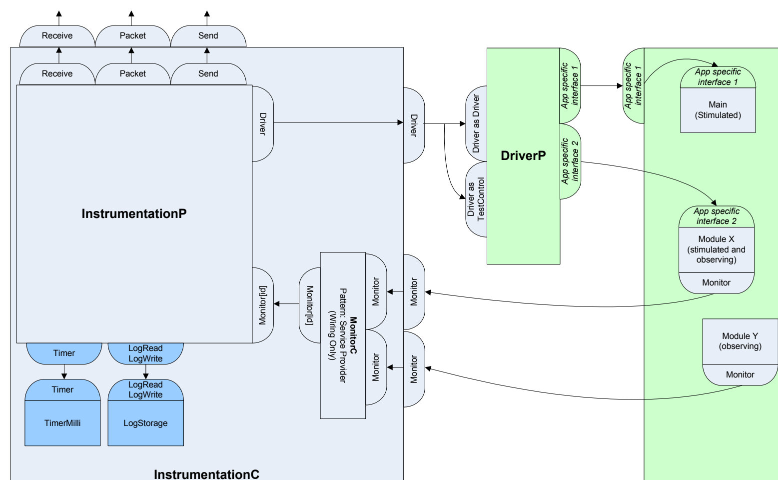
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TinyOS Instrumentation

Tonio Gsell



SEMESTER THESIS

Fall Term 2007

Supervisor: Matthias Woehrle
Supervisor: Dr. Jan Beutel
Professor: Dr. Lothar Thiele

December 2007

Contents

1: Introduction — *1*
- 1.1 Related work . . . 2
  - 1.1.1 TinyOS . . . 2
  - 1.1.2 NesC . . . 2
- 1.2 Test platforms . . . 3
  - 1.2.1 DSN . . . 3
  - 1.2.2 Serial interface . . . 3

2: Basic concept of the Instrumentation — *5*
- 2.1 Architecture overview . . . 5
- 2.2 Protocol . . . 6
  - 2.2.1 Class . . . 7
  - 2.2.2 Error flag . . . 8
  - 2.2.3 Time stamp flag . . . 8
  - 2.2.4 Payload length . . . 8
  - 2.2.5 Identifier . . . 8
  - 2.2.6 Time stamp . . . 8
  - 2.2.7 Packet number . . . 9
  - 2.2.8 Payload . . . 9

3: Implementation — *11*
- 3.1 Internal packet flow . . . 11
- 3.2 Driver . . . 11
- 3.3 Monitor . . . 12
- 3.4 Error handling . . . 13
- 3.5 Functionality and buffers . . . 13
  - 3.5.1 Control packets . . . 14
  - 3.5.2 Command packets . . . 15
  - 3.5.3 Instrumentation replies . . . 15
- 3.6 Storage . . . 16
- 3.7 DSN interface . . . 17

4: Evaluation *19*
4.1 RAM/ROM requirements 19
4.2 Dummy Application 20
4.2.1 Test settings 21
4.2.2 Serial - Test results 21
4.2.3 DSN - Test results 22
4.3 Multihop Oscilloscope 22
4.3.1 Test setting 22
4.3.2 DSN - Test results 23
4.4 Future implementations 23
4.4.1 Flash EBUSY handling 23
4.4.2 Monitor ID Parser 23
4.4.3 Echo 23
4.4.4 State storage 24
4.4.5 Packet storage requirements 24
4.4.6 Flash lifetime 24

5: Summary *25*

A: Little HOWTO *27*
A.1 Configuration file 27
A.2 Module file 28
A.3 Flash partitioning 28
A.4 Makefile 29
A.5 Compiler 29

B: Instruction set *31*

C: Error table *33*

D: Makefile *35*

E: TinyOS - Wiring *37*

Tables

B-1 State command table 31
B-2 Storage command table 31

C-1 Error table 33
C-2 Error table 34

Figures

2-1 Instrumentation component 5
2-2 Test core engine 6
2-3 Protocol 7

3-1 Instrumentation Data Flow 12
3-2 Instrumentation implemented buffers 14
3-3 Process Packet buffer 16

4-1 ROM Requirements 20

E-1 The TinyOS wiring graph 37

Preface

In my semester thesis about TinyOS Instrumentation, I have been working on a piece of code implementing a generic interface between a supporting test infrastructure and an application under test using TinyOS. There were already some conceptual ideas and a first code framework available, which made it much easier for me to find my way into the topic. After reading lots of related work, I developed an appropriate concept and some programming guidelines. Buffer requirements had to be defined, state machines and a solid error handling had to be implemented. Finally, I have rewritten almost everything of the existing code and added lots of new functionality. During the development phase, I have been repeatedly testing the component over the serial interface. A final twenty-node-test on DSN successfully passed without any problems. The code and all its functionality is well documented and ready to use.

1

Introduction

Today's increasing use of sensor networks, its scaling potential and the need for application specific distributed software solutions, necessitates a test infrastructure. A wireless sensor network is a network consisting of distributed nodes equipped with sensors to collect environmental data, e.g. temperature, humidity, or motion at different locations. Testing sensor networks has been shown to be quite a hard and annoying task. Wireless communication can lead to partial loss of data, parts of the network can be temporarily unavailable or single nodes will run out of energy. Low level access over JTAG and UART can be used, providing only limited access to the internal state and the LEDs are offering very primitive visual inspection. Even though this is only a small part out of all spotted difficulties, the complexity a test infrastructure has to handle with is considerable.

Given that a sensor network cannot just be tested as a black box, ways have to be offered to gather information about its internal state. Visibility can be guaranteed for example by wiring all nodes to a supporting backbone network. This approach has already been taken by the MoteLab testbed [1]. On the other hand, a more flexible wireless approach is to design a supporting wireless test platform [2].

To offer software developers an easy way to connect the test platform to the wireless network under test, a well defined interface has to be implemented. Our approach is to program a TinyOS library component which provides the functionality on each deployed node to grant the needed connectivity and visibility. It can be easily connected to any application under test and offers a well defined interface to the supporting test platform. The visibility will be granted by an on-/offline monitor interface. In other words, monitored data can be directly sent over the supporting network while testing, or be stored at the node's flash for later analysis. In addition to that, the component will allow the user to stimulate the application for special testing reasons.

The component will be easy to use and its functionality as comprehensive as necessary but also as slim as possible. Thus it will have as little impact on the application under test as possible.

1.1 Related work

As mentioned above, the idea is to implement a TinyOS component. This section will give some background information about the TinyOS's design concept and its programming language NesC.

1.1.1 TinyOS

TinyOS [3] is an open source operating system for wireless sensor networks. It uses a component-based architecture and an event-driven concurrency model. It is in particular designed for hardware systems with limited resources concerning memory, processing power and energy constraints, which have to operate autonomously and safely over months or even years. The operating system executes upcoming events as soon as possible, so the hardware can immediately jump back into sleep mode again saving as much energy as possible. This is important because todays platforms like Telos or Mica are highly limited in its resources.

The TmoteSky [4] node, for example, consists of an TI MSP430 processor running at 8MHz, 10KB of RAM, 48KB of flash memory and a Chipcon CC2420 wireless transceiver operating at 2.4GHz. It includes sensors for light, temperature, and humidity. In all following tests, the TmoteSky platform has been chosen to represent the different available platforms.

For the application development, different components can be connected, which in collectivity form the whole application. This leads to an easy implementation concept to assemble the needed functionality specific to the sensor nodes hardware. The concept of the individual components is inspired by currently used hardware, i.e. an event handler is implemented which responds with an event done call, signaling that all work has been processed.

A Timer component is on the one hand already usable from the TinyOS's component library. This Timer is a general purpose timing interface, which is implemented on each different platform, but does not have to support platform-specific components like for offering an Alarm interface. Thus on the other hand, to use platform-specific timing facilities, a new component can be written supporting its functionality. Wiring it to the rest of the components just offers the desired functions without having to rewrite the existing one.

All this together leads to a highly flexible operating system, which can be used on lots of different platforms like Telos/TMoteSky, Mica2 and many more.

1.1.2 NesC

Originally TinyOS was developed in the program language C. But the unique application area made it necessary to design a new programming concept, which optimally supports the operating system. Therefore the programming language NesC [5] has been developed.

NesC is using a component-based programming concept. There are two different types of components: Modules implement the whole functionality of the interfaces between different components and Configurations describe the wiring of the unique

interfaces. Except event handling across interfaces (wiring), all code is written in usual C style. So it is quite easy for a skilled C programmer to write its own NesC components. Modules can use interfaces from other components in the so called *uses* section and provide functionality encapsulated in interfaces for other components in the *provides* section as shown in Listing 1.1.

```
provides {
  interface myComponent;
}
uses {
  interface otherComponent;
}
```

Listing 1.1
The Module's 'provides'- and 'uses'-section.

With the provided NesC compiler/linker the source code can be compiled into machine code. There is also a JavaDoc-like documentation *make* command, which generates a comprehensible http documentation containing all interfaces and components.

1.2 Test platforms

The simplest approach to test wireless sensor networks is to use the node's LEDs. Of course this is not a really convincing test platform. The serial interface can be used to locally log data. Otherwise, a more sophisticated test platform can be used like DSN.

1.2.1 DSN

The Deployment Support Network [2] is a wireless network designed to support wireless sensor networks in their development phase. It operates as a stable backbone network offering testing, debugging and logging services. The DSN-Server can be operated by clients sending commands or requesting information by remote procedure calls (RPC). It sends and collects all data from the deployed DSN-Nodes.

The main goal is to be able to develop and debug embedded wireless systems in a real environment. There is no need for physical access to the wireless sensor target network. This offers a lot of new application areas, for example large scale outdoor sensor network testing.

1.2.2 Serial interface

The serial interface can be used to communicate with the Instrumentation component, if local access is granted. So in a first testing step it could be used to directly monitor data circumventing additional difficulties, which could occur with a wireless backbone like DSN. This has been used to verify the functionality of the Instrumentation before deploying it on DSN (see Section 4).

2

Basic concept of the Instrumentation

The test core engine (see Figure 2-1) is a generic test component for TinyOS programs. It is designed to be deployed in many different applications to support testing sensor networks. It offers visibility to the sensor network under test and consistency checks of the logged data even if resets or reboots occur.

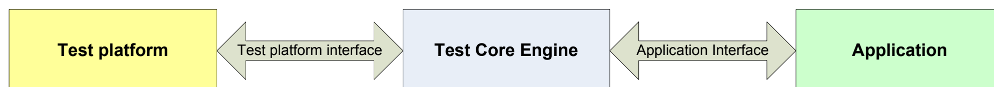


Figure 2-1
The test core engine connects to the test platform over the test platform interface on the left and to the application over the application interface on the right.

2.1 Architecture overview

Figure 2-2 shows the test core engine in detail. On the one hand, it offers a generic communication interface to the supporting test infrastructure, the test platform interface. It is designed to receive incoming Control and Command packets and to send outgoing Monitor and Error packets. On the other hand, it implements an application-specific interface to the program under test, the application interface. This can be used to inject data to the application under test using the Driver, as well as to log incoming data over the offered Monitor facility.

To guarantee the consistency of the logged Monitor data as well as internally raised errors, there are variable buffers implemented to handle bursty traffic and the node's flash can be used to store data.

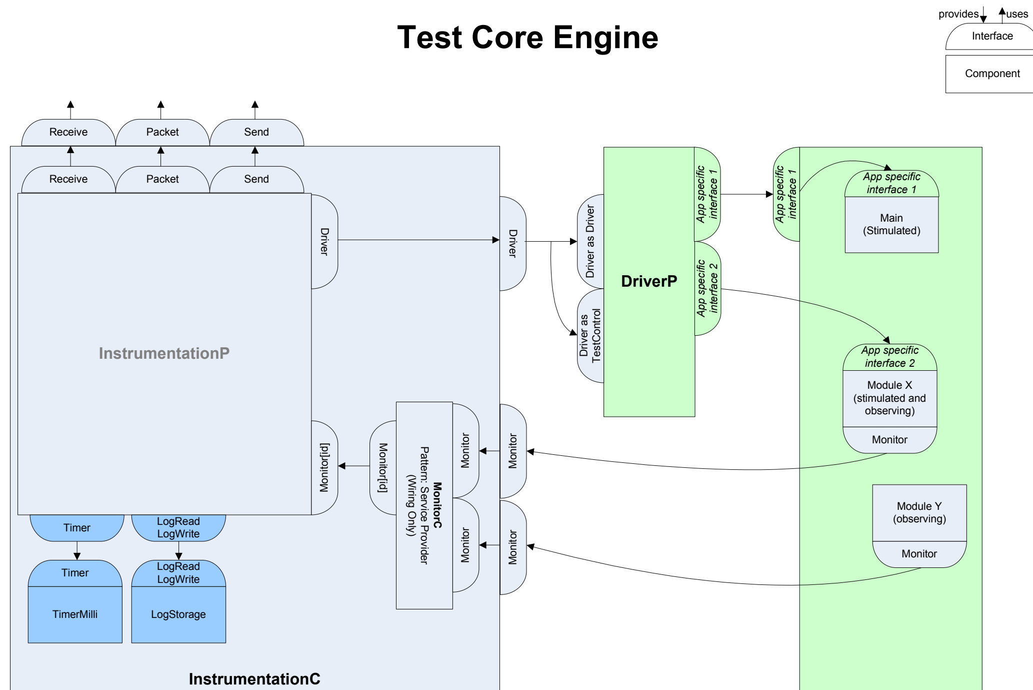


Figure 2-2
Block diagram of the test core engine with its two interfaces, upper left corner: test platform interface - right: application interface. Additional Timer and LogStorage components in the left lower corner of the instrumentation component.

2.2 Protocol

The protocol used with the instrumentation is designed to be quite simple as shown in Figure 2-3. All packets received and sent over the test platform interface, as well as stored temporarily in buffers or permanently in the flash, are using it.

The protocol header is defined in the Instrumentation header (Instrumentation.h) as shown in Listing 3.1.

```
typedef struct inst_header_t {
    nx_uint8_t flags;
    nx_uint8_t payloadLength;
    nx_uint16_t identifier;
    nx_uint32_t timestamp;
    nx_uint16_t packetNumber;
} inst_header_t;
```

Listing 2.1
The protocol's header structure.

Both buffer and flash storage are designed to handle only whole packets regardless of its actual content. In other words, one packet stored or buffered will always

require the size of payload offered by the next lower layer protocol, which in the case for TinyOS is TOSH_DATA_LENGTH. This in turn has some space for further improvement which will be discussed in Chapter 4.

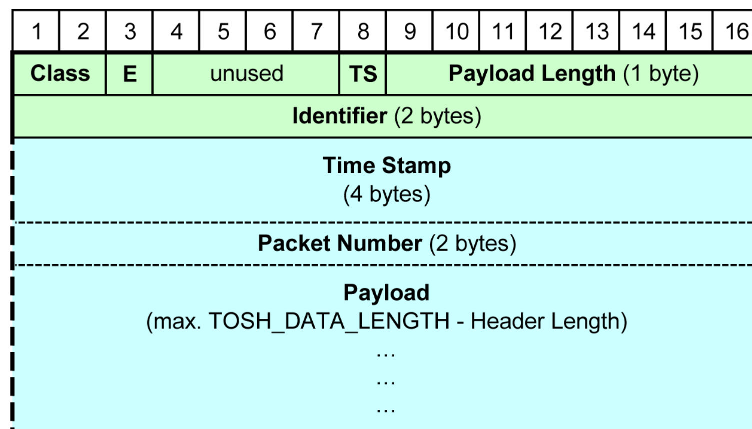


Figure 2-3
The Protocol used to communicate with the instrumentation component.
Solid lines stand for the fixed header, while the dashed one stands for the optional time stamp, packet number and payload.
E - error flag
TS - time stamp flag.

2.2.1 Class

The first two bits in the first byte are used to identify the packets class affiliation. There are four different classes available which differentiate the internal data flow channels.

2.2.1.1 Control class

On the one hand, the Control class offers the interface for the client to communicate with the internal state of the test core engine and its implemented functionality. For example, it is used to switch from sending packets directly to the platform interface in order to store them in the motes internal flash storage facilities.

On the other hand, using the DSN test platform, the Control class is used to send back informational packets to the client, using the DSN test platform. This includes error packets or informational packets like getting the Unix compile time.

2.2.1.2 Driver class

The Driver class provides the access point to the application. It is used to stimulate the program under test, for example polling where the application would be answering with a Monitor call. What actually will be stimulated is up to the application developers need and can, under the limitation of the payload size, be implemented as necessary. Only the user can generate Driver class packages; there will never be any generated by the test core engine.

2.2.1.3 Test Control class

The Test Control class is actually only an alias for a Driver and is "physically" the same, but logically differentiated for clear separation. Its function is to provide a module to stimulate the program under test in its setup period, e.g. modifying its state.

2.2.1.4 Monitor class

The Monitor class is used to extract information from the application under test. Each time a Monitor is called, a new packet with the class Monitor will be generated and routed as internally defined. No Monitor packets will be accepted which have not been generated by a Monitor call; instead, this would lead to an error.

2.2.2 Error flag

The third bit in the first byte is the error flag bit. If an error occurs a packet with the error flag set will be generated (see Section 3.4).

2.2.3 Time stamp flag

The eighth bit in the first byte has to be set, if the packet should include the four byte time stamp field.

2.2.4 Payload length

This one byte field (second byte) is used to specify the length of the attached payload. It is internally used to verify the packet length and is the only consistency check.

2.2.5 Identifier

The two bytes Identifier field (byte number 3 and 4) can be used in three different ways depending on the packets class affiliation:

If used with the Control class, the identifier will be separated into an upper part byte and a lower part byte. The upper part is used to address the state or storage functionality of the test core engine, while the lower part is addressing the different functions of them.

If used with the Driver or Test Control class, the identifiers purpose can be assigned by the developers of the application under test.

If used with the Monitor class, the identifier will contain the calling MonitorId (see Section 3.3).

2.2.6 Time stamp

On the one hand the time stamp (byte numbers 5 to 8) is used to accurately time the commands sent to the application under test. The commands are sequentially executed. If some or all packets include the time stamp field, their execution will be delayed for the specified amount of time.

On the other hand, the time stamp will be set to the local mote time for Monitor as well as Error packets. Its function is to be used for further interpretations of the collected data.

2.2.7 Packet number

The packet number (byte numbers 9 and 10) is only used with internally generated packets. Each time a new packet including error packets has been or should have been created the packet number counter will be stored in the packet number field and increased by one. For example, if a Monitor call has created a new packet but the Process Packet buffer is already full, the packet number counter will be increased by one and the appropriate error packet will be created with the next higher packet number. If instead the error buffer is also full, we can at least discover that something went wrong by analyzing the packet number gaps.

2.2.8 Payload

The payload (byte numbers 11 up to TOSH_DATA_LENGTH) can be used to send data to the application under test over the Driver or Test Control interface. And it is used to store the data passed by a Monitor call. The size of the payload is limited by the size offered by the next lower layer protocol, which in the case for TinyOS is TOSH_DATA_LENGTH minus the summation of the used header fields.

In case of a Monitor call the payload should never exceed $TOSH_DATA_LENGTH - 10bytes$!

3

Implementation

The Implementation of the test core engine will be discussed following its internal packet streams and buffer facilities. The current functionality and the problems encountered so far will be outlined in the following. For future work, please refer to Section 4.4.

3.1 Internal packet flow

The Instrumentations internal flow control can be divided into five independent packet streams (see Figure 3-1).

1. Packets arriving at the test platforms receive interface are temporarily stored in the packet specific buffer (Control or Command buffer) and processed by the test core engine as appropriate.
2. Driver and TestControl packets stored in the Command buffer will be unpacked and then forwarded in a first in - first out (FIFO) manner also depending on their time stamp delay to the applications Driver or TestControl interface.
3. Monitor logging content arriving at the applications monitor interface will be packed into a Monitor packet and temporarily stored in the Process Packet buffer by the test core engine.
4. Depending on the internal state, Error and Monitor packets will be stored in flash and read out again after a specific command call.
5. Internally generated error or informational packets as well as Monitor packets can be directly sent by the test core engine over the test platforms send interface.

3.2 Driver

The driver is a NesC-conform Dispatcher [5] used to forward packets to the application under test. It is up to the developers of the application to implement the actual

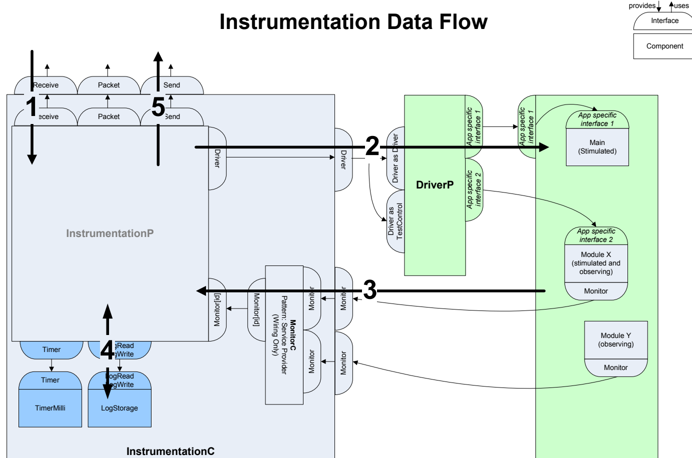


Figure 3-1
The Instrumentation flow chart.

1. *Test platform interface (receive) –> Test core engine*
2. *Test core engine –> Application interface (Driver/TestControl)*
3. *Application interface (Monitor) –> Test core engine*
4. *Flash read/write*
5. *Test core engine –> Test platform interface (send)*

behavior of the injected driver parameters. In return to a Driver call, a callDriver-FunctionDone event has to be raised with *error_t* as parameter which in case of FAIL will be leading in a future implementation to an exponential back-off call retrial (see chapter 4). Currently, in case of *FAIL*, an error packet will be generated (see Appendix C).

There are implemented default callDriverFunctions, too, which drop all incoming Driver packets. This can be necessary in case of an application under test exclusively using the Monitor facilities of the test core engine without any need of using the Driver functionality.

The Driver interface implemented is shown in listing 3.1:

```
interface Driver {
        command error_t callDriverFunction(uint16_t id, \\
                        void* payload, uint8_t len);
        event void callDriverFunctionDone(error_t error);
}
```

Listing 3.1
The Driver interface.

3.3 Monitor

The Monitor interface is a NesC conform generic Service instance [5] used from the application under test to monitor desired data, for example to reply to polling requests from the Driver side or when an in-line statement is reached. There can be up to 65536 monitors instantiated with a unique Monitor ID. Each Monitor receives a unique ID at compile time, which will be passed as the identifier argument to the packet generating mechanism. So all Monitor packets sent over the test platform interface can be attributed to its originating calling instance in the application's source code. (See also Section 4.4.2).

3.4 Error handling

Two different ways of error handling are implemented into the test core engine. On the one hand, there is the usual *error_t* based NesC conform error handling scheme and on the other hand, there is the special error flag bit implemented in the instrumentation protocol (as mentioned in Section 2.2) to be raised for a generated error packet. If an error packet has been generated it will be processed as a usual packet by temporarily storing it in the Process Packet buffer. If the internal test core engine state is set to send all packets directly over the test platform send interface with the optional implemented delay, the error packet will be forwarded in a FIFO manner when it is its turn. On the other side if the internal test core engine state is set to store the packets in the mote's flash facilities, the error packet will be stored in the same FIFO manner in flash.

For debugging reasons or for any other use, this can be defined by INST_DEBUG_MODE in the Makefile (see Appendix D). With this define set any error packet regardless of the internal store state, will be forwarded to the test platform send interface.

3.5 Functionality and buffers

Three different buffers are implemented in the test core engine: the Command buffer, the Control buffer and the Process packet buffer. They can be varied at compile time in size as needed to handle bursty input sequences of packets arriving both on the test platforms receive interface and on the applications monitor interface. All three buffers are designed as FIFO using the TinyOS's PoolC combined with QueueC.

PoolC implements an allocation pool of a specific type of memory objects, offering *get, put* and *size* functions. In this case it allocates the structure *inst_buffer_t* defined in the Instrumentation header (Instrumentation.h) as shown in Listing 3.2.

```
typedef struct inst_buffer_t {
    uint8_t packet[TOSH_DATA_LENGTH];
} inst_buffer_t;
```

Listing 3.2
The inst_buffer_t struct used to store the Instrumentation packets.

In accordance to the TinyOS design outline, the packet size has been chosen to fit accurately into the lower layer payload size.

QueueC is an interface to a FIFO list (queue) which holds the *inst_buffer_t*-pointers from the allocated pool. It offers *dequeue, enqueue, head* and *size* functionality.

The FIFO is also used for the state machines handling the three buffer types. In case of a new packet arriving at an empty buffer, the appropriate packet handler will be started and the state machine is leaving its idle state. If there are new packets arriving at the buffer while the first one is still being processed, they will be handled when the first one has left and so on. In case of an overflow, the packets will be dropped and an error packet is generated. After all packets have been processed, so

that the FIFO turns to empty again, the state machine will jump back to idle and wait for a new incoming packet.

The buffer design (see Figure 3-2) is closely bound to the internal data flow model as discussed above (see Figure 3-1).

1. The Control buffer handles all incoming Control class packets arriving at the test platforms receive interface.
2. The Command buffer handles all incoming Command class packets arriving at the test platforms receive interface.
3. The Process Packet buffer handles all incoming Monitor class packets arriving at the applications monitor interface, as well as all internally generated Error and Informational packets.

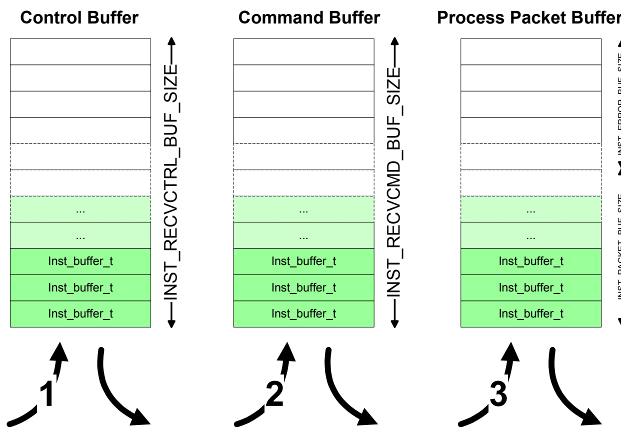


Figure 3-2
The Instrumentation buffer implementation.

1. *Test platform interface (receive) –> Control buffer*
2. *Test platform interface (receive) –> Command buffer*
3. *Application interface (Monitor) and internal generated error or informational packets –> Process Packet buffer*

3.5.1 Control packets

Over the test core engines Control functionality the internal state behavior can be set and different functions can be called.

State functionality:

- To be able to verify that the intended piece of software has been flashed to a chosen mote, the get Unix compile time function is implemented. The compile time generated IDENT_UNIX_TIME will be returned.
- Setting and unsetting the packet flow route to the motes storage facilities.

Storage functionality (see also Section 3.6):

- Readout the whole flash.
- Readout specified sequence of the flash.
- Erase the whole flash.

For a detailed instruction set, please refer to Appendix B.

3.5.2 Command packets

There are three different ways to time the stimulating driver packets:

1. Packets can be timed on the sender side with no time stamp included in its header, what means that the client controlling the test platform infrastructure has to use its own timer facilities. This will normally lead to low Command buffer requirements on the test core engine side. However, it can lead to unfeasible timing.
2. The internal test core engine Command timer can instead be used to cover over with the timing constraints in the chosen accuracy. The current implementation uses millisecond accuracy, but it is only a wiring matter to change to an other one.

   For this approach the Command buffer must be dimensioned to hold all needed Driver and TestControl packets, which have to include the well calculated time stamp header field. The first time a packet arrives in the Command buffer, the timer will be started with the specified delay in the packet's time stamp header field. The following packets reside in the Command buffer. The first packet will be processed when the timer fires. After processing it, the timer starts again with the delay in the next packet's time stamp field. This procedure will be repeated till all packets have been processed.
3. A combination out of 1 and 2.

3.5.3 Instrumentation replies

The Process packet buffer is responsible for both Monitor and Error packets. A first idea was to split them up into two different buffers with fixed size, which would have been easier to implement. But concerning the importance of error packets in our debug setting and the requirement to save as much storage as possible, a new buffer approach was chosen:

The buffer is now divided into two different parts belonging together (see Figure 3-3) separated in size by compile time assigned INST_PACKET_BUF_SIZE and INST_ERROR_BUF_SIZE which can be set in the Makefile (see Appendix D). The middle boundary between the Packet buffer size and the error buffer size can be seen as a semi permeable partition which leads to a heavier emphasis of the error packets. In case of an error overflow on its reserved page, if there is still place on the Monitor packets page, errors can leak into it and will only be dropped in case of a total buffer overflow. Instead, Monitor packets are already dropped in case of an overflow on their page raising an error. This assures that only errors can fill up the whole buffer.

Neither of the two buffers should be set to zero!

There are two directions in which packets can leave the Process Packet buffer:

They will be sequentially (FIFO) leaving the test core engine over the test platforms send interface until the buffer is empty again. If desired a delay timer can be set in the Makefile (see Appendix D) defined as INST_PACKET_SEND_DELAY which will

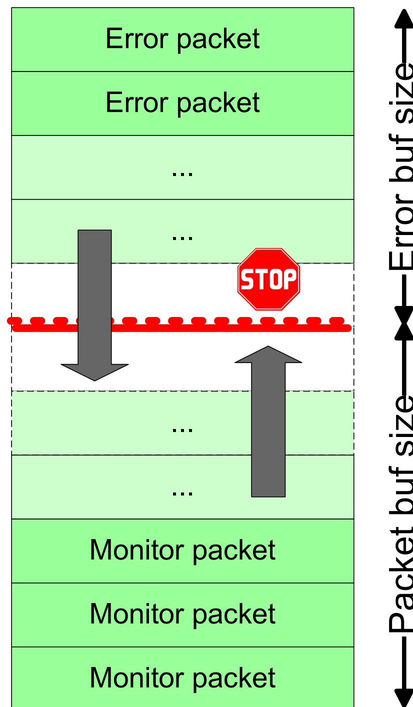


Figure 3-3
The Process Packet buffer is separated in two different parts:

1. *Error packet*
2. *Monitor packet*

If there is a bursty error traffic the Monitor packet buffer side can, in case of an error-side overflow, be used for further error packet buffering. While Monitor packets will be dropped in case of an Monitor-side overflow, which guarantees that the generated error packets can still be buffered.

probably lead to less congestion in the supporting debug network depending on its congestion avoidance facilities. In case of DSN it is advisable to choose the default value of 200ms or at least a value above 100ms. If it is set to zero the buffer will be flushed out only delayed in between each packet by the amount of time needed between a send call and its send done event. On the other hand, the packets can leave the Process Packet buffer to be stored in the node's storage facility.

3.6 Storage

Packets can be sequentially stored in storage for the following reasons, which is why TinyOS's LogStorageC [6] with its interfaces LogWrite and LogRead has been chosen.

Whether a node in a debugged network reboots in some circumstances, powers off for whatever reason or the test criteria forces the surrounding debug instance to come into operation just after the test has finished, the logging facility offers to safely store all desired packets. After testing or whenever needed, the stored packets can be read out from the storage facility and sent over the test platform interface.

Reading out packets takes place in the same manner as packets sent directly over the test platforms send interface. A packet will be read from flash, put into the Process Packet buffer and sent out, after the specified delay time the next packet in flash will be read and so on. There are two read commands implemented in the test core engine. On the one hand, all packets stored can be sequentially read out. On the other hand, only a part of the stored packets starting at a specified number

of packets ending at the desired amount of packets can be read out. This could be used for example in case of a partial loss of packets while reading the whole log to recover the lost packets.

Before the first write or read operation on the flash, it has to be erased at least once. Therefore and for any other new logging start the LogWrite erase function can be called by command, which will clear all pages in flash and set the log pointer back to its beginning.

With the INST_LOG_CIRCULAR defined in the Makefile (see Appendix D) set to *FALSE* the flash will be just filled up once and all following packets are discarded. On the other hand, if set to *TRUE* the flash will be written in a circular manner. After it has been filled up once an error is raised telling that from now on old packets will be overwritten.

The file *volumes-stm25p.xml* used with the TMote, which divides the storage in different segments, always has to be in the compile folder if the test core engine is used with a tmote. It has to be used to define the amount of logging space needed for each specific test. In case of the tested TmoteSky the maximum amount of available storage is 48 kByte. For further explanations please refer to TEP 103 [6] or the TinyOS Tutorial section [7]

3.7 DSN interface

To be able to use DSN with the test core engine, a little interface called DSNConnect had to be implemented. The used DSN command, passing the packets to its destination, is transferring data as an ascii-hex stream. Because the test core engine interprets all data as binary, a ascii-hex to binary converter has been written. So all arriving packets over the test platform interface will get converted before being forwarded to the test core engines receive interface. The conversion has to be set with INST_ASCII_TO_HEX in the Makefile (see Appendix D) if the test core engine is used with DSN.

4

Evaluation

The evaluation of the test core engine is divided into four different subsections. In the first part, the RAM/ROM requirements of the instrumentation will be discussed, which should be as small as possible, so that the application under test can use as much memory as possible. The module has been tested with a two way approach. On the one hand, a so called Dummy Application was built to test it under stress and well defined conditions. On the other hand, the well known TinyOS program Multihop Oscilloscope has been wired to the test core engine to produce some output. In the last part, future implementation ideas are discussed, which came up during the development time.

4.1 RAM/ROM requirements

The test core's ROM requirements depend on its code size, on the number of Monitor calls needed from the application under test, on the implemented Driver calls and on already used components from the application under test, which can be shared with the test core engine (see Figure 4-1). For example, the Multihop Oscilloscope (see Section 4.3), which shares the Timer but not LogStorage component with the Instrumentation, produces a 11KB ROM overhead. Currently, requirements are already really slim but not yet fully optimized. There is still place for improvements which can be dealt with in a future approach. If we look at the amount of ROM used per Monitor call, it's in average 24 bytes. Thus, it is not the amount of different Monitors used that is of interest, but the number of calls used in the application under test.

However, if we look at the test core's RAM requirements it only depends on the chosen buffer sizes. As all three buffers store the same packet structure, we can see it as one big buffer. After compiling the test core engine with lots of different buffer sizes, the conclusion leads to a 32 bytes increase per one packet slot buffer increase. This is a minimum overhead of 4 bytes per fully used packet, which results from the QueueC, plus 28 bytes of payload resulting totally in 32 bytes. And a maximum

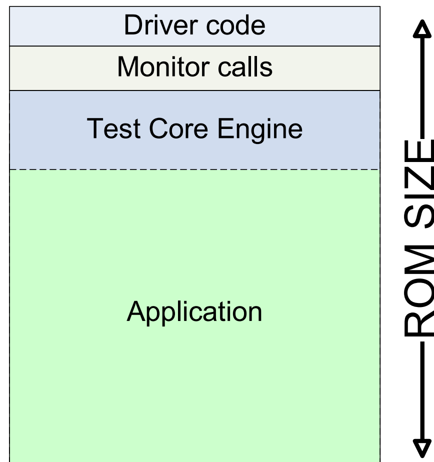


Figure 4-1
The ROM requirements consits out of four elemnts:

- *The test core engine's code size.*
- *The Driver code implemented in the application under test.*
- *The number of Monitor called from the application under test.*
- *And the code size of the application.*

Shared components used from the application under test and from the test core engine will have influence on the ROM requirements as well.

overhead of 22 bytes in case of an error packet in a buffer slot, which comes from 4 bytes QueueC overhead plus 18 bytes of wasted payload space.

4.2 Dummy Application

The Dummy Application has been coded for testing reasons only. It is wired to the test core engine using the two Driver interfaces and calling six different Monitors. The only thing it does is to wait for Driver calls. On an incoming call it raises a 100ms one-shot timer which then calls one out of the six Monitors using the identification number. This Monitor will then be called as many times as stored in the first payload byte passing the whole payload back to the test core engine (see Listing 4.1. This has been used to generate some bursty Monitor traffic testing the instrumentation under stress conditions.

```
event void Timer.fired() {
  uint16_t i;
  if (_Monitor == 1) {
    for(i=0;i<_Payload[0];i++)
    call Monitor1.logMonitor(_Payload, _Len);
  }
  else if (_Monitor == 2) ...
}
```

Listing 4.1

The DummyApplication's Monitor calling code segment included in the Timer event.

4.2.1 Test settings

Two different test settings have been used with the Dummy Application to verify the functionality of the test core engine:

The first one is used to test the instrumentation without intentionally producing any errors. It starts with a *get Unix compile time* command, verifying that the proper application is running on the mote, followed by a flash erase, which has to be done after reprogramming in order to use the flash. Between each command a five second delay is implemented, so that no congestion slows down DSN. Then, the internal state is switched to store all packets to the node's storage facilities. So from now on no more packets will be sent over the test platform interface. This will be used to verify the test core engine's storage functionality. Now six Driver packets are sent to use the Dummy Application's Monitor call reply mechanism mentioned above. With different identification numbers all six Monitors called at least once. So we can verify the returning IDs on their correctness. The used payload varies from three to the maximum of eighteen bytes assuring its proper availability.

Using the default Makefile values offers a ten packets Process Packet buffer. So in the test setting, different values in the first payload byte were used which did not exceed ten. This guarantees that there will be no buffer overflows as a result of too many Monitor calls simultaneously. Now if everything works, there should be 31 packets stored in the logging facility. The next command reads out all data from the flash, allowing us to see if all packets have been properly saved. This automatically changes the internal state back to sending the packets over the test platform interface. As the stored packets should survive a reboot of the nodes, we now turn off its power supply for a few seconds. Then, we read out again all stored packets followed by a partial readout (see Appendix B). This will be used to prove the consistency of the packets in the mote's flash.

The second test setting is used to intentionally produce some errors verifying the correctness of the test core engine's error handling facility. It also starts with a *get Unix compile time* command followed by a single Driver packet. But this time the first payload byte is greater than ten. This leads to a buffer overflow, because there are too many Monitor calls at the same time. This produces an error packet with code 1301 (see Appendix C). All generated packets are directly sent over the test platform interface.

4.2.2 Serial - Test results

Using the nodes serial port offers a reliable communication method. There is no radio transmission needed to successfully collect all generated packets. So we can expect to receive all data without any loss at all. To verify the expected data, ZOC [8] has been used, a well known terminal emulator. The commands are sent directly in ascii-hex code and the nodes reset button has been used to emulate the power off phase.

For the first test setting the expected amount of packets is 70:

- 1 Unix compile time packet
- 31 packets from the first flash read + 1 end of flash packet
- again 31 packets from the second flash read + 1 end of flash packet after reboot
- 5 packets from the partial flash read command

All seventy packets successfully arrived after entering all commands manually using ZOC.

For the second test setting the expected amount of packets is 12:

- 1 Unix compile time packet
- 10 Monitor packets
- 1 error packet reporting the buffer overflow

All twelve packets successfully arrived.

4.2.3 DSN - Test results

In contrast to the one node serial test, the DSN test offers a much more realistic environmental setting. 20 nodes have been chosen to run simultaneously operated by remote procedure calls (RPC) offered by the DSN-Server. All nodes are started at the same time. After that, one node after the other receives the first ten commands in ascii-hex. This is the phase until power off. When all 20 nodes are powered off by RPC, one after another is powered on again and the last two commands are sent, reading all logs simultaneously and then only five packets out of the flash.

Again for the first setting the same 70 packets as with the serial setup are expected to reach the DSN-Server for each of the 20 nodes.

All 20 nodes answered with 70 packets.

In the second error test setting, all twenty nodes answered with 12 packets. They are again the same packets as within the serial setup.

4.3 Multihop Oscilloscope

The Multihop Oscilloscope is a real application shipped with the TinyOS's source code. It is a simple data-collection application. All nodes in the sensor network periodically sample their default sensor and broadcast a message every few readings. The root node collects all received messages.

4.3.1 Test setting

With the Multihop Oscilloscope the idea is to show the Instrumentation in use with a real application. The only change made to the original Multihop Oscilloscope code is to wire the Instrumentation component to it and use two Monitor calls. The Monitors will be called whenever data is been sent and when the root node receives a message.

The Driver interface is not wired to the application. Hence the default Driver command will be called each time a Driver packet arrives at the test platform receive

interface. This will lead to an error packet generation, telling us the absence of its wiring.

This time twenty nodes are flashed with the altered Multihop Oscilloscope application. Again the test starts with a *get Unix compile time* command, verifying that the proper application is running on the mote. Then an arbitrary Driver packet is being sent to trigger the default Driver implementation to answer with an error packet (see Appendix C). The whole test will take one minute, so that enough messages have been generated.

4.3.2 DSN - Test results

All twenty nodes answered with the predicted packets:

- 1 Unix compile time packet
- 1 Default Driver command error packet
- Some Monitor packets containing the sensor data

There are no packet losses encountered either; all packet numbers are increased by one continuously.

4.4 Future implementations

All work on the test core engine described so far has been implemented during this semester thesis. There is still open work to do which will be discussed in this section.

4.4.1 Flash EBUSY handling

Currently, if the flash is already busy by the application under test and the test core engine tries to store some packets, an error packet will be generated. This can lead to a sudden buffer overflow by a huge amount of generated error packets. A good thing would thus be to implement some kind of back off timer, retrying to store a packet in case of a received *EBUSY* dropping the packet after a specified retry number. All the important sections in the source code have been marked with a *TODO*.

4.4.2 Monitor ID Parser

To be able to accurately associate the used Monitors to its Ids, it would be advantageous to write a little parser, which analyzes the precompiled C code. As the Monitor interface is a NesC conform generic Service Instance, its ID's will be assigned at compile time from the NesC compiler. Therefore, the different ID's will be already hard-coded in the precompiled C code. So they could be easily parsed and written to the desired output.

4.4.3 Echo

If not guaranteed by the test platform, you cannot be sure if your command really reached the test core engine. For example if used with DSN, that is exactly the case. So it would be useful to implement a simple command echo to have at least a small hint if the command really reached its destination. This could be easily implemented in the test cores receive interface.

4.4.4 State storage

In case of a node reset or reboot, all internal state variables will be set back to its initial state. For example, the packet number counter will be set back to zero. So an idea is to use the TinyOS's ConfigStorage to store the state of the test core engine in the flash periodically, which could be restored after a nodes reboot. This could be really useful for long time tests.

4.4.5 Packet storage requirements

Because of the low storage capacity of today's sensor nodes it may be necessary to lower or if possible eliminate the packet storage overhead. Till now each packet in the flash uses the maximum allowed size of *TOSH_DATA_LENGTH* bytes regardless of its actual one. This could be achieved by storing variable log sizes. But the good thing about the up to date implementation is its simpleness. So It will always be a trade off between ROM and flash requirements.

4.4.6 Flash lifetime

Flashes and everything else will not live forever. Its lifetime heavily depends on the number of read/write cycles. So decreasing the amount of writing to flash per incoming packets and reading from it to send packets over the test platform interface could increase the lifetime of the nodes storage facility. To achieve this goal, we have to bundle packets together and read/write them simultaneously. But this would lead to a higher packet loss probability in case of a node reset. So again it is a trade off between lifetime and packet loss probability.

5

Summary

The instrumentation component with its high ease of use can be without much effort wired to an existing TinyOS application. It is designed to test any application offering enough flexibility to stimulate and monitor data. With its Makefile, the test core engine can easily be fitted to the developers needs, for example to adjust the different buffers to handle bursty traffic. A strong error handling is implemented, which allows the user to detect any abnormality occurring during test phase. Last but not least, the test core engine is readily usable as the evaluation with the Multihop Oscilloscope shows.

A

Little HOWTO

The instrumentation component can be easily used in a TinyOS program as any other component. This howto is written for the TMoteSky.

A.1 Configuration file

First of all you have to wire the InstrumentationC to the chosen test platform component in your configuration file. Therefore you have to wire TestingCommands, TestingReplies, as well as TestConfig to the test platform component. In the following example, the DSN component is used as test side component, so we have to wire as well DSNConnectC.DSN to DSNC.DSN.

```
components InstrumentationC as Inst;

components DSNConnectC;
DSNConnectC.TestingCommands <- Inst.TestingCommands;
DSNConnectC.TestingReplies <- Inst.TestingReplies;
DSNConnectC.TestConfig <- Inst.TestConfig;

components DSNC;
DSNConnectC.DSN -> DSNC.DSN;
```

Listing A.1
Wiring all needed interfaces between the Instrumentation and the DSN component.

This would already be sufficient if you only used the test core engines internal functionality. But as the intention is normally not only to do so, you can wire the needed number of Monitors to your application.

```
components new MonitorC() as Monitor1;
App.Monitor1 -> Monitor1;
components new MonitorC() as Monitor2;
App.Monitor2 -> Monitor2;
```

On the other hand, and this must only be done if needed, the Driver and TestControl interface can be wired to your application.

```
App.Driver <- Inst.Driver;
App.TestControl <- Inst.TestControl;
```

A.2 Module file

To finish the wiring of your application you also have to wire the module file as follows:

List all used Monitors in the *uses* section.

```
interface Monitor as Monitor1;
interface Monitor as Monitor2;
```

And if needed list the Driver and TestControl in the *provides* section.

```
interface Driver;
interface Driver as TestControl;
```

Now you can call the Monitors within your code whenever it is required.

```
call Monitor1.logMonitor(Payload, Len);
call Monitor2.logMonitor(Payload, Len);
```

If you have chosen to use the Driver interface too, you have to implement the command callDriverFunction which has to return a normal *error_t* message handled by the test core engine.

```
command error_t Driver.callDriverFunction(
 uint16_t id, void* payload, uint8_t len)
command error_t TestControl.callDriverFunction(
 uint16_t id, void* payload, uint8_t len)
```

And it must signal a callDriverFunctionDone with parameter SUCCESS on succeeding or FAIL on failing to process the incoming command.

```
signal Driver.callDriverFunctionDone(SUCCESS);
signal TestControl.callDriverFunctionDone(SUCCESS);
```

A.3 Flash partitioning

Now the source code should be ready to use and we can go on with the storage requirements. Before compiling we have to create or at least alter the file *volumes-stm25p.xml* if already available, which stores the flashes partition information. If the storage facility is only used by the test core engine you can use all the available memory for it, otherwise you have to consider the different requirements and find a good balance. This file has to be placed in the compiling directory.

Volumes-stm25p.xml could for example look like this:

```
<volume_table>
<volume name="INSTRUMENTATION" size="131072" />
</volume_table>
```

Listing A.2
Volumes-stm25p.xml used to partition the flash.

A.4 Makefile

The last step before compiling is to specify all the test specific parameters in the makefile (see Appendix D), the different buffer sizes, the send delay, the amount of packet resends on fail, if the flash should be used in circular mode or errors should be sent at any time directly over the test platform interface in the debug mode. If used with DSN set USART to zero for deployment with the supporting DSN-node or to one if used with the serial interface.

A.5 Compiler

Compile it and hope for the best.

When the program has been flashed to the node(s) you can send a get Unix compile time command to see if the right answer is coming back.

If everything went fine, the test core engine is now ready to use.

B

Instruction set

State commands

Command ascii-hex	Command description
0x0000	Get the Unix compile time of the actual image.
0x0001	Set the internal storage flag. All packets will be stored to flash.
0x0002	Unset the internal storage flag. All packets will be directly sent over the test platform interface.

Table B-1: State command table

Storage commands

Command ascii-hex	Command description
0x0100	Read out all data from flash (unsets the internal storage flag).
0x0101	Read out a part of the data from flash using the two leading payload bytes (unsets the internal storage flag). The first byte is used to specify the beginning packet (numbers of stored packets not packet numbers). The second byte is used to specify the amount of packets to be read.
0x0102	Erase the flash (this has to be done at least once after programming the node).

Table B-2: Storage command table

C

Error table

Error code		Error description
decimal	hex	
		Receive
1000	0x3e8	received packet length to long
1001	0x3e9	received packet length to short
1002	0x3ea	received packet length and header included length doesn't match
1003	0x3eb	invalid Class received
		Control
1200	0x4b0	control buffer full
1201	0x4b1	control queue full (shouldn't happen!!!)
1202	0x4b2	RecvCtrlPool.put(..) failed (shouldn't happen!!!)
2000 + idUp	0x7d0 + ...	wrong idUpperPart passed
2300 + idLow	0x8fc + ...	wrong idLowerPart passed for state
2600 + idLow	0xa28 + ...	wrong idLowerPart passed for storage
		Packet processing
1300	0x514	processed payload length is too long
1301	0x515	packet Buffer full
1302	0x516	processPacketBuffer is empty, could not send or store (shouldn't happen!!!)
1303	0x517	ProcessPacketPool.put(..) failed
		Packet sending
1400	0x578	sending busy
1401	0x579	send done failed

Table C-1: Error table

Error code		Error description
decimal	hex	
		Driver/TestControl
1600	0x640	driver done failed
1601	0x641	testcontrol done failed
1602	0x642	command buffer full
1603	0x643	command queue full (shouldn't happen!!!)
1604	0x644	RecvCmdPool.put(..) failed
1605	0x645	Default Driver command called, Drivers not wired to the application.
		Monitor
1300	0x514	processed payload length is too long
		Packet storage
1800	0x708	erase busy
1801	0x709	log seek failed
1802	0x70a	flash full, from now on overwrite (only with circular log)
1803	0x70b	end of log reached
1804	0x70c	erase done failed
1805	0x70d	append failed
1806	0x70e	append done failed
1807	0x70f	sync failed
1808	0x710	sync done failed
1809	0x711	log read busy

Table C-2: Error table

D

Makefile

```
FILENAME=DummyApplication
COMPONENT =$(FILENAME)C
CFLAGS += -DTOSH_DATA_LENGTH=28
CFLAGS += -DUSART=0

#errors are not stored (directly sent)
CFLAGS += -DINST_DEBUG_MODE
#convert incoming ascii-hex stream to binary (needed only in DSNConnect)
CFLAGS += -DINST_ASCII_TO_BIN
#if used with SerialDemo use with default value or higher!!!
#Packets are sent with delay (no bursts , Default=200)
CFLAGS += -DINST_PACKET_SEND_DELAY=200
#Receive Control Packet Buffersize (Default=10, don't set to zero)
CFLAGS += -DINST_RECVCTRL_BUF_SIZE=10
#Receive Command Packet Buffersize (Default=10, don't set to zero)
CFLAGS += -DINST_RECVCMD_BUF_SIZE=10
#Send/Store Packet Buffersize (Default=10, don't set to zero)
CFLAGS += -DINST_PACKET_BUF_SIZE=10
#Error Buffersize (Default=10, don't set to zero)
CFLAGS += -DINST_ERROR_BUF_SIZE=10
#Log overwritten when full? (Default=FALSE)
CFLAGS += -DINST_LOG_CIRCULAR=FALSE
#Maximum Packet resend on fail (Default=5)
CFLAGS += -DINST_PACKET_RESEND_MAX=5
#Maximum Packet restore on fail (Default=5)
CFLAGS += -DINST_PACKET_RESTORE_MAX=5
CFLAGS += -I./dsn # dsn
include $(MAKERULES)
```

Listing D.1
The Instrumentation Makefile

E

TinyOS - Wiring

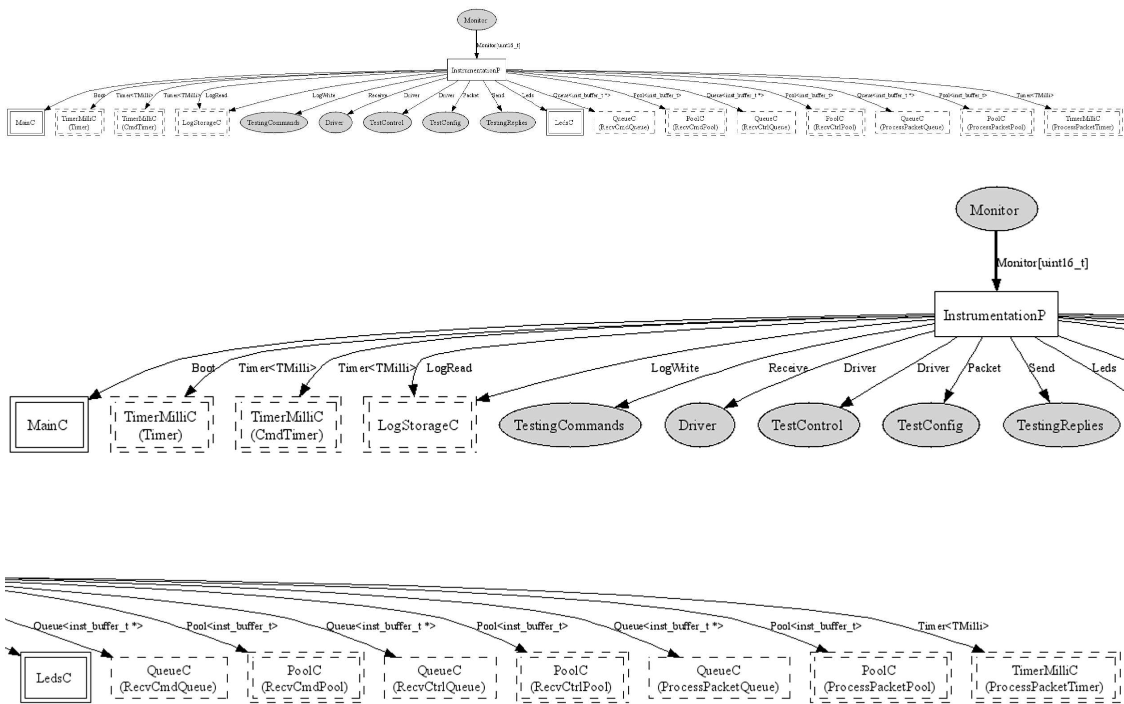


Figure E-1
Wiring graph generated with the 'make doc' command.

Bibliography

[1] Werner-Allen, G., Swieskowski, P., ,Welsh, M.: Motelab: A wireless sensor network testbed. In: Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSNŠ05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS), IEEE, Piscataway, NJ (2005)

[2] Dyer, M., Beutel, J., Kalt, T., Oehen, P., Thiele, L., Martin, K., Blum, P. Deployment Support Network - A toolkit for the development of WSNs. EWSN'07, Delft, Jan 2007.

[3] TinyOS, http://www.tinyos.net, Jan 2008.

[4] TMote Sky datasheet http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf, Jan 2007.

[5] Levis, P,. TinyOS Programming. www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf, Oct 2006

[6] Gay, D., Hui, J. Permanent Data Storage (Flash). TEP 103, Status: Final, TinyOS-Version: 2.x, http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep103.html, Jan 2008.

[7] TinyOS 2.0 Tutorials http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/index.html, Aug 2007

[8] ZOC - A Powerful Terminal Emulator and Telnet/SSH Client for Power Users. http://www.emtec.com/zoc/, Jan 2008

Herbstsemester 2007

SEMESTERARBEIT

für
Tonio Gsell

Betreuer: Matthias Woehrle
Stellvertreter: Jan Beutel

Ausgabe: 24. September 2007
Abgabe: 11. Januar 2008

Instrumentation for TinyOS Platforms

Einleitung

Wireless Sensor Networks (WSNs) ist eine neue Technologie, die in verschiedensten Gebieten eingesetzt wird. Dabei hat sich gezeigt, dass viele Installationen im Einsatz nicht das gewünschte Ergebnis erzielen. Eine möglicher Ausweg aus dieser Situation ist das System ausgiebig zu testen, bevor es installiert wird. Dafür kann man Simulatoren benutzen oder dedizierte Testbeds, die echte Hardware benutzen und damit dem System-Betrieb in der realen Welt sehr nahe kommen. Ausgiebiges Testen kann mehrere dieser Platformen benutzen, z.B. durch einen generellen Ansatz wie in beschrieben. Die Basis für Testen ist ein Zugang zu dem System, den einzelnen Devices-Under-Test (DUT), d.h. den Sensor Knoten. Dafür müss die eingebettete Software instrumentiert werden um Ereignisse zu überwachen mit einem sogenannten Monitor oder zu stimulieren über eine Treiber (Driver). Die Instrumentation muss auf der einen Seite flexible sein um für den Entwickler benutzbar zu sein. Auf der anderen Seite muss die Instrumentierung die Applikation so wenig wie möglich stören.

Aufgabenstellung

1. **Erstellen Sie einen Projektplan und legen Sie Meilensteine sowohl zeitlich wie auch thematisch fest. Erarbeiten Sie in Absprache mit dem Betreuer ein Pflichtenheft.**
2. **Machen Sie sich mit den relevanten Arbeiten im Bereich Sensornetze, Eingebettete Software und Software Instrumentierung vertraut.**
3. **Arbeiten Sie sich in die relevanten Technologien ein: DSN, und Tmote.**

4. **Verständniss von TinyOS-2.x Applikationen, er Instrumentierung von TinyOS-2.x Applikationen und der Anbindung eines Test-Moduls.**
5. **Verstehen Sie die spezifischen, relevanten Eigenschaften der Tmote Sky Platform in TinyOS (Flash). Erarbeiten Sie Buffer Anforderungen für Schnittstellen und Konzepte für das Verarbeiten von gleichzeitigen Zugriffen.**
6. **Erstellen Sie ein Konzept wie der momentan vorhandene Prototyp verbessert werden kann und erweitert werden muss.**
7. **Implementieren Sie ihr Konzept in einem lauffaehigen Prototypen anhand einer gegebenen Applikation.**
8. **Analysieren Sie Ihren Prototypen in Testläufen. Optional kann eine einfache Test Infrastruktur z.B. in Form eines Skripts erstellt werden.**
9. **Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.**

Durchführung der Semesterarbeit

Allgemeines

- **Der Verlauf des Projektes soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.**
- **Sie verfügen über PC's mit Linux/Windows für Softwareentwicklung und Test. Für die Einhaltung der geltenden Sicherheitsrichtlinien der ETH Zürich sind Sie selbst verantwortlich. Falls damit Probleme auftauchen wenden Sie sich an Ihren Betreuer.**
- **Stellen Sie Ihr Projekt zu Beginn der Semesterarbeit in einem Kurzvortrag vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums Ende Semester.**
- **Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern. Verfassen Sie dazu auch einen kurzen wöchentlichen Statusbericht (email).**

Abgabe

- **Geben Sie zwei unterschriebene Exemplare des Berichtes spätestens am *11. Januar 2008* dem betreuenden Assistenten oder seinen Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.**
- **Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Source- und Objectfiles, Konfigurationsfiles, benötigten Directorystrukturen usw. bestehen bleiben. Der Programmcode sowie die Filestruktur soll ausreichen dokumentiert sein. Eine spätere Anschlussarbeit soll auf dem hinterlassenen Stand aufbauen können.**