SensorOS: Programming Manual

Robert Moore

http://www.mooredesign.com.au/sensoros/

October 29, 2009

Version: 0.9

Abstract

This document provides an overview of the structure of the SensorOS operating system and the information needed to program for it in the context of both application and platform development. This document assumes that the reader has knowledge in C++ programming, the OSI networking model and basic operating systems knowledge (in particular scheduling, interrupts, dynamic memory allocation and race conditions).

Contents

List of Code Examples

List of Figures

List of Tables

1 Introduction

1.1 Background

SensorOS is an event-driven, FIFO scheduled embedded operating system (OS) designed for use with sensor networks. It is a simple OS that is suitable for using as an example for teaching OS concepts or for any application where a minimalist operating system is required for an embedded event-driven environment. SensorOS is based on the TinyOS[1] operating system, but is implemented in C++ and has a strong focus on portability and compatibility between development environments.

SensorOS exhibits the following properties:

- Minimalist
- Low overhead
- Portable
- Supports networking: including a well defined interface between the data link layer and network layer and explicit support for multiple data link layers
- Power aware
- Excellent documentation (to reduce learning / time overhead)
- Encompasses many of the useful patterns, ideas and abstractions present in TinyOS
- A focus on making it easy to port between platforms (through the design, code structure and documentation of the operating system)
- A focus on compatibility with a wide range of development environments (both in terms of hardware platforms and compilers)

1.2 When to use SensorOS

If you are working on an event-driven system and you want a simple operating system with a FIFO scheduler then SensorOS is a good choice. It is also a good choice if you are teaching embedded operating systems and you want a simplistic OS to illustrate various OS concepts (such as scheduling, networking and communication, race conditions etc.).

SensorOS was designed for use in a sensor network environment, and as such it is also suited for such applications. If your development environment is GCC on Linux then it is recommended that you consider using TinyOS (particularly if your platform or processor has been ported to TinyOS already) because there is a large community of developers working on it and a large base of existing applications and add-ons that you can leverage. If you don't want to spend the effort porting your platform to TinyOS, or you are working with a development environment not supported by TinyOS (but is supported by SensorOS, see section 1.5) then SensorOS is a good choice (even if you have to port your system to SensorOS, this document should ensure that it is much easier than porting to TinyOS).

[1] See http://www.tinyos.net/

1.3 SensorOS Structure

The SensorOS structure consists of a number of core OS files that define the basic SensorOS functionality, a number of platforms each exposing a particular hardware platform (or platforms if they are similar and can be exposed in the same code with conditional compilation) and a number of applications that each perform some function using SensorOS. Each application can be targeted at a single hardware platform, or can be written to be platform-independent. Section 2.4.3 gives more information.

The directory structure of SensorOS is as follows:

- **/** - The root directory contains the other SensorOS directories and a Makefile for compilation with GCC with development environments that have access to make (see section 1.6.2)
  - **/os/** - The os directory contains all the core SensorOS files, such as the classes described in section 2.3
  - **/platforms/** - The platforms directory contains a subdirectory for each platform SensorOS has been ported to, with the platform-specific files contained in that directory
  - **/applications/** - The applications directory contains a subdirectory for each application you develop with SensorOS, with the source files defined for each application within each subdirectory
  - **/unit_tests/** - The unit_tests directory contains a subdirectory for each platform that has unit tests within which, each unit test is contained inside a separate subdirectory

Appendix A shows all the files and directories that are currently contained within SensorOS.

In order to compile a SensorOS application, you need to set the include directories for your compiler (usually using the -I flag) to include the os directory, as well as the platform directory for the platform you are using and the application directory for the application you are compiling. There is a Makefile defined in the root SensorOS directory that automatically does this for you if you are using GCC.

This structure is very similar to TinyOS but there isn't the idea of "chips" that can be utilised by multiple platforms. If this sort of functionality is required then it is easy enough to simply extend the include directories available to the compiler to include the directory for any common code that is created (see section 1.6 for more information about the compilation process for SensorOS).

1.4 Coding and Naming Conventions

A set of coding and naming conventions have been adopted for SensorOS to ensure that all code is consistent, maximising readability and maintainability of code. The SensorOS coding conventions are based on the TinyOS coding conventions outlined by Yannopoulos and Gay (2006).

1.4.1 Names

- Directory names are lower case
- Header files have a .h extension, C++ source code files have a .cpp extension and any

assembly language files should have an extension appropriate for the compiler the code is written for

- File names are lower case with underscores to delimit words; the filename should be the same name as the class it defines
- Class names are camel case, starting with an upper case letter
- Basic types are lower case, with underscores delimiting words and end in '_t'
- Constants are all upper case, with underscores delimiting words
- Variables, functions and methods are camel case, starting with a lower case letter
- Primitive routines are lowercase with underscores delimiting words and are prefixed with two underscores (e.g. `__enable_interrupts()`)

To illustrate these naming concepts, code listings 1 and 2 give example header and source code files.

```
1  /**
2   * Example Class definition file
3   *
4   * The example class prints out hello world n number
5   *  of times when the helloWorld() method is called.
6   * This class illustrates various points from the
7   *  SensorOS coding conventions
8   *
9   * @author Robert Moore <rob@mooredesign.com.au>
10  * @version 1.0.0
11  * @date 2009-08-30
12  */
13
14 #ifndef _EXAMPLE_CLASS_H
15 #define _EXAMPLE_CLASS_H
16
17 class ExampleClass {
18
19     public:
20         ExampleClass(uint8_t numTimes);
21         ~ExampleClass();
22         void helloWorld(void);
23     private:
24         static const char HELLO_WORLD_STRING[15];
25         uint8_t numTimes;
26 };
27
28 #endif
```

Code Listing 1: Example header file (example_class.h)

1.4.2 Scope Delimiters

Scope delimiters have the left curly bracket directly after the block specifier (e.g. if, for, while, function name etc.) followed by a single space; the right curly bracket appears by itself on a line at the same indentation level as the line with the block specifier. All code inside a block is

```
1  /**
2   * Example class implementation file
3   *
4   * @author Robert Moore <rob@mooredesign.com.au>
5   * @version 1.0.0
6   * @date 2009-08-30
7   */
8
9  // Include main.h for SensorOS core files
10 #include <main.h>
11
12 const char ExampleClass::HELLO_WORLD_STRING[] = "Hello World!\r\n";
13
14 /**
15  * Example class constructor
16  *
17  * @param numTimes The number of times to print the
18  *  hello world message when helloWorld() is called
19  */
20 ExampleClass::ExampleClass(uint8_t numTimes) : numTimes(numTimes) {}
21
22 /**
23  * Example class destructor
24  */
25 ExampleClass::~ExampleClass() {}
26
27 /**
28  * Print hello world message numTimes times
29  */
30 void ExampleClass::helloWorld() {
31     uint8_t i;
32     for(i=0; i<numTimes; ++i) {
33         Debug::print(HELLO_WORLD_STRING);
34     }
35 }
```

Code Listing 2: Example source file (example_class.cpp)

indented one level and each indentation level is marked by a single tab character at the start of the line.

The helloWorld() method in code listing 2 gives an example of nested blocks with correct formatting (if inside function).

1.4.3 Primitive Types

SensorOS makes use of a set of typedef'd data types as opposed to the built-in primitive C/C++ types in the interest of making efficient use of memory and improving portability. Each platform will define these typedef's to accomodate the bit size of the primitive types for the compiler being used. Table 1.1 describes each of these types. Each usage of these types should be checked to ensure that the type chosen will be able to always store the possible values that variable can hold (to prevent overflow) and the smallest possible type is used, without breaking the previous constraint (to reduce memory use).

Typedef	Description
bool	Boolean value
uint8_t	8-bit unsigned integer
int8_t	8-bit signed integer
uint16_t	16-bit unsigned integer
int16_t	16-bit signed integer
uint32_t	32-bit unsigned integer
int32_t	32-bit signed integer
uint64_t	64-bit unsigned integer
int64_t	64-bit signed integer
ptrdiff_t	(pointer width)-bit signed integer

Table 1.1: SensorOS primitive data types

1.4.4 Classes, Header files and includes

All classes are to be defined by two files; a header file containing the definition for the class (see code listing 1 for an example) and a source file containing the static variable declarations and function implementations (see code listing 2 for an example).

The header file should begin with a **`#ifndef`** to search for the existence of a symbol with a name that is constructed by:

1. Taking the header file filename
2. Converting it to uppercase
3. Changing the .h extension to `_H`
4. Prefixing it with an underscore

For example, example_class.h in code listing 1 has the symbol as `_EXAMPLE_CLASS_H`. This then needs to be followed by a **`#define`** of the same symbol. Following this is the class definition and any typedefs, macro defines or other artefacts that need to be in the header file, finally the header file should end in a **`#endif`**. The **`#ifndef`**, **`#define`** and **`#endif`** are a standard C/C++ technique to ensure the header file can't be included multiple times.

Typedefs and **`#define`**s should only be placed in the header file if they are needed in files other than the corresponding source file for the class. Otherwise, they should be placed in the source file directly to minimise their scope and minimise the possibility of name conflicts.

The source file should start with a **`#include`**`<main.h>` in order to include the functionality of SensorOS, assuming everything has been set up correctly, this will also give it the definitions contained in the corresponding header file automatically. Whilst this defers from the usual practice of including the header file into the relevant source file and then that header file including other functionality needed by the source file, it avoids include order dependency issues. This is covered further in section 2.4.10.

1.4.5 Commenting

The commenting standard adopted by SensorOS, like in TinyOS is the use of Javadoc[1] style comments. Every file should have a comment at the top giving a one line overview of the file, as well as a description of any important notes, and also the author, date last modified and version number of the file.

All functions and methods should have a comment explaining the function purpose, inputs and outputs in the source file (or the header file for an interface or a template or virtual class).

1.5 Requirements to use SensorOS

In order to use SensorOS you will need:

- A C++ compiler that at the very least supports the EC++ standard as well as templates, the compiler does not need to have the Standard Template Library (STL)
- A hardware platform with a port to SensorOS (currently only the Freescale HCS12 on a Dragon 12 development board), or the ability to port your platform to SensorOS (see section 3)
- An appropriate amount of ROM and RAM; see tables 1.2 and 1.3 for an idea of how much is needed, note:
  - The figures in the tables were generated with no compiler optimisation and as such are gross overestimates
  - The null application is not representative of a typical application, it is a minimalist representation of a possible application
  - These tables show memory usage for the dragon12 platform against the null application (2 Byte pointers, 24 signals)
  - There also needs to be RAM allocation for the stack and the heap
  - Heap use may be able to be determined statically by analysing use of new operator and taking into account the notes in table 1.2 for `SignalRouter`
  - Stack size depends on the application but shouldn't need to be more than a few hundred Bytes

[1] Javadoc is a tool for generating API documentation in Java and relies on a pre-defined commenting syntax in order to annotate classes, functions etc. with extra information, see
http://java.sun.com/j2se/javadoc/

Component	Data	Code	Const	Notes
`List<VoidFunctor>`	0	362	0	—
`main()`	0	75	0	—
`Network`	16	150	14	—
`SignalRouter`	54	336	14	Data memory use dependant on pointer size and number of platform and application signals (see sections 2.4.9 and 3.3.5) Every signal that has a signal handler registered to it will have heap usage (4 Bytes per signal with registered handler(s), 4 Bytes per registered handler)
`Scheduler`	514	378	14	Data memory use dependant on pointer size
`VoidFunctor`	0	6	0	—
`VoidStaticFunctor`	0	36	0	—
`VoidInstanceFunctor<>`	0	0	0	Adds 98 Bytes of code memory per class that is used with it
Total	584	1343	42	—

Table 1.2: Core SensorOS RAM and ROM usage in Bytes

Component	Data	Code	Const	Notes
`Application`	4	32	54	—
`Atomic Section Routines`	0	11	0	—
`CanLink`	0	915	28	There is an extra 55 Bytes of code memory, 6 Bytes of const memory and 29 Bytes of data memory needed per CAN link that is compiled into an application (up to 5 links)
`Debug`	0	185	0	—
`Platform`	0	25	0	—
Total	4	1157	93	—

Table 1.3: Platform (dragon12) and Application (null) SensorOS RAM and ROM usage in Bytes

1.6 Compiling SensorOS

1.6.1 Compilation

As outlined in section 1.3, in order to compile SensorOS you need to set the include directories to the core OS files as well as the platform and application you are using. An example for GCC when compiling for the null application and dragon12 platform is:

```
1  ~/sensoros $ make
2  g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
       sensoros/applications/null/ -c os/main.cpp -o os/main.o
3  g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
       sensoros/applications/null/ -c os/network.cpp -o os/network.o
4  g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
       sensoros/applications/null/ -c os/scheduler.cpp -o os/scheduler.o
5  g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
       sensoros/applications/null/ -c os/signal_router.cpp -o os/signal_router.
       o
6  g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
       sensoros/applications/null/ -c platforms/dragon12/can_link.cpp -o
       platforms/dragon12/can_link.o
7  g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
       sensoros/applications/null/ -c platforms/dragon12/debug.cpp -o platforms
       /dragon12/debug.o
8  g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
       sensoros/applications/null/ -c platforms/dragon12/platform.cpp -o
       platforms/dragon12/platform.o
9  g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
       sensoros/applications/null/ -c applications/null/application.cpp -o
       applications/null/application.o
10 g++  -g -Wall os/main.o os/network.o os/scheduler.o os/signal_router.o
       platforms /dragon12/can_link.o platforms/dragon12/debug.o platforms/
       dragon12/platform.o applications/null/application.o -o application
```

As an example of using an IDE to set up the include directories, if a unit test is created for the dragon12 platform using IAR Embedded Workbench, the steps to set up the project would be:

1. Create a project in a subdirectory within the `sensoros/unit_tests/dragon12/` directory
2. Go to Project > C/C++ Compiler > Preprocessor
3. Add the following to the "Additional include directories" field:

```
1 $PROJ_DIR$\src
2 $PROJ_DIR$\..\..\..\os
3 $PROJ_DIR$\..\..\..\platforms\dragon12
4 $PROJ_DIR$\..\..\..\applications\null
```

4. Add the relevant source files, in the case of the dragon12 platform and the null application that would mean adding:
   - `sensoros/os/main.cpp` (Core)
   - `sensoros/os/network.cpp` (Core)
   - `sensoros/os/scheduler.cpp` (Core)

- `sensoros/os/signal_router.cpp` (Core)
- `sensoros/platforms/dragon12/can_link.cpp` (Platform)
- `sensoros/platforms/dragon12/debug.cpp` (Platform)
- `sensoros/platforms/dragon12/platform.cpp` (Platform)
- `sensoros/platforms/dragon12/iar.s12` (Platform)
- `sensoros/applications/null/application.cpp` (Application)

Since the compilation process uses include paths; platform and application developers must be careful to ensure there isn't filename conflicts otherwise particular files will become shadowed.

1.6.2 Makefile

If your development environment includes make, then you can make use of the Makefile that comes with sensoros. It automatically works out the dependencies and compilation instructions needed. In order to use it you will typically only need to edit it between the following lines of code:

```
1  ###
2  # Edit Below Here
3  ###
4
5  # Set platform
6  PLATFORM = dragon12
7  # Set application
8  APPLICATION = null
9  # Set compiler
10 COMPILER = g++
11 # Set compiler flags
12 FLAGS = -g -Wall
13
14 ###
15 # Don't Edit Below Here
16 ###
```

A description for each variable is given below:

- **PLATFORM**: The name of the directory of the platform you are compiling for
- **APPLICATION**: The name of the directory of the application you are compiling for
- **COMPILER**: The name of the compiler you are using
- **FLAGS**: Any flags you are passing to the compiler

The Makefile is configured to work with a GNU C compiler (e.g. -o, -c and -I flags). If you are using a compiler that has different command line arguments, then you will need to edit the Makefile to change these flags.

In order to use the Makefile you will also need to ensure there is a Makefile inside the directories for the application and platform that you are compiling for. These Makefiles need to contain a variable called `APPLICATION_DEPS` and `PLATFORM_DEPS` respectively. These variables should contain a space delimited list of all the header file dependencies for the application.h and platform.h files respectively.

In addition to these variables, if any of the .cpp files in the application / platform have any dependencies apart from main.h, then these can be added as a space delimited list to a dependency variable for that file. The dependency variable should be named by:

1. Taking the path of the .cpp file relative to the base sensoros/ directory
2. Changing /'s to underscores
3. Removing the .cpp extension
4. Add a suffix of `_DEPS`

Table 1.4 outlines some dependency variable names, for an actual example see `sensoros/platforms/dragon12/Makefile`.

Filename	**Variable Name**
sensoros/os/network.cpp	os_network_DEPS
sensoros/platforms/dragon12/can_link.cpp	platforms_dragon12_can_link_DEPS
sensoros/applications/null/some_class.cpp	applications_null_some_class_DEPS

Table 1.4: Makefile dependencies variable names

1.7 Software License

SensorOS is released under the MIT License:

2 Developing Applications

2.1 Overview

The core classes of the SensorOS operating system are described in the UML Class Diagram are described in Appendix B. The base platform-specific classes required by SensorOS are described in the UML Class Diagram shown in Appendix C. Finally, the base application-specific class is described in the UML Class Diagram shown in Appendix D.

This section is split into five sub sections:

1. A step-by-step guide is given to developing a SensorOS application, which acts as a quick reference guide and contains links to the relevant sections in the rest of the document
2. The base classes as described in the appendices mentioned above will be explained and their APIs revealed
3. A range of important points to know when programming SensorOS applications are discussed
4. A thorough description of how SensorOS applications can perform networking is given
5. A description is given of the yet to be implemented Timer subsystem is given

2.2 Creating a New Application (Step-by-step)

1. The `readme.txt` file for the platform(s) being coded for should be read, in particular the Notes section and License section
2. A sub directory needs to be created for the application inside the `sensoros/applications/` directory (see section 1.3)
3. A `readme.txt` file needs to be created for the application (see section 2.4.2)
4. If an IDE is being used to manage the project then include directories should be set up to point to the `sensoros/os/` directory, the directory of the platform being used and the application directory itself (see section 1.6)
5. If make is being used a Makefile needs to be created (see section 1.6.2)
6. An `application.h` file needs to be created; this file needs to at a minimum define a prototype for the Application class (see section 2.3.10)
7. An `application.cpp` file needs to be created that at a minimum should define the methods for the `Application` class. The following also need to be done in this file (after their dependencies are fulfilled):
   (a) The init method should register some function or method with the `SIG_SYSTEM_BOOTED` signal and that function or method should begin the application processing (see section 2.4.4)
   (b) The init method should register a class that implements `NetworkLayer` (see section 2.3.5) with the `Network` class (see section 2.3.3)
   (c) The init method should call any initialisation methods required (see section 2.4.8), and register signal handlers for any application signals (see sections 2.4.9 and 2.3.2)
8. An `application_signals.h` file needs to be created with a list of the signals the application will use (see section 2.4.9)
9. An `application_network_info.h` file needs to be created with definitions for the `network_address_t` type and the `NODE_ADDRESS` constant (see section 2.5.4) and any platform-targeted networking code mentioned in the platform documentation (see section 2.4.3)
10. A class needs to be created that implements `NetworkLayer` (see section 2.3.5)
11. Classes need to be created to encapsulate the application code (including sensor interfacing) and higher networking layers (see sections 2.5.8 and 2.4.1)
12. The `readme.txt` file and `Makefile` file should be kept up to date

2.3 Core Classes and Interfaces

2.3.1 Scheduler

Overview The scheduler in SensorOS is based on the scheduler in TinyOS; it is a basic, non-preemptive FIFO scheduler of "tasks" that have no parameters and return nothing (i.e. a `void functionName(void)` function or method). In fact, a task is represented by the `task_t` type, which is defined as `typedef VoidFunctor const * task_t` (see section 2.3.6). The Scheduler implements the Singleton creational design pattern (Gamma et al. 1995), so you can use the code as outlined in code listing 3 to get a reference to the scheduler.

```
1 Scheduler* scheduler = Scheduler::getInstance();
```

Code Listing 3: Getting a reference to the scheduler

As in TinyOS, once the scheduler has exhausted its task list it will call `Platform::sleep()` to put the processor into a low power state to conserve power use. This can be very important in sensor networks, because nodes will often be powered by limited power sources and it may be difficult or impossible to renew depleted power sources or gain new power sources, in some applications, sensor nodes are deployed and then left there until they die (run out of power). (Akyildiz et al. 2002; Qi et al. 2001; Sinopoli et al. 2003; Xu 2002)

The source code for this class is located in sensoros/os/scheduler.cpp and sensoros/os/scheduler.h as shown in Appendix A.

Posting a task To post a task to the scheduler you simply need to create a functor representing the function / method you would like called, and then post that functor to be run by the scheduler. Code listings 4, 5 and 6 give examples of how to post a static function, static method and instance method respectively. Note that the code listings show the code to statically allocate the functors, which is the recommended way of using them, as outlined in section 2.4.8. Alternatively, assuming your development environment supports it and you have weighed up the risks, you could also use dynamic allocation to the same effect.

```
1 void myTask(void);
2 static const VoidStaticFunctor myTaskFunctor(&myTask);
3 ...
4 scheduler->postTask(&myTaskFunctor);
```

Code Listing 4: Posting a static function to the scheduler

```
1 class MyClass {
2 public:
3     static void myTask(void);
4 }
5 static const VoidStaticFunctor myTaskFunctor(&MyClass::myTask);
6 ...
7 scheduler->postTask(&myTaskFunctor);
```

Code Listing 5: Posting a static method to the scheduler

```
1 class MyClass {
2 public:
3     MyClass();
4     void myTask(void);
5 }
6 static MyClass myClass();
7 static const VoidInstanceFunctor<MyClass> myTaskFunctor(&myClass, &MyClass
      ::myTask);
8 ...
9 scheduler->postTask(&myTaskFunctor);
```

Code Listing 6: Posting an instance method to the scheduler

"Tasks" in SensorOS can be posted multiple times since there is no mechanism in the scheduler to determine if a task has already been posted. This gives the user a basic kind of priority scheme if they would like to post particular tasks more times than other tasks. It does mean that the user must be careful to ensure that no adverse effects can result from a task being posted multiple times (or implement some sort of mechanism so that a particular task can only be posted once at a time).

A maximum of 256 tasks can be posted at any one time, if a 257th task is posted then the postTask() method will return an error status of EFULL.

API Table 2.1 outlines the public API for the Scheduler class and code listing 7 gives the prototypes of the public methods along with the comments describing their inputs, outputs and any points of note.

Method	Scope	Description / Prototype
getInstance	Static	Returns the singleton instance of the Scheduler class `Scheduler* Scheduler::getInstance()`
pushTask	Instance	Pushes the given task onto the end of the task queue `error_t Scheduler::pushTask(task_t task)`
runNextTask	Instance	Runs the task at the head of the queue and returns whether or not there is a task left on the queue `bool Scheduler::runNextTask()`
taskLoop	Instance	Enters an infinite loop that runs the next task on the queue and sends the processor into sleep mode when the queue is exhausted `void Scheduler::taskLoop()`

Table 2.1: Scheduler class public API

```
1  /**
2   * Returns the singleton instance of the Scheduler class
3   * @return The instance
4   */
5  Scheduler* Scheduler::getInstance()
6
7  /**
8   * Pushes the given task onto the end of the task queue
9   * Contains an atomic section
10  * @param task The task to push onto the queue
11  * @return The status of the operation (either SUCCESS or EFULL)
12  */
13 error_t Scheduler::pushTask(task_t task)
14
15 /**
16  * Runs the task at the head of the queue and returns whether or
17  *  not there is a task left on the queue
18  * Contains an atomic section
19  *  @return Whether or not there is a task left on the queue
20  */
21 bool Scheduler::runNextTask()
22
23 /**
24  * Enters an infinite loop that runs the next task on the queue and
25  *  sends the processor into sleep mode when the queue is exhausted
26  * Calls Platform::sleep()
27  */
28 void Scheduler::taskLoop()
```

Code Listing 7: The prototypes and comments for the public methods of the Scheduler class

2.3.2 SignalRouter

Overview In TinyOS, signals are routed between components in the system by wiring them together via configurations (Levis 2006). This "wiring" capability is not possible in C++, consequently another solution was devised whereby a SignalRouter class (loosely based on the Observer behavioural design pattern (Gamma et al. 1995)) signals all the event handlers registered to a particular event when that event occurs. The SignalRouter implements the Singleton creational design pattern (Gamma et al. 1995), so you can use the code as outlined in code listing 8 to get a reference to the signal router.

```
1 SignalRouter* signalRouter = SignalRouter::getInstance();
```

Code Listing 8: Getting a reference to the signal router

The source code for this class is located in sensoros/os/signal_router.cpp and sensoros/os/signal_router.h as shown in Appendix A.

Signal Representation Signals are represented in SensorOS by the **`signal_t`** enumerated type. The definition for this type is shown in code listing 9. Each platform and application can define up to 127 signals each, which are added to the **`signal_t`** enumeration by the two **`#include`**s. Each signal should begin with "`SIG_`". Two signals that will always be defined are `SIG_SOFTWARE_INIT` (which is a platform related event) and `SIG_SYSTEM_BOOTED` (which is when system initialisation is complete and the application can begin).

```
1 // signal_t Type
2 typedef enum {
3     SIG_SOFTWARE_INIT=0,
4     SIG_SYSTEM_BOOTED,
5     #include <platform_signals.h> // Can define a max of 127 signals
6     #include <application_signals.h> // Can define a max of 127 signals
7     NUM_SIGNALS // Must be last!
8 } signal_t;
9 typedef uint8_t signal_iterator_t; // NUM_SIGNALS can't be more than 256
```

Code Listing 9: Definition of **`signal_t`**

Registering a signal handler To register a signal handler to be called when a particular signal occurs, a functor needs to be made for the function / method to be called in the same way that they are made for the scheduler (see section 2.3.1). Code listing 10 demonstrates this for a functor called `mainFunctor` being registered to the `SIG_SYSTEM_BOOTED` signal.

```
1 // mainFunctor is already defined as a const VoidFunctor
2 // signalRouter is already defined as a reference to the signal router
3 //  see code listing 8
4 signalRouter->registerHandler(SIG_SYSTEM_BOOTED, &mainFunctor)
```

Code Listing 10: Registering a signal handler for the `SIG_SYSTEM_BOOTED` signal

Signalling an event To signal an event you simply call the signal method on the SignalRouter class, passing in the event that occurred. Typically you would only do this in a platform

definition for SensorOS (see section 3), but in some cases it may be necessary to write conditionally compiled platform-specific code that includes event signalling in order to implement specific functionality (see section 2.4.3). The sequence diagram in figure 2.1 demonstrates how signals are handled after the Signal Router is informed of an event.

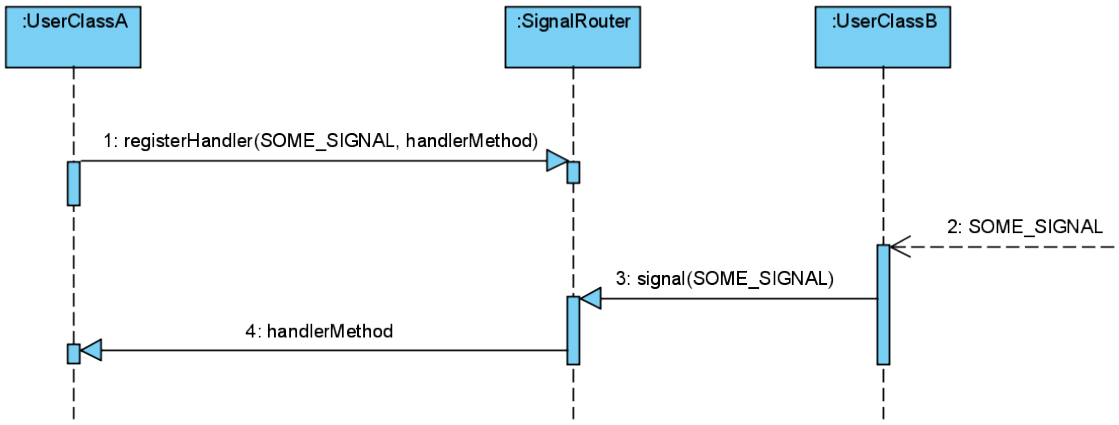


Figure 2.1: Sequence diagram illustrating how events are signalled and handled in SensorOS

API Table 2.2 outlines the public API for the SignalRouter class and code listing 11 gives the prototypes of the public methods along with the comments describing their inputs, outputs and any points of note.

Method	Scope	Description / Prototype
getInstance	Static	Returns the singleton instance of the SignalRouter class `SignalRouter::SignalRouter* getInstance()`
~SignalRouter	Instance	SignalRouter destructor: deletes dynamically allocated handler lists `SignalRouter::~SignalRouter()`
signal	Instance	Signals to the signal router that a particular event has occurred and any registered event handlers for that signal should be notified `void SignalRouter::signal(signal_t signal)`
registerHandler	Instance	Registers a functor to be called when a particular event occurs `void SignalRouter::registerHandler(signal_t signal, VoidFunctor const* handler)`

Table 2.2: SignalRouter class public API

```
1  /**
2   * Returns the singleton instance of the SignalRouter class
3   * @return The instance
4   */
5  SignalRouter* SignalRouter::getInstance()
6
7  /**
8   * Registers a functor to be called when a particular event occurs
9   * Dynamically allocates memory, either explicitly for a list or implicitly
10  *  via List::push_back()
11  */
12 void SignalRouter::registerHandler(signal_t signal, VoidFunctor const*
       handler)
13
14 /**
15  * Signals to the signal router that a particular event
16  *  has occurred and any registered event handlers for
17  *  that signal should be notified
18  * @param signal The signal that has occurred
19  */
20 void SignalRouter::signal(signal_t signal)
21
22 /**
23  * SignalRouter destructor: deletes dynamically allocated handler lists
24  */
25 SignalRouter::~SignalRouter()
```

Code Listing 11: The prototypes and comments for the public methods of the SignalRouter class

2.3.3 Network

Overview The Network class in SensorOS was designed with a number of goals in mind:

- To facilitate some well defined structure in the way the data link and network layers interact
- To explicitly allow multiple data link layers
- To overcome the problem of a lack of bi-directional wiring in C++ (making it hard for networking layers to reference data link layers and vice versa)

The Network class implements the Singleton creational design pattern and is loosely based on the Mediator behavioural design pattern (Gamma et al. 1995). You can use the code as outlined in code listing 12 to get a reference to the network object.

If you would like to see information about the DataLinkLayer and NetworkLayer classes, see sections 2.3.4 and 2.3.5 respectively.

```
1 Network* network = Network::getInstance();
```

Code Listing 12: Getting a reference to the network

The source code for this class is located in sensoros/os/network.cpp and sensoros/os/network.h as shown in Appendix A.

Registering the network layer It is the responsibility of the application to assign a network layer to the Network class sometime before the `SIG_SYSTEM_BOOTED` event (i.e. in `Application::init()`). In order to do this, a code snippet similar to code listing 13 would need to be used.

SensorOS only has explicit support for a single network layer, because most sensor network implementations should only require this. If multiple network layers are required, then the network layer class that is assigned to the network object will need to be some sort of proxy class (Gamma et al. 1995) between the data link layers and the network layers.

```
1 static MyNetworkLayer myNetworkLayer();
2 ...
3 // network is already defined as a reference to the network
4 //  see code listing 12
5 network->assignNetworkLayer(&myNetworkLayer)
```

Code Listing 13: Registering the network layer with the network object

For more information about using the data link layers and networking in general with SensorOS, see section 2.5.

Referencing a data link layer To get a reference to a data link layer, you simply need the data link layer id (1..N), and then use code like that shown in code listing 14.

API Table 2.3 outlines the public API for the Network class and code listing 15 gives the prototypes of the public methods along with the comments describing their inputs, outputs and any points of note.

```
1 // network is already defined as a reference to the network
2 //  see code listing 12
3 // dataLinkId is predefined (of type link_id_t) as the id
4 //  of the data link you want to reference
5 DataLinkLayer dataLink = network->getLink(dataLinkId);
```

Code Listing 14: Referencing a data link layer

Method	**Scope**	**Description / Prototype**
getInstance	Static	Returns the singleton instance of the Network class `Network* Network::getInstance()`
assignLink	Instance	Registers a data link layer with the network `void Network::assignLink(DataLinkLayer* link)`
getLink	Instance	Returns a reference to the data link with the given id `DataLinkLayer* Network::getLink(link_id_t linkId)`
assignNetworkLayer	Instance	Registers the network layer with the network `void Network::assignNetworkLayer(NetworkLayer* networkLayer)`
getNetworkLayer	Instance	Returns a reference to the network layer `NetworkLayer* Network::getNetworkLayer()`
getNumLinks	Instance	Returns the number of data links registered with the network `link_id_t Network::getNumLinks()`
getNodeAddress	Instance	Returns the network address of the current node `node_address_t Network::getNodeAddress()`

Table 2.3: Network class public API

```
1  /**
2   * Returns the singleton instance of the Network class
3   * @return The instance
4   */
5  Network* Network::getInstance()
6
7  /**
8   * Registers a data link layer with the network
9   * Will only add the data link layer if it is added
10  *  in order (e.g. link id 1 first, then 2 etc.) and the link id
11  *  is less than the NUM_DATA_LINKS platform defined constant
12  * @param link A reference to the data link layer to register
13  */
14 void Network::assignLink(DataLinkLayer* link)
15
16 /**
17  * Returns a reference to the data link with the given id
18  * @param linkId The id of the data link to return
19  * @return A reference to the data link layer
20  */
21 DataLinkLayer* Network::getLink(link_id_t linkId)
22
23 /**
24  * Registers the network layer with the network
25  * @param A reference to the network layer to register
26  */
27 void Network::assignNetworkLayer(NetworkLayer* networkLayer)
28
29 /**
30  * Returns a reference to the network layer
31  * @return A reference to the network layer
32  */
33 NetworkLayer* Network::getNetworkLayer()
34
35 /**
36  * Returns the number of data links registered with the network
37  * @return The number of data links
38  */
39 link_id_t Network::getNumLinks()
40
41 /**
42  * Returns the network address of the current node
43  * @return The node's network address
44  */
45 node_address_t Network::getNodeAddress()
```

Code Listing 15: The prototypes and comments for the public methods of the Network class

2.3.4 DataLinkLayer

Overview The DataLinkLayer class is an interface (see section 2.4.5) which is meant to be implemented by platform-specific classes. For more information about using the data link layers and networking in general with SensorOS, see section 2.5.

The source code for this class is located in sensoros/os/data_link_layer.h as shown in Appendix A.

API Table 2.4 outlines the API for the DataLinkLayer interface and code listings 16 and 17 give the prototypes of the methods along with the comments describing their inputs, outputs and any points of note.

Method	Scope	Description / Prototype
downFromNetwork	Instance	Gives a packet to send across the link `virtual error_t downFromNetwork(node_address_t destination, message_t* msg, data_length_t length, priority_t priority)`
cancel	Instance	Requests that the message pending sending in the given message buffer not be sent `virtual error_t cancel(message_t* msg)`
getLinkId	Instance	Returns the link id of this data link `virtual link_id_t getLinkId()`
maxPayloadLength	Instance	Returns the maximum length of the payload field `virtual data_length_t maxPayloadLength()`
getPayloadLength	Instance	Returns the length of the payload field in the given message buffer `virtual data_length_t getPayloadLength( message_t* msg)`
getTimestamp	Instance	Returns the length of the payload field in the given message buffer `virtual data_length_t getPayloadLength( message_t* msg)`
getSource	Instance	Returns the source address in the given message buffer `virtual node_address_t getSource(message_t* msg )`
getDestination	Instance	Returns the destination address in the given message buffer `virtual node_address_t getDestination(message_t * msg)`

Table 2.4: DataLinkLayer interface API

```
1  /**
2   * Gives a packet to send across the link
3   *
4   * @param destination The destination to send to, NULL if not applicable
5   *  or if broadcasting
6   * @param msg A pointer to a message buffer containing the packet in the
7   *  data field
8   * @param length The length of the packet to be sent
9   * @param priority The local priority of the message, smaller is greater
10  *  set to NULL if not applicable
11  *
12  * @return The success of the sending operation
13  *  SUCCESS If queuing the message for sending was successful
14  *  ESIZE If length is greater than the value returned from
15  *      maxPayloadLength()
16  *  EFULL If the send queue is full
17  *  EINVAL If length is 0
18  *  FAIL If there was some other error sending
19  */
20 virtual error_t downFromNetwork(node_address_t destination, message_t* msg,
21                                 data_length_t length, priority_t priority)
22
23 /**
24  * Requests that the message pending sending in the given
25  *  message buffer not be sent
26  *
27  * @param msg Pointer to the message buffer with the message to cancel
28  *  the sending of
29  *
30  * @returns The status of the cancel attempt:
31  *  SUCCESS if the message was still pending
32  *  FAIL if the message had already been sent
33  *  EINVAL if the message buffer wasn't recognised
34  */
35 virtual error_t cancel(message_t* msg)
36
37 /**
38  * Returns the link id of this data link (which then allows it to be
39  *  referenced from the Network class via the getLink method)
40  *
41  * @return The id of this data link
42  */
43 virtual link_id_t getLinkId()
44
45 /**
46  * Returns the maximum length of the payload field
47  *  i.e. MTU - header - footer
48  *
49  * @return The maximum payload field length
50  */
51 virtual data_length_t maxPayloadLength()
52
53 /**
54  * Returns the length of the payload field in the given message buffer
55  *
```

Code Listing 16: The prototypes and comments for the methods of the DataLinkLayer interface (Part 1)

```
1  */
2 virtual data_length_t getPayloadLength(message_t* msg)
3
4 /**
5  * Returns the timestamp in the given message buffer
6  *
7  * @return The timestamp in the given message buffer
8  */
9 virtual timestamp_t getTimestamp(message_t* msg)
10
11 /**
12  * Returns the source address in the given message buffer
13  *
14  * @return The source address in the given message buffer
15  */
16 virtual node_address_t getSource(message_t* msg)
17
18 /**
19  * Returns the destination address in the given message buffer
20  *
21  * @return The destination address in the given message buffer
22  */
23 virtual node_address_t getDestination(message_t* msg)
```

Code Listing 17: The prototypes and comments for the methods of the DataLinkLayer interface (Part 2)

2.3.5 NetworkLayer

Overview The NetworkLayer class is an interface (see section 2.4.5) which is meant to be implemented by a single application class (see section 2.5).

The source code for this class is located in sensoros/os/network_layer.h as shown in Appendix A.

API Table 2.5 outlines the API for the DataLinkLayer interface and code listing 18 gives the prototypes of the methods along with the comments describing their inputs, outputs and any points of note.

Method	Scope	Description / Prototype
upFromDataLink	Instance	Passes a filled buffer with received data to the network layer `virtual message_t* upFromDataLink(DataLinkLayer * linkLayer, message_t* msg, data_length_t length)`
sendDone	Instance	Allows the network layer to be informed that the send of a message has been completed and the associated message buffer is now free for use `virtual void sendDone(message_t* msg, error_t status)`

Table 2.5: NetworkLayer interface API

```
1  /**
2   * Passes a filled buffer with received data to the network layer
3   * Called within an atomic section to preserve integrity of message
4   *  buffer; this method must only perform short computation(s) and then
5   *  return ASAP
6   *
7   * @param linkLayer A pointer to the link layer that received the data
8   * @param msg A pointer to the message buffer with the received data
9   * @param length The length of the received payload (for convenience,
10  *  same value can be gotten with linkLayer->getPayloadLength(msg)
11  *
12  * @return A pointer to a valid message buffer that the link layer
13  *  can use for the next received packet
14  */
15
16 virtual message_t* upFromDataLink(DataLinkLayer* linkLayer, message_t* msg,
17                                   data_length_t length)
18 /**
19  * Allows release of the buffer pointed to by msg and informs
20  *  network layer of the status of sending of the message in that
21  *  buffer
22  *
23  * @param msg A pointer to the message buffer that can be freed
24  * @param status The status of sending the message in msg:
25  *  SUCCESS if the message was successfully sent
26  *  FAIL if the message failed to send
27  *  ECANCEL if the message was cancelled by a call to DataLinkLayer::cancel
        ()
28  */
29 virtual void sendDone(message_t* msg, error_t status)
```

Code Listing 18: The prototypes and comments for the methods of the NetworkLayer interface

2.3.6 VoidFunctor

A functor or function object is an object that can be called as if it is a function. Functors are used in SensorOS for storing which method or function should be called when events occur or when a scheduled task is run. In C++ functors can be created by overloading the () operator. (Silicon Graphics, Inc. 2009)

As shown in Appendix B, the VoidFunctor class is an abstract base functor class that encapsulates a function that takes no parameters and returns void i.e. `void functionName(void)`. It is a purely virtual class that is then overridden by the VoidStaticFunctor and VoidInstanceFunctor classes.

VoidStaticFunctor encapsulates a static method or function pointer, where as VoidInstanceFunctor encapsulates a pointer to an instance method within an instantiated object. Code listings 19, 20 and 21 show how to create and call a functor pointing to a function, static method and instance method respectively.

```
1 // Define function
2 void func(void);
3 // Initialise VoidStaticFunctor with function pointer
4 static const VoidStaticFunctor myFunctor(&func);
5 // Call the functor (calls the func() function)
6 myFunctor();
```

Code Listing 19: Creating and calling a VoidFunctor encapsulating a function pointer

```
1 // Define class with static method
2 class MyClass {
3 public:
4     static void func(void);
5 }
6 // Initialise VoidStaticFunctor with method pointer
7 static const VoidStaticFunctor myFunctor(&MyClass::func);
8 // Call the functor (calls the MyClass::func() method)
9 myFunctor();
```

Code Listing 20: Creating and calling a VoidFunctor encapsulating a static method pointer

The source code for this class is located in sensoros/os/void_functor.h as shown in Appendix A.

```
1  // Define class with instance method
2  class MyClass {
3  public:
4      MyClass();
5      void func(void);
6  }
7  // Initialise instance of MyClass
8  static MyClass myClass();
9  // Initialise VoidInstanceFunctor with method pointer
10 static const VoidInstanceFunctor<MyClass> myFunctor(&myClass, &MyClass::
       func);
11 // Call the functor (calls the myClass.func() method)
12 myFunctor();
```

Code Listing 21: Creating and calling a VoidFunctor encapsulating an instance method pointer

2.3.7 List

Overview Linked lists are a useful abstract data type (ADT) when:

- The number of items being stored is unknown
- No reasonable assumption can be made about the minimum / maximum number of items, or it is inefficient to allocate static storage for the potential maximum number of items (due to a large variation between the minimum and maximum number)
- The list won't be searched (since searching is $O(n)$ for linked lists)

One such area of application in SensorOS is the storage of event handlers in the SignalRouter class (see section 2.3.2). The number of signals that event handlers will be assigned to is unknown, the maximum number of event handlers for a given signal is unknown and the list is only used for iteration (not searching) when the event handlers are called in turn.

Thus a List class has been developed for use in SensorOS. This class is based on the interface of the standard template library (STL) list class. There are two reasons why a separate implementation was used in place of the STL class:

1. There is an Embedded C++ standard (EC++) that doesn't support the STL, so whilst many compilers may extend EC++ with the STL, many compilers won't have it, which conflicts with the goal of maximum compatibility across multiple development environments (IAR Systems nd; Plauger 1997)
2. The implementation given in SensorOS is optimised for embedded environments, with as little run-time overhead as possible, this does mean some trade-offs such as it is a singly-linked list as opposed to a doubly-linked list

The source code for this class is located in sensoros/os/list.h as shown in Appendix A.

Disadvantages The main disadvantage of the List class is that it uses dynamic memory allocation to create the links (and the SignalRouter also dynamically allocates the List classes as needed per signal as well to conserve memory space). Static allocation is preferred in SensorOS for the reasons discussed in 2.4.8.

The List class and ListLink class are both lightweight (both have two pointers, this is one of the differences from the STL implementation), so a small heap is needed to support them, and at the very least for signal handlers, all allocation should be done before `SIG_SYSTEM_BOOTED`, so run-time performance after the OS is operational shouldn't be affected.

As well as the dynamic allocation issue, it should be noted that the List class uses templating to allow it to be reused for other classes (than VoidFunctor, which is what SignalRouter uses lists of), meaning that for every class that is used in conjunction with the List class there will be more code memory needed.

Using the List class You can use the same sort of code that you would for the STL List class to perform basic operations on the list, such as appending and prepending elements, and iterating over the list. Code listing 22 gives an overview of the code required for these basic operations.

It should be noted that if the `DEBUG` macro is set to 1 then friend operations will be compiled with the List class, the ListLink class and the ListIterator class, so you can print then to an

output stream (e.g. cout) for debugging purposes.

API Table 2.6 outlines the API for the List class and code listing 23 gives the prototypes of the public methods along with the comments describing their inputs, outputs and any points of note. It should be noted that the `push_back` and `push_front` methods violate the naming conventions outlines in section 1.4.1, but are kept this way for compatibility with the STL interface.

Method	Scope	Description / Prototype
~List	Instance	List destructor: Deletes all the link objects allocated in this class `template<class TClass> List<TClass>::~List()`
push_back	Instance	Adds the given item to the end of the list `template<class TClass> void List<TClass>::push_back(TClass* obj)`
push_front	Instance	Adds the given item to the start of the list `template<class TClass> void List<TClass>::push_front(TClass* obj)`
insert	Instance	Inserts the given item to the list after the item pointed to by the given iterator `template<class TClass> void List<TClass>::insert(List<TClass>::iterator& iter, TClass* obj)`
begin	Instance	Returns the first item in the list `template<class TClass> ListLink<TClass>* List<TClass>::begin()const`
end	Instance	Returns the last item in the list `template<class TClass> ListLink<TClass>* List<TClass>::end()const`

Table 2.6: List class public API

```
1  // Define test class (MyClass) and some instances of it
2  class MyClass {
3  public:
4      MyClass(uint8_t value) : value(value) {};
5      uint8_t value;
6  };
7  MyClass one(1);
8  MyClass two(2);
9  MyClass three(3);
10 MyClass four(4);
11 MyClass five(5);
12 // Create a list, this is dynamic to match the usage in
13 //  SignalRouter, but this could also be a static instantiation
14 List<MyClass>* myList = new List<MyClass>();
15 // Add one to the back of the list
16 myList->push_back(&one);
17 // Get an iterator pointing to the current front of the list
18 // Uses the constructor of the iterator to assign the iterator
19 //  to an item
20 List<MyClass>::iterator i(myList->begin());
21 // Add two to the front of the list
22 myList->push_front(&two);
23 // Add three to the back of the list
24 myList->push_back(&three);
25 // Insert four after the item pointed to by iterator i
26 myList->insert(i, &four);
27 // Add five to the front
28 myList->push_front(&five);
29 // Iterate through the list with a new iterator, j
30 // Uses assignment to assign the iterator to an item
31 List<MyClass>::iterator j;
32 for (j=myList->begin(); j != myList->end(); ++j) {
33     // Dereference the iterator to get the class
34     //  reference, then print out the value in the class
35     //  using the Debug class
36     Debug::printf("%d\r\n", (*j).value);
37 }
38 // Note: Incrementing of the iterator MUST be preincrement
39 //  i.e. ++j instead of j++. Post increment is NOT supported
40 //  because it is less efficient (requires object copy)
41 // Note also: Only != and == are available for the iterator
42 //  and can be used to compare the iterator with an item in
43 //  the list or another iterator
```

Code Listing 22: How to use the List class and its iterator class

```
1  /**
2   * List destructor
3   *
4   * Deletes all the link objects allocated in this class
5   */
6  template<class TClass> List<TClass>::~List()
7
8  /**
9   * Adds the given item to the end of the list
10  *
11  * @param obj A reference to a TClass object to
12  *  be added to the list
13  */
14 template<class TClass> void List<TClass>::push_back(TClass* obj)
15
16 /**
17  * Adds the given item to the start of the list
18  *
19  * @param obj A reference to a TClass object to
20  *  be added to the list
21  */
22 template<class TClass> void List<TClass>::push_front(TClass* obj)
23
24     /**
25  * Inserts the given item to the list after the item
26  *  pointed to by the given iterator
27  *
28  * @param iter An iterator pointing to an item in the list
29  * @param obj A reference to a TClass object to
30  *  be added to the list
31  */
32 template<class TClass> void List<TClass>::insert(List<TClass>::iterator&
       iter, TClass* obj)
33
34 /**
35  * Returns the first item in the list
36  *
37  * @return The first item in the list
38  */
39 template<class TClass> ListLink<TClass>* List<TClass>::begin() const
40
41 /**
42  * Returns the last item in the list
43  *
44  * @return The last item in the list
45  */
46 template<class TClass> ListLink<TClass>* List<TClass>::end() const
```

Code Listing 23: The prototypes and comments for the public methods of the List class

2.3.8 Platform

Overview The Platform class acts as a global namespace exposing an interface for the operating system to interact with the hardware platform. The class only contains static methods and as such is never instantiated. It is implemented as part of a platform.

Additional methods can be defined aside from the ones mentioned in the API; the API simply outlines the minimum interface for the class.

API Table 2.7 outlines the API for the Platform class and code listing 24 gives the prototypes of the public methods along with the comments describing their inputs, outputs and any points of note.

Method	Scope	Description / Prototype
bootstrap	Static	Initialises the hardware at a very basic level `void Platform::bootstrap()`
init	Static	Initialise the hardware platform `void Platform::init()`
sleep	Static	Puts the hardware into a low power mode when there are no more tasks to run `void Platform::sleep()`

Table 2.7: Platform class public API

```
1  /**
2   * Initialises the hardware at a very basic level
3   *  e.g. CPU / memory mode selection
4   */
5  void Platform::bootstrap()
6
7  /**
8   * Initialise the hardware platform
9   * Can post tasks to be run after the method is finished
10  *  if there are order dependant tasks
11  */
12 void Platform::init()
13
14 /**
15  * Puts the hardware into a low power mode when there are no
16  *  more tasks to run. It should exit this mode either after
17  *  a specified amount of time, or preferably when there is
18  *  an interrupt that requires the platform to perform some
19  *  action, e.g. network activity
20  */
21 void Platform::sleep()
```

Code Listing 24: The prototypes and comments for the public methods of the Platform class

2.3.9 Debug

Overview The Debug class acts as a global namespace exposing an interface for debugging applications. The class only contains static methods and as such is never instantiated. It is implemented as part of a platform.

Additional methods can be defined aside from the ones mentioned in the API; the API simply outlines the minimum interface for the class.

API Table 2.8 outlines the API for the Debug class and code listing 25 gives the prototypes of the public methods along with the comments describing their inputs, outputs and any points of note.

The print and printf methods of the debug class can be implemented in a platform defined way, for example by sending characters to an RS232 interface. It should be noted that the use of the char type in these methods violates the coding conventions outlined in section 1.4.3, but is needed due to the compiler automatically using a `const char *` for static strings (e.g. `"static string"`).

The Debug class interface assumes the hardware platform has 8 LEDs. Not all hardware platforms will have this, but the Debug class will be implemented as part of the platform so this shouldn't be a problem.

If the Debug class needs some initialisation code run before it can function then the platform can define an init (or similar) method on the class and call it sometime before the `SIG_SYSTEM_BOOTED` event.

Additional methods can be defined aside from the ones mentioned in the API.

Method	Scope	Description / Prototype
print	Static	Prints a static string `void Debug::print(const char* string)`
printf	Static	Prints a formatted string `void Debug::printf(const char* format, ...)`
leds	Static	Lights a number of LEDs based on an integer value `void Debug::leds(uint8_t ledsValue)`
leds	Static	Lights (on unlights) a specific LED `void Debug::leds(bool ledValue, uint8_t led)`
toggleLed	Static	Toggles a specific LED on or off `void Debug::toggleLed(uint8_t led)`

Table 2.8: Debug class public API

```
1  /**
2   * Prints a static string
3   *
4   * Prints the given string to Serial Communications Interface (SCI) 0
5   *
6   * @param string The string to print
7   */
8  void Debug::print(const char* string)
9
10 /**
11  * Prints a formatted string
12  *
13  * Prints the given format string to Serial Communications Interface (SCI)
        0
14  * Uses the same syntax as the C stdio printf function
15  *
16  * @param format The format string
17  * @param ... Any values referenced in the format string
18  */
19 void Debug::printf(const char* format, ...)
20
21 /**
22  * Lights a number of LEDs based on an integer value
23  *
24  * If the integer given is say 6, then LED 1 and 2 will be lit
25  *  but LED 0 won't be
26  *
27  * @param ledsValue The value to set the leds to
28  */
29 void Debug::leds(uint8_t ledsValue)
30
31 /**
32  * Lights (or unlights) a specific LED
33  *
34  * @param ledValue Whether to light (true) or unlight
35  *  (false) the LED
36  * @param led The LED to modify (0-7)
37  */
38 void Debug::leds(bool ledValue, uint8_t led)
39
40 /**
41  * Toggles a specific LED on or off
42  *
43  * @param led The LED to toggle (0-7)
44  */
45 void Debug::toggleLed(uint8_t led)
```

Code Listing 25: The prototypes and comments for the public methods of the Debug class

2.3.10 Application

Overview The Application class acts as a global namespace exposing an interface for applications to be initialised by SensorOS. The class only contains static methods and as such is never instantiated. It is implemented as part of an application.

Additional methods can be defined aside from the ones mentioned in the API; the API simply outlines the minimum interface for the class.

API Table 2.9 outlines the API for the Application class and code listing 26 gives the prototypes of the public methods along with the comments describing their inputs, outputs and any points of note.

Method	Scope	Description / Prototype
init	Static	Initialise the application before interrupts are enabled `void Application::init()`

Table 2.9: Application class public API

```
1 /**
2  * Initialises the application before interrupts are enabled
3  */
4 void Application::init()
```

Code Listing 26: The prototypes and comments for the public methods of the Application class

2.4 Programming SensorOS Applications

2.4.1 Sensor Network Applications

Applications in a sensor network will have two main activities:

1. Interacting with the network; and
2. reading sensors.

The way that applications interact with the network is controlled by the implementation of the network layer and any further layers on top of it (see section 2.5.8).

The sensor subsystem has not yet been implemented in SensorOS, however it will be implemented as part of a future version. The sensor subsystem will be based on the one introduced in TinyOS; Source (the sensor API) and Sink (the application using the sensor) Independent Drivers (SIDs) (Tolle et al. 2008).

The SensorOS sensor subsystem will contain similar SID interfaces that use templating in order to achieve the same effect as the parameterised TinyOS interfaces. Because of the lack of bi-directional wiring in C++, a central `Sensor` class will need to be created to facilitate access to sensors. It is likely that the "wiring" of sensors to a particular application will still require platform targeted code (see section 2.4.3).

TinyOS has interfaces for accessing the ADC of platforms in order to get digitised values of analogue sensors. In SensorOS, it will be left to platform developers to expose access to digitised sensor values via the above mentioned interfaces.

2.4.2 Readme File

All platforms and applications should have a `readme.txt` file within their directory. This file should be edited in Markdown syntax[1]. The application readme file should contain the following sections:

- **Application** - The name of the application
- **Description** - A brief description of the purpose of the application
- **Author** - The authors name(s) as links to their email addresses
- **Date** - The date the application was last modified
- **Version** - The version of the application
- **Platforms** - A statement about the compatibility of the application with SensorOS platforms and a list of the platforms that it has been successfully run on
- **Notes** - Any points of note about the application, in particular documentation on any platform targeted code (see section 2.4.3)
- **License** - The software license you are releasing the application under

An example `readme.txt` file is shown below:

```
1 Application
```

[1] http://daringfireball.net/projects/markdown/syntax

```
2  ========
3
4  null
5
6  Description
7  ============
8
9  SensorOS application that simply prints when it initialises and
10 when the system is booted.
11
12 Author
13 ======
14
15 [Robert Moore](mailto:rob@mooredesign.com.au)
16
17 Date
18 ====
19
20 2009-09-27
21
22 Version
23 =======
24
25 1.0.0
26
27 Platforms
28 =========
29
30 This application should be compatible with all platforms, at present
31 it has been successfully used with the following platforms:
32
33 * dragon12
34
35 Notes
36 =====
37
38 This is a simple example application that can be used when
39 initially developing platforms to ensure they compile.
40
41 License
42 =======
43
44 MIT
45 ---
46
47 Copyright (c) 2009 Robert Moore
48
49 Permission is hereby granted, free of charge, to any person
50 obtaining a copy of this software and associated documentation
51 files (the "Software"), to deal in the Software without
52 restriction, including without limitation the rights to use,
```

```
53  copy, modify, merge, publish, distribute, sublicense, and/or sell
54  copies of the Software, and to permit persons to whom the
55  Software is furnished to do so, subject to the following
56  conditions:
57
58  The above copyright notice and this permission notice shall be
59  included in all copies or substantial portions of the Software.
60
61  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
62  EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
63  OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
64  NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
65  HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
66  WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
67  FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
68  OTHER DEALINGS IN THE SOFTWARE.
```

At this stage there is no automated documentation process for platforms and applications, however in a future version of SensorOS there will be a `sensoros/docs` directory with generated documentation for platforms and applications from the `readme.txt` and the source code (using Doxygen[1]).

2.4.3 Platform Targeted Application Code

While it is desirable to create applications that are 100% independent of platform, sometimes this is impossible. The way this problem is tackled with SensorOS is by conditional compilation. Each platform in SensorOS must define a macro `PLATFORM_<PLATFORM_NAME>`. So for the dragon12 platform, the macro `PLATFORM_DRAGON12` is defined. Code listing 27 gives a demonstration of using such conditional compilation to enable controlling software for CAN (Controller Area Network) modules 0 and 3 when compiling.

```
1 #ifdef PLATFORM_DRAGON12
2     // Use CAN module 0 as data link 1
3     #define MSCAN0 1
4     // Use CAN module 3 as data link 2
5     #define MSCAN3 2
6 #endif
```

Code Listing 27: Example of platform targeted code in an application for the dragon12 platform

Common problems that require platform targeted application code are:

- Differences in hardware capabilities of platforms (e.g. one platform might have one LED, whereas others may have 8)
- Lack of support in the core OS / platform code for a required feature (possibly should be abstracted as an extension to the OS rather than added to the application code using conditional compilation)
- Need to define macros to support conditional compilation of various components of the system (e.g. the 5 CAN modules on the HCS12 for the dragon12 platform can be indi-

[1] http://www.stack.nl/ dimitri/doxygen/

vidually enabled by defining `CAN<X>` as the data link id for that CAN module where X is 0 - 4, see code listing 27).

Where possible, platform targeted application code should be avoided, to ensure that applications are as portable as possible between platforms. Furthermore, there should be clear documentation about any such code in the application so that anyone wanting to use the application on a different platform knows what parts may not be compatible with their platform (see section 2.4.2).

2.4.4 main() Function

The actual main() function is defined as part of the core of SensorOS in the sensoros/os/-main.cpp file (see code listing 28). The two main parts of it as far as application developers are concerned are the `Application::init()` and `signalRouter.signal(SIG_SYSTEM_BOOTED)` lines. The former is where the application can perform any initialisation necessary before interrupts are enabled (such as network initialisation) and the later is where the application can actually start running because the system is fully initialised. It should be noted that any tasks posted to the Scheduler inside Application::init will be run before interrupts are enabled (this is so that any order dependent operations can be posted from Application::init, but run after it finishes).

In order for an application to make use of the `SIG_SYSTEM_BOOTED` event, it must create one or more functors pointing to functions or methods that should be called and then register them with that signal, like in code listing 10.

2.4.5 Interfaces

The ability to create interfaces using C++ is built into SensorOS by using the technique outlined by Rios (2005). Code listing 29 shows the definition of the interface macros; code listing 30 shows how to create an interface and code listing 31 shows how a class can implement an interface.

Because C++ supports multiple inheritance, you can use multiple interfaces in a class, or implement an interface in a class that already inherits from another class.

2.4.6 Preventing Race Conditions (Atomic Sections)

Race conditions can be defined as "Anomalous behaviour due to unexpected critical dependence on the relative timing of events." (FOLDOC 2002). Interrupts and their associated handlers mean the potential for race conditions exists in SensorOS. This is overcome by creating an atomic section around code that is vulnerable to race conditions. Because scheduling isn't preemptive in SensorOS (see section 2.3.1), it was decided that there was not a need for synchronisation mechanisms such as semaphores, and instead the atomic sections would consist of simply disabling interrupts for the period of the atomic section. Because SensorOS runs in supervisor mode and is designed for a uniprocessor environment, the usual problems with this approach are not relevant (Rinard 1998). Since interrupts are disabled during an atomic section it is *very important* that **the sizes of all atomic sections are as small as possible** and **the number of atomic sections is minimised**.

In order to create an atomic section in SensorOS you simply need to use the code shown in code listing 32.

```
1  /**
2   * Main system function (first function the system loads)
3   *
4   * @returns The status of the program, should never actually return
5   *  due to infinite scheduler loop
6   */
7  int main() {
8
9      // Platform bootstrap
10     Platform::bootstrap();
11     // Initialise the scheduler
12     Scheduler* scheduler = Scheduler::getInstance();
13     // Initialise the platform
14     Platform::init();
15     // Run any tasks
16     while (scheduler->runNextTask());
17     // Initialise the signal router
18     SignalRouter* signalRouter = SignalRouter::getInstance();
19     // Send a software initialisation signal
20     // Allows the platform initialisation to post event handlers
21     //  for this signal if they need to be deferred until after
22     //  initialisation
23     signalRouter->signal(SIG_SOFTWARE_INIT);
24     // Run any tasks
25     while (scheduler->runNextTask());
26     // Initialise the application
27     Application::init();
28     // Run any tasks
29     while (scheduler->runNextTask());
30     // Enable interrupts
31     __enable_interrupts();
32     // Call the system booted event
33     signalRouter->signal(SIG_SYSTEM_BOOTED);
34     // Spin in the scheduler loop
35     scheduler->taskLoop();
36     // Include for compiler, shouldn't ever be reached
37     return -1;
38 }
```

Code Listing 28: main() function

The definitions for **Atomic** and **EndAtomic** are shown in code listing 33. `__atomic_start()` and `__atomic_end()` are defined for each platform (see section 3.3.7), the basic idea is that the `__atomic_start()` routine returns whether or not interrupts were disabled before the atomic section and `__atomic_end()` uses this to determine whether or not it should enable interrupts.

Atomic and **EndAtomic** define a block and as such these commands must be treated the same as any block; you can't nest a block around only one of the commands (code listing 34 shows an example of invalid block nesting).

Since the atomic section is within its own block, the fact that **Atomic** declares a variable shouldn't affect your ability to declare multiple atomic sections in a single function since the `atomicValue` variable should be limited to the scope of the atomic section block.

```
#define Interface(name) \
    class name { \
        public: \
            virtual ~name() {};
#define EndInterface }
#define implements public
```

Code Listing 29: Interface macro definitions

```
Interface(MyInterface)
    virtual void someMethod(uint8_t someValue) = 0;
    virtual void someMethod2(uint8_t someOtherValue) = 0;
EndInterface;
```

Code Listing 30: Declaring an Interface

2.4.7 Interrupt Enabling / Disabling

If for some reason a blanket interrupt enable or disable is required then this can be performed using the `__enable_interrupts()` and `__disable_interrupts()` routines respectively. These are defined as part of a platform.

2.4.8 Static Allocation

Where possible, persistent variables in SensorOS should be declared and initialised statically, in particular if the objects are constant (e.g. VoidFunctors). There are a number of reasons this approach was chosen (Saks 2008):

- As outlined by Saks (1998); static allocation and initialisation of constant objects allows C++ compilers the ability to place those objects in ROM, freeing up valuable RAM space (important in low memory applications like sensor networks)
- Static storage allocation has no run-time cost (i.e. avoids **`new`** / `malloc` etc.)
- MISRA guidelines are against the use of dynamic memory allocation
- Dynamic memory allocation is (typically) non-deterministic, which isn't suited for real time operating environments
- Dynamic memory allocation fails at run-time if there is not enough memory (if static allocation is used, the lack of memory will become obvious at compile time)
- Static memory allocation avoids issues such as dangling pointers, heap corruption, memory leaks and fragmentation

There are a number of factors that need to be considered with static allocation however:

- Static initialisation has run-time overhead prior the `main()` being called
- Dynamic memory allocation is more flexible and efficient with memory use meaning memory is "squandered" if there are lots of intermittently used objects

Because the initialisation of static classes occurs before main is called, if there is any initialisation of objects that require core operating system functionality, then this should go in a method other than the constructor that is then called from `Platform::init()` or `Application::init`

```
1 class MyClass: implements MyInterface {
2 public:
3     MyClass();
4     void someMethod(uint8_t someValue);
5     void someMethod2(uint8_t someOtherValue);
6 }
```

Code Listing 31: Implementing an Interface

```
1 Atomic
2     // Race condition vulnerable code ...
3 EndAtomic;
```

Code Listing 32: Creating an atomic section

`()` as appropriate.

2.4.9 Application Signals

Application code can utilise the core signal handling functionality (see the SignalRouter class description in section 2.3.2) by defining a number of signals that the application can use. These signals are defined in the `application_signals.h` file within the application's directory. This file simply contains a list of signals separated by commas (any whitespace can appear before and after the commas). The signals should all begin with `SIG_` and be uppercase and with underscores between words (as outlined by the naming convention in section 1.4.1 for constants).

2.4.10 Main Include

In order to expose a source file to SensorOS, you need to add **`#include`**`<main.h>` to the start of the .cpp file. The corresponding .h definition file for the class should be included in the `application.h` file. This is different from the usual C++ practice of including the .h file in the .cpp file and then the .h file includes any necessary functionality the class needs. It is necessary to do it this way though to avoid circular dependency issues so that all the SensorOS types and classes are included in the correct order for all classes.

If there are any header files that need to be included that contain functionality only used in a particular source file only, then it should be included from within that source file rather than the `application.h` file.

2.4.11 Debugging

As discussed in section 2.3.9, each platform should define a `Debug` class that facilitates debugging using LEDs and string printing. When an application is deployed in a production manner these debug calls should not be present. In order to facilitate this, the application should be compiled with a `DEBUG_OFF` macro set to 1. This can be achieved by adding this as a constant in the command line call to the compiler (or via appropriate dialogs within an IDE).

```
1 #define Atomic {atomic_t atomicValue = __atomic_start();
2 #define EndAtomic __atomic_end(atomicValue);}
```

Code Listing 33: Definition of **`Atomic`** and **`EndAtomic`**

```
1 {
2     Atomic
3     // ...
4 }
5 EndAtomic;
```

Code Listing 34: Invalid block nesting in conjunction with atomic section

2.5 Networking

2.5.1 Overview

Table 2.10 outlines how the different OSI Reference Model layers correspond to networking in SensorOS. The main part of the OSI Reference Model that is explicitly implemented in SensorOS is the interface between the Data link and Networking layers, because this interface is between the platform code (the data link layer interfaces directly with the hardware physical layer, and as such needs to be implemented as part of the platform code) and the application code (the way the network layer is implemented is heavily dependent on the networking protocols used in any given application, and thus is implemented as part of the application code).

OSI Layer	**Implemented in SensorOS by**
Application	Application code (see section 2.4.1)
Presentation	Application code (optional)
Session	Application code (optional)
Transport	Application code (optional)
Network	Application code (implementing NetworkLayer interface, see section 2.3.5)
Data link	Platform code (implementing DataLinkLayer interface once per data link layer, see section 2.3.4)
Physical	Platform hardware

Table 2.10: Mapping the OSI Reference Model to SensorOS (Downs et al. 1998)

Any further networking code can be implemented as part of platform and / or application code as needed. SensorOS simply provides the basic framework to perform networking with the application and platform code having a distinct separation. For example, the platform could define Medium Access Control (MAC) and Logical Link Control (LLC) sub layers within a class implementing the DataLinkLayer interface and then implement Ethernet. Alternatively, it could expose a simple interface for the networking layer to interact with the networking hardware (see the CanLink class in the dragon12 platform). Similarly, the application could implement a TCP/IP stack, a full OSI stack or a simple network layer that the application code talks to directly. SensorOS is flexible enough to incorporate any number of networking possibilities.

2.5.2 Message Buffer

The interaction between the network and data link layers in SensorOS is based on the same concepts introduced in TinyOS 2 (Levis 2007), in that a specialised message buffer is used, which allows zero-copy semantics when transferring a packet between different data link layers and between the network layer and data link layer. It also makes it easier to copy a message to the physical layer since the header, data and footer regions of the packet can be kept contiguous for all data link layers. The end result is less memory use and less processing time when dealing with messages.

The way this works is via the `message_t` type, which is the type for the aforementioned message buffer. Code listing 35 gives the definition of the `message_t` type.

```
typedef struct {
    uint8_t header[sizeof(message_header_t)];
    uint8_t data[MAX_DATA_BYTES];
    uint8_t footer[sizeof(message_footer_t)];
    uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

Code Listing 35: Definition of the message_t message buffer type

The `message_header_t`, `message_footer_t` and `message_metadata_t` types are defined as part of a platform as the union of the structs that represent the header, footer and metadata for each type of data link layer present on that platform. For more information see section 3.3.1.

The `MAX_DATA_BYTES` constant is defined by the platform as the maximum number of bytes able to be transmitted by its data link layers.

When storing a network layer message in a message buffer, the message should go inside the data section of the message buffer. Because of the way the header, footer and metadata sections are defined, the network layer data (packet) will always be in the same position in the buffer. This is what facilitates the zero-copy semantics of the message buffer structure, a message may be received from one data link layer, passed to the network layer, who can then read the network packet inside and forward it to a different data link layer by passing a pointer to the same message buffer. The second data link layer simply sets the header and footer information in the message buffer, leaving the data section alone and can then send the frame contained in the message buffer.

It should be noted that the header is **not** aligned to the start of the header section, but instead right-aligned to the header section. This is to ensure that the header, data and footer sections are contiguous to assist with transferring the packet to the hardware. It should also be noted that the footer region is only contiguous if the network packet is `MAX_DATA_BYTES` bytes long. If the packet is shorter then it's reasonable for a data link layer to place the footer contiguously after the data packet (i.e. in the data region of the message buffer). The data link layer will need to store the size of the network layer packet in the **header** to be able to reconstruct the footer when receiving it though.

Figure 2.2 illustrates how the different sections are aligned in the message buffer for data link layers with different sized headers, footers and metadata sections. It outlines a hypothetical system that has both CAN and Ethernet data link layers, in this system `MAX_DATA_BYTES` would be set to 1500 (or less if the system didn't have enough memory to cope with that much data in the message buffers), which is the max payload for Ethernet. Because the header size of Ethernet is larger than that for CAN it determines the size of the header section of **`message_t`**, similarly with the meta data section (where an arbitrary 7 byte length is chosen for Ethernet). Because there is no need for a footer section with Ethernet, the footer section of CAN determines the size of the footer section in **`message_t`**. An example of contiguous footer storage with the data section is also shown for the CAN data link layer.

If the data link layer has to support variable sized headers or footers then it can either place them at the start of the header and / or footer regions of the message buffer so they have a defined offset or they can store the length of the header / footer in the metadata section and still store them contiguously with the data section. Obviously, if this is the case then the maximum header / footer size is what must be used in the `message_header_t` and `message_footer_t` unions.

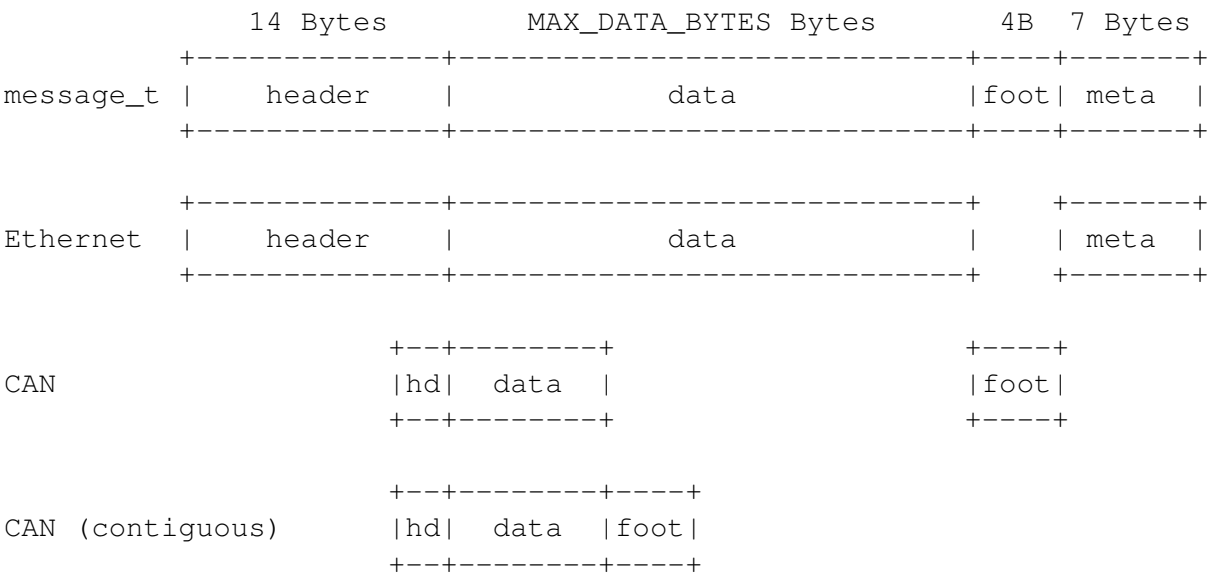


Figure 2.2: Illustration of message_t type and data placement for a hypothetical system with CAN and Ethernet data link layers

2.5.3 Accessing Data Link Layer Frame Information

The network layer is only allowed to access the data section of the message buffer. If it needs any information from the header / footer / metadata sections then it **must** use the interface provided by DataLinkLayer to access them, namely the following methods (see section 2.3.4 for more details):

- **data_length_t** `getPayloadLength(`**message_t*** `msg)`
- **timestamp_t** `getTimestamp(`**message_t*** `msg)`
- **node_address_t** `getSource(`**message_t*** `msg)`
- **node_address_t** `getDestination(`**message_t*** `msg)`

Obviously, the network layer must call these methods on the DataLinkLayer class corresponding to the data link that actually set the data in the message buffer.

2.5.4 Network Information Include

One of the application code files is `application_network_info.h`. The reason this file was created was to avoid circular compiler dependency issues. This file is included within `platform.h` before the platform networking code is defined.

This file defines two things:

1. The **node_address_t** type: must be big enough to store the largest node data link layer address
2. The `NODE_ADDRESS` constant: the data link layer address of the current node

The `NODE_ADDRESS` constant can be used as a network layer address as well, if the networking protocol in use allows node addressing using simple integers. In order to enable loading an application onto different nodes quickly, it is also valid to define the `NODE_ADDRESS` constant in

the command line call to the compiler, or via appropriate dialogs in an IDE.

2.5.5 Sending Messages

When sending a message from the network layer, the network layer class should prepare the packet it wants to send by putting it in the data section of a free message buffer (see section 2.5.2). Once the message has been prepared, the network layer needs to pass a reference to the filled message buffer to the data link layer it wants to send the message. It can get a reference to the relevant data link layer by using the id of the data link layer, in combination with code listing 14.

It can pass the message buffer to that data link layer by using the `downFromNetwork` method (see figure 2.3, code listing 16 and table 2.4). It needs to pass in the data link layer address of the node to send to on that link (or NULL if broadcasting), the message buffer pointer, the length of the network layer packet in the data section of the message buffer and the priority of the message (or NULL if not applicable). The method will then return a status:

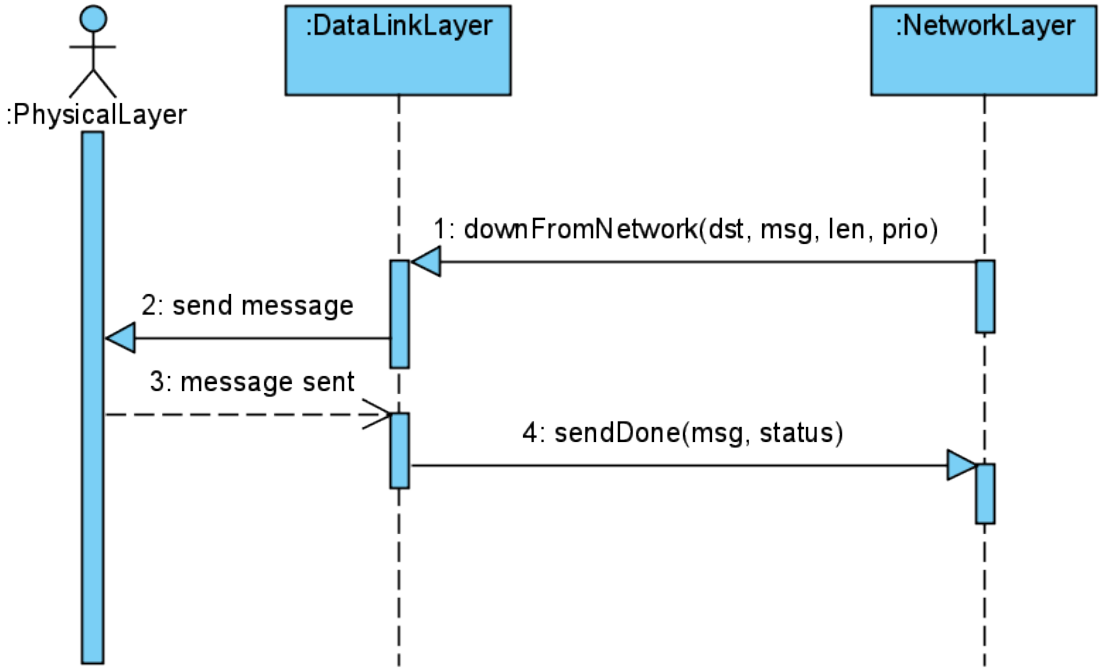


Figure 2.3: Sequence diagram illustrating how messages are sent in SensorOS

- **SUCCESS**: If queuing the message for sending was successful
- **ESIZE**: If length is greater than the maximum payload of the data link; the message was not sent
- **EFULL**: The send queue is full; the message was not sent
- **EINVAL**: If the length was 0; there was no message to send
- **FAIL**: There was some unknown / other error sending; the message was not sent

Note that the `ESIZE` return value occurs when a network layer packet is passed down which is too large for the data link to send. You can check the sending capacity of the data link by calling

the `maxPayloadLength` method on the corresponding DataLinkLayer class.

The `downFromNetwork` method queues a message for sending and as such the DataLinkLayer class retains the pointer to the passed in message buffer until it has finished sending. This means that once a message buffer is populated and passed to a data link for sending, the network layer can't touch the message buffer again.

Once the data link layer has finished sending the message it will notify the network layer via its `sendDone` method. The data link layer will pass in the message buffer pointer as well as the status of the sending process (typically, the status would be one of `SUCCESS`, `FAIL` or `ECANCEL`). Once the network layer receives the `sendDone` message for a particular message buffer it regains control of that message buffer and may continue using it. It should be noted that if the data link layer can confirm the message was sent immediately, then it is allowed to call `sendDone` before it returns from `downFromNetwork`, so the network layer should not do anything with the message buffer after the call to `downFromNetwork`.

Code listing 36 outlines an example class with one possible way of controlling which message buffers are free or used. It only allows the tracking of 8 buffers, but it's easy to extend the concept further. When sending a message, the class would simply have to use the code `message_t* msg = getFreeBuffer();` to secure a free message buffer (but check that it's not `NULL` before using it).

2.5.6 Cancelling a Message

It is possible for the network layer to cancel a queued message about to be sent by a data link layer. In order to do this, the network layer simply calls the `cancel` method on the relevant DataLinkLayer class, passing in a pointer to the message buffer it originally sent with the message to send. The method will either return:

- **SUCCESS** - If the message was still pending
- **FAIL** - If the message had already been sent, or there is some other problem cancelling the message
- **EINVAL** - If the message buffer passed in was not recognised

After a call to cancel that returns `FAIL`, if the `sendDone` message still hasn't been sent then it will run as usual. After a call to cancel that returns `SUCCESS`, the `sendDone` message will be called on the network layer with a status of:

- **SUCCESS** - If the cancel request to the hardware wasn't processed in time and the message was actually sent
- **ECANCEL** - If the message was successfully cancelled

2.5.7 Receiving Messages

When a data link layer receives a message addressed to it, it will pass the message on to the network layer via the `upFromDataLink` method (see figure 2.4, code listing 18 and table 2.5). Into this method, the data link layer will pass a reference to the DataLinkLayer class corresponding to the data link that received the message, a pointer to the message buffer that contains the received message, and the length of the network layer payload within the message buffer (this isn't a necessary parameter, since the same value can be gotten by calling `linkLayer->`

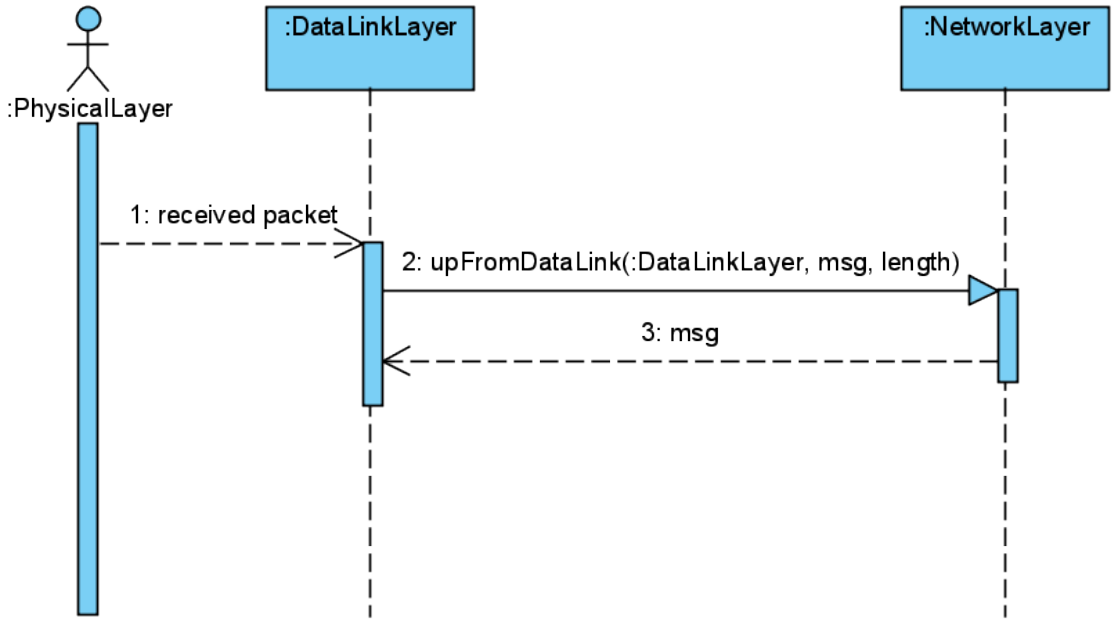


Figure 2.4: Sequence diagram illustrating how messages are received in SensorOS

`getPayloadLength(msg)`, but it's there for convenience). As mentioned in section 2.5.3, the other information in the frame can be retrieved with the relevant methods of the DataLinkLayer interface.

The `upFromDataLink` method is *called from within an atomic section* to preserve the integrity of the message buffer (the data link will normally have only a single message buffer for storing received messages), thus **the method must return as quickly as possible**. It should be noted that the return value of the method is a message buffer pointer. The reasoning behind this, is the method **must** return a pointer to a free message buffer that the data link layer can then use for the next message it receives. This leaves the network layer with three options for the `upFromDataLink` return value:

1. Return `msg` without touching it (i.e. drop the packet)
2. Copy out the network packet from the message buffer and return `msg`
3. Store `msg` for later processing (possibly posting a task to the scheduler to do this, see section 2.3.1) and return a (different) free message buffer pointer

It is **critically important** that the return value from the `upFromDataLink` method return value is a valid, free message buffer pointer, otherwise the system could become corrupted.

This concept of returning a free message buffer is borrowed from a similar concept in TinyOS 2. The main advantage it gives is summarised by Levis (2008):

> "This approach enforces an equilibrium between upper and lower packet layers. If an upper layer cannot handle packets as quickly as they are arriving, it still has to return a valid buffer to the lower layer. This buffer could be the msg parameter passed to it: it just returns the buffer it was given without looking at it. Following this policy means that a data-rate mismatch in an upper-level component will be isolated to that component. It will drop packets, but it will not prevent other components from

receiving packets. If an upper layer did not have to return a buffer immediately, then when an upper layer cannot handle packets quickly enough it will end up holding all of them, starving lower layers and possibly preventing packet reception."

2.5.8 Higher Layers

As mentioned in section 2.5.1, SensorOS is very flexible about what happens after a message is passed to the network layer from the data link layer, or about how the messages are passed to the network layer from higher layers. The application developer is left to their own devices to create an appropriate interface to the class implementing NetworkLayer. For instance, this can be in the form of:

- A full protocol stack (OSI, TCP/IP etc.)
- A simple interface for the application layer to send messages
- Sophisticated methods that provide particular services to higher layers
- A proxy class for a number of different network layers

For more comments about the application layer, see section 2.4.1.

```
1  class TestNetwork: implements NetworkLayer {
2  public:
3      TestNetwork() {
4          // Set all buffers as unused
5          used = 0xFF;
6      };
7      message_t* upFromDataLink(DataLinkLayer* linkLayer, message_t* msg,
8                                data_length_t length);
9      void sendDone(message_t* msg, error_t status) {
10         // Free the buffer used by msg
11         ptrdiff_t bufferId = (ptrdiff_t)(buffers - msg);
12         if (bufferId >=0 && bufferId <= 7) {
13             Atomic
14                 if (bufferId == 0) {
15                     used |= 0x1;
16                 } else {
17                     used |= 0x1 << bufferId;
18                 }
19             EndAtomic;
20         }
21         // ...
22     };
23 private:
24     // Array of message buffers
25     message_t buffers[8];
26     // Bit mask, a 1 indicates an unused buffer
27     uint8_t used;
28     /**
29      * Returns a pointer to a free message buffer
30      *
31      * The message buffer returned is reserved atomically
32      *  so it can't be used elsewhere
33      *
34      * @returns The message buffer pointer, or NULL if no
35      *  free buffers
36      */
37     message_t* getFreeBuffer() {
38         message_t* buffer = NULL;
39         uint8_t lowestFreeBuffer = 0;
40         // Extract lowest set bit
41         // http://realtimecollisiondetection.net/blog/?p=78
42         uint8_t lowestSetBit = used&-used;
43         uint8_t bit = lowestSetBit;
44         Atomic
45             // If there are any unused buffers
46             if (used > 0) {
47                 // Find log2 of lowestUnused to get the buffer number
48                 // Takes advantage of the fact that if only one
49                 //  bit is set in the integer, if the integer is 4 or less
50                 //  the log2 of it is itself shifted right one
51                 // For an 8-bit number, this loop will enter a maximum of
52                 //   two times
53                 while (bit > 0x4) {
54                     lowestFreeBuffer += 0x3;
55                     bit >>= 3;
```

Code Listing 36: Possible implementation for managing message buffers in NetworkLayer

```
                }
                lowestFreeBuffer += bit >> 1;
                // After extracting the lowest free buffer, claim it
                //  and set buffer appropriately
                used &= ~lowestSetBit;
                buffer = &buffers[lowestFreeBuffer];
            }
        EndAtomic;
        return buffer;
    };
};
```

Code Listing 37: Code listing 36 continued

2.6 Timers

The timer subsystem has not yet been implemented in SensorOS, however it will be implemented as part of a future version.

It has been decided to model the SensorOS timer subsystem on the TinyOS timer subsystem. The ability for TinyOS to pass in precision types at compile time isn't possible in C++ so a different way will need to be used for this. Also, TinyOS automatically allocates a unique timer id to a component that requests a timer via a nesC construct called unique, which takes a string and returns a unique number for every use of that string. A different technique will need to be devised for C++. One possible solution is to make use of a core operating system class which manages the timer hardware resources and classes would need to request the use of a timer and then release it when done. Unfortunately, a side effect of this is that if more than the number of hardware timers is requested then a run time error condition occurs and needs to be dealt with, whereas TinyOS throws a compile time error if this occurs. (Sharp et al. 2007)

3 Porting to Platforms

3.1 Overview

As outlined in section 1.1, one of the goals for SensorOS was to ensure that porting to different platforms was as easy as possible. This section is split up into three subsections:

1. A step-by-step guide to developing a SensorOS platform. Including a quick reference guide containing links to the relevant sections in the rest of the document
2. A discussion of important points when programming SensorOS platforms
3. A description of how SensorOS platforms can expose data links to applications

3.2 Creating a New Platform (Step-by-step)

1. A subdirectory needs to be created for the platform inside the `sensoros/platforms` directory (see section 1.3)
2. A `readme.txt` file needs to be created for the platform (see section 3.3.2)
3. A unit test should be created in `sensoros/unit_tests/<platform>/null` (see section 1.3) in order to test the basic platform functionality (and ability to compile) as it's created (see section 1.6 to see how to set up the project to be able to compile SensorOS), note that as extra classes are created (other than `Platform`, see section 2.3.8) unit tests should be created to fully test each class
4. If make is being used a Makefile needs to be created (see section 1.6.2)
5. A `platform.h` file needs to be created, the contents of the file should be (in order) (see section 3.3.1):
   (a) The platform definition constant needs to be defined
   (b) Any compiler constants need to be defined
   (c) The `NUM_DATA_LINKS` and `MAX_DATA_BYTES` constants need to be defined
   (d) `NULL`, `TRUE` and `FALSE` may need to be defined
   (e) The primitive SensorOS data types need to be defined
   (f) The **`link_id_t`**, **`data_length_t`**, **`priority_t`** and **`timestamp_t`** types need to be defined
   (g) The `size_t` type may need to be defined
   (h) The `Debug` class definition header file and any other files needed by the platform (e.g. register definitions, interrupt vector definitions) need to be included
   (i) The `application_network_info.h` application file needs to be included prior to all network related code
   (j) Definitions needs to be given for the `message_header_t`, `message_footer_t` and `message_metadata_t` unions (after the data link classes have been created)
   (k) The core `network.h` file needs to be included after the definitions of the above mentioned unions
   (l) Any data link class definition headers should be included after the `network.h` file
   (m) Definitions should be given for the `__disable_interrupts()` and `__enable_interrupts()` routines
   (n) Definitions should be given for the atomic section routines and **`atomic_t`** type
   (o) A definition should be given for the `Platform` class
6. A `platform.cpp` file should be created that at a minimum should define the methods for the `Platform` class (see section 2.3.8)
7. A definition header and implementation source file should be created for the `Debug` class (see section 2.3.9)

8. A definition header and implementation source file should be created for each type of data link available for the platform (see section 3.4
9. A `platform_signals.h` file needs to be created with a list of the signals the platform will use (see section 3.3.5)
10. If possible, the code should be tested under different compilers and operating systems to ensure that the code is compatible with as many development environments as possible (see section 3.3.3)
11. The `readme.txt` file and `Makefile` file should be kept up to date

3.3 Programming SensorOS Platforms

3.3.1 Platform Header (`platform.h`)

Overview The `platform.h` file is included into `main.h` and thus its definitions are exposed to every file in SensorOS (see sections 2.4.10 and 3.3.6). In order to fulfil all dependencies in the source, the order of statements in the `platform.h` file is very important. As such, the different "sections" of code in `platform.h` are represented by subsections within this section and placed in the same order that they need to appear in the `platform.h` file.

For an example `platform.h` file you can use `sensoros/platforms/dragon12/platform.h` as a reference.

Platform Definition Constant In order for applications to be able to conditionally compile platform targeted code (see section 2.4.3), each platform must define a platform constant. This constant must be named "`PLATFORM_`" followed by the directory name of the platform converted to uppercase. For instance, the definition for the dragon12 platform is **`#define`** `PLATFORM_DRAGON12`.

Compiler Constants As outlined in section 3.3.3, it may be necessary to define a compiler constant if there is only one supported compiler that doesn't have a compiler specific preprocessor constant. Furthermore, compiler specific settings may need to be enabled, for example for IAR EW on the dragon12 platform the **`#pragma`** `language=extended` command is issued to allow IAR language extensions.

Platform Constants All platforms need to define the following constants:

- `NUM_DATA_LINKS` - The maximum number of data links the platform can have
- `MAX_DATA_BYTES` - The maximum payload (in bytes) that can be transmitted by any data link (this will determine the size of the data section of the **`message_t`** type, see section 2.5.2)

Furthermore, if the compiler doesn't have definitions for `NULL`, `TRUE` or `FALSE` then these should be defined.

Where possible, constants should be defined using the **`enum`** construct to avoid using preprocessor macros (which can have side effects).

Primitive Types The basic SensorOS types as outlined in section 1.4.3 need to be defined. For some compilers this is possible by simply including the `inttypes.h` library file.

Platform Types Typedefs need to be given for the following types:

- **`link_id_t`** - Unsigned, needs to be large enough to hold `NUM_DATA_LINKS`
- **`data_length_t`** - Unsigned, needs to be large enough to hold `MAX_DATA_BYTES`
- **`priority_t`** - Unsigned, needs to be large enough to hold all data link message priorities
- **`timestamp_t`** - Unsigned, needs to be large enough to hold all data link timestamps

If the `size_t` type isn't defined without including compiler libraries (and those libraries aren't included) then the `size_t` type also needs to be defined.

Includes Any non-network related header files should now be included, this would incorporate the header file for the `Debug` class (see section 2.3.9). Examples of other header files would be definition files for register constants and interrupt vector addresses.

Networking The `application_network_info.h` application file needs to be included prior to all network related code (see section 2.5.4). After this include, definitions should be given for the header, footer and metadata structures for every type of data link a platform has. For example, the dragon12 has only the CAN data link, which has the definitions shown in code listing 38.

```
1  // MSCAN Message Buffer
2  typedef struct {
3      node_address_t source;
4      node_address_t destination;
5  } mscan_header_t;
6  typedef struct {
7      data_length_t length;
8      priority_t priority;
9      timestamp_t timestamp;
10 } mscan_footer_t;
11 typedef struct {} mscan_metadata_t;
```

Code Listing 38: Definition of the header, footer and metadata structures for the CAN data link in the dragon12 platform

Following these definitions, the `message_header_t`, `message_footer_t` and `message_metadata_t` unions need to be defined (see section 2.5.2). The definition for the dragon12 platform is shown in code listing 39.

```
1  // All Message Buffers (Used in conjunction with sizeof() in
2  //  message_t definition)
3  typedef union message_header {
4      mscan_header_t mscan;
5  } message_header_t;
6  typedef union message_footer {
7      mscan_footer_t mscan;
8  } message_footer_t;
9  typedef union message_metadata {
10     mscan_metadata_t mscan;
11 } message_metadata_t;
```

Code Listing 39: Definition of the `message_header_t`, `message_footer_t` and `message_metadata_t` unions for the dragon12 platform

Finally, the core `network.h` file needs to be included, followed by the header definition files for all data link layer classes.

Interrupt Control Definitions should be given for the `enable_interrupts()` and `disable_interrupts()` commands (see section 2.4.7).

Atomic Sections Definitions should be given for the **`atomic_t`** type and the `__atomic_start()` and `__atomic_end()` routines as outlined in section 3.3.7.

Platform Class A definition should be given for the `Platform` class (see section 2.3.8).

3.3.2 Readme File

As outlined in section 2.4.2; all platforms and applications should have a `readme.txt` file within their directory, which should be edited in Markdown syntax. The platform readme file should contain the following sections:

- **Platform** - The name of the platform
- **Description** - A brief description of the platform being supported
- **Author** - The authors name(s) as links to their email addresses
- **Date** - The date the platform was last modified
- **Version** - The version of the platform
- **Processor** - The processor(s) the platform is developed for
- **Compilers** - A list of the compilers that the platform is compatible with and any notes about how to set up the compiler for compilation with that platform
- **Notes** - Any points of note about the platform, in particular any documentation on any aspects of the platform requiring application code to function (see section 2.4.3) and documentation on how the Debug string printing functions work
- **License** - The software license you are releasing the platform under

An example `readme.txt` file is shown below:

```
1  Platform
2  ========
3
4  dragon12
5
6  Description
7  ===========
8
9  SensorOS platform for the [Wytec Dragon12 Development
10 Board](http://www.evbplus.com/dragon12_hc12_68hc12_9s12_hcs12.html)
11
12 Author
13 ======
14
15 [Robert Moore](mailto:rob@mooredesign.com.au)
16
17 Date
18 ====
19
20 2009-09-27
21
22 Version
23 =======
24
25 1.0.0
```

```
26
27  Processor
28  =========
29
30  [Freescale 9S12/HCS12](http://www.freescale.com/webapp/sps/site/
31  overview.jsp?nodeId=06258A25802570256D)
32
33  Compilers
34  =========
35
36  IAR Embedded Workbench (EW) and GCC compatible
37
38  When compiling with IAR EW, the following files need to be added
39  to the project:
40
41  * sensoros/os/main.cpp
42  * sensoros/os/network.cpp
43  * sensoros/os/scheduler.cpp
44  * sensoros/os/signal_router.cpp
45  * sensoros/os/signal_router.cpp
46  * sensoros/platforms/dragon12/can_link.cpp
47  * sensoros/platforms/dragon12/debug.cpp
48  * sensoros/platforms/dragon12/platform.cpp
49  * sensoros/platforms/dragon12/iar.s12
50  * Any application source files
51
52  And "Additional include directories" need to be added (Project >
53  C/C++ Compiler > Preprocessor) as (assuming the project resides in
54  a subdirectory of the application's directory (e.g. sensoros/
55  applications/<application_name>/iar/)):
56
57  * $PROJ_DIR$\..\..\..\os
58  * $PROJ_DIR$\..\..\..\platforms\dragon12
59  * $PROJ_DIR$\..
60
61  Notes
62  =====
63
64  There wasn't any macros defined by IAR EW, so in platform.h
65  __IAR__ is set if the compiler is not GCC. If the platform is
66  extended to include any more compilers then the setting of __IAR__
67  will need to be modified.
68
69  If an application wants to use any of the 5 CAN modules then it
70  should define CAN<X> (where X is 0 - 4, and refers to CAN module
71  X) as the link id number that the corresponding CAN module should
72  be. These definitions need to go in application_network_info.h and
73  should obviously be used as a conditional compilation for this
74  platform only, e.g.:
75
76          #ifdef PLATFORM_DRAGON12
```

```
77                  #define MSCAN0 1 // Use CAN module 0 as data link 1
78                  #define MSCAN3 2 // Use CAN module 3 as data link 2
79          #endif
80
81  Note: you must set the data link ids in ascending, incremental
82  order or the assignment of the data link ids to the network will fail.
83
84  The debug class will print strings to SCI0 at a baud of 19200 with 1
85  start bit, 8 data bits, 1 stop bit and no parity bits.
86
87  License
88  =======
89
90  MIT
91  ---
92
93  Copyright (c) 2009 Robert Moore
94
95  Permission is hereby granted, free of charge, to any person
96  obtaining a copy of this software and associated documentation
97  files (the "Software"), to deal in the Software without
98  restriction, including without limitation the rights to use,
99  copy, modify, merge, publish, distribute, sublicense, and/or sell
100 copies of the Software, and to permit persons to whom the
101 Software is furnished to do so, subject to the following
102 conditions:
103
104 The above copyright notice and this permission notice shall be
105 included in all copies or substantial portions of the Software.
106
107 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
108 EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
109 OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
110 NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
111 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
112 WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
113 FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
114 OTHER DEALINGS IN THE SOFTWARE.
```

At this stage there is no automated documentation process for platforms and applications, however in a future version of SensorOS there will be a `sensoros/docs` directory with generated documentation for platforms and applications from the `readme.txt` and the source code (using Doxygen).

3.3.3 Compiler Compatibility

In order to facilitate the goal of maximum development platform compatibility (see section 1.1), platforms should ensure that their code is compiler independent where possible, and for code that needs to be compiler specific conditional compilation should be used. This would be done using preprocessor constants that a particular compiler defines, for example GCC defines the constant `__GNUC__`. If a particular compiler doesn't define any constants then there are two

options:

- If the compiler is the only supported compiler with this problem, then a custom constant could be defined if none of the other supported compilers' constants are defined (see `sensoros/platforms/dragon12/platform.h`'s definition of `__IAR__` for an example)
- A preprocessor constant can be defined in the command line call to the compiler, or via appropriate dialogs inside an IDE

For example, the dragon12 platform was made compatible with IAR Embedded Workbench (EW) and GCC. The following are defined per compiler with conditional compilation in that platform:

- Register definitions: IAR EW has a special syntax with an symbol to declare address of variables, whereas GCC allows casting of the address as a pointer and dereferencing that pointer, see `sensoros/platforms/dragon12/hcs12_registers.h`
- Interrupt handlers: IAR EW requires the use of a **`#pragma`** declaration, whereas GCC allows a cast to determine the location of the function, see `sensoros/platforms/dragon12/hcs12_vectors.h` and `sensoros/platforms/dragon12/can_link.cpp`
- Definition of `__atomic_start()` and `__atomic_end()`: GCC allows sophisticated inline assembler code that allows these routines to be defined within the C++ code (see `sensoros/platforms/dragon12/platform.cpp`), whereas in IAR EW it was necessary to define a custom assembler file (see `sensoros/platforms/dragon12/iar.s12`)

3.3.4 Interrupt Service Routines

This section outlines how interrupt service routines (ISRs) are defined and created for the dragon12 platform. The approach presented may be useful for other platforms.

One of the types defined in the `sensoros/platforms/dragon12/hcs12_vectors.h` file is `interrupt_t` which is set to the return type of an interrupt service routine (ISR). Furthermore, a `SET_ISR` macro is defined which takes the name of the interrupt and the name of the ISR and ensures a pointer to the ISR is set up in the interrupt vector table such that the ISR is called when the particular vector is called. This approach works well for processors with vectored interrupt handlers. A different approach may be needed for other interrupt handler set ups.

The definition of `interrupt_t` and `SET_ISR` is given in code listing 40.

An example of the definition of an ISR for this platform is given in code listing 41. In this example, the code for CAN link 4 successfully sending a message is defined. `can4Link` is an instance of the CANLink class for CAN module 4. There is a conditionally compiled **`#pragma`** definition for IAR EW, since it needs this to define ISRs (hence why `SET_ISR` is blank for IAR EW). The last thing to explain is `MSCAN4TX_VECTOR`, which is simply set the numerical value of the vector address for the `MSCAN4TX` interrupt (see `sensoros/platforms/dragon12/hcs12_vectors.h` for its definition).

3.3.5 Platform Signals

Platform code can utilise the core signal handling functionality (see the SignalRouter class description in section 2.3.2) by defining a number of signals that the platform can use. These signals are defined in the `platform_signals.h` file within the platform's directory. This file

```
1  // Interrupt handler typedef: GNU
2  #ifdef __GNUC__
3      #define  interrupt_t void __attribute__((interrupt))
4      // Define macro to allow ISR's to be registered
5      // http://www.embedded.com/columns/programmingpointers/192503651
6      typedef void (*pointer_to_ISR)(void);
7      #define SET_ISR(VECTOR, ISR) *reinterpret_cast<pointer_to_ISR *>(VECTOR
           ) = ISR
8  #endif
9  // Interrupt Handler typedef: IAR
10 #ifdef __IAR__
11     #define interrupt_t __interrupt void
12     // IAR uses a #pragma to define vector addresses, make SET_ISR to
           nothing
13     #define SET_ISR(VECTOR, ISR)
14 #endif
```

Code Listing 40: Definition of `interrupt_t` and `SET_ISR` for dragon12 platform

```
1  #ifdef __IAR__
2      #pragma vector=MSCAN4TX_VECTOR
3  #endif
4  interrupt_t can4TXISR(void) {
5      // Signal a CAN 4 Send event
6      SignalRouter::getInstance()->signal(SIG_CAN4_SENT);
7      // Call the sendDone method for this link
8      can4Link.sendDone();
9  }
10 SET_ISR(MSCAN4TX_VECTOR, can4TXISR)
```

Code Listing 41: Definition of an ISR for dragon12 platform

simply contains a list of signals separated by commas (any whitespace can appear before and after the commas). The signals should all begin with `SIG_` and be uppercase and with underscores between words (as outlined by the naming convention in section 1.4.1 for constants).

3.3.6 Main Include

In order to expose a source file to SensorOS, you need to add **`#include`**`<main.h>` to the start of the .cpp file. The corresponding .h definition file for a class should be included in the `platform.h` file. This is different from the usual C++ practice of including the .h file in the .cpp file and then the .h file includes any necessary functionality the class needs. It is necessary to do it this way though to avoid circular dependency issues so that all the SensorOS types and classes are included in the correct order for all classes.

If there are any header files that need to be included that contain functionality only used in a particular source file only, then it should be included from within that source file rather than the `platform.h` file.

3.3.7 Atomic Sections

As shown in section 2.4.6, atomic sections consist of assigning a variable (`atomicValue`) of type **`atomic_t`** to the result of calling `__atomic_start()`, followed by the code inside the atomic

section and finally the call to `__atomic_end()` (with `atomicValue` passed into that routine). The return value from `__atomic_start()` must contain information about whether interrupts where enabled when that routine was called (either way, when the routine returns, interrupts must be disabled), and then `__atomic_end()` uses this information to determine whether interrupts should be re-enabled (only if they were enabled before `__atomic_start()` was called).

The definition for **`atomic_t`**, `__atomic_start()` and `__atomic_end()` must be inside `platform.h`, and the implementation of the routines must be present as part of the platform source code. For examples see `sensoros/platforms/dragon12/platform.h` (for the definitions), `sensoros/platforms/dragon12/platform.cpp` (for the GCC implementation) and `sensoros/platforms/dragon12/iar.s12` (for the IAR EW implementation).

3.4 Networking

3.4.1 Overview

Section 2.5 provides an overview of how networking is performed from the perspective of a SensorOS application. The platform perspective for networking revolves around the exposure of the `DataLinkLayer` interface (see section 2.3.4) to applications (more specifically the network layer via a class implementing the `NetworkLayer` interface, see section 2.3.5). The platform should implement a class for each type of data link layer present in the hardware platform.

The following subsections detail some of the finer points that need to be taken into consideration when writing data link layer classes.

3.4.2 Header and Footer Access

As outlined in section 2.5.2, headers and possibly footers will be stored contiguously with the network packet stored in the data section of the **`message_t`** type. In order for the data link layer class to be able to set and get fields from the header / footer it needs to get a reference to its header / footer data structure by using a relative pointer reference from the data section of the message buffer. In order to facilitate this operation, a number of macros are defined in the core `network.h` file. Code listing 42 gives the definition of these macros.

```
1 #define MESSAGE_HEADER(msg, type) ((type*)((msg)->data-sizeof(type)))
2 #define MESSAGE_FOOTER(msg, type) ((type*)((msg)->footer))
3 #define MESSAGE_FOOTER_CONTIGUOUS(msg, type) ((type*)((msg)->footer)-
    getPayloadLength(msg))
4 #define MESSAGE_METADATA(msg, type) ((type*)((msg)->metadata))
```

Code Listing 42: Definition of the message buffer access macros

Code listing 43 outlines an example usage of the `MESSAGE_HEADER` macro for the dragon12 platform `CanLink` class. The `MESSAGE_FOOTER` and `MESSAGE_METADATA` macros function in the same way. The `MESSAGE_FOOTER_CONTIGUOUS` macro allows the footer to be stored contiguously with the network packet stored in the data region (even if it's smaller than the size of the data section of **`message_t`**), but it uses `getPayloadLength`. This means it must be used inside a method of a class implementing `DataLinkLayer` (it should be anyway) and whatever requirements for `getPayloadLength` to return the correct length must be fulfilled for the message buffer in question.

```
1 /**
2  * Returns the source address of the message in the given message buffer
3  * @return Message source
4  */
5 node_address_t CanLink::getSource(message_t* msg) {
6
7     return MESSAGE_HEADER(msg, mscan_header_t)->source;
8 }
```

Code Listing 43: Example usage of the `MESSAGE_HEADER` message buffer access macro

3.4.3 Sending Messages

The data link layer class must meet the semantics of interacting with the network layer class for sending messages and then notifying the network layer when sending is complete (via its `sendDone` method) as described in sections 2.5.5 and 2.5.6. In order to do this the class will need to be able to store the message buffer pointers for messages it is queuing to send so that it can:

- Keep a reference of the message data it needs to send until it is actually sent
- Facilitate cancellation of messages (by accepting a message buffer pointer of the message to cancel)
- Notify the network layer class which message has finished sending (by passing across a message buffer pointer)

The data link class can have an internal buffer for queuing of messages for sending, or it can utilise whatever hardware sending queue is available, or there can be no queue and it can encapsulate the sending of a single message at a time. Either way, enough storage should exist in the class to store the appropriate number of message buffer pointers.

3.4.4 Receiving Messages

In order to facilitate the message buffer semantics associated with the `upFromDataLink` method in the `NetworkLayer` interface (see section 2.5.7), each data link class will need to have an instance variable that can hold a message buffer, which then acts as the initial receive buffer. Separate to this, a pointer to a receive message buffer will need to be kept that initially points to the aforementioned message buffer, but is changed to the return value from `upFromDataLink` after the receive buffer is filled and passed to that method. The initial receive buffer is never referenced apart from when the receive buffer pointer is initialised. When a message is received, the message buffer pointed to by the receive buffer pointer should be filled and then passed to the network layer.

3.4.5 Endianness

If a sensor network is to be formed from a range of platforms, the possibility of conflicting endianness can be realised with network packet exchange. This problem is taken care of in TinyOS by using the external types nesC construct (more specifically, adding an `nx_` prefix to types ensures they are big endian and aligned to byte boundaries (Levis 2007)). Such a construct is not possible with C++, and as such any platform that is not big endian should ensure that all packets are sent in big endian format, and received packets are extracted back into a format compatible with that processor.

References

Akyildiz, I. F., W. Su, Y. Sankarasubramaniam, and E. Cayirci. 2002. A survey on sensor networks. *IEEE communications magazine* 40 (8): 102–114.

Downs, K., S. Spanier, M. Ford, T. Stevenson, and H. Lew. 1998. *Internetworking technologies handbook*. Cisco Press.

FOLDOC. 2002. race condition. http://foldoc.org/race+condition, (accessed 26 September 2009).

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

IAR Systems. n.d. Extended embedded C++. http://iar.com/website1/1.0.1.0/467/1/, (accessed August 30, 2009).

Levis, P. 2006. TinyOS programming. http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf, (accessed 21 February 2009).

Levis, P. 2007. TEP 111: message_t. http://www.tinyos.net/tinyos-2.x/doc/html/tep111.html, (accessed September 25, 2009).

Levis, P. 2008. TEP 116: Packet protocols. http://www.tinyos.net/tinyos-2.x/doc/html/tep116.html, (accessed September 25, 2009).

Plauger, P. 1997. Embedded C++: An Overview. *Embedded Systems Programming* 10: 40–53.

Qi, H., S. S. Iyengar, and K. Chakrabarty. 2001. Distributed sensor networks - a review of recent research. *Journal of the Franklin Institute* 338 (6): 655–668.

Rinard, M. C. 1998. Lecture 5: Implementing synchronization operations. http://www.cag.csail.mit.edu/~rinard/osnotes/h5.html, (accessed 26 September 2009).

Rios, J. L. 2005. Using interfaces in C++. http://www.codeguru.com/cpp/cpp/cpp_mfc/oop/article.php/c9989/, (accessed 6 April 2009).

Saks, D. 1998. Static vs. dynamic initialization. *Embedded Systems Programming* 11: 19–22.

Saks, D. 2008. The yin and yang of dynamic allocation. *Embedded Systems Design* 21 (5): 12.

Sharp, C., M. Turon, and D. Gay. 2007. TEP 102: Timers. http://www.tinyos.net/tinyos-2.x/doc/html/tep102.html, (accessed May 28, 2009).

Silicon Graphics, Inc.. 2009. Function objects. http://www.sgi.com/tech/stl/functors.html, (accessed 17 September 2009).

Sinopoli, B., C. Sharp, L. Schenato, S. Schaffert, and S. S. Sastry. 2003. Distributed control applications within sensor networks. *Proceedings of the IEEE* 91 (8): 1235–1246.

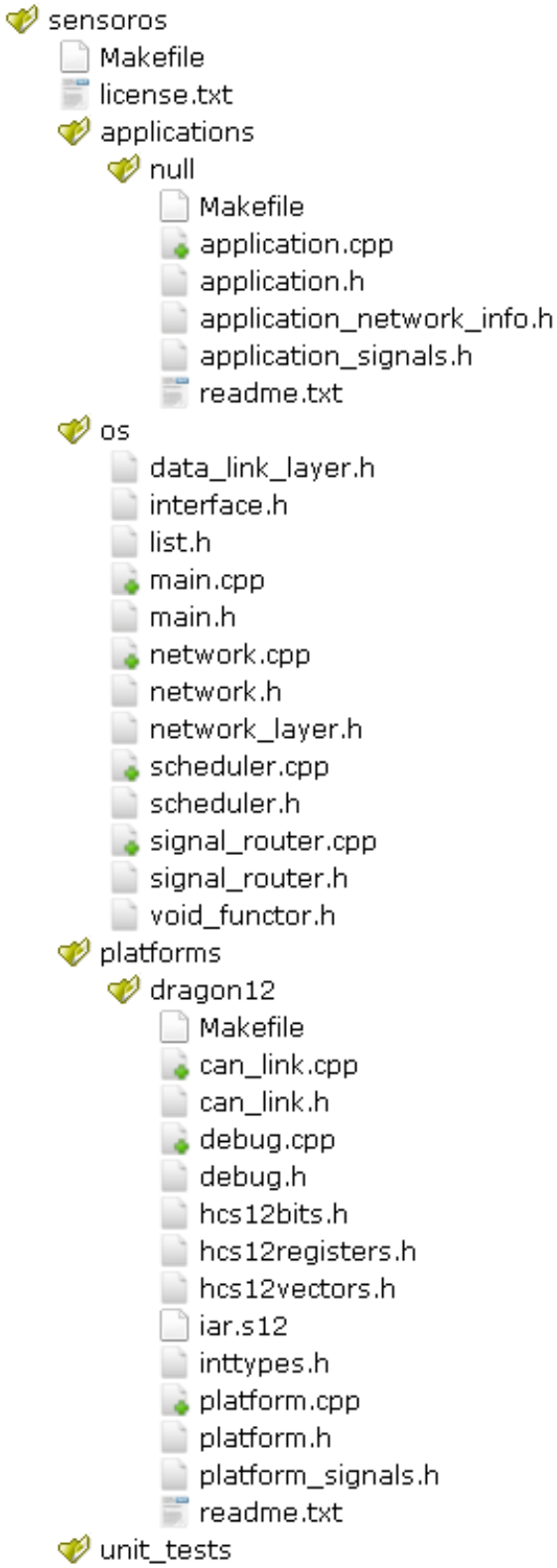
Tolle, G., P. Levis, and D. Gay. 2008. TEP 114: SIDs: Source and sink independent drivers. http://www.tinyos.net/tinyos-2.x/doc/html/tep114.html, (accessed September 26, 2009).

Xu, N. 2002. A survey of sensor network applications. *IEEE communications magazine* 40 (8): 102–114.

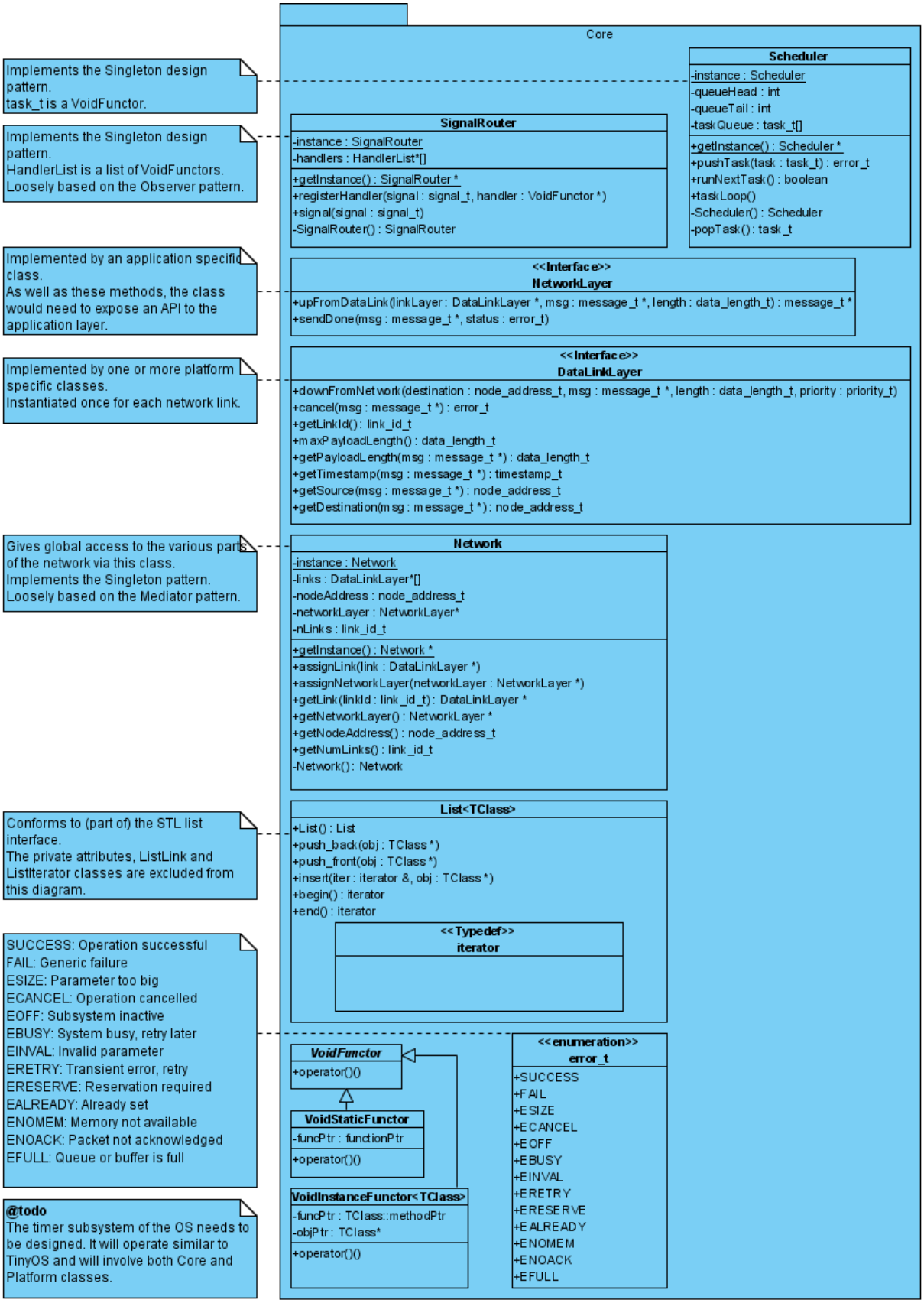
Yannopoulos, I. and D. Gay. 2006. TEP 3: Coding standard. http://www.tinyos.net/tinyos-2.x/doc/html/tep3.html, (accessed August 30, 2009).

Appendix A: File Structure

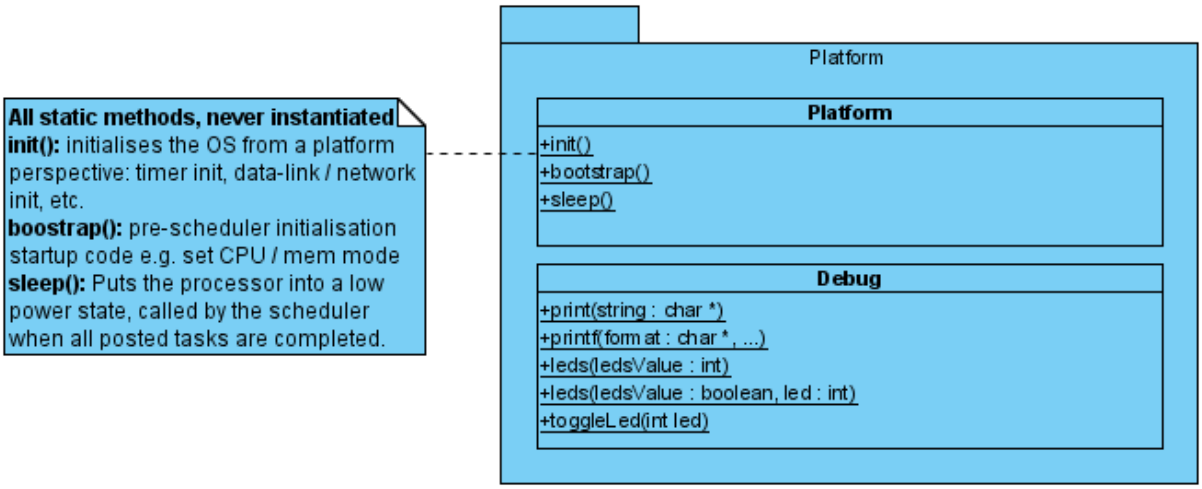
Note: the directories and files within the `unit_tests` directory have been omitted for brevity.



Appendix B: SensorOS.Core - UML Class Diagram



Appendix C: SensorOS.Platform - UML Class Diagram



Appendix D: SensorOS.Application - UML Class Diagram