Writing iSense Applications User Guide

coalesenses
research to innovate

Coalesenses GmbH Maria-Goeppert-Str. 1 23562 Lübeck Germany www.coalesenses.com sales@coalesenses.com

Document history

Version	*Date*	*Changes*
0.9	27.6.2008	Initial version
1.0	24.9.2008	doc complete
1.1	24.9.2008	adaptations to changes in iSense
1.2	31.10.2009	New Layout
1.3	17.08.2010	Some adaptations to changes in iSense, added sections on classes memory considerations and exceptions
1.4	12.02.2011	Some adaptations to changes in iSense
1.5	10.05.2012	Adaptation to release of iSense Firmware 2.0

Contents

1. About this User Guide

This guide assists you with the first steps to writing your first iSense application. It shows you

- how to setup a new application
- how to use the software concepts of iSense, including
  - sending debug output via the Gateway Module
  - sending radio packets to neighboring devices
  - using tasks and timeouts
  - taking advantage of the automatic power management
  - using the routing protocols that are part of iSense
  - synchronizing the devices' clocks
- how to compile and flash applications.

In this user guide,

- files and folders are represented in the Arial typeface,
- code fragments, function names etc. are represented in the `Courier New` typeface,
- GUI elements such as button descriptions etc. are represented in "quotation marks",
- titles of other documents are presented in *Italic* type.

This manual assumes that the reader has successfully installed the iSense development environment, and obtained the iSense standard firmware. For further information on these steps, consult the *Development Environment Setup User Guide* [1].

In addition, it is assumed that the user is familiar with the use of iShell. For further information on iShell, consult the *iShell User Guide* [2].

2. Creating an Application Folder

The easiest way to get to a new application folder is copying an application folder such as iApps/iSenseDemoApplication. For the remainder of this section, let's assume that iApps/iSenseDemoApplication is copied to iApps/myApp

Note that if the application folder you copied was under SVN control, you must delete all the .svn entries in myApp and its subdirectories, because otherwise the version control system will get confused.

In order to be able to import the new application into Eclipse as a project, you must slightly edit the .project file in myApp:

- in line 2, change `<name>iSenseDemoApplication</name>` to the desired project name, e.g. `<name>My first application</name>`
- around line 39, search for the entry

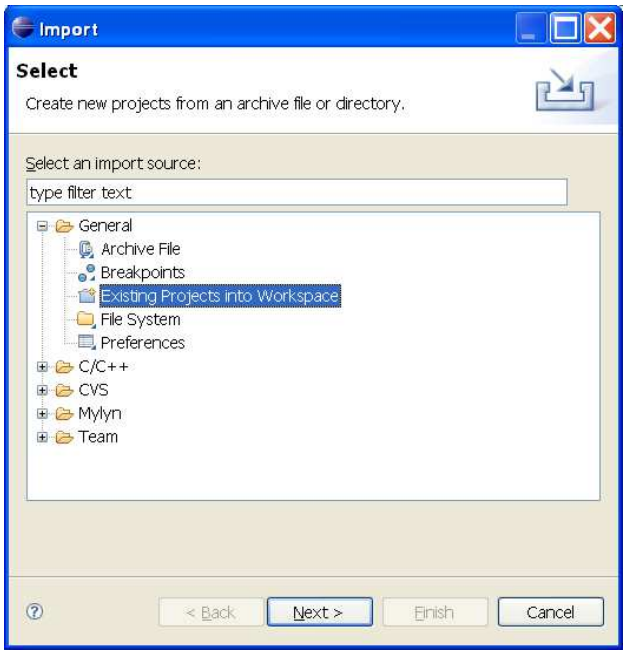
```
<dictionary>
	<key>org.eclipse.cdt.make.core.build.location</key>
	<value>\iSenseDemoApplication</value>
</dictionary>
```

and set it to the new directory name, e.g.

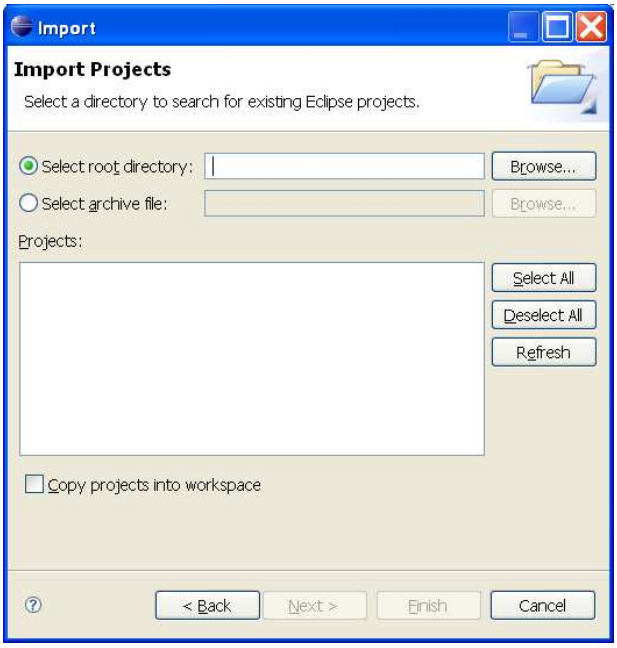
```
<dictionary>
	<key>org.eclipse.cdt.make.core.build.location</key>
	<value>\myApp</value>
</dictionary>
```

After these two steps, you can import your application into Eclipse.

Choose File → Import to open the Import dialog.



Choose “Existing projects into workspace”, and click on “Next”.



Click on Browse…, and select your application directory, i.e. “myApp”. After doing do, your application project should appear in the projects list of the above dialog. If it doesn’t, typical reasons are

- You chose a project name in the .project file that is already in use
- The .project file contains non-valid XML

Be sure not to check “Copy projects into workspace” in the above dialog. Finish the import by clicking on “Finish”.

As a result, you will find your Application project in the “Project Explorer”

3. Writing an Application

In your newly imported project, open the .cpp file in the src folder of the project. The application source code file opens.

The minimum that an application file must contain is an application factory. It is invoked by the operation system when the node is starting up.

```
isense::Application* application_factory(isense::Os& os)
{
	return new MyApplication(os);
}
```

It can be used to hand the custom application to the OS. Your custom application must be derived from `isense::Application` defined in `<isense/application.h>`, where it can inherit two key functions:

```
class MyApplication :
	public isense::Application
{
public:
	///From isense::Application
	virtual void boot (void);

};


//------------------------------------------------------------------
void
	MyApplication::
	boot(void)
	{
	  os().debug("App::boot");
	  os().allow_sleep(false);
	}
```

The method

```
virtual void boot (void);
```

is called by the OS when the node is booting up, after the application factory was invoked. In the above example, debug output written (see next section). To do so, a method of the OS is invoked. Access to the OS is done via

```
Os& os();
```

that return a reference to the OS. For a complete documentation of the OS API, please consult your doxygen docs. For the use of `allow_sleep(bool)`, please refer to the power management section.

3.1.Debug output (via the UART with attached iSense GatewayModule)

Note that you need to have enabled the UART to use the features described within this section. To enable the UART, ensure that

```
#define ISENSE_ENABLE_UART
```

is uncommented in your `config.h` (if you compile the iSense firmware manually), or check the option "iSense Configuration→Peripherals→UART (Serial I/O)" for your platform on the iSense web compile interface.

You can send character output to the outside world using the methods `Os::debug` and `Os::fatal`:

```
//-------------------------------------------------------
** Logs the given string depending on the log mode to a uart or
 *  radio.
 */
void debug( const char *format, ...);

//-------------------------------------------------------
/** Logs the given string depending on the log mode to a uart or
 *  radio.
 */
void fatal( const char *format, ...);
```

They work similar to the well-known sprintf functions in regular C, i.e. integer values etc. can be printed using the % notation.

For example

```
uint16 i = 128;
os().debug("value of i is %d, hex=%x", i, i);
```

will output „value of i is 128, hex=0x80".

The destination of the debug output can be set using `Os::set_log_mode`, the corresponding constants can be found in iSense/src/isense/os.h.

```
//-------------------------------------------------------
/** Sets the log mode to the given value, e.g.
 * (ISENSE_LOG_MODE_UART0 | ISENSE_LOG_MODE_RADIO)
 */
void set_log_mode( uint8 mode ) {log_mode_ = mode; }
```

On the iSense platform, the debug and fatal output default to UART 0, so they can be observed in iShell if the sensor node is connected to a PC using a gateway module.

3.2.Sending and receiving with neighboring devices

To transmit packets, you must use the current instance `isense::Radio` class defined in `<isense/radio.h>`. A reference to it can be obtained by calling `Os::radio()`. The `send`-method of the `isense::Radio` requires the destination address, the length of the buffer to be transmitted, the buffer, options.

```
//------------------------------------------------------------------------
/** Sends out data over the radio. Actual implementation may vary,
 * so also check at the actual implementations.
 *
 * \param dest_addr Destination address of the data
 * \param len Length of the payload on octets
 * \param buf pointer to the payload. Copied by the radio.
 * \param options Send options. For possible values see #tx_options
 *  and check actual implementations for details
 * \param sender Pointer to an object that want to receive a
 *  confirmation after sending. If no conf is desired, pass NULL.
 */
virtual void send(      const ISENSE_RADIO_ADDR_TYPE dest_addr,
                        link_layer_length_t len,
                        const uint8 * buf,
                        uint8 options,
                        Sender* sender );
```

The destination address is a 16 or 64 bit value (depending on whether you use 16bit short addressing or 64 bit extended addressing, note that if not configured differently, the default address length will be 64Bit) indicating the intended receiver of the packet. If you specify `ISENSE_RADIO_BROADCAST_ADDR`, this packet is received by any other device in the broadcasting range.

The length parameter indicates the buffer length.

Amongst others, the options tell the radio to perform an acknowledged or unacknowledged transmission. To use the default options, specify `LinkLayerInterface::ISENSE_RADIO_TX_OPTION_NONE`, if you want to send with an acknowledgement (only possible if not broadcasting), specify `LinkLayerInterface::ISENSE_RADIO_TX_OPTION_ACK`.

The sender parameter may be `NULL` or point to a class implementing the `isense::Sender interface`.

```
//------------------------------------------------------------------------
/** Interface for getting a sent confirmation after sending radio
 * packets. Objects that send radio packets and want to be notified
 * about the sending state must implement this interface and hand
 * over a pointer to themselves when calling the send_down method of the
 * radio. After the packet was sent or after the sending finally failed,
```

```
 * confirm is called.
 */
class Sender
{
 public:
   virtual ~Sender(){}
   //-----------------------------------------------------------------
   /** Indicates the state of send_down call. confirm is called after the
    * sending succeeded or finally failed.
    * \param state Indicates whether the sending was successful or not.
    *         Refer to #radio_state above for possible values.
    *
    * \param tries Indicates how often the radio tried to deliver the pkt.
    * \param time Time when the packet was sent out
    */
   virtual void confirm (uint8 state, uint8 tries, Time time)= 0;
};
```

If this parameter is not `NULL`, the radio will notify the sender about success or failure by calling its `confirm` method (in the interrupt context). In case of unacknowledged transmission, it is called as soon as the packet has been send by the radio. If you have requested an ack, it is called as soon as the acknowledgement arrived or after n failed tries. The time parameter indicates the time of transmission end.

A typical simple call to send without a confirmation might look like this:

```
uint8 outbuffer[10]
outbuffer[0] = MY_RADIO_PACKET_TYPE;
os().radio().send(ISENSE_RADIO_BROADCAST_ADDR, 10, outbuffer,
Radio::ISENSE_RADIO_HEADER_OPTION_NONE, NULL );
```

To receive packets, your application that should receive packets must implement the `isense::Receiver` interface. It can be found in `<isense/link_layer_interface.h>` and looks as follows:

```
//-----------------------------------------------------------------------
/** Interface for receiving radio packets. When the radio interface
 * receives a packet, the receive method is called. All different
 * radio implementations provide one or multiple functions to register
 *  receiver objects.
 *
 */
class Receiver : public iSenseObject
  {
  public:
    virtual ~Receiver(){}
    //-----------------------------------------------------------------
    /** This function is called by the radio where the Receiver is
     * registered when a packet is received.
     *
     *  \param len length of the packet to enqueue
     *  \param buf pointer to the payload of the packet (payload is copied)
     *  \param src_addr sender address of the packet
```

```
     *  \param dest_addr destination address of the packet
     *  \param signal_strength signal strength indication for the packet
     *  \param signal_quality signal quality indication for the packet
     *  \param sequence_no link layer single hop sequence number
     *  \param interface id of the interface that received the packet
     *  \param rx_time time of reception of that packet
     */
    virtual void receive (    link_layer_length_t len,
                              const uint8 * buf,
                              ISENSE_RADIO_ADDR_TYPE src_addr,
                              ISENSE_RADIO_ADDR_TYPE dest_addr,
                              uint16 signal_strength,
                              uint16 signal_quality,
                              uint8 sequence_no,
                              uint8 interface
                              Time rx_time);
  };
```

Hence, first of all each class that wants to receive data must implement the receive method. It is called whenever a packet is received. It is passed the payload length, the buffer with the payload, source and destination address, and, besides other information, the id of the interface that received the packet.

As a convention, the first byte of the payload must contain the packet type information. By comparing this byte with the type you are interested in, you can determine whether this packet is targeted at this receiver. (As a result, you must place this type information in the first byte when sending). Applications should use packet types between 32 and 63, as types between 0 and 31 are reserved to iSense protocols, and packet types larger than 63 are used for IPv6. For an overview of the iSense packet types currently used, refer to the `LinkLayerInterface::packet_types` enum in `<isense/ link_layer_interface.h>`

By evaluating the interface information, you can find out where the payload came from. Inferface ids less than 128 represent hardware interfaces (currently only "0 = primary radio interface" is used), whiles interface ids greater than 127 represent software interfaces (currently only "0xFE = routing protocol" is used).

Second, you must register this receiver with the `isense::Dispatcher` in order to actually receive packets. This could be done in the application's `boot`-method:

```
os().dispatcher().add_receiver(this);
```

A normal receiver obtains packets addressed to the MAC-address of the device it runs on or the broadcast address. Its *receive*-method is called in the normal context by processing the task-queue. A receiver must filter for its expected packets itself, e.g., by dropping packets for the broadcast address.

3.3.Tasks, Timeouts, and the differences inbetween

Wireless sensor network applications often need to do things periodically, in a certain time interval or at a certain point in time. The iSense software system provides two central mechanisms to address this issue: task and timeouts.

An application can register with the OS to receive a callback when the time has come. This is where the difference between tasks and timeouts becomes important:

- timeout callbacks are called within the interrupt context, and hence are hardly ever delayed, but the callback handler must not take long for execution, because otherwise other interrupt might be affected. Other than task, timeouts can be deregistered again (in case they did not fire yet).
- task callbacks are the regular application context. They cannot interrupt other activities, and hence may be delayed. On the other side, their execution can take an arbitrary time, and is hence suited for longer activities such as complex computations.

Both concepts include the implementation of an interface.

Defined in `<isense/timeout_handler.h>`:

```
class TimeoutHandler :  public iSenseObject
{
  public:
    /**
     * Called in the interrupt. Implement this for fast
     * processing routines.
     */
    virtual void timeout ( void* userdata ) = 0;

};
```

Defined in `<isense/task.h>`:

```
class Task : public iSenseObject
{
  public:
    /**
     * This method is called by the os in normal context.
     */
    virtual void execute( void* userdata ) = 0;
};
```

Once a class (such as your application) inherits these and implements the corresponding method, it is a timeout (or task respectively) and can be registered with the OS for later callback.

As for timeouts, the OS provides three methods for registering timeouts being called at a certain time, after a certain interval from now on and for deregistering such callbacks where the timeout did not fire yet (i.e. their timeout time is still to come).

```
/** Registers a timeout that fired at the specified time.
 *
 * \param interval Point in time when the callback will be called
 * \param toh Timeouthandler, whose timeout method is to be called
 * \param userdata an arbitrary pointer that is passed back again
 * to the callback
 *
 * \return The timeout id (can be used the delete the timeout later),
 *  or TIMEOUT_REGISTER_FAILED if the registration failed.
 */
uint8 add_timeout_at(Time time, TimeoutHandler *toh , void* userdata);
```

```
/** Registers a timeout that fires after a certain time.
 *
 * \param interval Time period after which the callback will be called
 * \param toh Timeouthandler, whose timeout method is to be called
 * \param userdata an arbitrary pointer that is passed back again
 * to the callback
 *
 * \return The timeout id (can be used the delete the timeout later),
 * or TIMEOUT_REGISTER_FAILED if the registration failed.
 */
uint8 add_timeout_in(Time interval, TimeoutHandler *toh , void* userdata);


/** This method removes the timeout that returned id on registration,
 * but only if the corresponding timeout handler euqals t. This
 * implicitly means that only the handler that is registered can
 * remove the timer (or at least its pointer must be known)
 *
 * \param id Timeout id of the timeout to be removed
 * \param t Timeouthandler that would have been called on timeout
 *
 * \return  true if the timer has been removed, false otherwise.
 */
bool remove_timeout(uint8 id, TimeoutHandler* t );
```

For task, the OS provides three methods for registering tasks to be executed as soon as possible, at a certain time, or after a certain interval from now on.

```
//-------------------------------------------------------------------------
/** This method enqueues the given task into the queue of tasks
 *  Its execute method is called during the main control flow
 *
 * \param task object implementing the Task interface, whose execute
 * method is called
 * \param userdata an arbitrary pointer that is passed back again
 * to the callback
 *
 * \return true if task was enqueued successfully, false otherwise (e.g.
 * if the task queue is full)
 *
 */
bool add_task(Task *task, void* userdata);

//-------------------------------------------------------------------------
/** This method enqueues the given task in the queue of lower priority
 *  tasks. Its execute method is called during the main control flow
 *
 * \param interval Point in time when the callback will be called
 * \param task object implementing the Task interface, whose execute
 * method is called
 * \param userdata an arbitrary pointer that is passed back again
 * to the callback
 *
 * \return true if task was enqueued successfully, false otherwise (e.g.
 * if the task queue is full)
 */
```

```
bool add_task_at(Time time, Task *task, void* userdata);

//---------------------------------------------------------------------
/** This method enqueues the given task in the queue of lower priority
 *  tasks. Its execute method is called during the main control flow
 *
 * \param interval Time period after which the callback will be called
 * \param task object implementing the Task interface, whose execute
 * method is called
 * \param userdata an arbitrary pointer that is passed back again
 * to the callback
 *
 * \return true if task was enqueued successfully, false otherwise (e.g.
 * if the task queue is full)
 */
bool add_task_in(Time interval, Task *task, void* userdata);
```

All methods for registering tasks and timeouts at the OS feature a so called `userdata` parameter. It can be used to attach additional information to a task or timeout, that is then passed to the `execute` and `timeout` methods when they are called. If you don't use the feature, simply pass `NULL`.

3.4.Power management: sleeping and waking up again

The iSense software system features an automatic power management system. It manages the different power modes to achieve minimum energy consumption.

Although the system works automatically in the background, the different software components must indicate to the OS which power modes they can tolerate for the current operation.

Three power modes are distinguished:

- full operation: all hardware features are fully functional
- doze mode: all peripherals and interrupt sources are functional, radio packets can be received and sent (if the radio is not turned off manually)
- sleep mode: radio and controller are switched off, but IO interrupt sources, tasks and timeouts are still functional.

The OS holds a counter for sleep and doze mode. If that counters are 0, sleeping or dozing are allowed. Each software component can disallow entering sleep and doze mode by calling the according OS methods that increment or decrement the according counters.

```
//---------------------------------------------------------------------
/** This method increments or decrements a counter for the permission
 * of automatic sleep. If the counter is zero automatic sleep is allowed.
 *
 * \return Returns true if the operation has been successful, false
 * if the variable had already the desired value.
 */
bool allow_sleep(bool allow);

//---------------------------------------------------------------------
/** This method increments or decrements a counter for the permission of
 * automatic doze. If the counter is zero automatic doze is allowed.
```

```
 *
 * \return Returns true if the operation has been successful, false if the
 * variable had already the desired value.
 */
bool allow_doze(bool allow);
```

After boot, sleeping and dozing are allowed by default. If the application wants to prevent sleeping, it must call

```
os().allow_sleep(false);
```

for example in its `boot` method. This prevents the system from sleeping until

```
os().allow_sleep(true);
```

is called again. The same holds for `allow_doze`.

For the application, sleeping and waking is transparent except for the fact that no radio packets can be received when sleeping. The application can operate as usual, as the system automatically wakes up in time when timeouts or tasks are due.

Note that allowing sleeping or dozing does not necessarily mean that the device actually goes to sleep or doze mode, as other software components can also prevent sleeping or dozing if their current operation cannot tolerate sleeping or dozing. An example for that is the TDMA radio that prevents sleeping during the active sending and receiving times (see doce documentation for the TDMA radio).

If your application (or any other object) needs to know about the wake-sleep and sleep-wake transitions, it can inherit the `SleepHandler` interface:

```
class SleepHandler :
      public iSenseObject
{
public:
      virtual ~SleepHandler(){}


      /** This method is called before the device goes to sleep.
       * In contrast to hibernate, it indicates the that memory
       * will be retained during sleep. Derived classes can do
       * here whatever is required before sleep. But: BE QUICK.
       *
       * Note, that tasks and timeouts registered here are NOT
       * called before sleep.
       *
       *  \return true if application aggrees to going to sleep
       * mode, false otherwise
       */
      virtual bool stand_by (void) = 0; // Memory held
```

```
        /** This method is called by the os after wake up from sleep.
         * Derived classes can use it e.g. deserialize themselves from
         * non-volatile memory in case system memory was not retained
         * during sleep. However, note that this currently never happens.
         *
         * It is necessary that your application does not take longer
         * than 1ms/SLEEP_HANDLER_MAX_COUNT.
         *
         *  \param memory_held Indicates whether the memory was retained
         * during sleep.
         */
        virtual void wake_up (bool memory_held) = 0;

};
```

The application (or any other object) can than be registered for being called with the OS,

```
os().add_sleep_handler(sh);
```

where `sh` must inherit from the SleepHandler interface. `stand_by` is then called when the device goes to sleep, whereas `wake_up` is called when the device wakes up. Note that the calls to `stand_by` and `wake_up` delay the process of actually going to sleep or waking up, so you should not do things that take long (such as writing to an UART or the I2C bus) within. Also not that tasks and timeouts added to the OS from theses methods will not be service before sleep.

Currently, hibernate is never called, as no device so far supports going to sleep without the memory being held.

3.5. Using routing protocols

This section shows how to use the routing protocols. Here, the bidirectional quality routing is used to show how to use routing protocols exemplarily.

Note that you need to have enabled the bidirectional quality routing to use the features described within this section. To enable the bidirection quality routing, ensure that

```
#define ISENSE_ENABLE_NEIGHBORHOOD_MONITOR
#define ISENSE_ENABLE_QUALITY_ROUTING
```

are uncommented in your `config.h` (if you compile the iSense firmware manually), or check the options “iSense Configuration→Functionality→Neighborhood Monitoring“ and “iSense Configuration→Functionality→Quality Unicast Bidirectional Routing Protocol“ for your platform on the iSense web compile interface.

In order to use a routing protocol, it must be instantiated manually. First, create a member variable in your application (or any other object) definition:

```
...

#include <isense/protocols/routing/neighborhood_monitor.h>
#include <isense/ protocols/routing/bidi_quality_routing.h>

...

class MyApplication :
      public isense::Application
{

...

private:
      BidirectionalQualityRouting* q_;

};
```

You should then initialize it in the constructor:

```
//-----------------------------------------------------------------------
void
      MyApplication::
      MyApplication(void) : q_(NULL)
      {
            ...
      }
```

You then have to instantiate the routing protocol, for example in the boot method:

```
//-----------------------------------------------------------------------
void
      MyApplication::
      boot(void)
{
      os().debug("App::boot");

      //prevent sleeping to be ready to receive at all times
      os().allow_sleep(false);

      //instantiate the neighorhood monitor
      NeighborhoodMonitor* n = new NeighborhoodMonitor(os());

      //instantiate the routing protocol
      q_= new BidirectionalQualityRouting(os(),n);

      //register application at the dispatcher
      os().dispatcher().add_receiver(this);
```

```
}
```

The bidirectional quality routing can use the neighborhood monitor to use especially such links that exhibit a low packet drop rate. Hence, the above example instantiates the neighborhood monitor (which automatically registers itself as a promiscuous receiver at the dispatcher). This step is not required for all routing protocols, and is also optional for use with the bidirectional quality routing.

Next, the bidirectional quality routing is instantiated. As all routing protocols register themselves at the dispatcher automatically, it is not required to register the protocol manually.

Finally, the application is registered at the dispatcher to allow it to receive radio data.

Note that all devices in the network must have instantiated the routing protocol for the routing to work.

To send data to a device, a route must be created first:

```
ISENSE_RADIO_ADDR_TYPE destination_addr = 0x158D0000181234ULL;
q_->create_route(destination_addr);
```

Be sure to append `ULL` to the literal specifying the address in order to instruct the compiler to interpret the specified numeric value as an unsigned 64 bit value. To determine the addresses of your devices, you can use the “Flash Programmer” plugin of the iShell tool [2].

To detect the presence of neighboring devices in your wireless network, you can query the neighborhood monitor’s `get_neighbors` method:

```
//------------------------------------------------------------------------
/**
 * Returns an array of neighbor structs with the requested information on all
 * neighbors. The array has a size of ISENSE_MAX_NEIGHBORHOOD_SIZE.
 * Depending on the actual number of neighbors, only the first few structs contain
 * valid data. They can be recognized by their id (which differs from 0xFFFF). All
 * empty structs have the addr field set to 0xFFFF
 *
 * \param prop a member of the neighbor_property enum to choose
 *        what information to retrieve
 * \return an array of neighbor structs.
 *
 * \note Memory for storing the struct is allocated by get_neighbors, and
 *       must be freed by the caller.
 */

     neighbor* get_neighbors( neighbor_property prop);
```

It returns an array of `ISENSE_MAX_NEIGHBORHOOD_SIZE neighbor structs`:

```
//------------------------------------------------------------------------
/**
 * Struct for returning information about the neighbor whose ID is addr. Value
 * contains whatever information was selection by the neighbor_property
 */

  typedef struct {
      ISENSE_RADIO_ADDR_TYPE addr; /**< address of the neighbor */
      uint8 value; /**< requested value for that neighbor */
  } neighbor;
```

Be aware that to detect the presence of devices, the neighborhood monitor requires that the neighbors already sent data to the device where the neighborhood monitor is running.

If you desire to automatically detect the presence of devices over multiple hops in your wireless network, you will need a separate mechanism for that, e.g. devices flooding a packet to all nodes in the network. Such a mechanism is out of the scope of the document and not described here.

Note that the route creation can fail (i.e. if your network is partitioned). The method

```
virtual bool route_available(ISENSE_RADIO_ADDR_TYPE dest );
```

which is part of the routing interface (and hence is available to each routing protocol) can be used to check whether a route to a node is available, i.e. if the route creation process was successful. However, keep in mind that creating a route can take a while, and call it a reasonable time after creating the route. Some routing protocols also provide the possibility to register a hander that is called when route creation succeeds or fails. Refer to the code documentation of the used routing protocol for details.

If the route creation was successful, data can be sent to the node:

```
ISENSE_RADIO_ADDR_TYPE destination_addr = 0x158D0000181234ULL;

//initialize payload buffer
uint8 buf[50];

//set packet type the custom type (200),
//allowed values between 128 and 255
buf[0] = 200;

//initialize payload buffer
for (uint8 i = 1; i < 50; i++)
      buf[i] = i;

os_.debug("sending data to %x", destination_addr);

//send playload
q_->send(destination_addr, 50, buf);
```

On the destination device, the payload can be received by adding code to the `receive` method of the application. Note that the corresponding application must be registered at the dispatcher.

```
void
MyApplication::
receive(    link_layer_length_t len,
            const uint8 * buf,
            ISENSE_RADIO_ADDR_TYPE src_addr,
            ISENSE_RADIO_ADDR_TYPE dest_addr,
            uint16 signal_strength,
            uint16 signal_quality,
            uint8 seq_no,
            uint8 interface,
            Time rx_time)
{
      // check packet type
      if (buf[0]== 200)
      {
            os().debug("received %d bytes, seq_no %d from %x to %x",
                        len, seq_no, src_addr, dest_addr);
      }
}
```

Note that the actual routing packet (consisting of the payload prefaced by the routing header) is not only passed to the routing protocols `receive` method, but also to the applications `receive` method. Hence, only the type check prevents the application from dealing with that routing packet. When the routing protocol receives a routing packet destined to the node it runs on, it unpacks the payload, and resubmits the payload to the dispatcher. The dispatcher then delivers the payload the all registered receivers, including the application.

3.6. Using time synchronization

This section shows how to use the time synchronization protocol.

Note that you need to have enabled the time synchronization to use the features described within this section. To enable the time synchronization, ensure that

```
#define ISENSE_ENABLE_TIME_SYNC
```

is uncommented in your `config.h` (if you compile the iSense firmware manually), or check the options “iSense Configuration→Functionality→ Time Synchronisation“ for your platform on the iSense web compile interface.

In order to use it, it must be instantiated manually. First, create a member variable in your application definition (or any other object) and initialize it in the constructor:

```
...

#include <isense/protocols/time_sync/confirm_time.h>
...

class MyApplication :
      public isense::Application
```

```
{

...

private:
      TimeSync* ts_;
...

};

//------------------------------------------------------------------------
void
      MyApplication::
      MyApplication(void) : ts_(NULL)
      {
            ...
      }
```

You then have to instantiate the time synchronization protocol, for example in the boot method:

```
//------------------------------------------------------------------------
void
      MyApplication::
      boot(void)
{
      os().debug("App::boot");

      //prevent sleeping to be ready to receive at all times
      os().allow_sleep(false);

      //instantiate the time synchronization protocol
      ts_ = new TimeSync(os());

      //register application at the dispatcher
      os_.dispatcher().add_receiver(this);
}
```

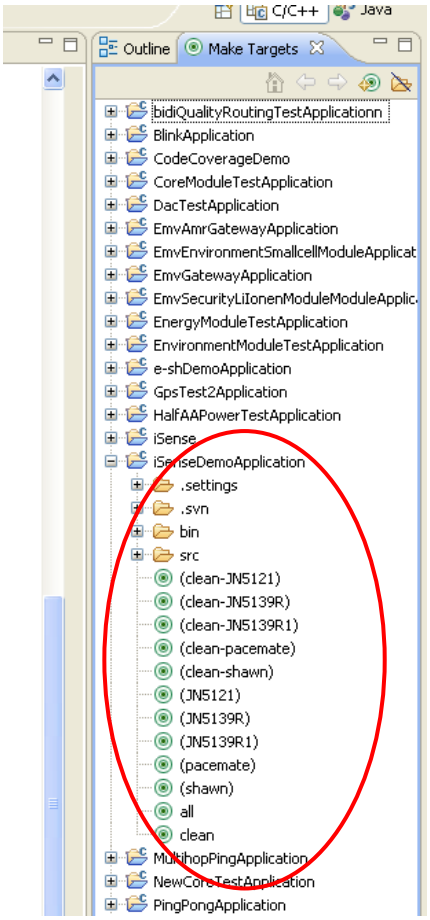
To trigger the time synchronization process, call:

```
      //trigger the time synchronization process
      ts_->launch_time_sync();
```

Note that all devices in the network must have the time synchronization protocol instantiated for the synchronization process to work.

4. Compiling applications

For each application imported into Eclipse, the “Make Target” view contains a number of make target definitions for different platforms.



For the iSense devices, you have to compile your application for “JN5139R” or “JN5139R1”, depending on the chip revision on your iSense CoreModule. Each CoreModule has a sticker on the metal shielding showing the employed chip type. If it shows the type number “JN5139-001-M01R”, you will need the “JN5139R” target, the number “JN5139-001-M01R1”, you will have to compile for the “JN5139R1” target.

Double clicking the correct target will launch the make process. If no errors occur, the output in the console view should look similar to this:

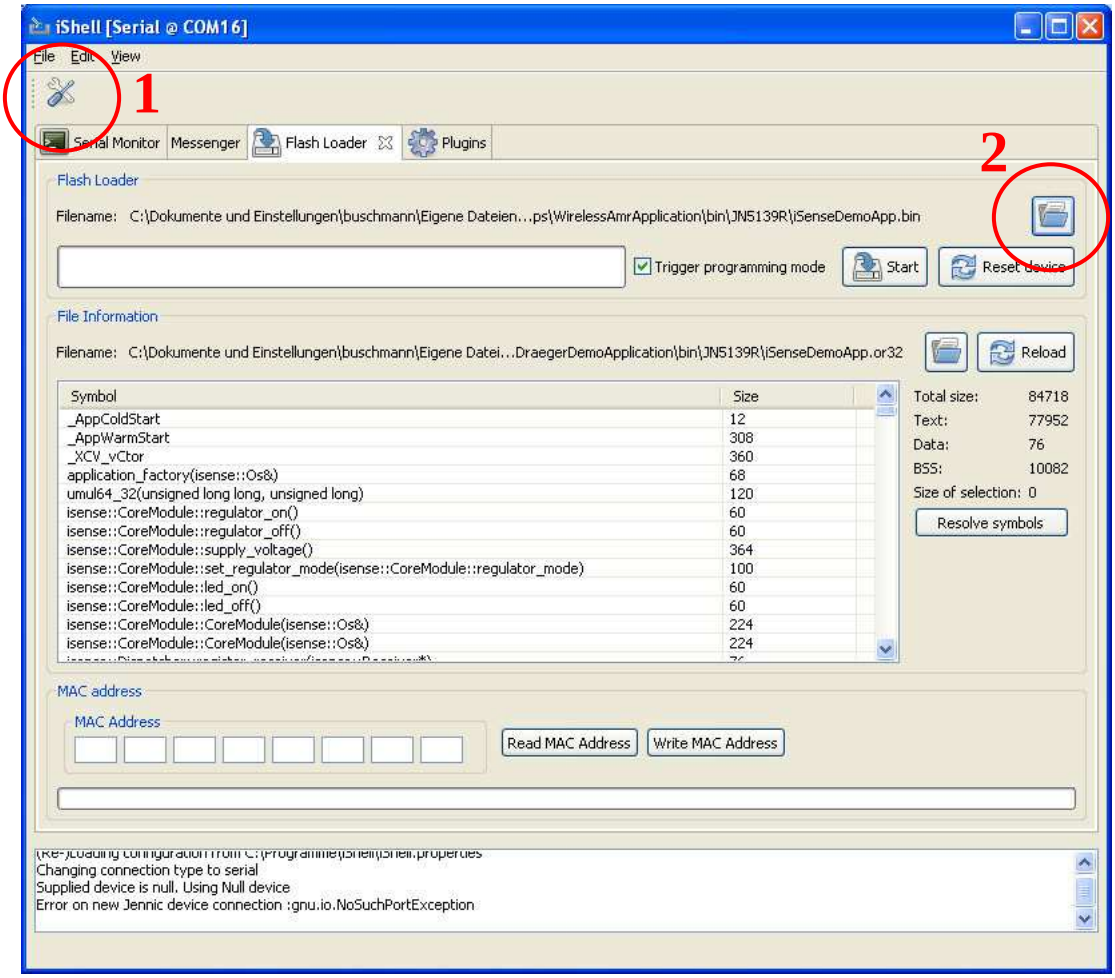
```
make -k JN5139R
----- Building for JENNIC 5139R target ------
----- Compiling (Jennic JN513xR) src/iSenseDemoApp.cpp
----- Linking to bin/JN5139R/iSenseDemoApp.or32
----- Copying to or32 bin/JN5139R/iSenseDemoApp.or32
   text     data  bss   dec   hex   filename
  68880     100   9520  78500 132a4 bin/JN5139R/iSenseDemoApp.or32
----- Build finished
```

```
----- Done  ------
```

The major outcome is a .bin file, which can be fould in `/iApps/myApp/bin/JN5139R` or `/iApps/myApp/bin/JN5139R1`, depending on the make target. This file must then be flashed onto the iSense device.

5. *Flashing and Using iShell for Viewing Debug Info*

Connect the iSense CoreModule to be flashed to an iSense GatewayModule, and plug the USB or serial cable to your computer. Now, start iShell, select the correct serial port by clicking on the options symbol (1) and change to the "Flash Loader" tab (see below).



Choose the .bin file to be flashed by clicking on the file chooser button in the "Flash Loader" part of the tab (2). Then click on "Start", and the flashing will start. Make sure that the "Trigger programming mode" option is checked. The progress bar indicates the state of the flashing process. While flashing, the "Start" button will change to "Cancel". You can abort the flashing process by clicking on "Cancel". However, note that the program on the connected node will be corrupted if the flash process is aborted once the flash process started.

After the flashing completed, change to the "Serial Monitor" tab. If the log mode is set to UART0 in your application software (which is the OS default, so you don't need to do anything to have it set like this), the CoreModule is connected to the PC via a iSense Gateway, and your application's `boot` method contains an `os().debug(…)` statement, you should see the output in the serial monitor.

If you see additional cryptic character in the window, change the mode to packet mode by choosing "View → Packet Mode".

6. Writing your own classes

If you need more functionality than available through using iSense firmware classes, you can easily extend your code by writing your own classes.

As usually (but not necessarily) the code of each class resides in a separate file, you should create a *.cpp and a *.h file in the src folder of your application.

Do not forget to add the new file to the list of files to be compiled. This is done by editing the Makefile.properties file in your application folder: add your *.cpp file to the line that starts with `BLDSRCS =`, the different files in that line must be separated by a white space.

To achieve a proper simulation of the heap allocations when running iSense applications in the Shawn simulator, you must derive your class from `iSenseObject` (if it is not derived from any other iSense class such as `Task` or so).

7. Pitfalls, Memory Considerations and Exceptions

Albeit iSense supports dynamic memory allocation similar to a PC, a few limitations must be kept in mind.

First of all, the size of the heap depends on your application's code size, as the iSense Core Modules feature a shared memory for code and data. The bigger your application binary is, the smaller is the heap and vice versa. If the heap is full, `malloc` and `new` will return `NULL`.

Currently, the maximum heap size is 128kB. Consequently, the maximum size of an allocated memory block is also 128kB.

In addition, certain alignment requirements must be considered: each variable must be aligned according to its size, but at most to 32 bits. This means that 8 bit variables such as `int8`, `uint8` and `char` can be located at an arbitrary address, whereas 16 bit variables such as `uint16` must be located at even addresses and larger variables such as `uint32` or `uint64` but also `structs` and `classes` must be located at addresses divisible by 4 without remainder.

The compiler as well as the dynamic memory will automatically ensure proper alignment. If required for proper alignment, it will automatically insert so called padding between two variables. Consider the example below:

```
typedef struct {
      uint16 var1;
      // compiler insert 2 bytes of padding here to align var2 correctly
      uint32 var2;
      uint16 var3;
      //compiler inserts 2 bytes of padding to align next vars correctly
} mystruct1;

typedef struct {
      uint32 var2;
      uint16 var1; // no padding here
      uint16 var3;
} mystruct2;

os().debug("size of mystruct1: %d size of mystruct2: %d",
      sizeof(mystruct1), sizeof(mystruct2));
```

The according output looks as follows:

```
size of mystruct1: 12 size of mystruct2: 8
```

This shows that the compiler adds 2 bytes of padding after `var1` and `var3` in `mystruct1`, but not in `mystruct2`. In other words, by sorting the variables from big to small in `mystruct2`, 33% of the size of `mystruct1` could be saved without any loss of functionality.

Note that arrays feature the alignment of their base types, i.e. a `uint8` array will not be particularly aligned, a `uint16` array will be two-aligned and so on.

However, the compiler's automatic alignment can be overruled by type casts. Consider the example below:

```
uint8 buf[8];
mystruct1* ptr;
ptr = (mystruct1*)buf;
ptr->var1 = 5;
```

In line 3, the `uint8` array's address is assigned to `ptr`, which is of type `mystruct1*`. Consequently, `ptr->var1` can be accessed in line 4. However, remember that `uint8` arrays are not aligned in a particular way. At run time, in line 8, the compiler will calculate the address of `var1` as `ptr`+4, and as `ptr` is unaligned just as `buf` was, so is `ptr`+4. Here, the code will try to access a 32 bit variable that might be not 4-aligned.

As a result, the CPU will issue an alignment exception, output an error message (if `ISENSE_ENABLE_FATAL_DEBUG` is enabled), and reset the Core Module. Note that probably this error would not always occur, as `buf` might be 4-aligned by chance, which makes such bugs hard to hunt.

Some help is offered by the exception's fatal error message (if enabled): it states the reason for the reset, as well as the program counter (PC) and the address that the CPU tried to access:

```
JPM: unaligned mem access
addr=0x4006841 PC=0x4004cbf caller=0x400586e
```

In the above, `addr` indicates the accessed memory address, `PC` indicated the address of the code that attempted that access, and `caller` points to the code that called the method in which the PC address lied. Here, it can be tracked that the instruction at 0x4004cbf in the memory tried to access the memory address 0x4006841, and that the calling address was 0x400586e. The last two addresses can help you to find the methods in which the error occurred: You can trace the memory location of all methods in your code by browsing the linker map file Map.txt (which is located next to the .bin file)

Below, a typical excerpt of a linker map file is shown: besides other stuff, it lists the starting address of all methods.

```
.text._ZN26MyApplication4bootEv
                0x04004cb2      0x150 bin/JN5148/MyApplication.o
                0x04004cb2                MyApplication::boot()
.text._ZN26 MyApplication7executeEPv
                0x04004fd0      0x131 bin/JN5148/MyApplication.o
                0x04004fd0                MyApplication::execute(void*)
```

Here, the application's `boot()` method is located at 0x4004cb2, followed by the `execute(void*)` method starting at 0x400d3f0. Hence, the address from the error method

(0x400fcbf) is located within the `boot()` method (0x4004cb2 < 0x4004cbf < 0x400d3f0) which in turn means that the code that caused the exception can be found in `boot()`.

The alignment exception is just one in a row of other exceptions that can occur, the most important ones are listed and explained below:

Error message	*Description*
`JPM: bus error`	The code tried to access a memory address that does not point to a valid memory location. This usually indicates that a `NULL` pointer or an uninitialized pointer was dereferenced.
`JPM: stack_overflow`	The stack (which grows downwards) got too big and exceeded the defined lower bound. This usually happens if the stack is screwed, or if a recursive function does not terminate. Note that this exception occurs only on CM20X/CM30X Core Modules with a JN5148 CPU, as the JN5139 CPU on the first generation Core Modules (CM10X) does not support a lower stack bound.
`JPM: illegal instr`	The CPU loaded an instruction that it cannot decode (i.e. that is unkown or of invalid format). This usually happens if code was overwritten with data, e.g. if data is written to an invalid pointer address.
`JPM: unaligned mem access`	The code tried to access a variable that is not properly aligned. This usually happens if an uninitialized pointer is dereferenced or if an unproper cast was done (see above).

Also, you should be aware that variables are not automatically initialized (except for global objects, of which contructors are called upon boot). Hence, it is the programmer's task to initialize all variables, including all member variables of classes, before using them.

Finally, be aware that the compiler does not support dynamically sized arrays on the allocated on the stack. Consequently, the following code will not work:

```
uint8 size = 5;
    :

    :
uint8 buffer[size]; // <-- will not work properly
```

Instead, arrays allocated on the stack must have a static size fixed at compile time.

```
uint8 buffer[5]; // <-- will work properly
```

If you require dynamically sized arrays, you should instead allocate them dynamically on the heap.

8. References

[1] coalesenses Development Environment Setup User Guide, online available at http://www.coalesenses.com/index.php?page=development-environent

[2] coalesenses iShell User Guide, online available at http://www.coalesenses.com/download/UG_ishell_1v4.pdf

coalesenses GmbH
Maria-Goeppert-Str.
23562 Lübeck
Germany

www.coalesenses.com
sales@coalesenses.com