Eclipse Plugin for TinyOS Debugging

Semester Thesis

Silvan Nellen
snellen@ee.ethz.ch

Advisors:
Benjamin Sigg
Philipp Sommer

Supervisor:
Prof. Dr. Roger Wattenhofer

Distributed Computing Group
Computer Engineering and Networks Laboratory (TIK)
Department of Information Technology and Electrical Engineering

February 2009

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Abstract

TinyOS is a widely used open source operating system for embedded systems written in NesC. NesC code is first compiled into a C program which is then processed by an ordinary C compiler. Since there is no dedicated NesC debugger a normal C debugger is used for debugging. th C debugger is unaware of the NesC code so the user has to have some knowledge about the implementation of the NesC compiler, something a TinyOS developper should not have to worry about. This paper presents a solution which allows the developer debug a TinyOS application without being aware of the implementation of the NesC compiler. A TinyOS debug plug-in for Eclipse is presented. The Eclipse plug-in facilitates debugging by integrating support for variable examination, breakpoints and automatic launching of debug sessions. First testers have found the presented Eclipse plug-in to be a useful tool for TinyOS developers.

Contents

1 Introduction

Sensor networks are a new class of distributed systems [1]. Instead of processing data generated by humans like a conventional computer they become an integral part of their environment. Sensor nodes are deployed in a terrain and form a self organised wireless network. With a multitude of sensors they gather information about their environment. Possible applications for sensor networks include habitat and environment monitoring, health care, industrial machinery surveillance, home automation and military surveillance (Joachim Hof [2]). Given the broad range of applications many hardware platforms for sensor nodes exist. Common requirements for sensor node platforms are unattended operation, low energy consumption and dynamic adaption to the environment. Popular platforms include BTnode[1], the MICA platform[2], the TelosB platform [3] and the Intel Mote2 [3].
Several commercial and non commercial operating systems for sensor nodes exist. MANTIS[4] is a freely available lightweight operating system for sensor nodes. It was developed at the Department of Computer Science at the University of Colorado at Boulder and implements multi threading to support parallel execution. MANTIS is written in standard C. The ZigBee[5] standard defines the layers of the protocol stack. Several commercial implementations of the standard exist for example by Ember[6], TI[7], Freescale[8] and Jennic[9]. Contiki[10] is an highly portable open source operating system for embedded systems. It is designed for microcontrollers with small amounts of memory and provides multithreaded parallelism and a energy efficient radio communication mechanism. TinyOS [4] is a very popular open source operating system developed at the University of Berkeley in California. It has been ported to over a dozen platforms and is used by over 500 research groups and companies.

[1] http://www.btnode.ethz.ch/
[2] http://www.xbow.com
[3] http://www.xbow.com/Products/productdetails.aspx?sid=252
[4] http://mantis.cs.colorado.edu
[5] http://www.zigbee.org
[6] http://www.ember.com
[7] http://www.ti.com/
[8] www.freescale.com/zigbee
[9] http://www.jennic.com/
[10] http://www.sics.se/contiki/

Figure 1.1: From left to right: BTnode, MICA, TelosB and Intel Mote2.

1.1 TinyOS and NesC

TinyOS was designed to meet the requirements of sensor networks such as limited resources, event-centric concurrent applications, and low-power operation. The provided libraries include network protocols, distributed services, sensor drivers and data acquisition tools. It has an event driven concurrency model based on asynchronous events and deferred computation called tasks. Events are a software abstraction of hardware events such as completion of sensor sampling or the arrival of a packet on the radio. Events trigger the execution of a corresponding handler. Event handlers have the highest priority and always run until completion. To keep the OS responsive a handler should take as little time as possible to execute. Complex calculations in TinyOS are supported by tasks. Tasks are executed sequentially and can be preempted by events. They can be posted by event handlers or other tasks and are executed in first in first out (FIFO) order.
TinyOS uses a simple memory model. All memory is allocated statically, tasks and event handlers share one memory space and there is no memory protection mechanism. To prevent race conditions on data shared between tasks and event handlers TinyOS supports the definition of atomic sections. Code in an atomic section is guaranteed to run atomically.
The operation system services such as sending/receiving messages and reading sensors values are accessed via split phase interfaces. A request for a service returns immediately. The result of the request is reported via callback at a later time.
TinyOS is implemented in NesC. NesC is an extension to the C language which was designed to implement the concepts and execution model of TinyOS ([5]). Components are the basic concept behind NesC. A component is a functional unit that implements a set of services specified by interfaces. An interface is an abstract definition of commands and events. A command is typically a request to a component to perform a service, an event signals the completion of that service. A component has two classes of interfaces: those it provides and those it uses. Interfaces are bidirectional. Users of an interface may call commands and have to implement event handlers. Providers of an interface have to implement commands and may signal events. Figure 1.2 shows two modules. Module A uses interface I which means it can call commands and receives events. Module B provides the interface I, it implements the commands and signals events.
There are two types of components: modules and configurations. Modules implement code for calling commands and receiving events. They contain private state variables and data buffers and declare private functions for internal use. Configurations are used to wire components together. Each interface a component uses has to be wired to a component that provides that interface. For example a module called Blink (Figure 1.3) which flashes the on board LEDs of the sensor node declares to use an interface called Leds. An application that uses this component wires the interface Leds to a module which provides the Leds interface (Figure 1.4). On the first line of the implementation two modules Blink and LedsC are declared. On the second line the Leds interface is wired to the LedsC configuration which provides that interface.

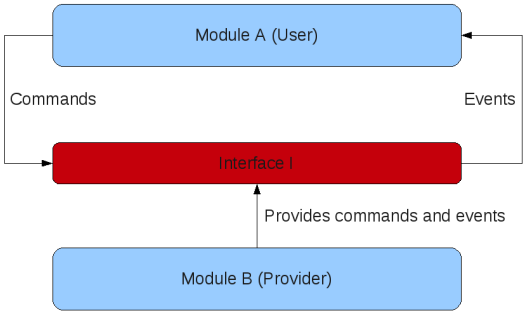

Figure 1.2: The NesC component model.

```
File Blink.nc
module Blink {
  uses interface Leds;
  ...
}
implementation {
...
}
```

Figure 1.3: The definition of the Blink module.

```
File BlinkApp.nc:

configuration BlinkAppC {
}
implementation {
  components Blink, LedsC;
  Blink.Leds -> LedsC;
}
```

Figure 1.4: The wiring of the Blink module.

TinyOS applications are compiled with the NesC compiler nescc[11]. The compilation is done in two steps.

1. All NesC source files are pre-processed to generate standard C code which is written into one file.

2. The generated code is compiled using gcc. The resulting binary contains all code necessary to run the application.

1.1.1 Debuging TinyOS sensor nodes

Debugging is an integral part of software development [6]. It not only helps to find and correct mistakes it also give insight into the inner workings of a program. With no human friendly input/output it is hard to learn what is going on inside a sensor node. One way to learn about the state of the node is using the serial console to print debug messages or send debug messages over the radio. These approaches are invasive, they require changes in the code, recompilation and they alter the program flow. In some cases they may simply not work, for example when the operating system crashes it might not be possible to send any messages over the radio or the serial console.
A way to solve this is to debug using specialised hardware while the application is running on the device. The set up for on chip debugging using the JTAG interface of the node is shown in Figure 1.5. A JTAG adapter is connected to the USB port of the developer PC. The adapter implements the JTAG protocol with chip specific extensions. The application running on the node can be interrupted at any time and breakpoints can be set on instructions. Once the program is stopped the memory and registers of the chip can be read and written.

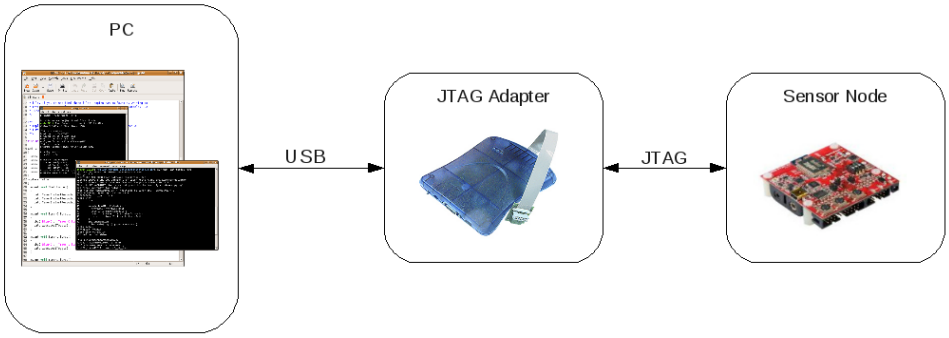

Figure 1.5: Debug setup.

The TinyOS documentation[12] suggests to use the GNU debugger (GDB) on the development PC. Since GDB does not have a NesC specific mode the generated C code is debugged. The NesC preprocessor uses the line directive to provide a mapping from the generated C code to the original NesC code. This means that single stepping through the code displays the correct line number in the NesC files. The component model of NesC is mapped to the component unaware C language by using the names of symbols (variables, functions) to store the information about which symbol belongs to which component.

[11] http://sourceforge.net/projects/nescc
[12] http://www.tinyos.net/tinyos-1.x/doc/nesc/nesc-debugging.html

1.2 Goals of this Project

The goal is to simplify debugging of TinyOS applications. Debugging the C code generated during compilation is not entirely trivial and requires some knowledge about the inner workings of the NesC compiler, something an ordinary TinyOS developer should not have to worry about. To free the developer from this burden was an important goal of this project. Graphical integrated development environmens (IDEs) like Microsoft Visual Studio, NetBeans and Eclipse are becoming more and more a standard for software development. With syntax highlighting editors, integrated compilers and support for automatic building they make programming easier. Many of them include a graphical debugger which allow to start a program in debug mode. The running process can be interrupted to examine it or even change its state. This project aims at bringing this experience to the TinyOS development community. Eclipse is a widely used extensible open source IDE for which several solutions for TinyOS development exist (see Section 1.3). None of them provides debugging support. In this project the Yeti 2 TinyOS plug-in for Eclipse was extended with basic graphical debugger support.
The following tasks have been identified:

- Implementation of an interface between Java and the TinyOS-Debugging-Tools (ice-gdb, avr-gdb, gdb).
- Implementation of a mapping between original NesC -Code and the C-Code the debugging tools work with.
- Implementation of a plug-in for eclipse which supports at least setting breakpoints and reading variables.
- Concepts and implementation to automate the steps between compilation and debugging.
- Modification of Yeti 2 to seamlessly integrate the newly developed plug-in. Yeti 2 should not be dependent on the debug plug-in

For this project the following scenario was considered. The TinyOS Yeti 2 Eclipse plug-in is used to develop software for a sensor node based on the Atmel AVR micro-controller and the JTAG interface is used to access the on chip debug module of the node.

1.3 Related Work

Several Eclipse[13] plug-ins for TinyOS development have been created.
The Yeti 2 eclipse plug-in [7] provides an IDE for TinyOS 2.x development. The integrated NesC parser allows real time code validation, syntax highlighting, context sensitive code completion and hyperlink navigation across files and to definitions. Compilation and flashing from within Eclipse and stub generation for interfaces is also provided. The component graph displays a graphical representation of the modules and wirings of the TinyOS application. It has no support for debugging.
The NESCDT[14] eclipse plug-in provides automatic detection of NesC files, a NesC editor with syntax highlighting and auto completion for keywords and interface members. It has no support for debugging or automatic building.

[13] http://www.eclipse.org
[14] http://docs.tinyos.net/index.php/NESCDT

TinyDT[15] is an eclipse plug-in that implements an IDE for TinyOS 1.x. development. Its features include syntax highlighting, code completion for interfaces, automatic build support, support for multiple target platforms. TinyDT includes a NesC parser which is used to build an in memory representation of the nesC application. This includes the component hierarchy, wirings, interfaces and the NesC documentation. TinyDT has no support for debugging.
The VDB debugging utility[16] provides a way of debugging TinyOS applications. It provides an assert function and utilities to write log messages to the on chip EEPROM. VDB includes a terminal (VDBTerm) which is connected to the node via serial line and displays the output generated by print statements. Debugging using VDB is invasive and requires recompilation of the code.

[15] http://tinydt.sourceforge.net
[16] http://www.cs.virginia.edu/ lg6e/tool/debug.html

2 Design

This chapter describes the design of the TinyOS debugging solution implemented in this project. The architecture will be introduced briefly in section 2.1. The rest of this chapter discuses the components in more detail.

2.1 Architecture

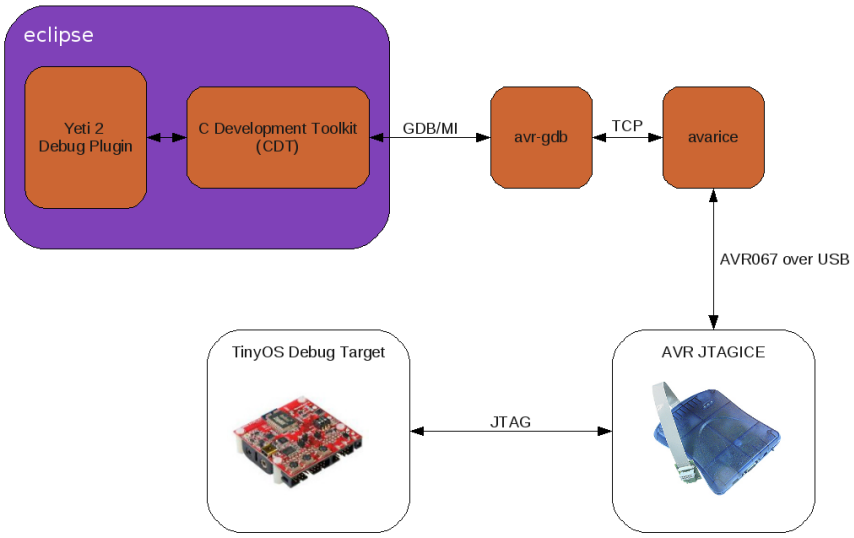

Figure 2.1: Set up of a debug session.

Figure 2.1 shows the set up for debuging sensor nodes equiped with an AVR micro-controller running TinyOS. The Yeti 2 debug plugin uses the C Development Tools plugin to communicate

with a AVR specific variant of the gnu debugger (GDB[1]) called avr-gdb. When launching a debug session avr-gdb connects to AVaRICE[2]. GDB sends debug commands to AVaRICE which translates them to the AVR067 protocol supported by the Atmel JTAG ICE. The JTAG ICE controls the on-chip-debugging facilities of the target device.
The components of the debug session form a chain of dependency with the sensor node being the first and the Yeti 2 debug plugin being the last element. The rest of this chapter discuses the most important elements of the TinyOS debug chain.

2.2 JTAG

The AVR JTAG ICE adapter is the second element in the debug chain. It interfaces with the on-chip debug system of the sensor node ([8]). It allows to monitor and control the execution of applications running on the AVR chip. When a program is running the JTAG ICE will continuously poll the target device to see if a breakpoint has been reached. If this is the case the program counter, all registers and the content of the RAM are read out and transmitted to the PC connected to the device. Once stopped the code can be executed instruction for instruction ("single stepping") or the device can be set back to run mode.
The AVR JTAG ICE implements the JTAG standard for testing integrated circuits. JTAG is the common name for the IEEE 1149.1 standard entitled "Standard Test Access Port and Boundary-Scan Architecture". The standard was established by the Joint Test Action Group (JTAG) in 1985. It distinguishes four basic hardware elements [9].

1. Test access port (TAP)
2. TAP controller
3. Instruction Register (IR)
4. A set of data registers (TDR)

The TAP controller generates clock and control signals for the JTAG circuitry on the chip. The TAP controller is implemented as a finite state machine. The instruction register is serially written and controls the state transitions of the TAP controller. The test access port consists of the following pins: Test data in (TDI), Test data out (TDO), Test clock (TCK), Test mode select (TMS) and Test reset (TRST) which is optional.
The basic principle behind IEEE 1149.1 is a boundary scan architecture. In a boundary scan device every input and output pin is supplemented with a memory cell. The memory cells of the integrated circuits on a board form a shift register called the Boundary-Scan register (BSR). The BSR is a mandatory part of the data registers. The first cell is connected to the test data in (TDI) pin, the last cell is connected to the test data out (TDO) pin. The shift register is clocked with the TCK pin. In normal mode the cells simply forwards the signal on the pin. In test mode the value for each pin can be set by shifting it from the TDI pin through all preceding registers to the destination. Similarly the value of a pin can be read by shifting it through all the succeeding registers to the TDO pin. Figure 2.2 shows an example of a boundary scan enabled circuit board. Every chip on the board is equipped with memory cells on the boundary (the input output pins) to the other chips. The boundary scan memory cells of the chips on the board are linked together so that the pin of every chip on the board can be accessed through the test pins.

[1] http://www.gnu.org/software/gdb/
[2] http://avarice.sourceforge.net/

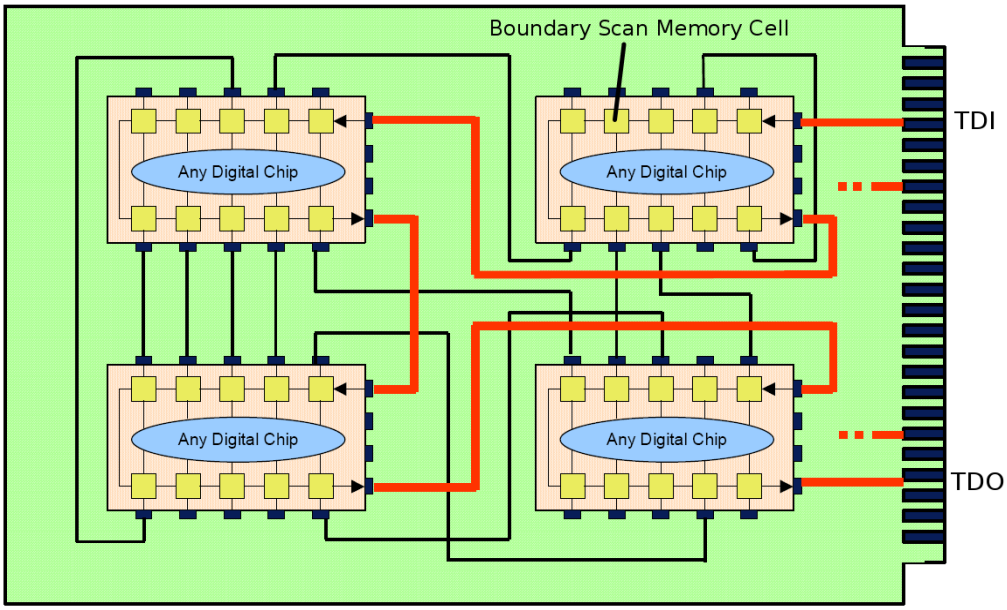

Figure 2.2: An example of a boundary scan architecture. (Picture from [10])

Atmel extended the JTAG interface of AVR micro-controllers non standard commands to allow reading memory and internal registers as well as stepping through code and setting breakpoints [8].

2.3 GDB/AVaRICE

The Eclipse IDE is designed to rely on GDB to examine the state of programs during execution. According to [11] GDB handles four main tasks :

- Starting programs specifying any parameters that might affect its behaviour.
- Stop the program on predefined conditions (Breakpoints, Watchpoints).
- Examining the state of the program when it is stopped (Stack trace, examine variables)
- Change the state of the program (single stepping through code).

GDB supports programs written in C/C++ and has partial support for Modula-2, Pascal, Fortran and objective C. The standard way to use GDB is via the command line interface. There are several front ends such as ddd, Xxgdb and IDEs such as eclipse and netbeans which can interface with GDB using the machine friendly GDB/MI (MI stands for machine interface) interface. GDB/MI involves three parts: commands sent to GDB, responses to these commands and notifications about events. Each command results in one response, either indicating success or failure. Notifications are generated when the state of the target changes (for example when a breakpoint is hit). To debug a program it is either started from within GDB or GDB is attached to a running process on the local machine. The generic serial protocol can be used to interface with a remote target if it is not powerful enough to run GDB.
The AVR micro-controllers used in this project support a customized version of the JTAG protocol[8]. GDB does not support the JTAG protocol directly, instead AVaRICE is used to translate the commands of GDB to the proprietary AVR06 protocol that the JTAG ICE understands

(see [12]). AVaRICE runs on the developer machine and listens on a TCP port for incoming connections. GDB connects and communicates with AVaRICE using the GDB serial protocol. The commands it receives from GDB are translated and sent to the Atmel AVR JTAG ICE adapter.

2.4 Eclipse

Eclipse is an extensible platform for building integrated development environments (IDE). The Eclipse IDE consists of a set of plug-ins which use the Eclipse Software development kit (SDK). The SDK is developed in the Java language and therefore supports a wide range of operating systems like Microsoft Windows, Mac OSX, Linux, Solaris, AIX and HP-UX. The architecture of Eclipse is shown in Figure 2.3. The basic building blocks of eclipse are ([13]):

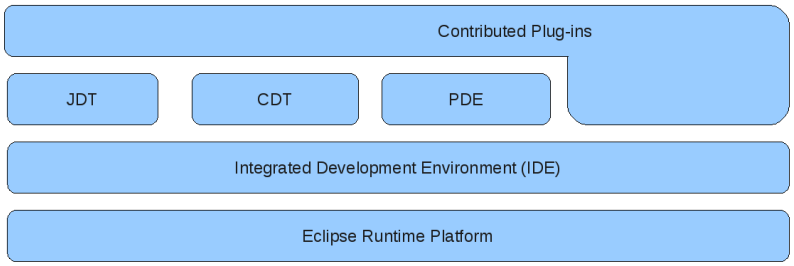

Figure 2.3: Architecture of the Eclipse SDK.

Eclipse Runtime Platform The runtime platform provides basic services like a plugin registry, resources, user interface and update facilities. The plugin registry is used to manage and load installed plug-ins. Resources provides a platform independent view of files and folder and projects.

Integrated Development Environment The integrated development environment provides common functionality necessary to develop programs. This includes a centralized dialog to define preferences, a search engine to search files and present results, a language independent debug model and support for version control repositories.

Eclipse is designed to be flexible. Plug-ins are built upon other plug-ins and the eclipse platform. Each plugin can expose part of its functionality in extension points. Other plug-ins can use these points to extend the functionality. Several plug-ins for different programming languages have been developed. The Java development tools (JDT[3]) implement an IDE for developing applications in Java. The Plug-in Development Environment (PDE[4]) provides tools to create, develop, debug, build and deploy Eclipse plug-ins.

[3] http://www.eclipse.org/jdt/
[4] http://www.eclipse.org/pde/

The Eclipse graphical user interface (GUI) is termed the ”Workbench”. It consists of views, perspectives and editors.

View A view is a visual component within a workbench. Views typically display information about the resources (files, folders) of a project, open editors and display information about the current editor. For example the package explorer is used to browse and open projects.

Perspective A perspective is a collection of views and editors which are displayed together on the workbench. For example the debug perspective displays all relevant view for debugging applications.

Editor Editors are used to edit source files. Multiple editors may be opened on one workbench window.

By default all eclipse projects and settings are stored in one directory called the workspace. The workspace to use is determined at start up and can not be switched during runtime.
To start the program under development is termed ”launching” in Eclipse. An extensible framework for dealing with launching is provided. Launching is centred around two entities: *launch configuration types* and *launch configurations*. The type of the configuration defines what happens when the configuration is launched. For example the Java launch configuration type is used to launch Java applications, the Java debug configuration is used to debug Java application. Each launch configuration type has a graphical user interface to define the parameters of the launch. A set of parameters of a launch are called a *launch configuration* in Eclipse.

2.4.1 The C Development Tools (CDT)

The C development tools (CDT[5]) plugin provides a fully functional IDE for C and C++ development in eclipse. It provides support for building applications with the GNU compiler collection (gcc) using the make command. A Makefile generator is provided to integrate the configuration of the build system. It provides a launcher for C/C++ applications. The CDT includes a code editor supports syntax highlighting, source code completion and error detection. The visual debuging support is built around the GNU debugger (GDB). It supports displaying a thread list, an interface to view and change variables, support for setting breakpoints and a list of all loaded shared modules.
Although built around the GNU C development tools (gcc, gdb) which are natively available in linux and Mac OSX the CDT can be used on windows using Cygwin[6] or MinGW[7].

2.5 The Yeti 2 Debug Plugin

The Yeti 2 Debug Plugin is the last component in the TinyOS debug chain. Its task is to provide seamless integration of TinyOS debugging into Eclipse. This section describes the features that were designed and implemented to solve the tasks identified in section 1.2.

2.5.1 Interface to the Debugging Tools

The implementation of an interface between Java and the TinyOS-Debugging-Tools was delegated to the C development tools (CDT). Since the binary being debugged was compiled from

[5] http://www.eclipse.org/cdt/
[6] http://www.cygwin.com/
[7] http://www.mingw.org/

C source the Yeti 2 debug plugin designed to provide a layer around CDT. The C program being debugged is represented with what is termed the C model. It includes a model which abstracts a C program and provides an API to access the variables. CDT implements an interface between the Eclipse framework and GDB including an API which allows to set breakpoints and access the variables of the program being debugged.

2.5.2 Mapping of C to NesC and Support for Breakpoints

The task of mapping C variables to the original NesC variables was implemented with a view called ”Component Variables Browser”. It displays a list of the components of the NesC program. Each item in the list can be expanded to reveal the variables and arrays belonging to the component. The value of the variables is shown and arrays can be expanded to reveal their elements. By double clicking on a variable a dialog to change the value is opened.
The NesC text editor of the Yeti 2 plugin was extended to support setting breakpoints via the context menu of the left ruler or by double clicking on the ruler. The command is intercepted by the debug plugin and forwarded to CDT.

2.5.3 Automate Debugging and Integration into Yeti 2

A launch configuration type for TinyOS debugging was defined to support automatic launching of debug sessions via a user friendly interface. On launch the launch configuration type automatically starts an AVaRICE process. If the process is still alive after a configurable time interval the launch is delegated the CDT if not the launch is aborted. The AVaRICE process is controlled with the standard debug view of Eclipse, the output is redirected to the Eclipse console. The launch configuration type implements a TinyOS launch configuration dialog through which the user defines the parameters for GDB and AVaRICE. This includes the TCP port over which they communicate and the time interval after which the status of the AVaRICE process is checked when launching. Each launch configuration is associated with a TinyOS project and a binary which is used to retrieve debug information. The launch configuration interface implements a selection dialog that lets the user choose a binary.
All dependencies of Yeti 2 on the code in the Yeti 2 debug plugin were eliminated by providing extension points in Yeti 2 which were extended in the Yeti 2 debug plugin.

2.5.4 Software Components

In this chapter the software components of the developed plugin are discussed. The Yeti 2 TinyOS debug plugin consists of five components (Figure 2.4).

CDT abstraction layer To simplify the maintenance of the debug plugin all direct dependencies on the CDT are encapsulated in this component. This makes it easy to adapt the plugin to future changes in CDT. This component mainly contains wrapper classes around CDT classes.

Breakpoint extension In Eclipse each type of source file is handled by its own editor. Setting and removing breakpoints is heavily dependent on the language of the application being debugged. Eclipse provides a mechanism to extend text editors with the ability to set and display breakpoints. This component extends the NesC text editor to support setting and removing breakpoints.

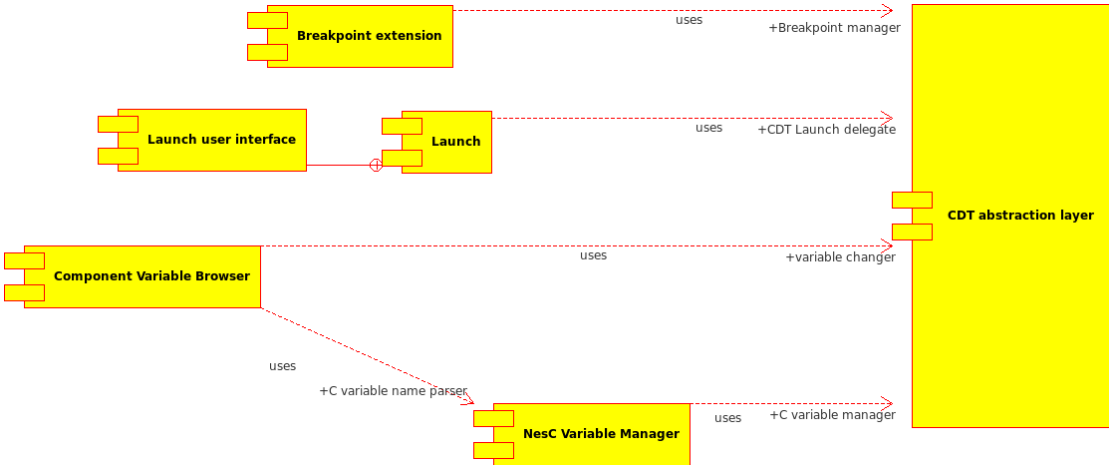

Figure 2.4: UML component diagram of the debug plugin.

Launch The Launch component of the plugin defines a debug configuration type for debugging TinyOS applications. It starts avarice before handing the control to the CDT launch delegate which invokes GDB. The configuration AVaRICE and GDB is defined using the user interface provided by this component.

NesC Variable Manager This component is responsible for managing the variables of the NesC application being debugged. It finds all variables defined in the running binary and filters out the ones belonging to the NesC application. This includes parsing the variable names to extract the module the variable belongs to.

Component Variable Browser This component implements the view in eclipse to display the variables grouped by the components they are defined in. The Component Variable Browser uses the NesC Variable Manager to obtain the list of variables of the application being debugged. For each variable the current value is displayed. The view allows to change the values of the displayed variables.

3 Implementation

This chapter gives detailed insight into the implementation of the YETI 2 Debug plug-in. The Eclipse platform defines a set of interfaces and actions common to many debuggers known as the debug model. This includes stack frames, variables, breakpoints and threads and some common actions such as suspending, stepping, resuming and terminating. The platform provides a user interface which is used to access the model. The C development tools provide an implementation of a debug model specific to the C and C++ language.
The basic approach of the TinyOS debug plug-in was to use CDT to interface with GDB and adapt the graphical user interface so that TinyOS applications can be debugged. This was implemented by adding an additional layer that provides the TinyOS debug interface and uses CDT to implement its functionality.

3.1 CDT abstraction layer

The access to the CDT was isolated into a separate package to ease maintenance and updates if something in CDT changes. The basic approach to do this was to outsource all functionality which requires direct access to classes in CDT. In this section the most important classes of the CDT abstraction layer are discussed.

CDTVariableManager CDT provides a mechanism to add static variables to the debug model. The `CDTVariableManager` is an implementation of `tinyOS.debug.variables.IVariableManager`. This interface provides methods to retrieve all static variables that are defined in the binary and register static variables with CDT.

CDTBreakpointToggleTarget The `CDTBreakpointToggleTarget` implements the interface `org.eclipse.debug.ui.actions.IToggleBreakpointsTarget` defined in the standard debug model of eclipse. Only the interface for line breakpoints was implemented.

CDTLaunchConfigurationDelegate This class simply creates a new CDT launch delegate and delegates the launch to it.

3.2 Launching

The extension point `org.eclipse.debug.core.launchConfigurationTypes` was used to define a new launch configuration type for TinyOS projects. The corresponding launch delegate is implemented in `tinyOS.debug.launch.LaunchConfigurationDelegate`. The launch configuration allows to specify a program which provides an interface between GDB and the JTAG device called a GDB proxy. There are three types of proxies that can be used: AVaRICE, a custom command or none. The first thing the launch delegate does is search the configuration for the configured proxy. It reads the configuration and creates a process to run the command. The started process is added to the launch which means the proxy process is displayed to the user as a part of the debug session. Figure 3.1 shows the generic interface of Eclipse that displays the status of the process and allows to terminate and reset the proxy. The standard output of the proxy process is redirected to the eclipse console. After start up a configurable time is waited to give the proxy time to launch. If the process has not exited it is assumed that the proxy has initialised and is waiting for connections.

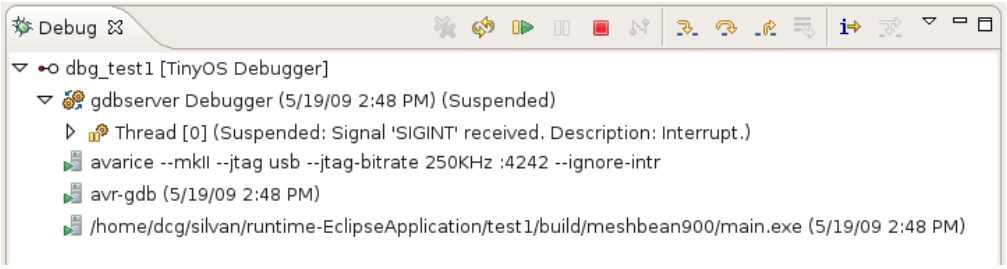

Figure 3.1: List of all processes belonging to a launch

After the GDB proxy has been started successfully the nature of the project is checked. Each project in Eclipse has a set of natures[1] which are used to create an association between the projects and plug-ins. Projects containing C code are associated with the CDT C nature. The CDT launcher will abort the launch if it the project being launched has no C nature. Before delegating the launch to the CDT launcher the C nature is added to the TinyOS project if it is not already present.

3.3 Breakpoints

The `org.eclipse.ui.editorActions` extension point is used to extend TinyOS editor so that breakpoints can be set by double clicking on the left ruler or via context menu. The extension is defined by specifying the identifier of an editor and a class which implements `org.eclipse.ui.IEditorActionDelegate`. The implemented action delegate uses the standard breakpoint manager of Eclipse to remove breakpoints. Requests to set a breakpoint are passed to the CDT breakpoint manager. CDT implements the communication from Eclipse to GDB. The only information needed to set a breakpoint is the file and the line number. Together with the binary GDB is able to calculate the address of the instruction on which the hardware breakpoint is set. Since the NesC compiler marks the the generated C code with the file and line number corresponding to the NesC code the breakpoints are set automatically on the right

[1] http://www.eclipse.org/articles/Article-Builders/builders.html

instruction. The Eclipse framework is used to store breakpoints persistently. This means that breakpoints remain over the course of several debug sessions and when Eclipse is restarted.

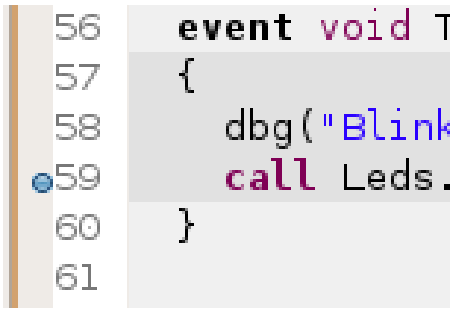

Figure 3.2: Breapoint annotation on the left ruler of an NesC editor.

The default implementation of a text editors displays breakpoints for files inside the workspace (Figure 3.2), files that are not part of the workspace are opened in special wrappers that do not display breakpoints by default. Annotations in text editor are implemented in the annotation model of the editor. To enable breakpoints for external editors a special annotation model was implemented in the debug plug-in. The model registers as breakpoint listener. Every time a breakpoint in the file the annotation provider belongs is created it adds the annotation for the breakpoint to the model of its editor. To keep the Core Yeti 2 plug-in independent from the debug plug-in an extension point called `TinyOS.EditorAnnotationModelProvider` was created. When an external file is opened this extension point is used to create the annotation of the editor.

3.4 Variables

The NesC compiler uses the variable names to map the component model of NesC to C. For example a NesC variable or function called X in module M will be maped to a C variable called `M<sep>X`. The module and variable name is separated by a platform dependent separator token (`<sep>`). The default separator is $. Commands and event handlers are linked statically on compile time. Each event that a component may receive and each command a component implements is mapped to a C function (see Figure 3.3 and 3.4).

```
Event handler naming scheme:
<Component name><sep><interface name><sep><event name>();

Example:
BlinkC__Boot__booted();
```

Figure 3.3: Mapping of NesC events to C.

The `NesCVariableNameParser` is a primitive parser for variable names. It provides methods to split a variable name into a module part and the variable part and a method to check whether a variable name was generated by NesC. The `NesCVariableNameParser` requires an object that implements the interface `INesCSeparatorProvider` (Figure 3.5) as constructor argument. The `NesCVariableNameParser` used this object to retrieve the variable separator of the platform being debugged. The default implementation of the interface

```
Command naming scheme:
<Calling module><sep><interface name><sep><command name>();

Example:
BlinkC__Leds__led2Toggle();
```

Figure 3.4: Mapping of NesC commands to C.

(NesCSeparatorFromCDTLaunch) relies on the environment wrapper of the Yeti 2 core plug-in. The environment wrapper runs the make file of the project in dry mode which display the commands that would be run without actually running them. The separator is parsed form the parameters of the ncc command.

```
public interface INesCSeparatorProvider {
        String getSeparator();
}
```

Figure 3.5: The interface to retrieve the variable separator.

By default the CDT debug model does not track the values of static variables. Since all variables (apart from local variables) in TinyOS are allocated statically CDT does not track them. The variable module of the debug plug-in implements a debug event listener (NesCVariableListener) which is invoked when events such as starting/stopping and suspending a program occur. Whenever an event is captured a list of all static variables is retrieved and the NesC variables are filtered out. A list of all NesC variables that are not yet registered with the CDT is compiled and the variables are added.
The org.eclipse.ui.views extension point was used to add a view called ”Component Variables Browser” (Figure 3.6). It displays the defined NesC variables grouped by the modules in a tree. Whenever the debug context (the function that is currently executed) changes the view is refreshed. First a list of all modules is compiled and a tree node for each module is created and all variables of the module are added as children if they are not already present.

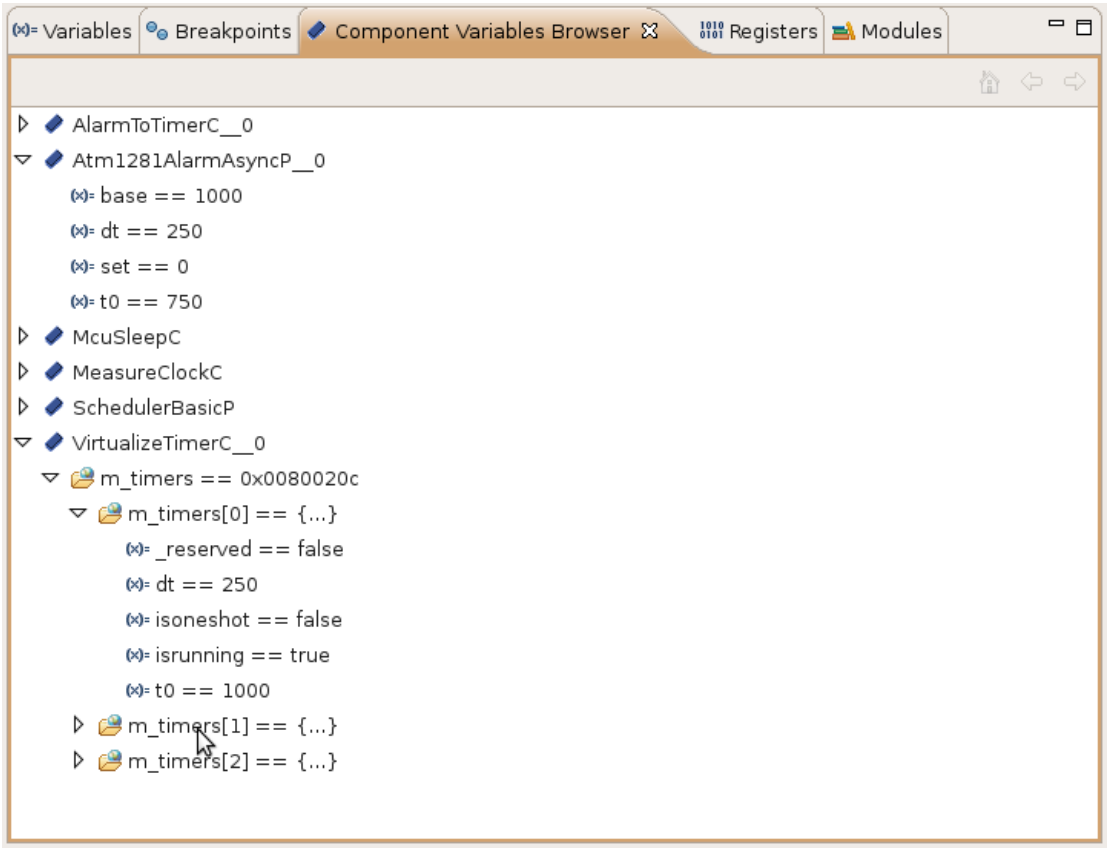

Figure 3.6: The Component Variables Browser

4
Conclusion and Future Work

The vision of intelligent dust involves thousands of self organised wirelessly networked nodes being deployed in an environment. With a multitude of sensors they gather information about the environment. Possible application scenarios range from environmental monitoring to home automation.
TinyOS is a widely used open source operating system for sensor nodes written in NesC a component based C dialect. NesC programs are compiled to C and then to binary. Since there is no dedicated NesC debugger the generated C code has to be debugged. This requires some knowledge about the inner workings of the NesC compiler which should no be required to develop a program. In this paper a debugging solution based on Eclipse is presented. The developed Eclipse TinyOS debug plug-in was designed to provide a mapping from the generated C code to the NesC code and an easy way to set up debug sessions. It supports setting breakpoints in NesC files with a simple double click. The variables defined in a NesC application are displayed grouped by the components they are defined in. The variables can be examined and refactored. The plug-in allows to start a debug session with a simple click. First testers have found the presented Eclipse plug-in to be a useful tool for TinyOS developers.
While being an important first step towards developer friendly TinyOS debugging there are several features which were excluded in the presented solution. This includes watchpoints. Watchpoints are defined on certain expressions and interrupt the program when the value of the expression changes. The ability to set breakpoints on signals would be an interesting extension. Support for debugwire, a proprietary debug interface for chips with too few pins to support JTAG and an ignore list in component variables browser would be other minor improvements.

A User Guide

This chapter documents the features provided by the plugin.

A.1 Launching

The first thing to do is to create a launch configuration. This section shows how to do this for a TinyOS project and describes the launch configuration options.

A.1.1 The Main Configuration Tab

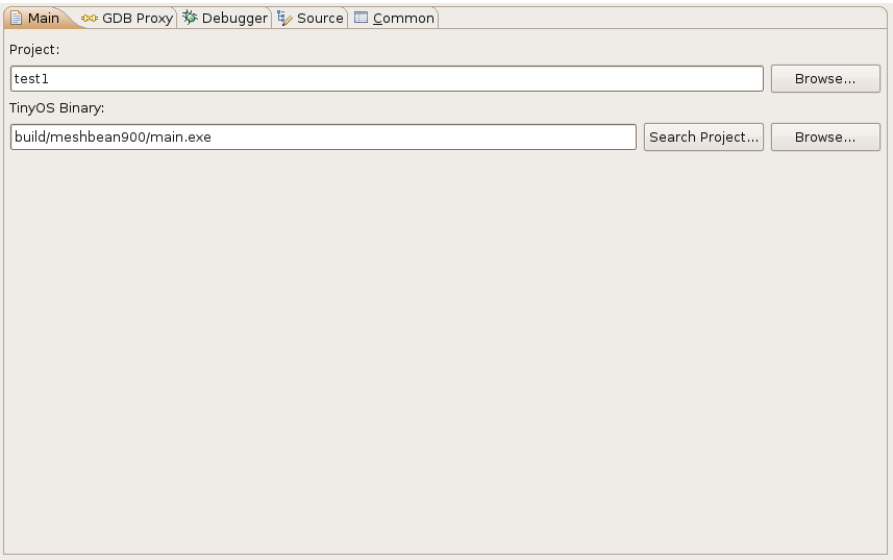

Figure A.1: The Main configuration tab

In the Main tab (Figure A.1) the project is defined. The binary being debugged can either be defined by browsing the file system or by searching for binaries (files with the extension .exe) in

the choosen project. The binary is used by the debugger to map filenames and line numbers to addresses for setting breakpoints. The global variables defined in the programm are determined by parsing the binary file.

A.1.2 The GDB Proxy Tab

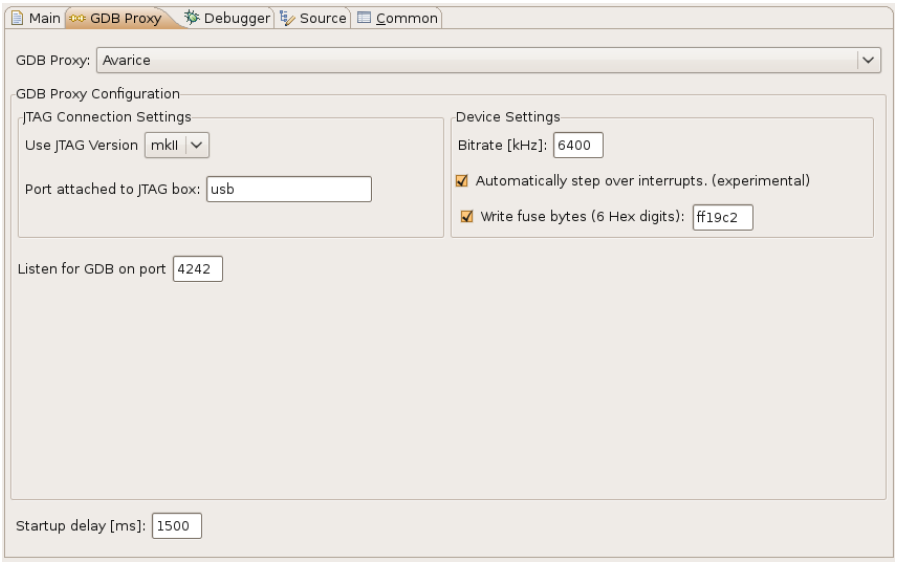

Figure A.2: The avarice configuration tab

In the proxy tab (Figure A.2) a GDB proxy is defined. A GDB proxy is a command that is started just before the actual debugger during the launch. It provides an interface through which GDB communicates with the target. Three options for the configuration of the GDB Proxy are available: ”Avarice” [1], ”User defined command” and ”None”. The ”User defined command” can be used if the target device requires a GDB proxy other than avarice. The specified command will be started during the launch.
The launch of the debugging session will be delayed by the time defined in **Startup delay** after starting the proxy. This can be used to give the GDB Proxy command enough time to initialize before GDB attempts to connect. The default value is 1500 miliseconds.

Avarice

The Avarice configuration provides an intuitive graphical user interface through which the arguments for the avarice command are defined. In the area ”JTAG Connection settings” the parameters for the connection between avarice and the JTAG device are defined.

Use JTAG version Defines the JTAG version to use. (Default: mkI)

Port attached to the JTAG box Defines the port to which the JTAG device is attached (Default: /dev/avrjtag)

The area ”Device Settings” is used to define the settings of the device being debugged.

[1] http://avarice.sourceforge.net/

Bitrate Defines the bitrate the JTAG device uses to communicate with the avr target device. The bitrate must be less than 1/4 of the frequency of the target device. Valid bitrates for a JTAG ICE mkI are 1000 kHz, 500 kHz, 250 kHz or 125 kHz, for a JTAG ICE mkII 22 kHz through 6400 kHz. (Default: 1000 kHz)

Automatically step over interrupts Avarice will automatically step over interrupts. This is particularly usefull when stepping through a program line by line. If this is disabled then avarice will also step through interrupt handle routines which may not be the intended behaviour. This feature is experimental and works only if the target device is not fused for compatibility. (Default: true)

Write fuse bytes Write the fuse bytes defined in the input box. The fuse bytes must consist of 6 hexadecimal digits in the format EEHHLL. EE stands for the extended fuse bytes, HH for the high fuse bytes, LL for the low fuse bytes. If the checkbox is disabled, the value in the input box will be completely ignored. (Default: checkbox disabled, fuse bytes ff19c2)

GDB uses the GDB/MI (machine interface), a line based text oriented interface to communicate with avarice. The communication runs over the TCP port specified in the **Listen for GDB on port** input box. (Default: 4242)

A.1.3 The Debugger Tab

The debugger tab is used to define the parameters of the debugger. The debug plugin will alway launch a CDT debug session with gdb running in server mode.

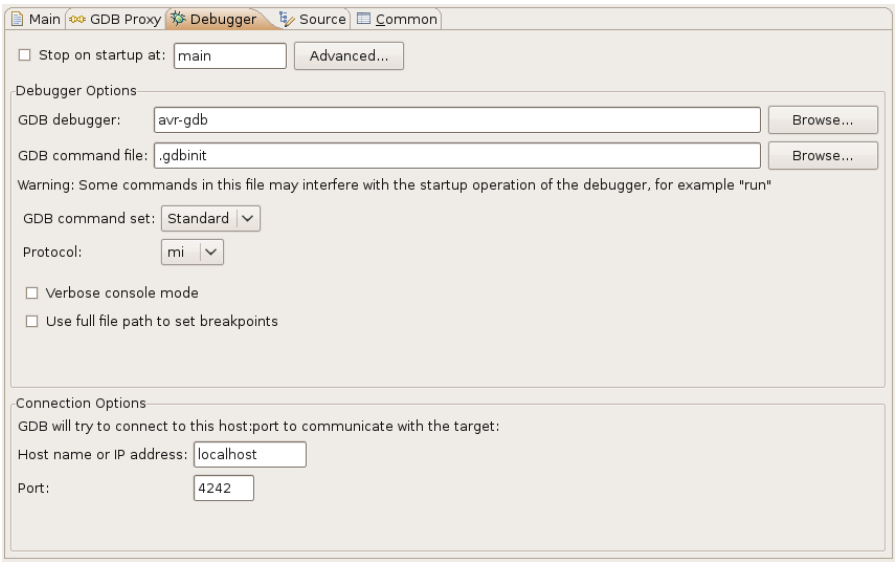

Figure A.3: The Debugger tab

Stop startup at Stop startup at the symbol defined in the text box. (Default: main)

GDB Debugger Defines the command which is used to start the debugger (Default: avr-gdb)

GDB Command file GDB will execute the commands in the given file on startup (Default: .gdbinit)

GDB Command set Defines the command set to use to communicate with the GDB proxy. (Default: /dev/avrjtag)

GDB Protocol Defines the protocol to use to communicate with the GDB proxy. (Default: mi)

Verbose console mode If enabled GDB will print all commands it sends to the GDB proxy on the console (Default: disabled)

Use full path to set breakpoints Use full file path to set breakpoints. For example use blink/src/Blink.nc:71 instead of Blink.nc:71 (Default: disabled)

Host name or IP address Defines the host gdb will connect to on startup (Default: localhost)

Port Defines to port gdb will use to connect to the proxy. (Default: 4242) If avarice is used as proxy this value will be automatically synchronized with the setting in the GDB Proxy tab.

A.2 Breakpoints

Breakpoints can be set by double clicking on the left ruler of a NesC editor or via the context menu by right clicking (Figure A.4). Breakpoints can be set before launching the debug session as well as during the session. All defined breakpoints are displayed in the breakpoint view (Figure A.5)

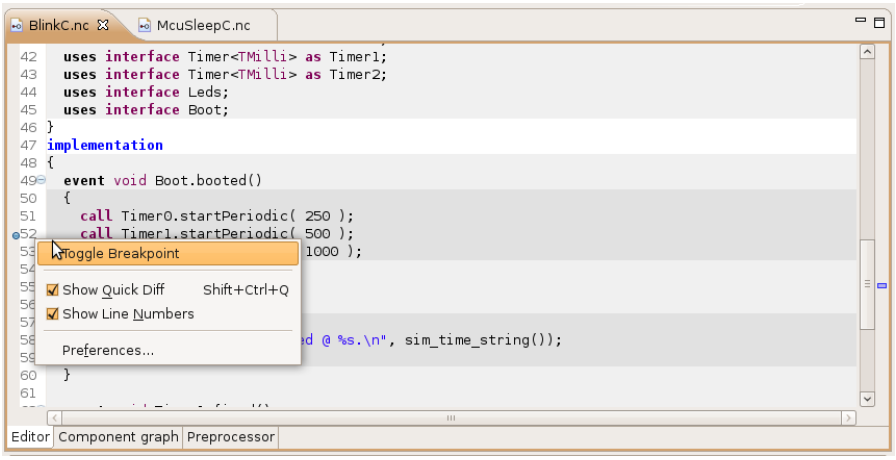

Figure A.4: Setting a breakpoint using the context menu

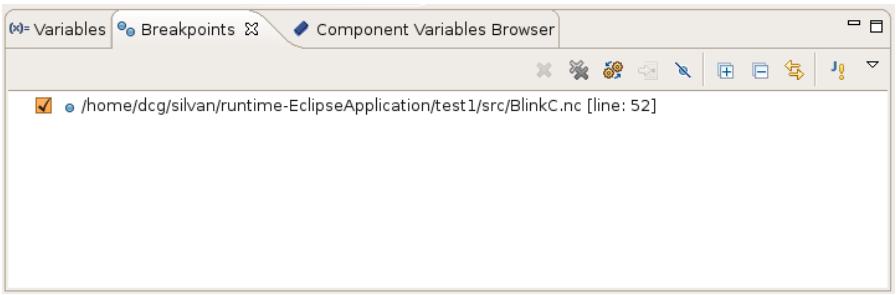

Figure A.5: The standard Eclipse breakpoint view

The currently defined breakpoints are displayed in the standard Eclipse breakpoint view (Figure A.5).

A.3 Variables

All variables defined in components are displayed in the Component Variables Browser (Figure A.6). The variables are listed in a tree viewer ordered by components. The components can be expanded to examine all variables of a component. The value of the variable can be changed by double clicking on the variable, or via the context menu by right clicking on the variable (Figure A.7). The new value of the variable is set in the appearing ”Change Value...” dialog (Figure A.8).

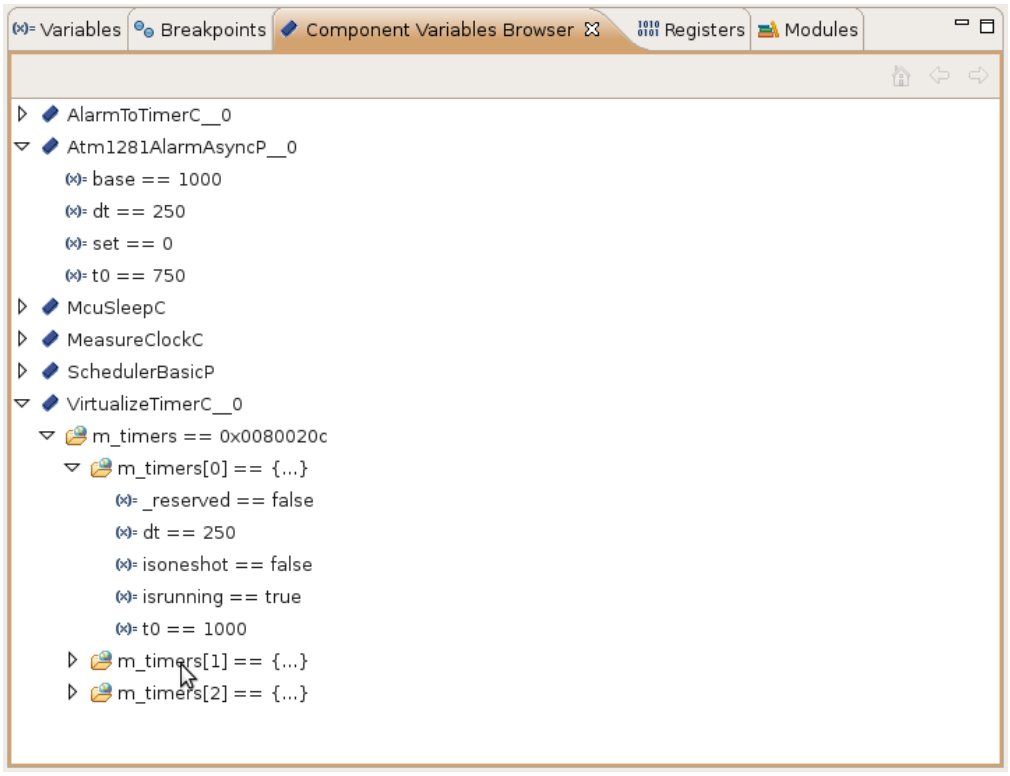

Figure A.6: The Component Variables Browser

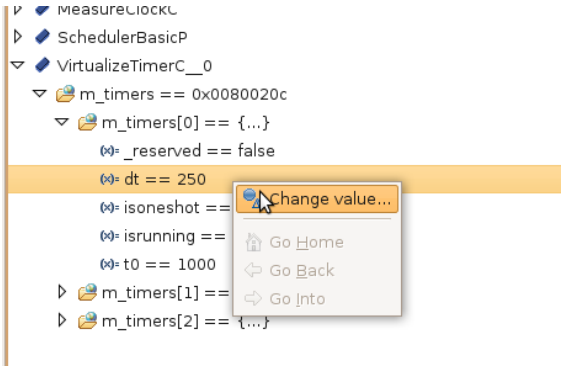

Figure A.7: The context menu for changing the value of variables

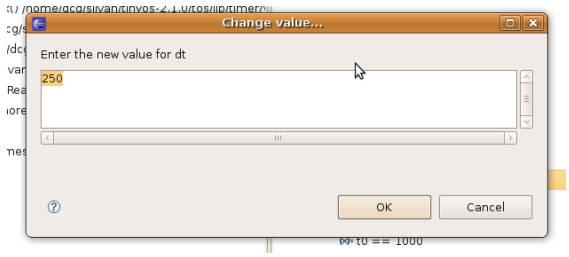

Figure A.8: The refactor dialog

Local variables can be examined via the standard way using the variable view.

Bibliography

[1] C. S. Raghavendra, Krishna M. Sivalingam, and Taieb Znati, editors. *Wireless sensor networks*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[2] Hans-Joachim Hof. Applications of sensor networks. In *Algorithms for Sensor and Ad Hoc Networks*, pages 1–20, 2007.

[3] Robert Adler, Mick Flanigan, Jonathan Huang, Ralph Kling, Nandakishore Kushalnagar, Lama Nachman, Chieh-Yih Wan, and Mark Yarvis. Intel mote 2: an advanced platform for demanding sensor network applications. pages 298–298, 2005.

[4] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*. 2005.

[5] Philip David Culler Eric Brewer David Gay Philip Levis. nesc 1.1 language reference manual, 2003.

[6] Matthew Telles and Yuan Hsieh. *The Science of Debugging*. Coriolis Group Books, Scottsdale, AZ, USA, 2001.

[7] Benjamin Sigg. Yeti 2 - tinyos 2.x eclipse plugin. Master's thesis, ETH, 2008.

[8] Atmel. *AVR JTAG ICE User Guide*. Atmel Corporation, 2001.

[9] Harry Bleeker Peter van den Eijnden Frans de Jong. *Boundary-scan test: a practical approach*. Springer, 1993.

[10] Ben Bennetts. Boundary scan tutorial, 2001.

[11] Richard Stallman Roland Pesch Stan Shebs. *Debugging with gdb*. Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA, ninth edition edition, 2009.

[12] Atmel. *AVR067: JTAGICE mkII Communication Protocol*. Atmel Corporation, 2006.

[13] David Carlson. *Eclipse Distilled*. Addison Wesley Professional, Februar 2005.