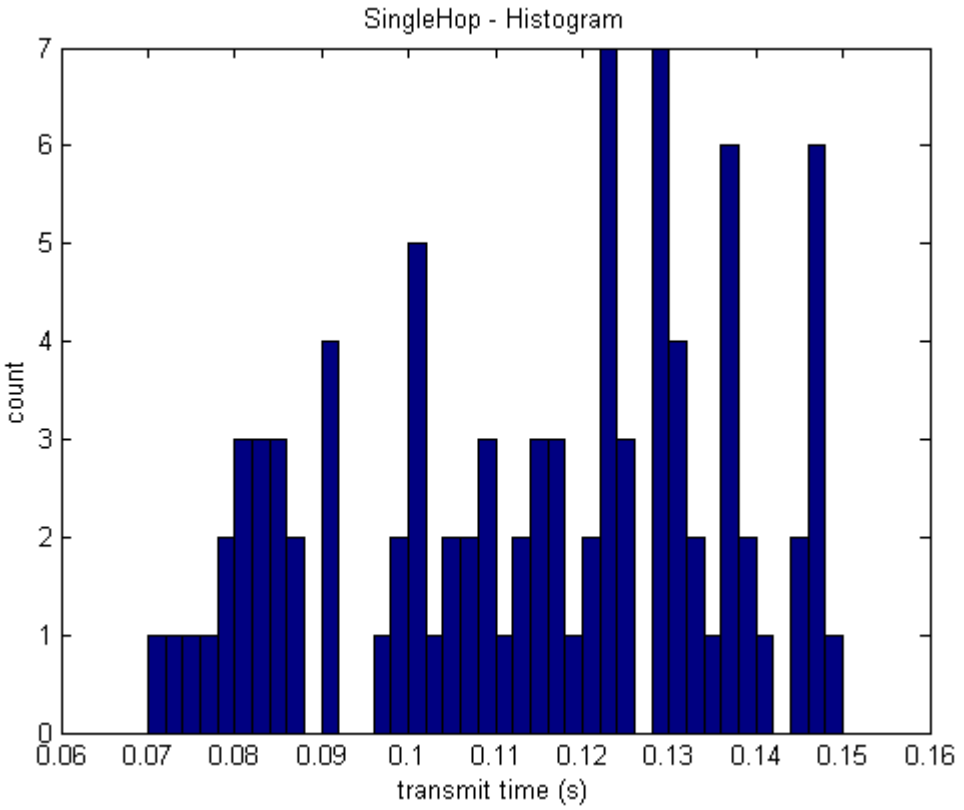

Abb. 9 (Laufzeit: SingleHop)

Mittelwert: 0.12143; Varianz: 0.0007536 (Ermittelt aus Messwerten)
Aufgetragen sind auf der X-Achse die Intervalle mit 2ms-Auflösung und auf der Y-Achse die Anzahl der in das zugehörige Intervall fallenden Messwerte.

Man erkennt deutlich, dass keine Normalverteilung vorliegt. Es deutet eher auf eine Gleichverteilung der Werte hin, da innerhalb des Intervalls 70-150ms alle Messwerte mit relativ ähnlicher Wahrscheinlichkeit auftreten. Insgesamt 91 Messwerte liegen in dem betrachteten Gesamtintervall.
Die Ursachen für diese starken Variationen der Laufzeiten können mit dem Grundprinzip des Betriebssystems TinyOS gedeutet werden.
Da es sich um ein Multitaskingsystem handelt, sind exakte Ausführungszeiten des Codes nicht zu erwarten. Ebenfalls spielt der allgemeine ‚Traffic', also der normale Datenverkehr ein Rolle. Er kann durch Störungen beeinflussen, ob z.B. eine Preambel früher oder später erkannt wird. Da die Preambel standardmäßig 18 Bytes beträgt, können hierdurch beträchtliche Schwankungen entstehen.

Dass die Gesamtzeit der Übertragung im Mittel über 100ms liegt, ist im Wesentlichen auf die Struktur einer TinyOS-Message zurückzuführen. Da das Übertragungsprotokoll großen Wert auf Sicherheit legt (18 Bytes Preambel, CRC, etc.) ist die zu übertragende Datenmenge verglichen mit den übertragenen Nutzdaten (hier: 2 Ints) relativ groß.
Hinzu kommen Extraberechnungen wie die Übertragungsstärke, die zusätzliche Zeit veranschlagen.

Eine SingleHop-Übertragung in insgesamt unter 30ms, wie es z.B. zum Regeln des ‚Invertierten Pendels' des ‚Lehrstuhls für Automatisierungstechnik der TU Kaiserslautern' ist also mit dem Standard-AM-System von TinyOS nicht möglich.

Des Weiteren ist während dem Versuchsablauf zu beobachten, dass sich das System von Zeit zu Zeit aufhängt, was nur durch Abschalten beider Komponenten (Basisstation und Zielstation) und anschließendes Neustarten des Versuchs zu beheben ist.
Die Ursache hierfür liegt in der Tatsache, dass es sich bei der gewählten Programmstruktur in keinem Fall um ein ‚sicheres' Netzwerk handelt, sondern im Gegenteil um ein möglichst ‚schnelles' (unter Nutzung der implementierten Funktionen mit Default-Werten).
Sobald die Basisstation einmal die Rücknachricht der Zielstation nicht erhält, kann das ‚able_to_send'-Flag nicht rückgesetzt werden, und es wird keine neue Nachricht generiert (die rote LED bleibt also brennen).
Eine Abhilfe dieser Problematik schafft ein ‚intelligentes' System, in dem die einzelnen Komponenten untereinander in ständiger Kommunikation durch ‚Pings' stehen, in denen Verbindungsqualität, Nachbarschaften, schnellste Wege, etc. ermittelt werden können.
Durch Meldung und Rückmeldung kann sichergestellt werden, dass ein Datenpaket definitiv am Ziel angekommen ist.
Ein solches System jedoch würde unverhältnismäßig viel langsamer arbeiten, weshalb bei z.B. bei Hochgeschwindigkeitsregelungssystemen lieber ein schnelles Funksystem gewählt, und fehlende Messwerte durch Softwarelösungen herausgerechnet werden sollten.

8. Messung: Send-SendDone (V2)

8.1. Versuchsziel

Zum Vervollständigen des 1. Versuchs soll noch die Zeit bestimmt werden, die vergeht, zwischen:
Aufrufen des ‚Send'-Commands und dem Event ‚SendDone'.
Dies ist also exakt die Zeit, die es dauert, um eine generierte Struct mit Nutzdaten durch die Hierarchiekette nach unten zu reichen, CRC zu generieren, fehlende Übertragungsdaten zu ermitteln, kurz: die vollständige Nachricht zu generieren, und diese zu senden.

8.2.Versuchsvorbereitungen

Benötigtes Material:

- 1 MICA2-Node
- TDS-210 mit TDS-2CM
- Nullmodemkabel
- modifizierter Stecker zum Anschluss des MICA2 an TDS-210
- Software ‚WaveStar'
- Dateien: ‚Basisstation_5M.nc'
- Ordner: ‚Basisstation_5' mit Inhalt

In diesem Fall ist kein zweiter MICA2-Node nötig, da die Sende-Funktion einfach an den Timer gekoppelt wird, ohne das ‚able_to_send'-Flag zu nutzen. Denn nur dieses hat dafür gesorgt, dass erst nach erhalt der Rücknachricht von der Zielstation der Sendemodus wieder freigeschaltet wurde.
Der zentrale Code in ‚Basisstation_5M.nc' lautet daher:

```
      command result_t IntOutput.output(uint16_t value)
      {
      IntMsg *message = (IntMsg *)data.data;
            call Leds.redOn
            if (!busy)
            {
                  busy = TRUE;
                  message->val = value;
                  atomic {
                        message->src = TOS_LOCAL_ADDRESS;
            }
                  if (call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg),
&data))
                    return SUCCESS;
                  busy = FALSE;
            }
            return FAIL;
      }
```

```
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m)
{

}

event result_t Send.sendDone(TOS_MsgPtr msg, result_t success)
{
      if (busy && msg == &data)
      {
            call Leds.redOff();
            busy = FALSE;
            signal IntOutput.outputComplete(success);
             }
      return SUCCESS;
}
```

Das Event ‘Receive’ ist nun also völlig ohne Bedeutung, da keine zurücksendende Zielstation existiert. Hier sei nur noch einmal angeführt, dass das Event trotzdem implementiert sein muss, da es im Interface steht und der Compiler ansonsten einen Fehler liefert!

Die Installation geschieht wie im vorangegangenen Versuch: Der Applikationsordner muss sich im ‚Apps’-Verzeichnis der TinyOS-Installation befinden. Die Datei ‚Basisstation_5M.nc’ im ‚Lib’-Ordner.
Die Applikation wird wieder über ‚make mica2 install’, ausgeführt im Applikationsordner, installiert.

8.3.Versuchsdurchführung

Wie in Versuch 1 wird das Oszilloskop an den MICA2-Node angeschlossen und entsprechend justiert. Die Impulszeiten sind jetzt kürzer, jedoch ist die Clock-Frequenz gleich geblieben (4Hz). Daher empfiehlt es sich, dieselbe Display-Auflösung des TDS-210 wie zuvor zu wählen.
Erneut werden mehrere Tabellen mit ‚WaveStar’ erzeugt (hier: 40), exportiert und zu einer einzigen Datei zusammengefasst (hier: ‚2.csv’).
Entsprechend liefert die Auswertung durch MATLAB die Grafik in Abb. 6.
Erneut ergibt sich keine Verteilung, die einer Gaußkurve ähnelt.
Die Messwerte scheinen innerhalb des Intervalls 35ms<X<80ms recht gleichmäßig verteilt zu sein.
Die Auflösung der Edges in der vorliegenden Grafik ist dieselbe wie Abb. 5.
Es liegen 121 Messwerte zugrunde.
Eine Paketausfallrate ist natürlich nicht zu beobachten, da keine Übertragung betrachtet wird.
Die Ergebnisse decken sich recht gut mit denen des ersten Versuchs: multipliziert man die Übertragungszeiten des Sendevorgangs mit 2, erhält man ziemlich genau die Werte für ein Hop-And-Back-Routing wie in Versuch 1.

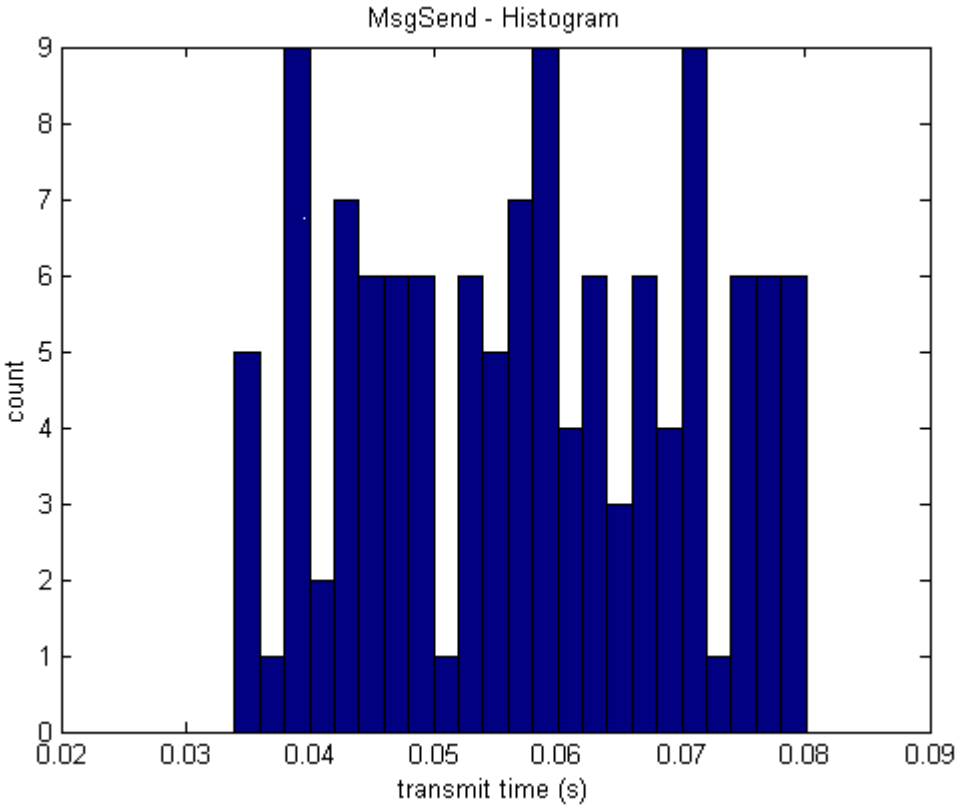

Abb. 10 (Laufzeit: Send-SendDone)

Mittelwert: 0.061337; Varianz: 0.00024252 (Ermittelt aus Messwerten)

9. Beispiele zu größeren Netzwerken

9.1. Netzwerk mit Router

Gemäß den beiden vorangegangenen Versuchen wird jetzt ein vergleichbares Setup für ein größeres Netzwerk mit einem Router zwischen Basis- und Zielstation angeben.

Sobald mehr als zwei MICA2-Komponenten an einem Netzwerk beteiligt sind, wird die Kommunikation schwieriger. Angenommen, ein Router sendet die von der Basis empfangene Nachricht an die Zielstation weiter. Da er als Router fungiert, muss er im Empfangsradius sowohl der Basis- als auch der Zielstation liegen. Das bedeutet, dass jede Nachricht, die der Router abstrahlt, von beiden anderen MICA2 empfangen wird.
Das wird insoweit zum Problem, als die Nachricht, die eigentlich nur für die Zielstation bestimmt war, bei der Basisstation unnötige Rechenzeit verbraucht. In derselben Zeit, in der die Basis die unnötige Message empfängt, verpasst sie eine evtl. wichtige Nachricht eines weiteren MICA2.
Ein Beispiel für eine solche Konfiguration bietet die Applikation ‚Basisstation_2' zusammen mit ‚Router_2' und ‚Zielstation_2'. Die zugehörigen ‚M'-Dateien sind ‚Basisstation_2M.nc', Router_2M.nc' und ‚Zielstation_2M.nc'.
Wie gehabt werden sie auf nun 3 MICA2-Nodes installiert. Eingeschaltet werden sie in der Reihenfolge ‚Zielstation', ‚Router', ‚Basis'.
Das System scheint zu funktionieren, da die LEDs Aktivität zeigen bis zu dem Moment, in dem das System aufgrund schon oben beschriebener Umstände abstürzt.
Wenn man sich jedoch den Code ansieht, sieht man, wie umständlich die Verwaltung der verschiedenen Messages ist:

Basisstation_2M.nc:

```
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m)
{
      IntMsg *eingang = (IntMsg *)m->data;
      if(eingang->src != TOS_LOCAL_ADDRESS)
      {
            call Leds.redOff();
            able_to_send = TRUE;
      }
      return m;
}
```

Das ‚Receive'-Event, das normalerweise das Flag ‚able_to_send' freigibt, um eine neue Nachricht zu ermöglichen, wird nun nur freigegeben, wenn die empfangene Nachricht im Datenfeld ‚src' **nicht** die eigene ID enthält. ‚src' enthält in diesem Netzwerk stets die ID desjenigen Nodes, der die jeweilige Nachricht generiert hat. D.h. eine Nachricht von der Basisstation enthält die ID der Basis im ‚src'-Feld des Payloads, eine rückläufige Nachricht vom Zielknoten enthält dessen ID an selbiger Stelle.
Da dieses Feld vom Router nicht editiert wird, kann die Basis anhand einer Prüfung dieser Variable erkennen, ob sie ihre eigene Nachricht vom Router empfangen hat, der sie eigentlich nur an die Zielstation weiterleiten soll.
Die Zielstation enthält einen ähnlichen Code.

Zielstation_2M.c:

```
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m)
{
      IntMsg *eingang = (IntMsg *)m->data;
      IntMsg *ausgang = (IntMsg *)data.data;
      if (!busy && eingang->src != TOS_LOCAL_ADDRESS)
      {     busy = TRUE;
            atomic {    eingang->count++;
                        ausgang->val = eingang->val;
                        ausgang->src = TOS_LOCAL_ADDRESS;
                        ausgang->count = eingang->count;
                  }
            call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &data);
      }
      return m;
}
```

Auch hier wird eine rückläufige Nachricht nur generiert, wenn zuvor eine Nachricht empfangen wurde, die im ‚src'-Feld **nicht** die eigene ID enthält:
if(eingang != TOS_LOCAL_ADDRESS).
Der Code des Routers ist sogar noch komplexer, da er sich zuerst einmal quasi im Netzwerk ‚orientieren' muss. Für dieses Netzwerk geht der Router davon aus, dass die erste eintreffende Nachricht von der Basis kommt. Er leitet sie weiter, ohne das ‚src'-Feld zu manipulieren. Sobald eine Message eintrifft, die im ‚src'-Feld nicht die ID der Basis hat, wird diese neue ID als die der Zielstation deklariert.
Im folgenden muss ständig unterschieden werden, ob zur Zeit auf eine Nachricht der Basis oder der Zielstation gewartet wird. Der Code dafür sieht folgendermaßen aus:

Router_2M.nc:

```
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m)
{
      IntMsg *eingang = (IntMsg *)m->data;
      IntMsg *ausgang = (IntMsg *)data.data;
      if(base != -1 && end == -1)
      // Zweite eintreffende Nachricht -> Zielstation
      {
            end = eingang->src;
            waiting_4_end = FALSE;
            waiting_4_base = TRUE;
      }
      if(base == -1 && end == -1)
      // Erste eintreffende Msg -> Zuordnung: Basis
      {
            base = eingang->src;
            waiting_4_base = FALSE;
            waiting_4_end = TRUE;
      }
      if(waiting_4_base == TRUE && waiting_4_end == FALSE &&
      eingang->src == base)
      {
            if (!busy)
                  {
                  busy = TRUE;
                  atomic {
                        eingang->count++;
                        ausgang->val = eingang->val;
                        ausgang->src = eingang->src;
                        ausgang->count = eingang->count;
                              }
```

```
                    call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg),
                    &data);
                    }
        }
        if(waiting_4_base == FALSE && waiting_4_end == TRUE &&
        eingang->src == end)
        {
                if (!busy)
                    {
                          busy = TRUE;
                          atomic {
                                eingang->count++;
                                ausgang->val = eingang->val;
                                ausgang->src = eingang->src;
                                ausgang->count = eingang->count;
                                      }
                          call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg),
                          &data);
                          }
                }
        return m;
}
event result_t Send.sendDone(TOS_MsgPtr msg, result_t success)
{
        if (busy && msg == &data)
        {
              if(waiting_4_base == TRUE && waiting_4_end == FALSE &&
              busy == TRUE)
              {     waiting_4_base = FALSE;
                    waiting_4_end = TRUE;
                    busy = FALSE;
              }
              if(waiting_4_base == FALSE && waiting_4_end == TRUE &&
              busy == TRUE)
              {     waiting_4_base = TRUE;
                    waiting_4_end = FALSE;
                    busy = FALSE;
              }
          }
```

Zunächst ist zu bemerken, dass einige Funktionen, wie der Hop-Counter, keinen direkten Nutzen für den Ablauf des Programms haben. Er ist lediglich als Beispiel für Datengewinnung aus dem Netzwerk heraus implementiert.
Man erkennt, dass eine ständige Fallunterscheidung notwendig ist, da auch der Fall zu betrachten ist, dass eine Nachricht aufgrund irgendwelcher Umstände nicht gesendet werden konnte. Daher muss das Kippen der ‚waiting_4_...'-Flags in den ‚SendDone'-Event-Block ausgelagert werden. Diese Flags sorgen dafür, dass in Abhängigkeit des ‚src'-Feldes in den eintreffenden Messages abwechselnd Nachrichten der Basis- und Zielstation weitergeleitet werden.
In diesem speziellen Fall wäre es eigentlich nicht nötig, eine Fallunterscheidung über das erste Zuweisen der Variablen ‚base' und ‚end' hinaus zu treffen: Basis und Zielstation würden sowieso keine zweite Nachricht aussenden, ehe sie nicht die Rückmeldung vom jeweils anderen Ende hätten – der Router würde also sowieso nur abwechselnd von beiden Seiten Nachrichten erhalten.
Jedoch muss bedacht werden, dass evtl. mehrere Router nach verschiedenen Seiten der Basis weg existieren. In dem Fall kann es vorkommen, dass der Router eine Nachricht der Basis erhält, die für einen anderen Netzwerkzweig bestimmt ist, obwohl er schon kurz zuvor von dort eine für ‚seinen' Zielknoten bestimmte Message erhalten hat.

In einem solchen Fall ist es wichtig, dass die zweite Nachricht nicht weitergeleitet wird, sondern weiterhin auf die Rückmeldung des Zielknotens gewartet wird. An dieser Stelle zeigt sich aber nun das schwerwiegende Problem dieses Systems:
Empfängt der Router eine Nachricht der Basis für einen anderen Netzwerkzweig, kann er evtl. die zeitgleich erfolgende Rückmeldung des Zielknotens nicht empfangen.
Der Router verbringt sinnlos Zeit damit, Nachrichten auszuwerten, die im Endeffekt vielleicht gar nicht für ihn bestimmt waren.

An dieser Stelle soll eine mögliche Lösung dieses Problems aufgezeigt sein:

9.2.Verbessertes Netzwerk mit Router

Nutzt man statt dieser Methode, eine Nachricht auszuwerten um den Bestimmungsort zu ermitteln, eine andere, die auf Direktadressierung beruht, lässt sich eine Menge Rechenzeit sparen.

Hierzu ist das Installationsverfahren zu nutzen, das schon zu Beginn dieses Berichts in Abschnitt 3 vorgestellt wurde.
Jedem MICA2-Node lässt sich eine feste ID zuweisen, indem man folgenden Installationsparameter nutzt:
`make mica2 install.x`,
wobei x der zugeteilten ID entspricht. Das AM-Modul von TinyOS arbeitet derart, dass in einer generierten Message die erste übertragene Nutzinformation (nach Preambel etc.) die Zieladresse ist. In sämtlichen bisher vorgestellten Codes lautete diese beim Aufruf der ‚Send'-Funktion:
`call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &data))`
Hierbei entspricht ‚TOS_BCAST_ADDR' einer Konstanten, die für ‚'Alle MICA2' steht. Die Nachricht wird also von allen Nodes empfangen.
Statt der Konstanten ‚TOS_BCAST_ADDR' kann hier auch eine Konstante angegeben werden. Auf diesem Weg kann ein MICA2-Node direkt angesprochen werden.
Dies funktioniert, da der Radio-Stack der Nodes eine Nachricht sofort verfallen lässt (‚drop'), wenn die erste Information nach der Preambel nicht die eigene ID oder der Globalcode (‚TOS_GLOBAL_ADDR') ist.
Es ist dann nicht mehr nötig, den gesamten Rest der Message zu empfangen, und der Node kehrt sofort in den Empfangsmodus zurück.
Auf diese Weise kann wertvolle Arbeits- und Rechenzeit gespart werden.
Einen MICA2-Node kann man also direkt ansprechen, indem der ‚Send'-Aufruf mit einer Klartext-ID erfolgt, oder aber eine Variable verwendet wird.
Letzteres könnte folgendermaßen aussehen:
Zu Beginn des Netzwerks läuft eine Initialisierungsroutine ab, in der alle Nodes ihre ID über den Global-Channel funken. Auf diese Weise erfährt jeder Node, welche anderen MICA2 in seinem Radius liegen. Diese IDs können in einem Array abgelegt werden.
Eine Pathfinding-Routine generiert Hop-Ketten, die aus den IDs derjenigen Nodes besteht, die eine Message passieren muss, um zu einem beliebigen Zielknoten zu gelangen.
Ein MICA2 muss also lediglich die Ziel-ID erhalten (unter der Voraussetzung, dass jedem Node alle anderen Verbindungen bekannt sind), um aus dem ID-Array eine Folge-ID herauszugreifen zu können, die die Nachricht zum nächsten Node weiterleitet.
Hier bietet sich ein Ansatz für eine dynamische Strukturverwaltung eines Netzwerks.
Um eine Veranschaulichung zu bieten, können die Applikationen ‚Basisstation_3', ‚Router_3' und ‚Zielstation_3' installiert werden.

Sie stellen grundsätzlich dieselbe Netzwerkstruktur dar, wie es auch schon das letzte Beispiel tat, nutzen jedoch Direktadressierung.
Für ein ordnungsgemäßes Funktionieren müssen folgende IDs zugewiesen werden:
Basisstation: 0
Router: 1
Zielstation: 2
Basisstation und Zielstation addressieren jetzt den Node mit der ID ‚1' an. In das Payload-Datenfeld ‚val' wird die Zieladresse der Nachricht eingetragen. Im Falle der Basis ist das 2, im Falle der Zielstation ist es 0.
In das Feld ‚src' wird die Adresse des Verfassers eingetragen, also 0 für die Basis, 2 für den Zielnode.
Der Router tut nichts anderes, als eine neue Nachricht zu generieren, in der alle Werte übernommen werden, und setzt als Zieladresse die Variable ‚val' ein.
Natürlich funktioniert dieses Verfahren bei zwei Routern nicht mehr. In dem Fall müsste entweder eine feste Reihenfolge vorgegeben werden, oder die Nodes müssten sich derartige Informationen wie oben beschrieben selbst besorgen.
Hier nur der Code des Routers:

```
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m)
{
      IntMsg *eingang = (IntMsg *)m->data;
      IntMsg *ausgang = (IntMsg *)data.data;
                    if (!busy)
                        {
                        busy = TRUE;
                        atomic {
                              ausgang->val = eingang->val;
                              ausgang->src = eingang->src;
                              }
                        call Send.send(eingang->val, sizeof(IntMsg),
                        &data);
                        }
return m;
}
```

Wie man sieht, ist keine großartige Verwaltungsroutine zu schreiben, die allerdings auch ein starres Netzwerk voraussetzen würde.

10 . Quellen und Verweise

Quellenangaben:

[1] Nelson Lee, Philip Levis, Jason Hill: ‚Stack.pdf'; im Lieferumfang enthalten

[2] Markus Reif: ‚Mica2_Layout.pdf'; in der Anlage

[3] Fa. Crossbow: ‚lesson1.html'; letztes Update: 31.07.2002; im Lieferumfang enthalten

[4] Berkeley, University Of California: TinyOS-Dokumentation; http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/uninstall.html ; letzter Zugriff am 09.01.2004

Verweise:

Homepage der Fa. Tektronix:	www.tektronix.com ; letzter Zugriff am 09.01.2004
Homepage der Fa. Crossbow:	www.xbow.com ; letzter Zugriff am 09.01.2004
Homepage von Berkeley, University Of Calaifornia:	http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/ ; letzter Zugriff am 09.01.2004

Kontakt

Prof. Dr.-Ing. habil. Lothar Litz
Technische Universität Kaiserslautern
Lehrstuhl für Automatisierungstechnik
Postfach 3049
67653 Kaiserslautern
Tel.: +49 (0) 631/205-4450
Fax.: +49 (0) 631/205-4462
E-Mail: litz@eit.uni-kl.de
URL: http://www.eit.uni-kl.de/litz

Dipl.-Ing. Oliver Gabel
Adresse wie oben
Tel: +49 (0) 631/205-4459
E-Mail: gabel@eit.uni-kl.de