



Wiselib School 2010  
Session 1  
Introduction & Preliminaries

Henning Hasemann

Institute of Operating Systems and Computer Networks

October 2010

# Outline

- 1 School organization
- 2 Motivation
  - A Library Of Algorithms
  - Platform Independence
- 3 Design Of The Wiselib
  - Memory Management
  - Abstraction With Templates
  - Concept Architecture
  - Integrating The Wiselib
- 4 The Development Environment

# Basics

- These Slides can also be viewed online at <http://www.ibr.cs.tu-bs.de/alg/winterschool/>
- There you'll also find material for practical lessons

# Session structure

Talk

*Hands On Session* (Practical lesson)

Discussion of the results

# Sessions

## Monday

**Session 1:** Introduction & Preliminaries (Henning Hasemann)

**Session 2:** Wiselib Programming Basics (Tobias Baumgartner)

## Tuesday

**Session 3:** Combining Algorithms Into Higher-Level Systems  
(Henning Hasemann)

**Session 4:** Insight Into Wiselib Internals (Tobias Baumgartner)

## Wednesday

Semantics Workshop

(Dr. Alexandre Passant, Dr. Marcel Karnstedt, Myriam Leggieri)

# Outline

- 1 School organization
- 2 Motivation
  - A Library Of Algorithms
  - Platform Independence
- 3 Design Of The Wiselib
  - Memory Management
  - Abstraction With Templates
  - Concept Architecture
  - Integrating The Wiselib
- 4 The Development Environment

# Outline

- 1 School organization
- 2 Motivation
  - A Library Of Algorithms
  - Platform Independence
- 3 Design Of The Wiselib
  - Memory Management
  - Abstraction With Templates
  - Concept Architecture
  - Integrating The Wiselib
- 4 The Development Environment

# Typical Problems In WSN Programming

- Theoreticians are not interested in programming
  - Ideally they just have to write their algorithms
  - And do not need to care about boilerplate code
- Practitioners are not interested in theory
  - Just need a good algorithm for their task
  - Without having to study the field for years

⇒ There is need for an algorithm library

- With lots of algorithms for all kinds of tasks
- That are easy to integrate into existing systems
- And are combinable
- And easily enhanceable



# The Wiselib

A library of about 50 algorithms, lots more to come! These are

- Extensible
- Combineable
- Exchangeable

Currently includes the following algorithm categories

- Clustering
- Graph Coloring
- Crypto
- Energy Preservation
- Localization
- Metrics
- Routing
- Synchronization
- Topology Control
- Tracking

# The Wiselib is...

- A C++ project
- Free (as in freedom), licensed under LGPL
- **NOT** a middleware (we will see later why)

<http://wiselib.org>

There you'll find:

- The Documentation Wiki
- The Wiselib Sourcecode
- The Bugtracker
- Instructions on how to download & install the Wiselib

# Wiselib Distributions

## Testing

- Under development
- Not necessarily tested on all platforms
- New things that may still change their interface
- “Release early, release often”
- <https://svn.itm.uni-luebeck.de/wisebed/wiselib/trunk/wiselib.testing>

## Stable

- Tested on all supported platforms
- Interfaces will not change anymore
- <https://svn.itm.uni-luebeck.de/wisebed/wiselib/trunk/wiselib.stable>

# Outline

- 1 School organization
- 2 **Motivation**
  - A Library Of Algorithms
  - Platform Independence**
- 3 Design Of The Wiselib
  - Memory Management
  - Abstraction With Templates
  - Concept Architecture
  - Integrating The Wiselib
- 4 The Development Environment

# Platform Independence

- When scientists all over the world work together, they likely use different experimentation environments
- The Wiselib aims to be **versatile**
  - So it can be used for **different tasks**
  - Which also require **different hardware**
- In lots of applications we need **heterogeneous nodes**
  - But do not want to write the same code again and again for each node type

→ We want the Wiselib to be platform independent!

# Platform Independence

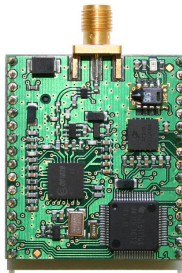
iSense



iMote2



ScatterWeb MSB



Tmote Sky



## Hardware

Jennic

Intel XScale

MSP430

MSP 430

## Operating System

iSense

TinyOS

Scatterweb / Contiki

Contiki / TinyOS

## ROM / RAM

128kB / 92kB

32MB / 32MB

48kB / 10kB

48kB / 10kB

## Memory Management

Dynamic

Dynamic

Static

Dynamic

## Programming Language

C++

nesC

C

C, nesC

# Platform Independence

- Some platforms do not provide dynamic memory
- And/or have limited RAM
- Some do not provide a C++ environment
  - No libstdc++
  - So no exception handling, RTTI, virtual inheritance, etc...

The “extremely portable” subset of C++

- C (except malloc / free)
- Static memory management
- “Simple” (non-virtual) inheritance
- Templates
- Use C-Headers (`<math.h>` instead of `<cmath>`)

**The Wiselib adheres to those conditions!**

# Outline

- 1 School organization
- 2 Motivation
  - A Library Of Algorithms
  - Platform Independence
- 3 Design Of The Wiselib
  - Memory Management
  - Abstraction With Templates
  - Concept Architecture
  - Integrating The Wiselib
- 4 The Development Environment



# Outline

- 1 School organization
- 2 Motivation
  - A Library Of Algorithms
  - Platform Independence
- 3 Design Of The Wiselib
  - Memory Management
  - Abstraction With Templates
  - Concept Architecture
  - Integrating The Wiselib
- 4 The Development Environment

# Memory Management

Platform independence demands:

- No `malloc/free` or `new/delete`
- Data can be allocated in 3 ways:
  - Global
  - Static
  - On the stack (function-local)
- Constructors of global/static variables will be called before `main()`
- ... in undefined order!
- That can be very undesirable:

```
1 Radio radio;  
2 SomeAlgorithm algo(radio); // Might receive uninitialized radio!
```

- Provide `init()/destruct()` methods, call them manually
- Hide initialization method of system objects ("Facets")

(More on this in Session 3)

# Outline

- 1 School organization
- 2 Motivation
  - A Library Of Algorithms
  - Platform Independence
- 3 Design Of The Wiselib
  - Memory Management
  - Abstraction With Templates**
  - Concept Architecture
  - Integrating The Wiselib
- 4 The Development Environment

# Inheritance

**Problem:** Virtual inheritance is not portable.

What would we use virtual inheritance for?

→ **Code reuse**

Base class provides functionality which can be used by derived class

- Still possible with non-virtual inheritance

→ **Abstraction**

Define an *interface* which *classes* can use to interact with each other

- An algorithm only has to know the interface of the things its using, the concrete implementation is exchangeable

We want both!

Do it with templates!

- The “interface” is given by a piece of documentation, called **Concept**
- An algorithm expects a template parameter for the type of the concrete class, which is called **Model**

# Abstraction

## Concept

- Describes behaviour of components
- E.g. “A Radio has a void send(char\*) method”
- Only documentation

## Model

- Actual class
- Implements any number of concepts
- E.g. A routing protocol may implement the radio concept
- ...so it can be used like one

# How Usable Is The Template Approach?

- There are other ways to provide abstraction
  - In C, one would usually abstract with **function pointers**
  - In C++ one would use **virtual inheritance**

How do they compare to the template approach?

# Abstracting with C function pointers

```
1 // C
2 typedef struct {
3     int (*value)(void);
4 } Concept;
5
6 int model_value() { return 5; }
7 Concept model = { .value = &model_value };
8
9 void algorithm(Concept *c) {
10     // pointer->pointer->function
11     int v = c->value();
12 }
13
14 int main(int argc, char** argv) {
15     algorithm(&model);
16 }
```

# Abstracting with virtual inheritance

```
1 // C++
2 class Concept {
3     public:
4         virtual int value();
5 };
6
7 class Model : public Concept {
8     public:
9         int value() { return 5; }
10 };
11
12 class Algorithm {
13     public:
14         // reference->vtable->function
15         void init(Concept& c) { v = c.value(); }
16         int v;
17 };
18
19 int main(int argc, char** argv) {
20     Model m;
21     Algorithm a;
22     a.init(m);
23 }
```



# Abstracting with templates

```
1 // C++
2
3 // concept "Concept" {
4 //   has an 'int value()' method
5 // }
6
7 class Model {
8     public:
9         int value() { return 5; }
10 };
11
12 template<typename Concept_P>
13 class Algorithm {
14     public:
15         // reference -> function
16         void init(Concept_P& c) { v = c.value(); }
17         int v;
18 };
19
20 int main(int argc, char** argv) {
21     Model m;
22     Algorithm<Model> a;
23     a.init(m);
24 }
```

# Comparing the results

After compiling (for jennic, using ba-elf-gcc/ba-elf-g++) with -Os:

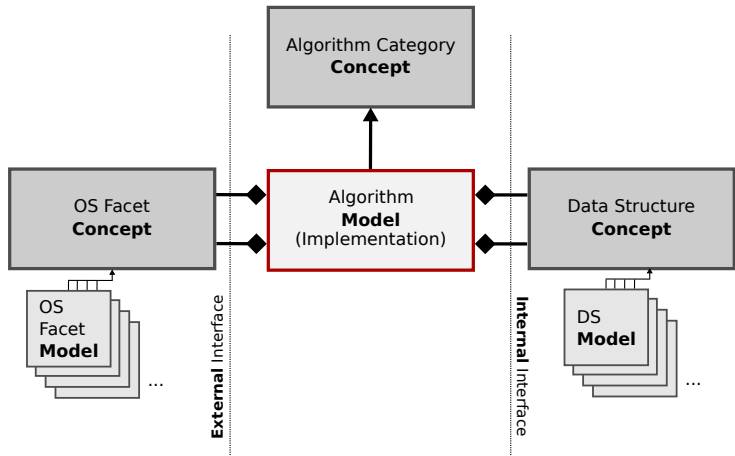
	text	data	bss	dec	hex	filename
1	56	4	0	60	3c	c.o
2	16	0	0	16	10	template.o
3	143	0	0	143	8f	virtual.o

- Template-based design is space efficient!
- Template-based design produces fast code!
- Template-based design is portable!

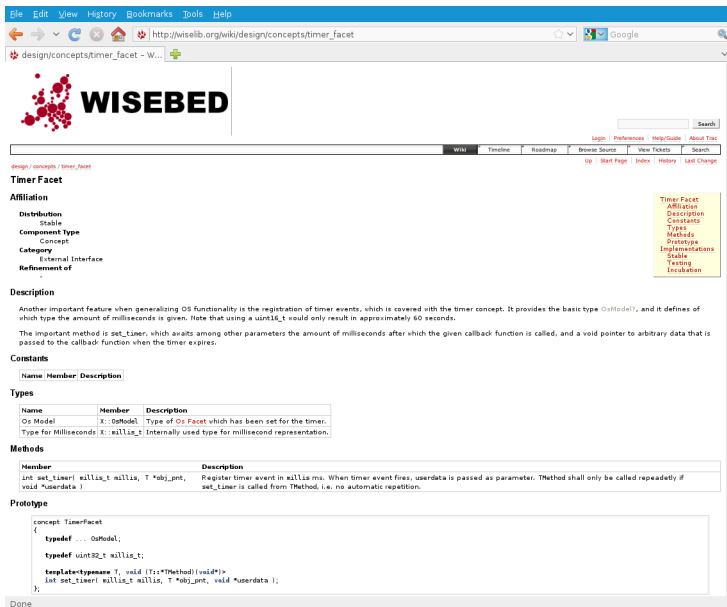
# Outline

- 1 School organization
- 2 Motivation
  - A Library Of Algorithms
  - Platform Independence
- 3 Design Of The Wiselib
  - Memory Management
  - Abstraction With Templates
  - Concept Architecture**
  - Integrating The Wiselib
- 4 The Development Environment

# Types Of Concepts



# How Does A Concept Look?



The screenshot shows a web browser window with the address bar displaying `http://wiselib.org/wiki/design/concepts/timer_facet`. The page features the Wiselib logo (a cluster of red dots) and the title "WISEBED". Below the logo, there is a navigation bar with links: "Wiki", "Timeline", "Readmap", "Browse Source", "View Tickets", and "Search". A search bar is also present. The main content area is titled "Timer Facet" and includes a sidebar with a table of contents. The table of contents lists: Affiliation, Distribution, Component Type, Category, Refinement of, Constants, Types, and Methods. The main content area is divided into sections: Affiliation, Description, Constants, Types, and Methods. The "Description" section contains text about generalizing OS functionality and the registration of timer events. The "Constants" section contains a table with columns: Name, Member, and Description. The "Types" section contains a table with columns: Name, Member, and Description. The "Methods" section contains a table with columns: Member and Description. The "Prototype" section contains a code block showing the C++ code for the TimerFacet class.

**Timer Facet**

**Affiliation**

<b>Distribution</b>
Stable
<b>Component Type</b>
Concept
<b>Category</b>
External Interface
<b>Refinement of</b>

**Description**

Another important feature when generalizing OS functionality is the registration of timer events, which is covered with the timer concept. It provides the basic type `OsModel`, and it defines of which type the amount of milliseconds is given. Note that using a `uint16_t` would only result in approximately 60 seconds.

The important method is `set_timer`, which awaits among other parameters the amount of milliseconds after which the given callback function is called, and a void pointer to arbitrary data that is passed to the callback function when the timer expires.

**Constants**

Name	Member	Description

**Types**

Name	Member	Description
Os Model	X: <code>OsModel</code>	Type of <code>Os Facet</code> which has been set for the timer.
Type for Milliseconds	X: <code>millis_t</code>	Internally used type for millisecond representation.

**Methods**

Member	Description
<code>int set_timer( millis_t millis, T *obj_ptr, void *userdata )</code>	Register timer event in millis ms. When timer event fires, userdata is passed as parameter. TMethod shall only be called repeatedly if <code>set_timer</code> is called from TMethod, i.e. no automatic repetition.

**Prototype**

```
concept TimerFacet
{
    typedef ... OsModel;
    typedef uint32_t millis_t;

    template<typename T, void (T::*TMethod)(void*)>
    int set_timer( millis_t millis, T *obj_ptr, void *userdata );
};
```

# Concept Organization

- Lots of models
  - Lots of concepts
  - Models that behave similar should share concepts
  - E.g. A routing algorithm should be usable like a radio  
For the user, both are just things that
    - Can receive messages
    - Can send messages to nodes
    - Only the neighborhood is different!
  - But a routing algorithm might have additional methods!
- We want a (loose) hierarchy of concepts
- We want to express concept inheritance
- We want to have “base concepts” for general things

# The OsModel Facet

```
1 concept OsModel {
2   typedef ... size_t;
3   typedef ... block_data_t; // "byte"-like type for buffers
4   enum ReturnValues { SUCCESS = ..., ERR_UNSPEC = ..., ... };
5
6   typedef ... Radio; // Wireless communication facet
7   typedef ... Timer;
8   typedef ... Debug; // Send debug messages
9
10  static const Endianness endianness; // WISELIB_LITTLE_ENDIAN or WISELIB_BIG_ENDIAN
11 }
```

- Holds platform properties (like endianness, size type, etc...)
- Constants for return values
  - Include at least SUCCESS and ERR\_UNSPEC (unspecified error)
  - May/will include more, similar to errno
- Holds types of other OS Facets

# Concept Inheritance

```
1 concept RadioFacet {
2     typedef ... OsModel;
3     typedef ... node_id_t;
4     typedef ... block_data_t;
5     typedef ... size_t;
6
7     typedef ... message_id_t;
8
9     enum SpecialNodeIds {
10         BROADCAST_ADDRESS = ...,
11         NULL_NODE.ID      = ...
12     };
13     enum Restrictions {
14         MAX_MESSAGE.LENGTH = ...
15     };
16
17     int enable_radio();
18     int disable_radio();
19
20     int send(node_id_t receiver, size_t len,
21             block_data_t *data );
22
23     node_id_t id();
24
25     // ....
26 };
```

← We “derive” another concept from this one:

```
1 concept VariablePowerRadioFacet
2     : public RadioFacet
3 {
4     // Everything in RadioFacet plus:
5
6     typedef ... TxPower;
7
8     int set_power(TxPower p);
9     TxPower power();
10 };
```



# Base Concepts

BasicAlgorithm
+init(...): int
+init(): int
+destruct(): int

StateCallback
+READY
+NO_VALUE
+INACTIVE
+register_state_callback(): int

State
+READY
+NO_VALUE
+INACTIVE
+state(): int

Request
+typedef value_t
+operator()(): value_t

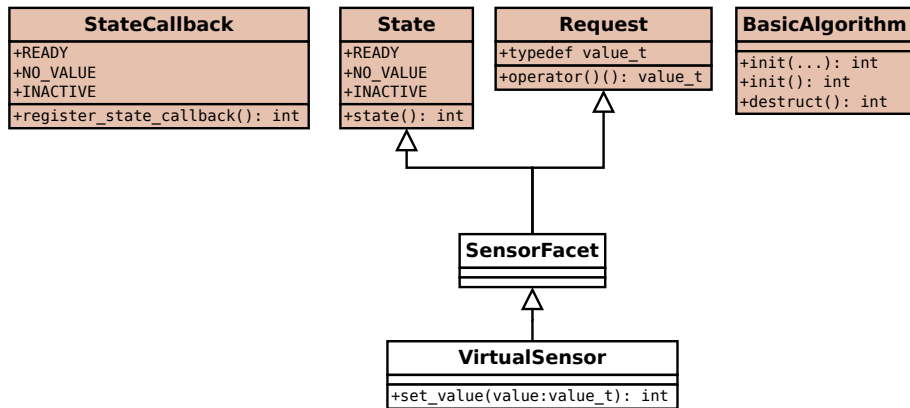
**Basic Algorithm** Manual initialization & destruction (so the order is defineable)

**Request** Produces values (can be polled with call-operator)

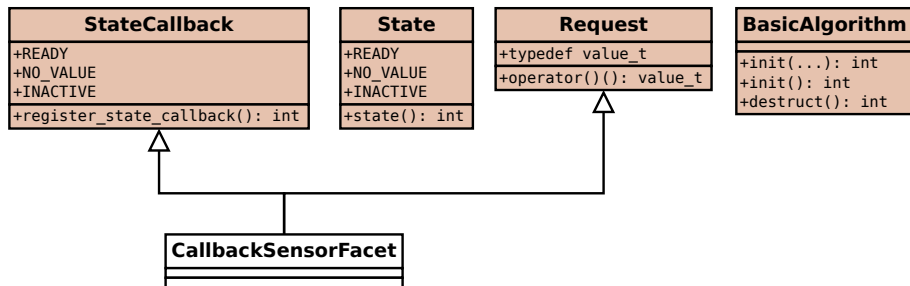
**State** Object is not guaranteed to be able to operate all the time

**StateCallback** Object can inform its user about state changes

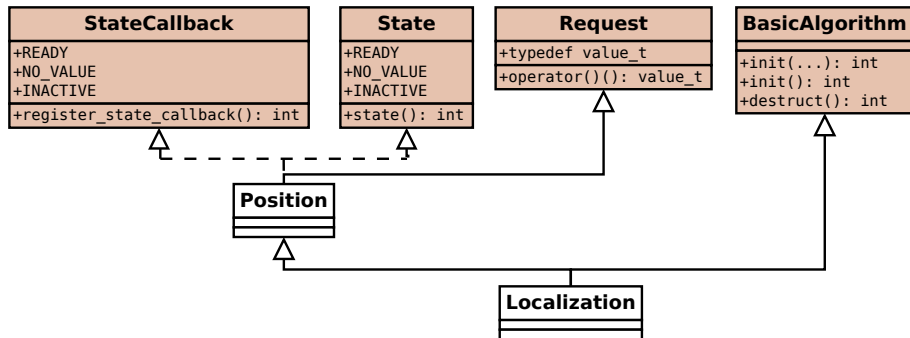
# Base Concepts



# Base Concepts



# Base Concepts



# Stackability

**Idea:** Things with similar behaviour should share a concept!

Routing algorithms behave like radios

- They send and receive data to other nodes
- Routing algorithms implement the *Radio Concept*

Localization algorithms produce a stream of values

- So do sensors!
- Localization algorithms implement the *Sensor Concept* or the *CallbackSensor Concept*

Etc. . .

## Benefit

- Say some algorithm uses a radio (i.e. transmits data)
- We can pass a routing algorithm instead
- And extend the algorithms functionality that way!

# Stackability



- Create arbitrary complex applications
- Just by plugging together algorithms

Here:

- 1 "Physical" radio by iSense
- 2 AES-Encrypted node-to-node radio
- 3 Routing, all packets AES-encrypted node-to-node
- 4 All packets AES-encrypted node-to-node,  
payload ECC encrypted end-to-end

...can be used like a single simple radio!

# Outline

- 1 School organization
- 2 Motivation
  - A Library Of Algorithms
  - Platform Independence
- 3 Design Of The Wiselib
  - Memory Management
  - Abstraction With Templates
  - Concept Architecture
  - Integrating The Wiselib
- 4 The Development Environment

# Integration Demands

- Wiselib components should be easily integrable into existing code
- We want and/or need the full power of the platform. Examples:
  - Dynamically discover attached sensors
  - Fine-tuned device configuration

BUT

- Sometimes you want to run the same application on different platforms
- Advanced hardware settings are relatively unimportant

Two different integration mechanisms needed!



# Integration Mechanisms

## Direct Integration

- Just use whatever parts of the Wiselib you like
  - ⊕ Retain full power of your platform
  - ⊕ Good if you have existing code
  - ⊖ Not portable

## Generic Application

- Write a Wiselib application class
    - ⊕ Can be compiled for all Wiselib backends
    - ⊖ You can only access the operating system through facets
    - ⊖ But functionality will be limited to a common subset
- E.g. you have to write “extremely portable” C++ (no `new/delete`, RTTI, exceptions, ...) in order to retain portability

# Direct Integration

```
1 // ...
2
3 void iSenseDemoApplication::boot(void) {
4     os_.debug("WiselibExample::boot");
5     routing_.enable();
6     routing_.reg_rcv_callback<
7         iSenseDemoApplication,
8         &iSenseDemoApplication::receive_routing_message>(this);
9
10    os_.allow_sleep(false);
11    os_.add_task_in(isense::Time(MILLISECONDS), this, 0);
12 }
13
14 // ...
```

- iSense specific code
- Wiselib specific code

# Generic Application

```
1#include "external_interface/external_interface.h"
2#include "external_interface/external_interface_testing.h"
3// ...
4
5typedef wiselib::PCOsModel Os;
6class DemoApplication {
7    public:
8        void init(Os::AppMainParameter& amp) {
9            radio_ = &wiselib::FacetProvider<Os, Os::Radio>::get_facet(amp);
10            debug_ = &wiselib::FacetProvider<Os, Os::Debug>::get_facet(amp);
11
12            algorithm_.init();
13
14            radio_>enable_radio();
15            debug_>debug(" Initialized.\n");
16        }
17
18    private:
19        Os::Debug::self_pointer_t debug_;
20        Os::Radio::self_pointer_t radio_;
21        SomeAlgorithm algorithm_;
22 };
23
24wiselib::WiselibApplication<Os, DemoApplication> demo_app;
25void application_main(Os::AppMainParameter& amp) {
26    demo_app.init(amp);
27 }
```

Platform selection Initialization: FacetProvider for OS facets / Manual for algorithms application\_main getting called by Wiselib ↔ OS adaptor

# Outline

- 1 School organization
- 2 Motivation
  - A Library Of Algorithms
  - Platform Independence
- 3 Design Of The Wiselib
  - Memory Management
  - Abstraction With Templates
  - Concept Architecture
  - Integrating The Wiselib
- 4 The Development Environment

# USB Sticks

USB Stick contains

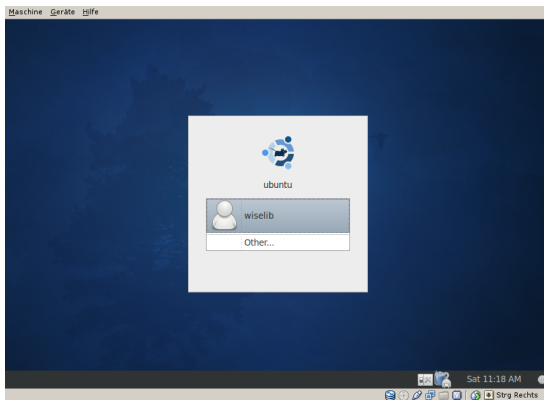
- A virtual machine image
- VirtualBox for using the image
- Naturally you can also install VirtualBox or VMWare from the internet

The VM image can also be downloaded at:

<http://www.ibr.cs.tu-bs.de/alg/winterschool/>

(Note that the bandwidth is limited)

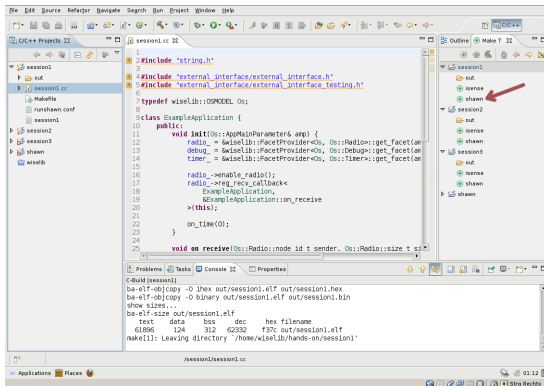
# First steps in the VM



- Get the supplied VM image running
- Log in (user: wiselib / password: wiselib)
- You may adjust your keyboard settings here

# Compiling A Wiselib Application

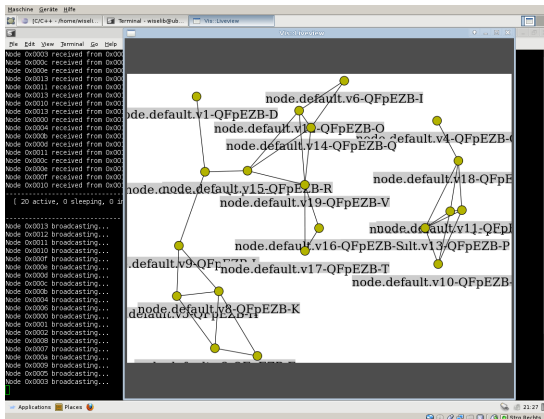
- Start Eclipse



- Verify you are in project “session1” and are editing “session1.cc”
- Double-click on “shawn” (arrow) to compile for shawn
- Verify there are no error messages
- Repeat for “isense”

# Running A Wiselib Application In Shawn

- Start a terminal (*Applications* → *Terminal*)
- Change into directory `hands-on/session1` (relative to `$home`)
- Type `./session1 < runshawn.conf`

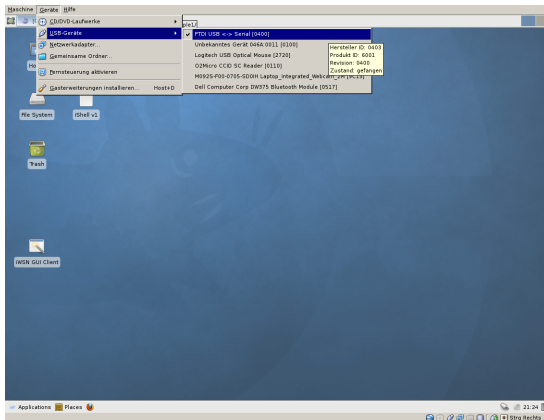


- Should look somewhat like this



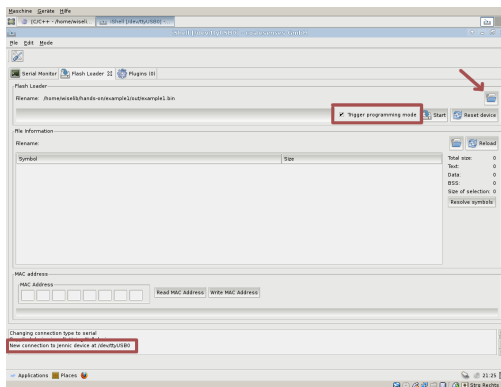
# Connecting An iSense Node

- Close or minimize terminal
- Connect an iSense Node



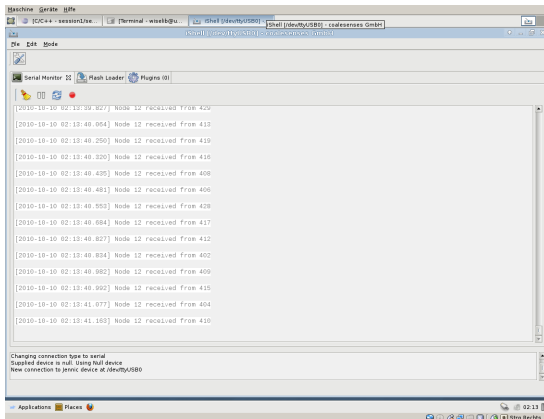
- Make sure you pass-through the USB access

# Flashing An iSense Node



- Verify connection to node was established
- Go to “Flash Loader”
- Verify “Trigger Programming mode” is activated
- Select (arrow) `/hands-on/session1/out/session1.bin`
- Push “Start”

# Observe Node Operation



- Go to “Serial Monitor”