Curtin
University of Technology

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SensorOS: Cross-Platform Operating System for Sensor Networks

by

Robert Daniel Moore

A thesis submitted for the degree of

Bachelor of Engineering in Computer Systems Engineering

Curtin University of Technology

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

TITLE: SensorOS: Cross-Platform Operating System for Sensor Networks

AUTHOR: Robert Daniel Moore

FAMILY NAME: Moore

GIVEN NAMES: Robert Daniel

DATE: October 30, 2009	SUPERVISOR: Clive Maynard
DEGREE: Bachelor of Engineering	OPTION: Computer Systems Engineering

ABSTRACT:

Smaller classes of unmanned aerial vehicles have reliability problems that result in high crash rates. A contributing factor to this is the fact that these UAVs have a lack of sophisticated sensors and a single point of failure for any sensors they do have. A unique sensor network has been proposed by the Defence Science and Technology Organisation to address this problem by increasing the reliability of the sensing and control subsystems. This project aims to design and implement a control system for this sensor network.

As part of this aim, an operating system targeted at sensor networks was developed in C++ based on the TinyOS operating system. It was designed to encompass many of the useful patterns and abstractions in TinyOS, but have a focus on compatibility between different development environments, easy portability between said environments and better structure for networking (including explicit support for multiple data-link layers).

INDEXING TERMS:

Sensor Network, Wireless Sensor Network, Distributed Network, TinyOS, Operating System, OS, SensorOS, Sensors, nesC, C++, Structural Health Monitoring, Dragon12, HCS12, MC9S12DG256B, Unmanned Aerial Vehicle, UAV, Data Fusion, Data Aggregation, Data Dissemination.

	GOOD	AVERAGE	POOR
TECHNICAL WORK			
REPORT PRESENTATION			

EXAMINER

CO-EXAMINER

SYNOPSIS

Smaller classes of unmanned aerial vehicles have reliability problems that result in high crash rates. A contributing factor to this is the fact that these UAVs have a lack of sophisticated sensors and a single point of failure for any sensors they do have. A unique sensor network has been proposed by the Defence Science and Technology Organisation to address this problem by increasing the reliability of the sensing and control subsystems. This project aims to design and implement a control system for this sensor network.

As part of this aim, an operating system targeted at sensor networks was developed in C++ based on the TinyOS operating system. It was designed to encompass many of the useful patterns and abstractions in TinyOS, but have a focus on compatibility between different development environments, easy portability between said environments and better structure for networking (including explicit support for multiple data-link layers).

Robert Daniel Moore
4A Logue Court
BALGA WA 6061

October 30, 2009

Professor Syed Islam
Head of Department
Department of Computer Engineering
Curtin University of Technology
GPO BOX U1987
PERTH WA 6845

Dear Professor Islam

I hereby offer this thesis titled *"SensorOS: Cross-Platform Operating System for Sensor Networks"* in partial satisfaction of the requirements of the degree of Bachelor of Engineering in Computer Systems Engineering.

I declare that the work contained within is entirely my own work except where stated or acknowledged otherwise.

Yours Sincerely

Robert Daniel Moore

ACKNOWLEDGEMENTS

The author would like to thank the project supervisor Clive Maynard, and the co-supervisor Cesar Ortega-Sanchez for their support, guidance and technical advice. Furthermore, the author would like to thank Stephen van der Velden from the DSTO for providing the opportunity to work on this project.

Thanks should also be given to Anthony Lawther, Helen Moore, Stephen Moore, Ryan Musgrave, James Pearce, Nigel Scott, Mariel Wallace and Kim Wisniewski for their proof reading assistance.

INDEX PAGE

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

1.1 Motivation for work

1.1.1 Unmanned aerial vehicles

Unmanned aerial vehicles (UAVs) are aircraft that can fly without a human crew. UAVs can be remote controlled, or have full or partial automation of control. The lack of human crew means UAVs are useful in dangerous, dirty or dull environments ("D-cube" missions; (Pastor Llorens et al. 2007)). Consequently, they have numerous military and civil applications including (Barrows 2002; Sarris 2001; Pastor Llorens et al. 2007):

- Military applications
  - Reconnaissance Surveillance and Target Acquisition (RSTA)
  - Surveillance for peacetime and combat Synthetic Aperture Radar (SAR)
  - Deception operations
  - Maritime operations (Naval fire support, over the horizon targeting, anti-ship missile defence, ship classification)
  - Electronic Warfare (EW) and SIGINT (SIGnals INTelligence)
  - Special and psyops
  - Meteorology missions
  - Route and landing reconnaissance support
  - Adjustment of indirect fire and Close Air Support (CAS)
  - Battle Damage Assessment (BDA)
- Environmental applications
  - Environmental research and monitoring
  - Pollution assessment and monitoring
  - Oceanography
- Emergency / security / monitoring applications
  - Fire-fighting

    - Border patrol
    - Search and rescue
    - Disaster and emergency management
- Industrial applications
    - Crop spraying and surveillance
    - Nuclear factory surveillance
- Communications relay

Historically UAVs have been used mainly for military applications, having been under development by the United States (U.S.) Department of Defence since the late 1950s (Schneider 2004, 17). Civil applications are becoming a reality with the development of smaller UAV classes with lower manufacturing and operational costs (Barrows 2002; Sarris 2001; Pastor Llorens et al. 2007).

1.1.2 UAV reliability problems

The U.S. Department of Defence recognises value in widespread adoption of UAV technology (Schneider 2004, 5); however, they are not widely used by the U.S. military services for a number of reasons including (Schneider 2004, 6):

> "[T]hese new systems cost more than anticipated, suffer from high accident rates because of subsystem reliability and operator error, and lack the combat survivability features of manned aircraft."

One of the main problems with subsystem reliability in UAVs is a lack of sensory capability, data amalgamation and decision making ability. This is combined with the fact that the sophisticated redundancies and sensors built into the sensor and control systems of manned aircraft cannot be used due to the cost, size and weight constraints associated with UAVs. Another contributing factor is that any sensors that UAVs do contain will typically have a single point of failure. (van der Velden et al. 2007)

This problem is starting to be addressed with the development of larger military UAVs with more reliable systems including manned aircraft quality components. These are viable since the cost and weight constraints are more relaxed in these UAVs. (Schneider 2004, 17-19)

However, the reliability problems with smaller classes of UAVs such as mini (or small) / micro sized UAVs[1] cannot be solved in this way because of the aforementioned cost and weight constraints. It should be noted that civil applications are more likely to use these smaller UAVs. (Barrows 2002; Pastor Llorens et al. 2007; van der Velden et al. 2007)

1.1.3 DSTO sensor network solution

Thus a solution is needed to improve the reliability of the sensing and control systems in smaller UAVs while taking the cost and weight constraints into consideration. The Defence Science and Technology Organisation (DSTO) have proposed a possible solution that involves the creation of a sensor network on the UAVs consisting of special micro-electro-mechanical system (MEMS) sensors (van der Velden et al. 2007; van der Velden and Powlesland 2008). More information about this system can be found in section 2.0.

1.2 Project scope

The aim of this project was to research and implement a control system for the sensor network mentioned in section 1.1.3. As part of this aim an operating system needed to be found on which to run the control system.

[1]According to regulation 101.240 the Australian Civil Aviation Safety Regulations 1998 a micro UAV is a UAV with a gross weight of 100 grams or less, a large UAV has a launch mass of 150 kilograms or more and a small UAV is not a large UAV nor a micro UAV. (Civil Aviation Authority 1998)

1.3 Project outcomes

The primary outcomes of this project are two-fold:

1. Preliminary research and design of the control system mentioned in section 1.2 has been undertaken, resulting in a requirements specification and categorised reading list.
2. Preliminary implementation has been undertaken resulting in an operating system with the goals of this particular control system in mind. The operating system is suitably generic and flexible such that it can also be used for applications other than the one for which it was designed.

The operating system has been created against a single hardware platform, but the ability to extend it to other platforms exists. A programming manual has been created to assist development of applications for the operating system and porting different hardware platforms to it.

A secondary outcome of this project is a comprehensive review of techniques and algorithms researched and developed for sensor networks from the initial research into these types of networks through to the present day. This review is provided in section 3.0 of this thesis.

1.4 Thesis outline

The purpose of this document is to outline the outcomes for this project including justification of decisions made and difficulties encountered.

It demonstrates that appropriate engineering practices and principles have been adopted throughout the work that was completed and suggests future work that could be implemented.

The section breakdown of this thesis is as follows:

Section 2.0	Provides an outline of the sensor network mentioned in section 1.1.3.
Section 3.0	Provides an overview of the research that was conducted into sensor networks to identify trends, best practices and current research.
Section 4.0	Covers the motivation, approach and outcome of the work to create a system requirements specification for the control system this project focused on.
Section 5.0	Describes the need to secure an appropriate operating system for the control system to run on, and the progress made in regards to that need.
Section 6.0	Provides a summary of the outcomes of this project and outlines some suggestions for future work that could be undertaken.

The thesis concludes with a number of appendices; the system requirements specification (Appendix A), the categorised reading list (Appendix B), some example nesC code snippets (Appendix C), the operating system programming manual (Appendix D) and a description of the digital files that accompany this thesis (Appendix E).

2.0 STRUCTURAL HEALTH MONITORING SENSOR NETWORK FOR MINI / MICRO UAVS

2.1 Multi Sensitivity Redundant Node (MSRN)

Most off the shelf MEMS sensors are sensitive to a single physical parameter and use compensation to minimise error from other parameters. Packaged sensors are typically around 5mm x 5mm x 2mm in dimension, one gram in weight and exhibit power use in the order of milli Watts. (van der Velden et al. 2007)

van der Velden et al. (2007) outline the idea of a Multi Sensitivity Redundant Node (MSRN), which consists of an array of MEMS sensors on a single silicon chip. The proposed MEMS sensors for the MSRN are special due to the fact that they are multi-sensing, which means they can measure a number of different physical parameters simultaneously. Consequently, MSRNs could have sensitivity to some or all of:

- Acceleration
- Angular velocity
- Temperature
- Pressure
- Infrared intensity
- Magnetic flux
- Fluid flow

It is suggested that the MSRN chips will be around the same size and weight as a single-sensing, commercially available MEMS sensor and use less power. Furthermore, the MSRN structure will yield multiple streams of information that are more accurate and robust against damage than the single stream of information from a single-sensing MEMS sensor.

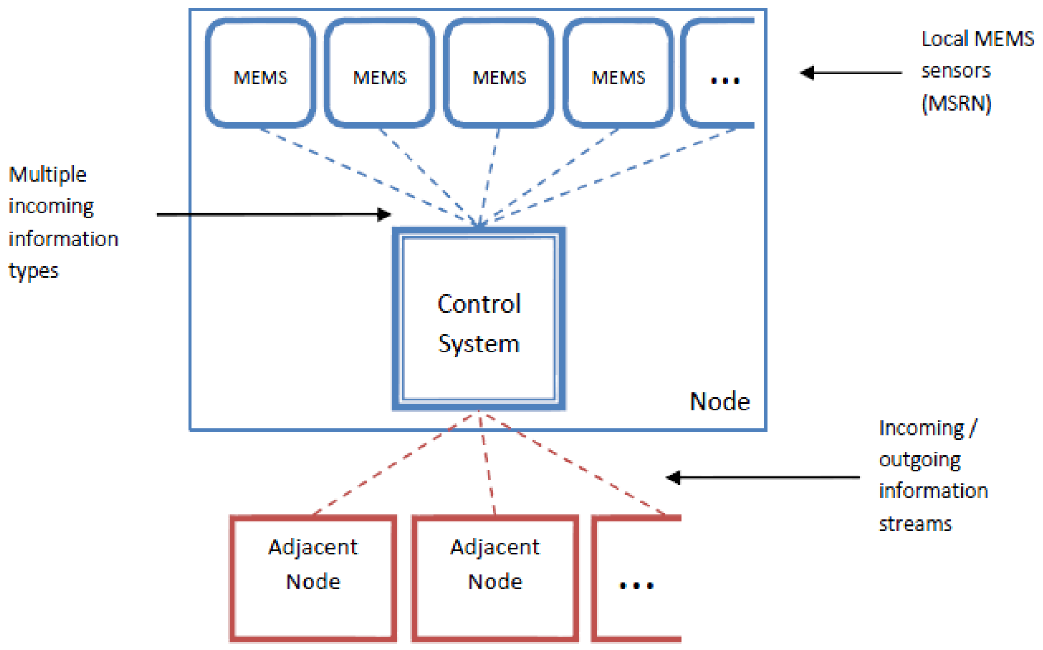

MEMS
MEMS
MEMS
MEMS
...
Local MEMS sensors (MSRN)
Multiple incoming information types
Control System
Node
Incoming / outgoing information streams
Adjacent Node
Adjacent Node
...

Figure 2.1: Block diagram of the proposed redundant sensor network from the perspective of a single node

The preliminary expectations of the sensing ability of the MSRN are outlined in Appendix A of the requirements specification contained within Appendix A of this thesis. It is envisaged that the MSRN will be able to sense a subset of these information types, the exact subset being configurable by the hardware controlling the MSRA (van der Velden, S. 2009, pers. comm., 3 February). (van der Velden et al. 2007)

2.2 Redundant sensor network

van der Velden and Powlesland (2008) propose that a redundant sensor network can be formed by combining each node in the network with a MSRN chip. A block diagram of what the system would look like from the perspective of a single node is shown in figure 2.1.

In order to promote redundancy within the network each node will be connected to a number of other nodes via point-to-point links. The redundancy within a node due to the MSRN and between nodes due to the multiplicity of links means that the network does not have a single point of failure. Consequently, the network is robust when nodes and / or links are damaged. (van der Velden and Powlesland 2008)

As well as participating in the sensor network it is proposed that each node could also participate in some or all of the following activities (van der Velden and Powlesland 2008):

- Actuation
- Computation
- Information storage
- Power generation
- Power storage
- Mission statement storage and interpretation

In order to keep the system robust these activities would be distributed across multiple nodes.

2.3 Physical layer

It is proposed that the point-to-point links will be used to transmit both power and data simultaneously. This facilitates flexible and robust power routing by utilising the structure of the network, allowing a reduction in the number of connections in the system without compromising data link multiplicity. Reducing the number of connections in turn reduces the vulnerability of the system to shorts and breaks. (van der Velden and Powlesland 2008)

A possible solution to this requirement is via an Acoustic Electric Feedthrough (AEF) connection (van der Velden, S. 2009, pers. comm., 2 February). This sort of connection allows power and data to travel across a physical medium (such as an aircraft bulkhead) via acoustic stress waves. (Moss et al. 2009)

The way this works is via two piezoelectric elements (made from Lead Zirconate Titanate (PZT)) wedged either side of a metallic plate. One PZT element will act as a transmitter when a charge is run through it. This in turn produces an acoustic stress wave that travels through the metal plate. This stress wave then causes a similar electric charge to be produced inside the receiving PZT element. (Moss et al. 2009)

2.4 Desired system characteristics

van der Velden and Powlesland (2008) outline the most important attributes the sensor network should exhibit:

- High robustness in the presence of damage, including graceful degradation where possible
- Medium network bandwidth
- Efficient power use
- Single physical path for both power and data

In order to reconfigure itself for optimum performance the system would need to take into account the available power and set the parameters for the sensors in order to maximise a set of predefined utility functions that assign importance to the degree of accuracy of the various parameters being sensed. (van der Velden and Powlesland 2008)

The following points were also made (van der Velden et al. 2007; van der Velden and Powlesland 2008):

- Sensor information transferred through the network should have believability, accuracy and temporal descriptors attached to it to improve robustness;
- To improve the robustness of the system, as well as transferring sensor information through the network, the network should also transfer control informa-

tion, routing information, power availability advertisements, power allocation requests and speculative advertising and information requests;

- Some sort of fault identification technique should be incorporated into the system to identify faulty sensors and discard their data; and
- The system will need to incorporate some sort of amalgamation scheme to combine the data streams from multiple sensors / nodes.

In order to allow interoperability with different hardware a goal of the system is that it is portable between different platforms. (van der Velden 2008, pers. comm., 25 August)

2.5 Application to mini / micro UAVs

The intended application of this system is for use in mini and micro sized UAVs to solve the reliability problems outlined in section 1.1.2 by forming a structural health monitoring sensor network on these UAVs. The size and weight characteristics of the MSRNs combined with the robustness and accuracy of the sensor network outlined in section 2.2 means that this particular sensor network is well suited to solving the aforementioned UAV problems.

If the AEF connection outlined in section 2.3 is used, then this sensor network can be retro-fitted to existing UAV designs since the connection can penetrate bulkheads without needing to tamper with their structural integrity (Moss et al. 2009).

3.0 SENSOR NETWORKS

3.1 General trends and applications

In order to determine the issues involved in developing sensor network control systems and identify the best practices, current research and thinking behind sensor networks extensive research was performed on available literature. The literature identified a clear trend towards the research and implementation of wireless sensor networks (WSNs).

WSNs consist of a number of distributed sensing nodes, which typically consist of a simple, low power microcontroller with a small amount of memory, a low power wireless transceiver and a limited power supply (Polastre et al. 2005; Vieira et al. 2003; Ricadela 2005). The node hardware will usually be low cost, commercial off-the-shelf (COTS) technology as opposed to more expensive and reliable hardware. A WSN will typically be constructed by randomly placing a large number of these nodes in the order of hundreds or possibly thousands in a dense, random pattern around a target area. Thus the relatively low reliability of the hardware is counteracted by redundancy in numbers. Once deployed the nodes will then organise themselves into an ad hoc network and communicate the desired sensor information to a base station for analysis or perform tasks and / or analysis amongst themselves. The WSN will then be left to its own devices until the nodes run out of power. (Akkaya and Younis 2005; Akyildiz et al. 2002; Qi et al. 2001)

WSNs are a relatively new field of research that have been facilitated by advances in MEMS and wireless communication technologies. The protocols and algorithms used in them are still being actively researched and developed. This research is necessary because the power efficiency required and the large number of nodes in WSNs means that established ad hoc networking protocols are not suitable. (Akkaya and Younis 2005; Akyildiz et al. 2002; Chong and Kumar 2003; Xu 2002)

The potential flexibility, fault tolerance, low cost and rapid deployability that WSNs yield lead to wide and varying applications, including (Akkaya and Younis 2005; Akyildiz et al. 2002; Xu 2002):

- Military
  - Battlefield surveillance
  - Enemy tracking and targeting
  - Reconnaissance
- Health
  - Patient monitoring
  - Biomedical implants
- Environment
  - Habitat monitoring
  - Observation and forecasting
  - Disaster area monitoring
- Commercial
  - Structural health monitoring of buildings
  - Inventory management
  - Product quality monitoring

With constant advances in miniaturisation the original goal of WSNs, the deployment of thousands of nodes the size of dust particles otherwise known as "smart dust" (Bose 2009), is likely to be reached in the near future. Another interesting possibility is the "human as a sensor" concept outlined by Bose (2009) where personal mobile phones can be used as sensor nodes in a large scale sensor network. This yields potential applications such as health and fitness monitoring, situational awareness (including possible integration with near-by smart dust networks) and environmental sensing (for example pollution tracking).

3.2 Sensor network control system considerations

3.2.1 Overview

Analysis of the sensor network literature leads to the identification of a number of key aspects that need to be considered when developing a sensor network system. The distributed nature of sensor networks, combined with the requirement for power efficiency and the potential for extremely large, randomly distributed ad hoc networks mean that unique solutions are required for each aspect. The aspects are listed below and then outlined in the following sections (with a focus on WSNs):

- Power management
- Data fusion
- Data aggregation
- Fault identification and tolerance
- Networking protocols and packet structure
- Network structure
- Data dissemination
- Time synchronisation
- Security

3.2.2 Power management

As outlined in section 3.1, WSN nodes will typically be deployed with a limited power source and be left to run until the power is exhausted. Maximising the lifetime of nodes, and thus the sensor network itself is an important goal of these types of sensor networks. This means that typically (Akyildiz et al. 2002; Szewczyk et al. 2007; Vieira et al. 2003; Younis and Fahmy 2004):

- The microcontroller chosen for the sensor nodes will typically have low power characteristics (such as the MSP430F149, which uses 400μA of current at an operating voltage of 3V);

- The node hardware will include the ability to monitor the amount of remaining power and the current power use, allowing the node to reconfigure itself based on the current power conditions;
- The operating systems will have power management capabilities; and
- The protocols and algorithms present in these sensor networks will focus on low / efficient power use, this is outlined further in the following sections.

3.2.3 Data fusion, data aggregation and fault tolerance

Since data transmission typically has a significant power cost associated with it the amount of data sent across the network needs to be minimised. The possibility of WSNs with thousands of nodes has been envisaged; in order for this to be feasible bandwidth usage per node would need to be minimised so the network can support the transmission of data from all nodes. This goal of minimising network bandwidth necessitates that the sensor values sent by nodes across the network be as compact as possible. With this in mind nodes need to use their computation ability to partially process the data ensuring that only the necessary information is transmitted. What this usually involves is combining multiple sensor values into a smaller number of values (possibly a single value): this process is known as data fusion. (Akkaya and Younis 2005; Akyildiz et al. 2002; Guestrin et al. 2004)

Further to this goal of bandwidth usage minimisation individual nodes need to use their computation ability to combine received values with their local value rather than forwarding fused sensor values received from other nodes through the network. Then the combined local and remote value can be forwarded to other nodes reducing the total bandwidth use. This process of fusing values from multiple nodes is called data aggregation. It should be noted that the ability to perform data aggregation is limited if the sensor network is spread out over a large physical area and the data generated is location-dependant. Depending on the density of node deployment however, there will still be scope for localised data aggregation. (Boulis et al. 2003; Solis and Obraczka 2004)

A number of factors can lead to incorrect sensor data eventually reaching the information sink (the base station in WSNs) (Chong and Kumar 2003; Solis and Obraczka 2004):

- The data aggregation algorithm favours energy-efficiency over data accuracy / freshness (that is data is aggregated and sent too infrequently)
- The data aggregation algorithm does not properly keep track of the dependencies in the information and problems such as double counting can occur, reducing the accuracy of the fused information
- Faulty / damaged sensor(s)
- Faulty / damaged node(s)

The first point can be addressed by ensuring the data aggregation algorithm (in combination with the data dissemination algorithm, see section 3.2.5) strikes an appropriate balance between power efficiency and data freshness and accuracy. The second point can be addressed by ensuring the data aggregation algorithm keeps track of data pedigree, but this means a more complicated algorithm that takes up more computation and power resources. The last two points can be addressed by employing fault detection and / or fault tolerance algorithms. (Solis and Obraczka 2004)

Data fusion algorithms can range from simple techniques such as averaging, histograms and voting to more complicated techniques such as model-based techniques that take into account how data is generated, curve fitting, Bayesian inference, artificial neural networks and fuzzy neural networks. Since the more complicated algorithms take up more computation and power resources nodes can be programmed to select the technique they use based on the current computational load and remaining power. (Chong and Kumar 2003; Guestrin et al. 2004; Klein 1999)

Fault tolerance can be built into data fusion algorithms by utilising the redundancy in the multiplicity of sensor readings. For example, Qi et al. (2001) outline a number of functions that allow for fault tolerant sensor fusion by detecting which readings are faulty; the M, S, Ω and N functions. Fault detection algorithms can be deployed independently of data fusion to identify faulty sensors, whose values can then be ignored. An example is the Bayesian algorithms outlined by Krishnamachari and Iyengar (2004). These fault detection algorithms could logically be extended to detecting nodes in the network that are generating faulty information.

An interesting technique that was proposed early on in sensor network research was the use of mobile agents to form a Mobile Agent Distributed Sensor Network (MADSN) (Qi et al. 2001). The basic idea behind this technique is that a special kind of software, called a mobile agent, migrates from node-to-node autonomously performing data processing. This data processing can include data fusion and it means that node data stays local with the mobile agent being transferred across the network in place of the data. It is claimed that this approach has a number of advantages, including (Qi et al. 2001; Lange and Oshima 1999):

- Reduced network load, reducing energy use
- Overcoming network latency, allowing support of real time requirements
- Networking protocols are encapsulated in the agent rather than each node
- The agents act asynchronously and autonomously from the host that created them, replacing the need for a constant connection to that host
- Agents can adapt dynamically to the network, allowing optimal configurations
- Agents can incorporate fault tolerance and improve the robustness of the network
- The network becomes more scalable since the data from each node is not transmitted through the network

This technique is not appropriate for all applications and a number of disadvantages have been identified. For example, if the algorithm the mobile agent is encapsulating is complex then the overhead of transferring that algorithm across the network cancels out any bandwidth reductions (Wang and Wang 2007). Other disadvantages include longer end-to-end latency (Chen et al. 2007) and security implications of executing code receiving through a potentially insecure network (Harrison et al. 1997). There has been limited research into this technique and current research efforts appear to be simple extensions of the original idea (Cao et al. 2007; Chen et al. 2007; Hla et al. 2008; Patel and Jain 2009); while MADSNs may have a greater importance in the future the technique does not appear to have progressed enough to implement.

3.2.4 Networking protocols and packet structure

The unique constraints of WSNs require new networking protocols that support power efficiency, large numbers of nodes, self organising capabilities, unstable network topologies and a potential lack of global addressing. Most research into WSNs has been on the data link and network layers with some initial research into the transport layer and application layer. (Akyildiz et al. 2002; Akkaya and Younis 2005)

A number of data link layer medium access control (MAC) protocols targeted at sensor networks have been developed, including the Self-Organizing Medium Access Control for Sensor Networks (SMACS) and Eavesdrop-And-Register (EAR) algorithms (Sohrabi et al. 2000) as well as various Carrier Sensing Multiple Access (CSMA), Frequency Division Multiple Access (FDMA) and Time Division Multiple Access (TDMA) adaptations (Guo et al. 2001; Shih et al. 2001). The common factor between these algorithms is they provide mechanisms for power saving modes so the transceiver can be turned off for periods of time. One consequence of having low power modes is that sensors may not receive messages while they are in these modes; the node has to wake up periodically to see if there is any incoming network data or new sensor data. Li et al. (2003) propose an algorithm where a particular number of waste bits are added to the start of all packets so that all nodes within range will know they are receiving a packet. (Akyildiz et al. 2002)

The responsibility of the data link layer to perform error control also needs to be considered. If an Automatic Repeat reQuest (ARQ) protocol is used then there is a requirement to send acknowledgements across the network increasing power and bandwidth use. Alternatively, Forward Error Correction (FEC) can be used, but that increases the size of each packet, also increasing power and bandwidth use. Some FEC techniques are too computationally expensive for deployment in sensor networks. There has been a reasonable amount of recent research into error correction techniques, and it appears that a selection should be made on an application by application basis (Busse et al. 2006; Jeong and Ee 2006; Li et al. 2009; Vuran and Akyildiz 2006). (Akyildiz et al. 2002)

As discussed in section 3.1, WSNs are deployed randomly resulting in a need to organise themselves. Furthermore, it is likely that information sinks will be more than one hop from at least some of the information sources. This necessitates the use of multi-hop routing protocols. As well as the aforementioned power efficiency and node multiplicity considerations the network layer protocols in sensor networks have a range of other factors that need to be considered (Akyildiz et al. 2002):

- They should take into account the data aggregation algorithm
- They should take into account the network structure and data dissemination strategy in order to support the required network structure and addressing
- In some applications, location awareness is important (for example enemy tracking)

Sensor network routing protocols can use a variety of metrics to determine optimal routing paths including maximum power available, minimum energy to transmit, minimum hop (same as minimum energy if hop energy use is constant across all links) and maximum value of the minimum power available along the route. Data-centric approaches such as speculative advertisements and information requests can also be used, which would likely use attribute based addressing (Krishnamachari et al. 2002; van der Velden and Powlesland 2008). Another potentially useful metric is link quality, which can be determined by link connectivity statistics; Woo et al. (2003) evaluate a number of different link quality estimation algorithms for this purpose as well as suitable neighbourhood table management algorithms. (Akyildiz et al. 2002)

A variety of network layer routing protocols have been developed for sensor networks, including:

- Sensor Protocols for Information via Negotiation (SPIN) (Heinzelman et al. 1999)
- Sequential Assignment Routing (SAR) (Sohrabi et al. 2000)
- Low-Energy Adaptive Clustering Hierarchy (LEACH) (Heinzelman et al. 2000)
- Directed Diffusion (Intanagonwiwat et al. 2000)
- Small Minimum Energy Communication Network (SMECN) (Li and Halpern 2001)
- Partial-partition Avoiding GEographic Routing (PAGER) (Zou et al. 2005)

Directed Diffusion in particular has received a lot of research interest (Braginsky and Estrin 2002; Chen et al. 2007; Chu et al. 2002; Kumar et al. 2003; Patel and Jain 2009).

The size of packets able to be carried by the network will be dependent on the data link layer and network layer protocols. The packet structure that a given sensor network uses will vary on a per-application basis depending on the goals of the network and the types of information being transmitted. Some initial work on application layer protocols that define packet structures are outlined by Akyildiz et al. (2002).

3.2.5 Network structure and data dissemination

Four sensor network goals affect the choice of an appropriate networking structure (Iyengar et al. 1994):

- Meeting any real-time constraints
- Minimising energy use by minimising data transmissions
- Tolerance to faults in the network
- There can be hundreds or thousands of nodes

The first two points lead to some sort of hierarchical structure so the number of hops between information sources and information sinks are minimised. The third point requires some sort of redundancy in the structure of the network. The final point requires that the network structure is extensible and scalable.

Network layer routing protocols will generally need to have information about the structure of the network, so they are very closely tied to the network structure. Some of the algorithms listed in section 3.2.4 specify restrictions on the network structure. Tree based networking structures have been considered since the work by Wesson et al. (1981) with the development of the Anarchic Committee (AC) and Dynamic Hierarchical Cone (DHC) structures. AC is a fully interconnected network allowing every node to communicate with every other, whilst this presents the best performance and fault tolerance it has serious scalability issues. DHC is a hierarchical structure that only allows communication between nodes in adjacent layers; this is scalable but is not very fault tolerant. Figure 3.1 gives a graphical representation of these structures. (Qi et al. 2001)

A combination of these structures was considered in the first network structure proposed for sensor networks: the flat tree network, where the network consists of a number of binary trees the roots of which are completely connected (Jayasimha et al.

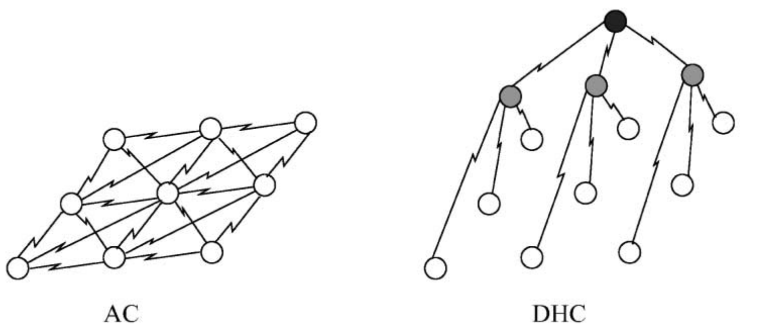


Figure 3.1: AC and DHC networking structures (Qi et al. 2001)

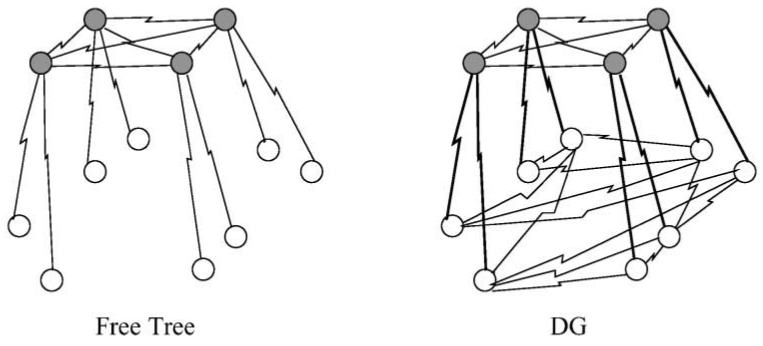


Figure 3.2: Flat tree and DG networking structures (Qi et al. 2001)

1991). Iyengar et al. (1994) improved the fault tolerance, extensibility and simplicity of routing algorithms of the flat tree structure by using a deBruijn Graph (DG) to connect nodes on the same level. Figure 3.2 gives a graphical representation of these structures. (Qi et al. 2001)

More recently, there has been the development of approaches that are more targeted towards large, unreliable ad hoc networks with severe power constraints by introducing more flexible, energy efficient clustering techniques that are more suited to data-centric networks. Examples include Hybrid, Energy-Efficient, Distributed (HEED) clustering (Younis and Fahmy 2004) and the Directed Diffusion algorithm mentioned in section 3.2.4.

Data dissemination is the strategy used to determine when information should be sent through the network and to whom it should be sent, with the overall goal of information sinks efficiently extracting data from the sensor network. It should be noted that the data aggregation algorithm, network layer protocol and the network structure all affect

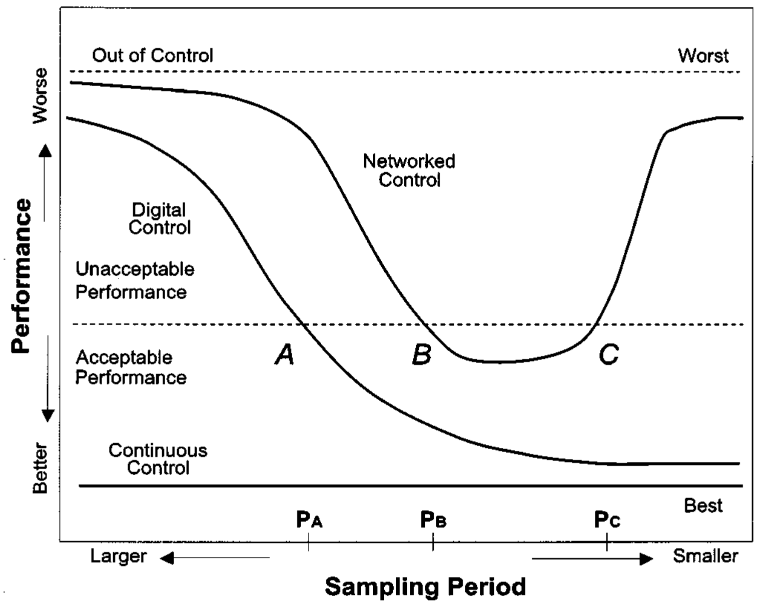


Figure 3.3: Sampling period versus performance for networked control systems (Lian et al. 2002)

the data dissemination strategy (in fact some of the algorithms span across all these realms). The data dissemination strategy must ensure that data is not sent out too frequently otherwise the network will be overloaded. Conversely, if data is sent out too infrequently then the information sinks will be acting on stale data and the system will not be able to function effectively. This balance is similar to the balance needed with sampling period in networked control systems, as shown in figure 3.3 (Lian et al. 2002). (Shenker et al. 2003)

A number of data dissemination strategies have been devised for sensor networks, including:

- Declarative Routing Protocol (DRP) (Coffin et al. 2000)
- Distributed Services (Lim 2001)
- Geographical and Energy Aware Routing (GEAR) (Yu et al. 2001)
- Data-Centric Storage (DCS) (Shenker et al. 2003)
- Sinks Accessing data From Environments (SAFE) (Kim et al. 2003)
- Deluge (Hui and Culler 2004)

- WaveScheduling (Trigoni et al. 2004)
- GRAdient Broadcast (GRAB) (Ye et al. 2005)
- Location-Aided Flooding (LAF) (Sabbineni and Chakrabarty 2005)
- Two-Tier Data Dissemination (TTDD) (Luo et al. 2005)
- Infuse (Kulkarni and Arumugam 2006)

The network layer protocols SPIN and Directed Diffusion mentioned in section 3.2.4 can also be considered to be data dissemination strategies.

3.2.6 Time synchronisation

As a distributed computing platform, sensor networks need time synchronisation between nodes. More specifically sensor nodes need to be synchronised to (Dai and Han 2004; Elson and Estrin 2001; Sivrikaya and Yener 2004):

- Ensure correct timestamping of packets and thus chronologically accurate interpretation of packets by information sinks;
- Allow coordination between nodes for operations such as data aggregation;
- Facilitate concurrent "wake up" periods to reduce message routing latency and allow longer "sleep" periods; and
- Allow correct functioning of time sensitive networking protocols such as TDMA.

There are a number of standardised time synchronisation protocols that have been developed including the Network Time Protocol (NTP), Simple NTP (SNTP), Global Positioning System (GPS) and IEEE 1588 protocol (Dopplinger and Innis 2007). These standardised protocols are typically not suitable for sensor networks because they are too computationally expensive, require too much transmission overhead and will typically deliver a timing precision that is more accurate than the network requires resulting in a waste of resources. Furthermore, the low cost hardware typically used in sensor networks is more susceptible to clock drift than the hardware on which the standardised time synchronisation protocols are implemented. They also do not take into account the increased packet delay variance from collision of broadcasted packets in the case of WSNs. (Dai and Han 2004; Elson and Estrin 2001; Elson and Römer 2003; Sivrikaya and Yener 2004)

A range of research has been conducted in order to provide efficient, long lifetime time synchronisation that more closely supports the goals of sensor networks:

- Elson and Estrin (2001) outline post-facto synchronisation, which allows nodes to stay synchronised after powering down by the use of broadcasted synchronisation signals and is appropriate for localised time synchronisation.
- Elson et al. (2002) outline Reference Broadcast Synchronisation (RBS), which utilises a broadcasted reference signal to create a clock that nodes can synchronise against.
- Sichitiu and Veerarittiphan (2003) outline two algorithms Tiny-Sync and Mini-Sync, which are simple, drift aware, fault tolerant time synchronisation algorithms that perform pair-wise synchronisation; they can scalably synchronise an entire network when combined with an appropriate hierarchical network structure.
- van Greunen and Rabaey (2003) present the Lightweight Tree-based Synchronisation (LTS) time synchronisation method, which is similar to the Tiny-Sync and Mini-Sync algorithms mentioned above.
- Sinopoli et al. (2003) outline the use of an NTP-like global time synchronisation in combination with lower precision time zones between groups of localised nodes and the ability to transform time readings between nodes. This allows a range of applications with different precision requirements to be accommodated.
- Hu and Servetto (2003) describe a method of time synchronisation where nodes collaborate to create an aggregate waveform observable simultaneously by all nodes, which simulates a "super node" generating a network-wide time reference

signal.

- Ganeriwal et al. (2003) describe the Timing-sync Protocol for Sensor Networks (TPSN), which provides network-wide sender-receiver time synchronisation in a simple, efficient and scalable manner.
- Maróti et al. (2004) describe the Flooding Time Synchronization Protocol (FTSP); a low bandwidth time synchronisation protocol, which leverages MAC layer time stamping and error compensation to provide a high precision time synchronisation that is robust to network faults.
- Dai and Han (2004) describe TSync; a lightweight synchronisation service that uses bidirectional, multi-channel broadcasting.
- Ishikawa and Mita (2008) have modified COTS Category 5 Ethernet cables and COTS hubs to provide constant time synchronisation in wired sensor networks independent of the data being transferred.

A good overview of the advantages and disadvantages of many of these approaches is given in (Sivrikaya and Yener 2004).

3.2.7 Security

Some WSN applications involve operation in hostile environments where security of wireless transmissions is an important consideration. In this case techniques need to be used to ensure the network can be protected from intrusion and spoofing, which can range from designing low profile networking hardware through to encryption of packets. It should be noted that packet encryption / decryption requires extra computation (and thus power) resources, as well as a requirement for the establishment of pairwise encryption keys (Eschenauer and Gligor 2002; Lee and Stinson 2004; Liu et al. 2005). (Chong and Kumar 2003)

Ashraf et al. (2008) outline a framework to perform security assessments of WSNs with the goal of deploying secure WSN applications. They developed the framework in recognition of the fact that security requirements differ from application to application and thus an analysis would need to be performed to determine the security requirements for every WSN application that is designed. Walters et al. (2007) give an excellent overview of the obstacles, requirements, attacks and defences related to WSN security.

3.3 Extraction of requirements for UAV sensor network

Comparison of the sensor network described in section 2.0 to WSNs yields the following observations:

- The system will exhibit the low cost hardware, power efficiency and fault tolerance characteristics present in WSNs;
- The system has point-to-point links rather than broadcasted wireless transmissions like WSNs; nodes will not participate in an ad hoc network, but rather exhibit a relatively fixed structure (apart from damaged nodes and links);
- The point-to-point links mean there is less flexibility in deployment of the network, and thus the number of nodes and density of node placement are both likely to be smaller than in WSNs;
- Nodes can have multiple data link layers since there will be multiple point-to-point links; this differs from WSNs, whose nodes will typically have a single data link layer for the wireless transmitter; and
- There is no base station like most WSN applications individual nodes will perform tasks and local data analysis.

Consideration of the system in light of the sensor network aspects discussed in the previous section result in the following implications:

- **Power management** – Power management is of similar importance to WSNs, so a power management scheme is definitely required. This scheme would need to incorporate the ability for nodes to detect their remaining power level and potentially allow nodes to request more power from power generation nodes. This should be possible with the flexibility afforded by the proposed physical layer.
- **Data fusion** – Data fusion will be required on all nodes that have an MSRN in order to consolidate the information generated by the MSRN.
- **Data aggregation** – Data aggregation will need to be applied to combine redundant information generated by MSRNs in multiple nodes.
- **Fault identification** – Fault identification should be deployed on each node to identify faulty sensors in the MSRN so their values can be ignored. Similarly, the data aggregation strategy could be combined with a fault identification algorithm to identify damaged nodes in the network. Depending on the flexibility of the power management system power could potentially be shut off to faulty nodes (the unique physical layer proposed should have the potential for this) to conserve power and reduce the likelihood of the node interfering with the correct operation of the system.
- **Fault tolerance** – The combination of data fusion, data aggregation, fault identification and intelligent routing will ensure there is inherent fault tolerance built into the system. It is possible that further fault tolerance could be built into the system by looking at individual aspects.
- **Data link layer protocol** – The data link layer protocol will need to be suitable for transmission across point-to-point links and be compatible with whatever physical layer is selected. If the Acoustic Electric Feedthrough (AEF, see section 2.3) physical layer is chosen, the data link layer will need to incorporate

protection against signal interference. The use of ARQ protocols is probably too much of an overhead for the network, particularly as the size of the network increases and messages travel multiple hops. FEC techniques are worth investigation depending on the capabilities of the data link layer protocol chosen.

- **Network layer protocol and data dissemination** – The network layer protocol should incorporate intelligent routing to overcome network faults and will need to be compatible with multiple links per node and the chosen network structure. The primary function of the network is dissemination of sensor data therefore, data-centric routing and data dissemination should be investigated as a possible implementation option. This would support the possible goal of the system to incorporate artificial intelligence into the network to maximise utility functions for data accuracy (van der Velden and Powlesland 2008). The data-centric approach also leads nicely to the speculative advertising and information request concepts mentioned by van der Velden and Powlesland (2008). The network layer will also need to facilitate functionality for the power management scheme, such as power allocation request messages.
- **Network structure** – van der Velden and Powlesland (2008) outline that a physical mesh networking structure strikes an appropriate balance between redundancy and link cost. Furthermore, depending on the number of nodes in the network it may be appropriate to form a logical hierarchical structure over the physical structure to allow the use of better data aggregation, time synchronisation and routing algorithms.
- **Time synchronisation** – Time synchronisation will be a necessary feature of the control system since the network will have real time actuation requirements and thus, sensor data will be chronologically important. Since point-to-point links will be used the time synchronisation protocols that rely on broadcasted signals are not appropriate.
- **Security** – The use of point-to-point links largely negates the need for security solutions and as such there is no initial need for security built into the control

system (van der Velden 2009, pers. comm., 30 January). Since the system is currently targeted at UAVs, which have potential military applications the security of the system should be considered at a later stage as there still exists the possibility of interception of network data from electromagnetic radiation (or acoustic waves in the case of AEF) emanating from the network links.

Research into the use of sensor networks for structural health monitoring yields a number of applications (Ishikawa and Mita 2008; Jaman and Hussain 2007; Paek et al. 2005) as well as two excellent reviews (Bischoff et al. 2009; Lynch and Loh 2006). However, all of the work in this area seems to be in the structural health monitoring of buildings and as such has little in common with the system being investigated. Furthermore, most of the structural health monitoring applications are WSNs.

4.0 SYSTEM REQUIREMENTS SPECIFICATION

4.1 Motivation

After the initial research into sensor networks the different aspects of the control system being designed were considered in turn. Firstly, to identify the application-specific issues involved and secondly, to identify suitable solutions, protocols and algorithms. After a number of weeks of collaborative communication with Stephen van der Velden from the DSTO, a basic outline of the control system started to emerge.

Many questions remained unanswered as there were no hard specifications for the hardware with which the system would be integrating. The MSRN chip described in section 2.1 was still being developed and the choice of physical layer and operating hardware were left as open decisions. The work on the control system was starting from scratch apart from the initial ideas proposed by van der Velden et al. (2007); van der Velden and Powlesland (2008).

In order to progress further it was thought that a System / Software Requirements Specification (SRS) should be created in order to:

- Formalise and summarise the initial sensor network research that was undertaken and provide a means of performing targeted research for the particular system;
- Outline the perceived perspective of the appropriate specification for the system for DSTO approval;
- Lock down the scope of the project to create a set of achievable goals;
- Specify an orderly and well structured development approach, with required flexibility for prototyping solutions for the different aspects of the system; and
- Identify the variables of the system that were still unknown and could not be directly answered.

4.2 Approach

It was decided that a casual approach be taken with the SRS document for a number of reasons:

- Save time to allow maximum time to work on the system design and implementation;
- Recognition that there could be further clarification or changes in the direction of various aspects of the system; and
- Provision of flexibility to perform rapid prototyping to work out various parts of the system.

Whilst the Institute of Electrical and Electronics Engineers (IEEE) SRS style was used to structure the document (IEEE 1998) the content of the document is much less formal and complete than would be seen in a typical SRS document. Furthermore, parts of the specification touch on implementation and design details, which should not normally be part of a requirements document.

The SRS outlined an approach that involved splitting up the system into three separate design and implementation phases. The first phase, "Basic System" outlines a very simplistic system in order to create a benchmark for the rest of the development:

- **Initialisation Mode** – Simple, periodic initialisation including time synchronisation and neighbour status determination.
- **Time synchronisation** – No specific details were given in the specification apart from the specification of a precision variable and the stipulation the algorithm should minimise power use.
- **Data fusion** – Simple averaging at a regular time interval, minimum and maximum values are kept to measure the spread of data (and thus act as primitive believability descriptors: the smaller the range the more believable the value is),

the number of sensors used to generate the data is also kept (this serves as a primitive accuracy descriptor: the more sensors there are the more likely the average value is accurate).

- **Fault tolerance** – Outliers will be ignored from the data fusion algorithm.
- **Data aggregation** – No data aggregation: incoming data from the network that is newer than the current value is taken at face value.
- **Data dissemination** – If a newer value is received from the network it is treated as described under data aggregation above, and then forwarded to all other network links. If a packet is received from the network with an older value the packet is dropped.
- **Networking protocols** – The network layer protocol will need to support the data dissemination algorithm described above. The data link layer needs to support point-to-point links, have MAC to overcome possible interference if an Acoustic Electric Feedthrough (AEF, see section 2.3) physical layer is used and fit the packet structure (see below)
- **Packet structure** – Data and system (time sync, probe, probe acknowledgement) packets, each data packet contains the value for a single information type, and consists of:
  - Information type
  - Average, minimum and maximum values
  - Number of sensors used to generate the average
  - Timestamp (data freshness descriptor)
  - Source address (assuming that nodes are addressable)
- **Flat network structure** – No hierarchy; all nodes receive all information.
- **Data access API** – The system will have a data access API so the application layer can access the sensor information without dealing directly with the network or the sensors.

The second phase "Multi-level" system is an extension of the phase one system with the introduction of a logical, hierarchical network structure (along with an appropriate network layer routing protocol) as well as more sophisticated data aggregation and dissemination strategies to improve fault tolerance and accuracy and reduce the network traffic and power use of the system. The choice of the network structure and data aggregation and dissemination strategies is left open to further research, but a list of points to guide the choice is made. It was envisaged that experience from implementing the phase one system, in combination with targeted research and potential rapid prototyping, would lead to appropriate choices.

The third phase of the system was outlined as a number of extensions to the phase two system that in combination would transform the system into the final "ideal" system. The extensions that were outlined are:

- **Utility function optimisation and MSRN reconfigurability** – Optimising system performance based on utility functions for data accuracy and reconfiguring the MSRN chips based on this optimisation.
- **Information requests and speculative advertisements** – Improving robustness and reducing network traffic by adoption of speculative advertisements and information requests combined with intelligent routing (possibly via the data-centric approach outlined in sections 3.2.4, 3.2.5 and 3.3).
- **Fault tolerance and detection** – Investigating integration of more intelligent fault tolerance and detection algorithms with the system to improve system robustness.
- **Power management** – Improve power efficiency of the system by introducing power management algorithms and different power modes.
- **Data fusion and aggregation** – Investigation of more intelligent data fusion and aggregation to improve fault tolerance and accuracy of the system. This could include integration with the networking protocols.

- **Hardware implementation** – The possibility of implementing parts of the system in hardware for speed, memory, power efficiency and portability reasons is outlined.

A number of unknown variables that needed to be determined were outlined in the specification:

- Sampling period for data fusion;
- Whether nodes should be addressable;
- Whether the source address should be put into the packet;
- Time synchronisation precision;
- Minimum synchronisation period;
- Maximum delay across the system;
- Maximum number of nodes; and
- Timestamp resolution.

The specification also outlined the need to secure an operating system to make development easier and to support the portability goal of the system. Initial research on a number of operating systems indicated the TinyOS operating system as the best choice and it was nominated as a possible choice in the specification, along with a list of its advantages and disadvantages.

4.3 Outcome

The written specification can be found in Appendix A. It has been left in its original state despite clarification on various points and terminology that has since been established. The overview of the specification in the previous section corrects these issues and links therefore, the specification should be read in conjunction with that section.

After the specification was presented to the DSTO there were some initial concerns about the value of an operating system and focusing on designing the system bottom up rather than performing research on some of the more advanced concepts up front (like integrating the artificial intelligence in the network protocols). Despite these concerns, it was agreed that the specification represented an appropriate approach for an undergraduate project and the specification was accepted (van der Velden 2009, pers. comm., February 24). In order to address the concerns that were raised extensive further research was performed. A more insightful analysis of the system was developed; this analysis is embodied in section 3.3. Furthermore, a justification of the need for an operating system is given in section 5.1.

The initial choices that were made to allow development to begin were:

- **Development platform** – One of the system goals was portability, so in essence the platform choice is arbitrary, hence the choice of the Freescale HCS12 on the Wytec Dragon 12 development board for availability and familiarity reasons.
- **Operating System** – TinyOS was chosen as the operating system on which the control system would be developed, since it seemed the best fit for the goals and requirements of the system, see section 5.4 for more information about TinyOS.
- **Variable selection** – Table 4.1 outlines the initial arbitrary values that were selected for the variables outlined in the specification to allow development to begin unhindered.
- **Packet size** – Given the values in table 4.1 the packet size needed is 7 bytes assuming the source address is stored in the data portion of the packet rather than a header.
- **Data link layer** – The Controller Area Network (CAN) protocol was chosen because it was natively supported on the development platform with multiple drivers (allowing testing of multiple links as required by the system). It is simple, it supports point-to-point links, it has automatic acknowledgement and re-

Variable	**Value**
Sampling period	10ms
Addressable nodes / source address in packet	Yes / Yes
Time synchronisation precision	1ms
Synchronisation period	1s
Maximum network delay	2ms
Maximum nodes	127
Timestamp resolution	16-bit

Table 4.1: Initial values for specification variables

transmission of packets and fits the packet size outlined in the last point. If the system is extended further requiring a larger packet size (for example higher precision data requiring more than 8 bits, which is an eventuality with the MSRN (van der Velden 2009, pers. comm., 24 February)) then the use of CAN would need to be reconsidered since the maximum CAN message size is 8 bytes.

In conjunction with the specification a reading list was created to categorise the reading material that had already been read and the material that had yet to be read so that relevant articles could be quickly identified as each aspect of the system was designed. This reading list has been supplemented with the extra research material that has been discovered since and can be found in Appendix B.

The specification outlined that the phase one system would be implemented as part of this project along with most (if not all) of the phase two system. As it eventuated this estimate turned out to be overly optimistic. Due to changes in project focus (that are outlined in section 5.0), the development of the phase one system did not begin.

5.0 OPERATING SYSTEM

5.1 Motivation

As outlined in section 2.4, one of the goals of the control system is that it must be portable so that it can run on a variety of platforms. There are a number of ways to accomplish this all of which involve defining an architecture independent Application Programming Interface (API) (Marcondes et al. 2006):

- **Virtual Machines (VMs) –** such as the Java Virtual Machine (JVM)
- **System call interfaces –** such as POSIX and WIN32
- **Hardware Abstraction Layers (HALs) –** used in many operating systems, including Windows and Linux

Virtual Machines allow binary compatibility between different hardware platforms from the compilation of the same source code. However, the memory and processor overhead of translating between the binary code the VM understands and the machine language the underlying hardware understands is typically too high to be feasible in an embedded system. There are ways to address the overhead such as Just-in-Time (JIT) and Ahead-of-Time (AOT) compilers, but there will still be come overhead with these approaches. The VM will need to be ported to the hardware platform before it can be used, which will be a complicated process. Also, the languages that are compiled to VM binary code will typically be too general to perform low-level functionality such as register access that are critical in an embedded system. This means the language would need to be combined with more conventional embedded systems development languages like C and assembly meaning extra work to port the system to each different hardware platform. (Barr and Frank 1997; Marcondes et al. 2006)

When system call interfaces are used they not only allow platform independence, they also allow operating system independence. The problem is that pre-existing interfaces are designed primarily with desktop systems in mind and are thus too bloated to be practically used in a limited memory / processing embedded environment. (Marcondes et al. 2006)

Hardware Abstraction Layers are software that directly communicate with the underlying hardware and provide an API for the operating system above to access. The use of HALs allows operating systems to be implemented in a portable manner as all of the hardware targeted source code can be separated from the operating system source code. Thus, if an operating system with appropriate HALs is used then applications written for that operating system will be portable. (Marcondes et al. 2006)

For an embedded system with tight memory and processing constraints the most suitable option to ensure portable code is the use of an operating system that utilises HALs to separate platform targeted code. The use of an operating system allows easier and faster development because of the services it provides (for example schedulers and timers).

5.2 Core requirements

In order to support the goals of the system being considered for this project (as outlined in section 2.4) the operating system chosen to develop this particular control system needed to have the following properties:

- Low computation and memory overhead (lightweight)
- Portable between hardware platforms (as described in the previous section)
- Support for networking
- Power aware
- Good documentation to minimise learning overhead

Furthermore, the operating system needed to run on the development platform chosen (see section 4.3) or be easy to port to that platform.

5.3 Choice

The following operating systems were considered for use with the control system being developed. It should be noted that these operating systems were shortlisted because of their lightweight nature; other common embedded operating systems such as Embedded Linux and Windows CE were immediately rejected because they are targeted at more powerful hardware than this system is intended to run on:

- **TinyOS** – Free, active development community, designed for WSNs, lightweight, power aware, networking support, good documentation, ported to a number of platforms (not including HCS12), a lot of the sensor network literature mentioned TinyOS, a range of algorithms and extensions are available for it. (UC Berkley n.d.)
- **SOS** – Free, similar to TinyOS, but uses dynamic module loading to give greater flexibility. However, it is no longer under active development. (UCLA NESL 2008)
- μ**Cos II** – Robust, portable (ported to HCS12), real-time, reasonably light weight. However, it is not free. (Micrium 2009)
- **eCos** – Free, portable (not including HCS12), built-in networking support, configurable component-based architecture allowing the kernel to be paired back for lightweight operation. (eCos n.d.)
- **FreeRTOS** – Free, active development community, portable (ported to HCS12), lightweight, supports real-time requirements, good documentation. No in-built networking or power management support. (FreeRTOS 2009)

FreeRTOS, eCos and TinyOS were further shortlisted from the above list, and a final choice made on TinyOS because of its in-built networking and power management as well as the fact it is more lightweight and there are a range of useful extensions and algorithms developed for it (including a network simulator). Initial reading of documentation and source code seemed to indicate that TinyOS was a much easier operating system to understand and program. The following disadvantages of TinyOS were noted:

- It was implemented in an unknown language nesC, which means a learning curve cost (however, nesC is an extension of the known C language and there is good documentation on it, so it was determined this should not have much of an impact).
- It would need to be ported to the development environment (however, due to the MSRNs and potentially the physical layer, no matter what operating system is chosen, there would have to be some porting and in this case the porting process would benefit by helping to understand the operating system and nesC).
- The system is targeted at WSNs rather than sensor networks in general (however, section 3.3 outlines that the system being developed has many similarities to WSNs and it was deemed this was acceptable).
- TinyOS does not have real-time guarantees (it was decided that this was an acceptable disadvantage because most events occurring in the system will be periodic, allowing static real-time analysis approaches to be utilised).
- Scheduling is FIFO non-preemptive (this means that CPU-bound tasks are not suitable for the operating system, however for this type of system one of the goals is to reduce computation overhead to minimise power use, so this is acceptable).

5.4 TinyOS

5.4.1 Overview

5.4.1.1 Introduction

It should be noted that the overview given in this section is by no means comprehensive: it is broad, simplistic and simply serves to outline the main parts of TinyOS and nesC with some detail for parts that are relevant in later sections.

TinyOS is an event-driven embedded operating system specifically designed for WSN applications. It is implemented in a language called nesC, which is an extended version of the C language that adds on (among other things) components and interfaces and the ability to statically “wire” components together bi-directionally. TinyOS uses simple, non-preemptive FIFO scheduling of “tasks”, which are encapsulated by a single function inside a component. TinyOS also provides interfaces for power management, timer management, sensor access and network access. TinyOS comes with a component library that includes networking protocols, distributed services, sensor drivers and data acquisition tools. (UC Berkley n.d.)

TinyOS has been ported to over a dozen hardware platforms including (Sourceforge.NET 2009):

- Intel Mote – ARM ARM7TDMI microcontroller (Nachman et al. 2005)
- Intel Mote 2 – Intel XScale PXA processor (Adler et al. 2005)
- Mica2 – Atmel ATmega128 microcontroller (Crossbow n.d.-b)
- MicaZ – Atmel ATmega128 microcontroller (Crossbow n.d.-c)
- Iris – Atmel ATmega128 microcontroller (Crossbow n.d.-a)
- BTnode – Atmel ATmega128 microcontroller (Zurich 2007)
- TelosB – Ti MSP430 microcontroller (Moteiv Corporation 2004; Polastre et al. 2005)

- eyesIFX – Ti MSP430 microcontroller (Lentsch 2005)
- SHIMMER – Ti MSP430 microcontroller (Kuris and Dishongh 2006)
- Microchip PIC microcontroller (Lynch and Reilly 2005; Korber et al. 2005)

The requirements to develop TinyOS applications are a GCC tool chain for the hardware platform and Linux in order to run the nesC compiler (which compiles the nesC code into GCC compatible C code). It also requires Make to be installed on the computer for the TinyOS build system. (Wiki 2009)

TinyOS conventions and extensions are documented by a system called TinyOS Extension Proposals (TEPs) (Levis 2007a). TinyOS development is conducted according to a set of coding conventions that are outlined in TEP 3 (Yannopoulos and Gay 2006).

5.4.1.2 nesC

nesC interface definitions are analogous to abstract classes in object-oriented languages (known as virtual classes in C++). They define a bi-directional interaction between two components: the provider and the user. Each interface can specify a number of commands that the provider component must implement and the user command may call. The interface can also specify a number of events that the user component must implement and the provider component may trigger. Both commands and events are encapsulated as function calls. Interfaces have unique, global names. nesC also provides the ability to pass static and run-time variables into interfaces. This is done via generic and parameterised interfaces respectively. Generic interfaces allow types and integers to be statically passed into an interface using the same syntax as templates in C++ (`InterfaceName<parameters...>`). Parameterised interfaces allow multiple interfaces to be used and referenced by an integer index: this uses array notation (`InterfaceName[index_type_t index]`). (Gay et al. 2005; Levis 2006; Levis and Gay 2009)

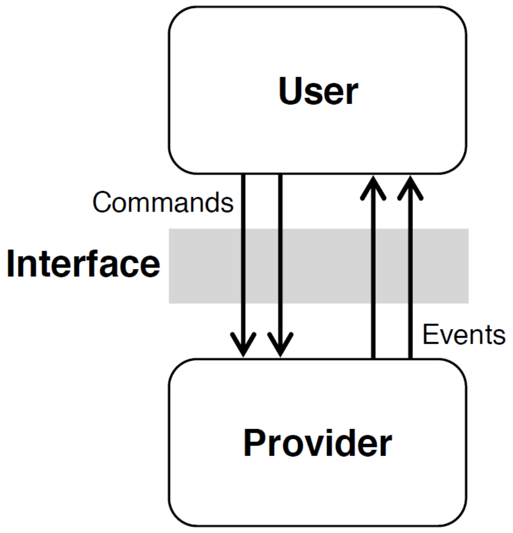


Figure 5.1: Illustration of the relationship between interfaces, components, commands and events in nesC (Levis and Gay 2009)

Figure 5.1 demonstrates the relationship between users, providers, interfaces, commands and events. The following code snippet gives an example of the syntax for declaring these constructs (Gay et al. 2005; Levis and Gay 2009):

```
// Standard Interface Example with a single command and event
interface InterfaceName {
    command some_type_t commandName(another_type_t parameterName);
    event some_type_t eventName(another_type_t parameterName);
}
// Example Generic Interface Definition: Queue
// async means the function can be called within
//  an interrupt handler
interface Queue<t> {
    async command void push(t x);
    async command t pop();
    async command bool empty();
    async command bool full();
}
```

Each component in nesC will specify a number of interfaces that it uses or provides. There are two types of components in nesC: modules and configurations. Modules explicitly define the commands of the interfaces they provide and the events of the interfaces they use, whereas configurations "wire" other components to the interfaces it uses and provides (as well as performing wiring between the interfaces of other components). These components can be either modules or other configurations and the components being wired can either be instantiated within the configuration or wired into the configuration. (Gay et al. 2005; Levis and Gay 2009)

By default, all components are singletons; there is only one instance of them in the system, and wirings to the same component name will mean wirings to the same component instance. There is a way to declare non-singleton components by the use of generic components, which can be instantiated with parameters in a similar way to constructors in object-oriented languages. Generic components can be declared using the **`generic`** keyword, and having parentheses after the component name (optionally with instantiation parameters). These instantiation parameters can include types by using the syntax **`typedef`** `type_name_t`. (Levis and Gay 2009)

The code snippets shown in Appendix C demonstrate the syntax for a number of nesC features, including:

- Declaration of private component variables
- Use of components in a configuration using the **`components`** keyword
- Instantiation of generic components using the **`new`** keyword, in conjunction with the **`as`** keyword for naming the instance
- Use of the **`as`** keyword for renaming interfaces, which allows a component to provide or use the same interface multiple times under different names
- Use of the **`call`** keyword for invoking commands
- Use of the **`signal`** keyword for invoking events
- Use of `->` and `<-` to wire the user of an interface to the provider of an interface
- Use of = to wire an interface that a configuration uses or provides to a component
- Use of dot notation to access the interfaces and commands of components
- Posting tasks (see next section for explanation)

Another feature that nesC provides is the ability to create atomic sections. This is applied via the **`atomic`** keyword, which can be added as a prefix before a statement or a {...} block. The atomic sections that this keyword forms are replaced at the beginning with a call to `__nesc_atomic_start` and at the end with a call to `__nesc_atomic_end`. These routines are defined for each platform and will typically be implemented by disabling interrupts and then re-enabling them (if they were not already disabled). (Gay et al. 2005)

Since nesC is compiled into C code it suffers from the global namespacing problem in C: every function, type and constant you declare is then available to everything else meaning naming conflicts can readily occur. nesC overcomes this by ensuring any type definitions, preprocessor constants, enumerations or functions defined within a module will be localised to that module. Any of these declarations that appear outside a module will work as in C. (Levis and Gay 2009)

5.4.1.3 Execution Model

The execution model of TinyOS is based on split-phase operations, run-to-completion tasks and interrupt handlers. Long-running I/O operations are performed via split-phase operations: a command will be invoked, which asks the relevant hardware to perform the operation, then when the hardware is finished an interrupt handler will signal an event to let the original component know the requested operation has been finished. The use of split-phase operations means that TinyOS has no need for threads (which use up memory, and are thus undesirable in the low memory WSN hardware environments) or polling loops (which consume power and execution resources). Any code in TinyOS is executed by an interrupt handler (which will typically signal an event, as described above) or by a task. Tasks are essentially a deferred procedure call, they can be posted at any time, and then will be executed at a later time. Any commands that need to be called by interrupt handlers must use the **`async`** specifier (as shown in the code snippet in section 5.4.1.2) to allow the nesC compiler to ensure there are no concurrency issues. (Levis and Gay 2009)

Tasks in nesC are encapsulated as a function within a module that returns void and has no parameters; the syntax is `task void myTask()`. A task can be posted to the scheduler by calling `post myTask()`, which returns immediately with true or false depending on whether the task was successfully posted to the scheduler. (Gay et al. 2005)

The scheduler in TinyOS is a first-in first-out (FIFO) structure that executes tasks to completion before executing the next task in the queue. When the task queue is exhausted the scheduler will put the processor into a low power mode until an interrupt wakes the processor (possibly posting more tasks to execute).

5.4.1.4 Networking

Data link layer level communications in TinyOS are based on three classes of interfaces: Packet interfaces for accessing message data and metadata, Send interfaces for sending packets and Receive interfaces for handling packet reception. The Send and Receive interfaces are distinguished by their addressing scheme, and may be parameterised to provide support for multiple higher level networking layer clients (the equivalent of ports in Transmission Control Protocol/Internet Protocol (TCP/IP)). (Levis 2008)

The base Packet interface simply exposes a way to determine the maximum payload length for the packet, as well as the ability to return a pointer to the payload portion of the packet and clear all fields in the packet. Its definition is as follows (Levis 2008):

```
interface Packet {
    command void clear(message_t* msg);
    command uint8_t payloadLength(message_t* msg);
    command void setPayLoadLength(message_t* msg, uint8_t len);
    command uint8_t maxPayloadLength();
    command void* getPayload(message_t* msg, uint8_t len);
}
```

This basic packet interface can be extended to provide more information in the packet. Such as, getting and setting destination address, source address and packet type (as in AMPacket, the active message Packet interface).

The underlying data structure that the Packet interfaces expose access to is the `message_t` data buffer type. This data buffer has been designed to facilitate 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 by keeping the header, payload and footer of a message contiguous. This results in less memory use and processing time when dealing with messages. The `message_t` type is defined as follows (Levis 2007b):

```
typedef nx_struct message_t {
    nx_uint8_t header[sizeof(message_header_t)];
    nx_uint8_t data[TOSH_DATA_LENGTH];
    nx_uint8_t footer[sizeof(message_footer_t)];
    nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

The `nx_` prefix is a nesC construct that allows the structure to be packed in big endian format regardless of the way the processor natively stores values, this means that different processors can be present on the network and exchange messages in an interoperable manner. (Levis 2006)

The `message_header_t`, `message_footer_t` and `message_metadata_t` types are defined as part of a platform. They are the union of all the header, footer and metadata structures for each type of data link layer present in the platform respectively. This way, the size of each field in the message buffer will be large enough to store all the different types of packets the system will handle. (Levis 2007b)

In order to keep the header, footer and payload fields contiguous, the header field is to be right-aligned in the header portion of the message buffer. Figure 5.2 visually demonstrates the placement of data in the `message_t` type for a hypothetical system

```
              14 Bytes       TOSH_DATA_LENGTH Bytes     4B  7 Bytes
          +--------------+----------------------------+----+-------+
message_t |    header    |           data             |foot| meta  |
          +--------------+----------------------------+----+-------+

          +--------------+----------------------------+    +-------+
Ethernet  |    header    |           data             |    | meta  |
          +--------------+----------------------------+    +-------+

                      +--+-------+                    +----+
CAN                   |hd| data  |                    |foot|
                      +--+-------+                    +----+

                      +--+-------+----+
CAN (contiguous)      |hd| data  |foot|
                      +--+-------+----+
```

Figure 5.2: Illustration of message_t type and data placement for a hypothetical system with CAN and Ethernet data link layers

that has CC2420 (IEEE 802.15.4) and CAN data link layers. It also demonstrates that the footer can be optionally stored in the data segment of the message buffer so it is contiguous with the payload when the payload is smaller than the total length of the data segment. (Levis 2007b)

The ability to access different fields within the message buffer is exposed by the aforementioned Packet interfaces. Each data link layer will have a component that exposes a Packet interface to set and get these fields before sending across that link and after receiving on that link. A hierarchy of Packet interfaces can be created using wiring to facilitate nested access for the hierarchy of networking layers as demonstrated by figure 5.3. (Levis 2007b; Levis and Gay 2009)

The ability to send and receive messages is facilitated by the Send and Receive interfaces respectively. Send provides commands to send and cancel a message and an event to indicate when the sending is finished. Receive simply provides an event for receiving packets. The Send interface can be extended to provide the ability to send with different addressing, such as the AMSend interface which allows the address of the node to send the packet to be specified. (Levis 2008)

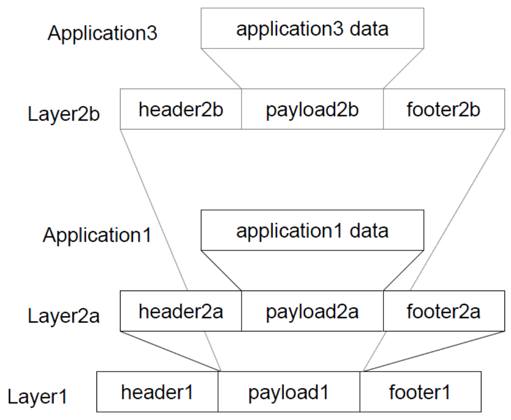


Figure 5.3: Wiring Packet interfaces into a hierarchy facilitates the ability to manage multiple network layers within the message buffer structure (Levis and Gay 2009)

At a basic data link layer level, TinyOS provides interfaces and modules to manage Active Messages (AM): a single-hop, unreliable packet. These packets have a source and destination address, provide synchronous acknowledgements and are allowed to occupy a variable length (up to a maximum value). They also have a type field identifying the protocol used to send that message. (Levis 2008)

TinyOS also provides higher level services for multi-hop communication with in-built components and interfaces for tree collection (motes organise themselves into a tree, with data collected and sent up the tree to the root; typically the base station) and dissemination (efficient distribution of a value across the entire network). (Levis and Gay 2009)

5.4.1.5 Timers

As outlined by Sharp et al. (2007) most microcontrollers offer a variety of features from their timer subsystems, such as:

- Several counters possibly with different widths and clocking options; and
- One or more compare registers per counter, allowing the ability to trigger interrupts, change output pins, change the counter value or capture the time of input pin changes.

Different microcontrollers have different frequency crystals (although it is claimed that a 32768Hz crystal is the most common frequency (Sharp et al. 2007)). As such TinyOS emphasises a 32-bit width timer but accommodates other precisions. Thus it is stated that the closest precision to a given timing increment (for example milliseconds as 1024 cycles per second) should be chosen based on the accuracy afforded by the given hardware platform. (Sharp et al. 2007)

TinyOS provides a number of interfaces with which to access the timer subsystem, most are parameterised by precision, but some by width. To facilitate the parameterisation of precision some "dummy" typedefs are created like so (ibid):

```
typedef struct { int notUsed; } TMilli; // 1024 ticks per second
typedef struct { int notUsed; } T32khz; // 32768 ticks per second
typedef struct { int notUsed; } TMicro; // 1048576 ticks per second
```

TinyOS provides the following interfaces for dealing with the timer subsystem (ibid):

- **Counter** - Allows the application to access the current time and manage overflows
- **Alarm** - Allows an event to be fired after a certain time is reached
- **BusyWait** - Allows a busy wait for a given period of time
- **LocalTime** - Facilitates access to a 32-bit counter without worrying about overflow conditions
- **Timer** - Allows periodic and one-shot time intervals to be flagged using an event

A platform is expected to implement Counters and Alarms for all the possible precisions and widths supported by the hardware timers available. They should be named using the convention Counter<P><W>C and Alarm<P><W>C, where <P> is the precision and <W> is the width. Furthermore, a platform must provide implementations for HilTimerMilliC and BusyWaitMicroC in order to allow core TinyOS components to function correctly. (ibid)

In order to allow a system to utilise an extended set of timing features a number of platform independent utility classes have been defined, namely (ibid):

- **AlarmToTimerC** - Converts a 32-bit Alarm to a Timer
- **BusyWaitCounterC** - Uses a counter to block for a specified amount of time
- **CounterToLocalTimeC** - Converts a 32-bit counter to LocalTime
- **TransformAlarmC** - Uses a widened counter to decrease precision or widen an Alarm
- **TransformCounterC** - Decreases precision or widens a counter
- **VirtualizeTimerC** - Uses a single Timer to create up to 255 virtual timers

5.4.1.6 Sensors

There are three main ways in which sensor access can be classified in TinyOS (Tolle et al. 2008):

- **Phase** - The time between requesting data and receiving data from a sensor can be enumerated by phase: single (the data is immediately received), split (the data is requested, and some time later the data is received by an event signal) and trigger (the data is triggered, rather than request driven).
- **Data Type** - The sensor data can either be scalar (single value) or stream data (multiple values).
- **Scope** - Sensors can be application or platform specific. Platform specific sensors are built in to the hardware platform. Application specific sensors are "plugged" into the hardware platform to go with a particular application. Application specific sensors will have the sensor API implemented as part of the application code, and platform specific sensors will have the sensor API implemented as part of the platform code.

TinyOS introduces the concept of Source (the sensor API) and Sink (the application using the sensor) Independent Drivers (SIDs) to provide a separation between application code and sensor code. This concept is realised by four interfaces (Tolle et al. 2008):

1. **Read**: Used for split-phase, scalar sensor readings (sensors with low sensing rates)
2. **Get**: Used for single-phase, scalar sensor readings (sensors with cached or readily available data)
3. **Notify**: Used for triggered, scalar sensor readings (sensors that are triggered, intended for low-rate events)
4. **ReadStream**: Used for split-phase, vectored sensor readings (useful for applications that can handle continuous streams of new sensor data or sensors that produce data at high rates)

All four interfaces are parameterised by a `val_t` type, which can be set to any type (for example **`uint8_t`** for 8-bit unsigned sensor data, **`int16_t`** for 16-bit signed sensor data).

5.4.1.7 Power management

Apart from the power management features of the component, as mentioned in section 5.4.1.3, TinyOS provides a number of interfaces and components that platforms can utilise to provide applications with the ability of managing the power level of the system, as well as the current power use and remaining power level (Szewczyk et al. 2007). Furthermore, TinyOS provides facilities for managing the power use of various resources by controlling when they are enabled (Klues et al. 2008).

5.4.2 Porting progress

The following list outlines the steps that were completed, and have yet to be completed, as part of the attempted port of TinyOS to the HCS12 / Dragon 12 development platform:

- **HCS12 Chip**
  - ✓ **Hardware Include File** – Created `tos/chips/hcs12/hcs12hardware.h` with inttypes.h, registers and interrupt vectors includes (and associated files), interrupt and atomic section routines and sleep and power routines
  - ✓ **General Purpose I/O (GPIO) Pins** – Created HplHcs12GeneralIOC configuration that exposes a GeneralIO interface for each GPIO pin and a HplHcs12PortIO interface for each GPIO port, see `tos/chips/hcs12/gpio/*.nc`
  - ✗ **McuSleepC Module** – Created a skeleton for this module (`tos/chips/hcs12/McuSleepC.nc`) but have not completed it
  - ✗ **CAN Components and Interfaces** – Have not attempted
  - ✗ **Timer Components and Interfaces** – Have not attempted
  - ✗ **ADC Components** – Have not attempted (possibly not relevant for this project)
  - ✗ **Other I/O** – Have not attempted to add the non-CAN I/O such as UART and I^{2}C (not relevant for this project)
- **Dragon12 Platform**
  - ✓ **.platform File** – Created `tos/platforms/dragon12/.platform`
  - ✓ **Hardware Include File** – Created `tos/platforms/dragon12/hardware.h`
  - ✓ **Platform Include File** – Created `tos/platforms/dragon12/platform.h`
  - ✓ **PlatformC and PlatformP Components** – Created `tos/platforms/dragon12/PlatformC.nc` and `tos/platforms/dragon12/PlatformP.nc`, which perform platform initialisation

- ✓ **PlatformLedsC Component** – Created `tos/platforms/dragon12/PlatformLedsC.nc`, which exposes the LEDs made available from `tos/platforms/dragon12/PlatformIOC.nc` and `tos/platforms/dragon12/PlatformIOP.nc`
- ☒ **Power Management** – Have not attempted to add power measurement or management components
- ☒ **I/O** – Have not attempted to add non LED I/O such as 7-segment displays, push buttons and LCD panel (not relevant for this project)
- ☒ **Sensors** – Have not attempted to add any Dragon12 sensors (not relevant for this project)

- **Make System Integration**
  - ✓ **Make Target File** – Created `support/make/dragon12.target`
  - ✓ **HCS12 Make Files** – Created `support/make/hcs12.rules` (not completely finished, there are redundant rules in there from the platform that the file was copied from and there is no load rule in there) and `support/make/debug.extra`
- **Testing**
  - ✓ **Compile against Null** – Completed successfully
  - ☒ **Run Null** – Not able to do
  - ☒ **Compile and run against Blink** – Not attempted
  - ☒ **Compile and run each new application in turn** – Not attempted
- **UAV Sensor Network**
  - ☒ **MSRN Integration** – Have not attempted
  - ☒ **Phase 1 Application** – Have not attempted
  - ☒ **Phase 2 Application** – Have not attempted
  - ☒ **Phase 3 Extensions** – Have not attempted

5.4.3 Difficulties encountered

After TinyOS was chosen as the operating system on which the system would be developed, a number of weeks were spent researching TinyOS and nesC in order to try and work out how to port TinyOS to a new platform and write nesC code. Even more weeks were spent trying to accomplish the actual port. A few reasons for this are that the learning curve for nesC slowed down progress and the documentation for porting, whilst a good first step, is by no means complete (Leopold 2008; Wiki 2008). However, the main reason is that the porting process itself is not streamlined: there are not clear boundaries of what needs to be implemented in a chip and a platform and it cannot be tested until all the base functionality is there. Essentially you are relying on the compiler to tell you what is missing in order to determine the minimum requirements. Then it may or may not work when loaded on the platform and there are a number of files that could have a bug if it does not work.

When these difficulties in the porting process first started to emerge, this led to the idea that it was possibly a better use of time to utilise an existing processor that TinyOS has already been ported to. However, after consideration, it was decided to keep on trying to port to the HCS12 for the following reasons:

- It was helping with the process of learning nesC;
- Even if one of the processors TinyOS has already been ported to was used, some sort of porting would need to occur because the platforms TinyOS has been ported to are not suited to this particular application, and the MSRN would need to be integrated;
- Multiple HCS12s on the Dragon12 development board were readily available at the university for network testing;
- Ordering multiple units of another platform would cost too much time and money;
- The HCS12 has multiple CAN ports, so multiple point-to-point links would be able to be tested; and

- Previous experience (and thus familiarity) with the HCS12 meant that time did not need to be spent learning a new hardware platform.

Once the platform was in a state where it compiled, an attempt was made to load the output onto the development board. After searching the Internet, Linux drivers for the P&E Micro Multilink Universal Serial Bus (USB) Background Debug Mode (BDM) cable that was being used were not able to be found. Further research led to the realisation that there were Linux drivers for the Coldfire BDM cable (albeit the parallel port version only), but there were not USB drivers for it (P&E Micro 2006). After posting on the P&E Micro Forum, there was confirmation that there are not any Linux drivers for the HCS12 BDM cable, USB or otherwise (P&E Micro 2009).

At this point a number of different options were considered:

- Attempting to load the output from Linux (in ELF, iHex, S19 and assembly formats) onto the microcontroller using IAR Embedded Workbench, but the output was not compatible
- Purchasing P&E Micro software to load output onto the microcontroller, but this would be hard to debug
- Finding software or hardware to program the HCS12 on Linux, but none could be found
- Choosing a different development platform: this means that the work done on porting the HCS12 is relatively fruitless and all the advantages of using the HCS12 platform mentioned above are lost
- Keeping the HCS12 platform and implementing an operating system based on TinyOS that can be loaded on the HCS12

An initial design of the last option demonstrated that the implementation of an operating system was feasible within the time constraints of the project, and therefore this was the chosen option.

5.5 SensorOS

5.5.1 Overview

The operating system mentioned in the previous section was designed and mostly implemented, it has been named SensorOS. When developing SensorOS the following features were incorporated:

- Encompassing many of the useful patterns, ideas and abstractions present in TinyOS so that the advantages of choosing TinyOS as the operating system for development could still be realised with the new operating system and the existing research and work on TinyOS could be leveraged.
- Keeping the operating system as simple as possible to reduce development time and make application development and porting as easy as possible.
- Ensuring that the operating system is as efficient as possible to conserve computation and power resources.
- A focus on making it easy to port between platforms so any developers using SensorOS would not come across the same problems that were faced in this project with TinyOS (through the design, code structure and documentation of the operating system).
- A focus on making an operating system that will be compatible with a wide range of development environments, for the same reason (both in terms of hardware platforms and compilers).
- Introducing more structure in the handling of networking, more specifically, a well defined interface between the platform and application networking code (making both porting and application development easier) and explicit support for multiple data-link layers (to directly support multiple point-to-point links like those in the application this project was looking at).

SensorOS is appropriate for use in:

- Sensor network applications where TinyOS cannot be used. Otherwise, TinyOS should be used so that the large developer community and base of existing applications and add-ons can be leveraged
- The possible exception to this is where a new platform needs to be ported, since porting is much easier in SensorOS than TinyOS
- Simple event driven systems where a FIFO scheduler is acceptable
- Teaching basic operating system concepts (such as scheduling, networking, communications and race conditions)

5.5.2 Language Choice

The two main goals of the operating system that affected the language choice were the need to support the patterns and abstractions in TinyOS and the goal of making the language portable between platforms and compilers. The first of those goals points to an object oriented language to support the component, interface and design pattern ideas present in TinyOS. The second of these goals points to a language such as C which has compilers for most hardware platforms, provides suitable low-level commands to interface directly with hardware (necessary for an operating system) but is high-level enough to be able to program in a platform-independent manner (with the help of the principle of hardware abstraction layers (HALs), another concept utilised by TinyOS).

A logical result when considering those two goals is the choice of C++ as the language for the operating system. This language choice allows the operating system to be built in a platform-independent manner via the use of well defined HALs and the adoption of various best practices for writing portable C/C++, such as the guidelines of MISRA C (Jones 2002).

It should be noted that there are a number of disadvantages in using C++ (over C or nesC):

- There are fewer embedded compilers that support C++ compared to C
- In particular, there is less compiler support for templates among embedded compilers since the Embedded C++ (EC++) standard does not support templating (Plauger 1997), however various compilers have been created which support extended versions of EC++ (IAR Systems n.d.; Plauger 1997)
- There are situations in which C++ code can be less efficient than plain C and nesC (for example virtual classes)
- There are situations in which the C++ output has more function calls than nesC because of its lack of bi-directional wiring and static linking of wired modules
- C++ lacks the ability to perform bi-directional wiring like nesC, so less efficient methods need to be used to leverage similar flexibility

It was felt that the advantages of using it outweighed the disadvantages given the goals that were trying to be accomplished. Of particular interest, Herity (1998) outlines a comprehensive list of why C++ is a good language for embedded software development.

5.5.3 Initial Design

As mentioned in section 5.4.3, an initial design was created before accepting the implementation of an operating system as the approach to take. This design was created by firstly identifying the main parts of TinyOS that could be put together to form an operating system:

- Leverage similar flexibility of bi-directional wiring
- Split-phase interfaces and event propagation
- Platform and application initialisation

- Timer subsystem
- Sensor subsystem
- Message buffer with zero copy semantics
- Interface between data link layer (platform code) and network layer (application code)
- Ability to build on top of network layer
- FIFO scheduler with power management
- Atomic Sections

The main problem that needed to be resolved was how to accomplish the flexibility afforded by nesC's bi-directional wiring in a language that did not natively support bi-directional wiring. This lack of flexibility is mostly a problem when code from platforms and code from applications need to interact with each other, since the exact interactions cannot be known at the time of writing. The bi-directional wiring problem was overcome by looking at each different part of the operating system that used such wiring in TinyOS and devising suitable replacements:

- **Split-phase interfaces and event propagation** – A singleton signal router could be created that is informed when an event occurs and could then inform any registered handlers. That way, the bi-directional wiring is simulated by the signal router acting as an intermediary, with two single-directional "wirings" to the signal router. However, the disadvantage of this approach is the addition of a layer of indirection and having to maintain information about the signal handlers. This means it is less efficient in terms of both computation resources and memory use. Regardless, this was thought to be the most elegant option.
- **Initialisation** – Applications and Platforms can be initialised without bi-directional wiring by including all the initialisation code for the application / platform in one place, this is not quite as flexible as the TinyOS method, but is simpler to understand.

- **Timers** – A singleton timer manager object could take care of allocating and releasing timers to applications or platforms that request them. Apart from the memory and computation disadvantages mentioned previously, this has the disadvantage that the number of timers requested is not known at compile time, leading to a possible run time error condition which needs to be dealt with appropriately.
- **Sensors** – A singleton sensor manager object could take care of facilitating access to the sensors, which has the aforementioned extra memory and computation disadvantages.
- **Interface between the data link layer and network layer** – A singleton network object can manage the connection between these layers. As there is only one network layer the extra memory overhead of this approach is not as significant.
- **Layers on top of the network layer** – These layers are all contained within a particular application so the interactions between objects can be resolved by application developers.

While many of these approaches have some disadvantages over the TinyOS approach, the end result is a system that is as flexible as TinyOS, simpler to understand (than layers of bi-directionally wired components) and the increased computation resources and memory use needed should not have too large an impact on the final system.

A decision was made to create interfaces for the networking interface between the data link layer and network layer. While interfaces are not directly supported in C++, clever utilisation of the preprocessor in combination with virtual classes allows the equivalent functionality and syntax (Rios 2005). It was also decided that the preprocessor could be used to simulate the `atomic` construct from nesC by introducing **`Atomic`** and **`EndAtomic`** commands that call `__atomic_start` and `__atomic_end` routines that a platform defines in the same way as the `__nesc_atomic_start` and `__nesc_atomic_end` routines in TinyOS.

The Timer and Sensor subsystems were left out of the initial design so that only the essential parts of the operating system could be focused on in the initial implementation, with these subsystems able to be added later.

The Packet interfaces in TinyOS allow higher layers to be wired on top of the network layer in a uniform manner. However, this still requires effort from application designers in order to implement the modules and the layers are different for every application. Thus it was decided that no specific functionality would be placed in the operating system to support networking above the network layer.

It was decided to make the scheduler a copy of the TinyOS scheduler, including the call to put the processor into a low power state when the queue is exhausted. The scheduler would be implemented as a singleton object in order to facilitate its controlled global access.

It was decided that the Platform, Application and Scheduler Initialisation could be taken care of by directly transforming the `main` function from TinyOS to the new operating system. In order to facilitate this transformation, in combination with the singleton objects mentioned above, the design included the creation of Platform and Application namespaces that each had static methods that can be called from `main`. The code from TinyOS and equivalent pseudocode for the new operating system is shown in table 5.1. The TinyOS code is from `tos/system/RealMainP.nc` in TinyOS 2.1.0. It should be noted that the operating system design included an extra two steps in the main function between steps six and seven, which were the initialisation of the application before interrupts are enabled (`Application::init()`) followed by the running of any posted tasks.

In combination with the formulation of C++ appropriate solutions for all relevant TinyOS features, a Unified Modeling Language (UML) design was completed to illustrate the potential signatures of all the major classes and interfaces. The final UML description of the resulting operating system can be found in the appendices of the programming manual found in Appendix D.

5.5.4 Development approach

Whilst developing the operating system, each part of the operating system was taken in turn, the initial design for that subsystem reviewed, the implementation performed and a unit test created and run with IAR Embedded Workbench. A second review was then undertaken after the section passed the unit test to analyse the implemented solution for efficiency and compiler and platform compatibility. The order of implementation for the subsystems was determined by initially analysing the order of importance of the subsystems for the operating system as a whole.

At various points the Makefile was run to test whether the code could compile with GCC, which tested the cross compiler compatibility of the operating system. When a bug was found in a unit test a combination of breakpoints, stepping, judicious use of the Debug class and the memory dump ability in IAR Embedded Workbench were used to find the cause of the bug. When the cause was found, a solution was formulated and the unit test re-run to determine if the problem had been resolved.

5.5.5 Difficulties encountered

A number of difficulties were encountered and resolved whilst developing the operating system, some of the more significant difficulties and solutions were:

- Learning advanced Make syntax in order to develop a set of Makefiles that make compiling SensorOS applications easy when using GCC. See the "Compiling SensorOS" subsection in section 1 of Appendix D for more information.

- Deciding how to write compiler targeted code such as interrupt handlers and register definitions. The solution used involved conditional compilation by utilising macros that identify the compiler being used. See the "Compiler Compatibility" and "Interrupt Service Routines" subsections in section 3 of Appendix D for more information.
- Choosing an implementation approach for the encapsulation of function pointers for scheduler tasks and signal router handlers. The chosen method was functor objects, which allow static and instance method pointers to be able to be used. This had the expense of an extra level of indirection due to the need of a virtual base class, but provided flexibility. See the "VoidFunctor" subsection in section 2 of Appendix D for more information.
- Deciding whether to allow tasks to be posted multiple times to the scheduler, or to adopt the TinyOS approach of allowing a task to be posted once only. In order to preserve efficiency by avoiding an $O(n)$ lookup of the scheduler task list to see if a task had already been posted, tasks were allowed to be posted multiple times. This had the added benefit that a primitive "priority" could be given to tasks by posting them multiple times. It does mean the application and platform developers need to be careful to ensure negative effects do not result from this fact. See the "Scheduler" subsection of section 2 of Appendix D for more information.
- Difficult to understand documentation on the CAN controller for the HCS12 and lack of mention that the bit timing registers need to be set within the CAN standard timing requirements for the module to function. Eventually the documentation was understood and the timing requirements problem was identified through unit testing.
- Creating a reusable, efficient CAN implementation including register access while minimising source code repetition. The selected solution was to create a static, constant array of the initial memory address for each 5 CAN modules in the HCS12, and adding an offset via a precompiler macro for the CAN module

being used. Theoretically, compiler optimisation should then be able to statically determine the memory address for each lookup, meaning the generated code is a lot more efficient. Unfortunately, the fact the CAN module identifier is an instance variable means this compiler optimisation cannot occur. A possible future solution is the use of templates to pass the can link identifier in as a static constant. However, this increases code memory since each CAN module will need a separate code implementation and would only be feasible if one or two CAN modules are being used.

- Deciding how to write platform targeted code in an application. The solution used involved conditional compilation by utilising macros that identify the platform being used. See the "Platform Targeted Application Code" subsection in section 2 of Appendix D for more information.
- Realisation that the C++ Standard Template Library (STL) was not available on all compilers and the list construct did not seem to be very efficient for usage as a simple singly-linked list. The solution was to create a singly linked list that had a subset of the STL list features, but conformed to the same interface as STL (including iterators) and was optimised for embedded systems. See the "List" subsection in section 2 of Appendix D.
- Working out how to ensure that as much memory is statically allocated as possible to improve system performance. See subsection "Static Allocation" in section 2 of Appendix D.

5.5.6 Remaining work

In order to finalise the first version of SensorOS the following actions will need to be taken:

- Detailed design and implementation of the timer subsystem, see subsection "Timers" in section 2 of Appendix D;
- Detailed design and implementation of the sensor subsystem, see subsection

“Sensor Network Applications” in section 2 of Appendix D;

- Final testing of the CANLink unit test;
- Testing of integration with the Null application;
- Creation of more sophisticated testing applications; and
- Using the operating system for the creation of an actual control system.

Order	Activity	Code
1.	Bootstrap	TinyOS: `platform_bootstrap()` New OS: `Platform::bootstrap()`
2.	Initialise Scheduler	TinyOS: `call Scheduler.init()` New OS: `scheduler = Scheduler::getInstance()`
3.	Initialise platform	TinyOS: `call PlatformInit.init()` New OS: `Platform::init()`
4.	Run any tasks	TinyOS: `while (call Scheduler.runNextTask())` New OS: `while (scheduler->runNextTask())`
5.	Initialise software	TinyOS: `call SoftwareInit.init()` New OS: `signalRouter = SignalRouter::getInstance()` `signalRouter->signal(SIG_SOFTWARE_INIT)`
6.	Run any tasks	TinyOS: `while (call Scheduler.runNextTask())` New OS: `while (scheduler->runNextTask())`
7.	Enable interrupts	TinyOS: `__nesc_enable_interrupt()` New OS: `__enable_interrupts()`
8.	Signal system booted	TinyOS: `signal Boot.booted()` New OS: `signalRouter->signal(SIG_SYSTEM_BOOTED)`
9.	Run scheduler loop	TinyOS: `call Scheduler.taskLoop()` New OS: `scheduler->taskLoop()`

Table 5.1: Main function: TinyOS versus the new operating system design

6.0 CONCLUSIONS

6.1 Analysis of the project outcomes

This project has a number of important outcomes, including:

- The implementation of an embedded operating system targeted at sensor networks;
- The formalisation of requirements for a structured development approach of the control system of the UAV sensor network outlined in section 2.5, see Appendix A;
- Extensive research into sensor networks, including the relevant points for the UAV sensor network control system; and
- Composition of a categorised reading list for any future developers working on the UAV control system.

As outlined in section 5.5.1, there are a number of other uses for SensorOS than sensor network control systems. It can also be used for simple event-driven systems and for teaching basic operating system concepts. The programming manual that was created for SensorOS ensures that any SensorOS application or platform development should present few problems.

The combination of the requirements specification (Appendix A), the summary of sensor network research from the perspective of a UAV sensor network (see section 3.3) and the reading list (Appendix B) ensure that any development that carries on from where this project concludes should have a solid starting ground.

6.2 Future Work

There are two avenues of future work that could be accomplished. Firstly, SensorOS should be completed by effecting all the points outlined in section 5.5.6. Secondly, the UAV sensor network control system should be designed, implemented and integrated with the finished MSRN and physical layer hardware.

A final note about SensorOS: future work would include the porting of SensorOS to different hardware platforms as required. SensorOS could be refined if a diverse range of developers created an equally diverse range of applications.

REFERENCES

Adler, R., M. Flanigan, J. Huang, R. Kling, N. Kushalnagar, L. Nachman, C. Wan, and M. Yarvis. 2005. Intel mote 2: an advanced platform for demanding sensor network applications. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, *2-4 November, 2009*, *San Diego, CA, USA*, 298. ACM.

Akkaya, K. and M. Younis. 2005. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks* 3 (3): 325–349.

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

Ashraf, A., M. Hashmani, B. Chowdhry, M. Mussadiq, Q. Gee, K. Rajput, and A. Qadeer. 2008. Design and analysis of the security assessment framework for achieving discrete security values in wireless sensor networks. In *Canadian Conference on Electrical and Computer Engineering*, *4-7 May, 2008*, *Niagara Falls, Canada*, 855–860. IEEE.

Barr, M. and B. Frank. 1997. Java: Too much for your system? *Embedded Systems Programming* 10 (5): 102–114.

Barrows, G. 2002. Future visual microsensors for mini/micro-UAV applications. In *7th IEEE International Workshop on Cellular Neural Networks and their Applications*, *22-24 July, 2002*, *Frankfurt, Germany*, 31–36. IEEE.

Bischoff, R., J. Meyer, and G. Feltrin. 2009. Wireless sensor network platforms. *Encyclopaedia of Structural Health Monitoring*: 1229–1238.

Bose, R. 2009. Sensor networks–motes, smart spaces, and beyond. *IEEE Pervasive Computing* 8 (3): 84–90.

Boulis, A., S. Ganeriwal, and M. Srivastava. 2003. Aggregation in sensor networks: an energy–accuracy trade-off. *Ad hoc networks* 1 (2-3): 317–331.

Braginsky, D. and D. Estrin. 2002. Rumor routing algorthim for sensor networks. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, *28 September, 2002*, *Atlanta, GA, USA*, 22–31. ACM.

Busse, M., T. Haenselmann, T. King, and W. Effelsberg. 2006. The impact of forward error correction on wireless sensor network performance. In *Proceedings of ACM Workshop on Real-World Wireless Sensor Networks*, *19 June, 2006*, *Uppsala, Sweden*. ACM.

Cao, Y., C. He, and L. Jiang. 2007. Energy-efficient routing for mobile agents in wireless sensor networks. *Frontiers of Electrical and Electronic Engineering in China* 2 (2): 161–166.

Chen, M., T. Kwon, Y. Yuan, Y. Choi, and V. Leung. 2007. Mobile agent-based directed diffusion in wireless sensor networks. *EURASIP Journal on Advances in Signal Processing* 2007 (1): 219–219.

Chong, C. and S. Kumar. 2003. Sensor networks: Evolution, opportunities, and challenges. *Proceedings of the IEEE* 91 (8): 1247–1256.

Chu, M., H. Haussecker, and F. Zhao. 2002. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. *International Journal of High Performance Computing Applications* 16 (3): 293.

Civil Aviation Authority. 1998. Civil Aviation Safety Regulations 1998 (CASR) Part 101.

Coffin, D., D. Van Hook, S. McGarry, and S. Kolek. 2000. Declarative ad-hoc sensor networking. In *Proceedings of the Conference on Integrated Command Environments*, *31 July, 2000*, *San Diego, CA, USA*, 109–120. SPIE.

Crossbow. n.d.-a. IRIS datasheet. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/IRIS_Datasheet.pdf, (accessed 21 February 2009).

Crossbow. n.d.-b. MICA2 datasheet. http://www.xbow.com/products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf, (accessed 21 February 2009).

Crossbow. n.d.-c. MICAZ datasheet. http://www.xbow.com/products/Product_pdf_files/Wireless_pdf/MICAZ_Datasheet.pdf, (accessed 21 February 2009).

Dai, H. and R. Han. 2004. Tsync: a lightweight bidirectional time synchronization service for wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review* 8 (1): 125–139.

Dopplinger, A. and J. Innis. 2007. Using IEEE 1588 for synchronization of network-connected devices. http://www.embedded.com/columns/technicalinsights/198500595, (accessed 20 October 2009).

eCos. n.d. eCos Home Page. http://ecos.sourceware.org/, (accessed 22 October 2009).

Elson, J. and D. Estrin. 2001. Time synchronisation for wireless sensor networks. In *IPDPS Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, *27 April, 2001*, *San Francisco, CA, USA*.

Elson, J., L. Girod, and D. Estrin. 2002. Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS Operating Systems Review* 36: 147–163.

Elson, J. and K. Römer. 2003. Wireless sensor networks: A new regime for time synchronization. *ACM SIGCOMM Computer Communication Review* 33 (1): 149–154.

Eschenauer, L. and V. Gligor. 2002. A key-management scheme for distributed sensor networks. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, *18-22 November, 2002*, *Washington, DC, USA*, 41–47. ACM.

FreeRTOS. 2009. FreeRTOS - Features Overview. http://www.freertos.org/, (accessed 22 October 2009).

Ganeriwal, S., R. Kumar, and M. Srivastava. 2003. Timing-sync protocol for sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, *5-7 November, 2003*, *Los Angeles, CA, USA*, 138–149. ACM.

Gay, D., P. Levis, D. Culler, and E. Brewer. 2005. nesC 1.2 language reference manual.

Guestrin, C., P. Bodik, R. Thibaux, M. Paskin, and S. Madden. 2004. Distributed regression: an efficient framework for modeling sensor network data. In *Proceedings of the Third International Symposium on Information Processing in Sensor Networks*, *26-27 April, 2004*, *Berkeley, CA, USA*, 1–10. ACM.

Guo, C., L. Zhong, and J. Rabaey. 2001. Low power ditributed MAC for ad hoc sensor radio networks. In *IEEE Global Telecommunications Conference*, Volume 5, *25-29 November, 2001*, *San Antonio, TX, USA*, 2944–2948. IEEE.

Harrison, C., D. Chess, and A. Kershenbaum. 1997. Mobile agents: Are they a good idea. *Mobile Object Systems: Towards the Programmable Internet* 1222: 25–47.

Heinzelman, W. R., A. Chandrakasan, and H. Balakrishnan. 2000. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Volume 8, *4-7 January, 2000*, *Wailea Maui, HI, USA*, 3005–3015. IEEE.

Heinzelman, W. R., J. Kulik, and H. Balakrishnan. 1999. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, *15-19 August, 1999*, *Seattle, WA, USA*, 174–185. ACM.

Herity, D. 1998. C++ in Embedded Systems: Myth and Reality. *Embedded Systems Programming* 11 (2): 48–71.

Hla, K., Y. Choi, and J. Park. 2008. The multi agent system solutions for wireless sensor network applications. *Lecture Notes in Computer Science* 4953: 454.

Hu, A. and S. Servetto. 2003. Asymptotically optimal time synchronization in dense sensor networks. In *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications*, *19 September, 2003*, *San Diego, CA, USA*, 10. ACM.

Hui, J. and D. Culler. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, *3-5 November, 2004*, *Baltimore, MD, USA*, 81–94. ACM.

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

IEEE. 1998. IEEE Standard 830-1998: IEEE recommended practice for software requirements specifications.

Intanagonwiwat, C., R. Govindan, and D. Estrin. 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, *6-11 August, 2000*, *Boston, MA, USA*, 56–67. ACM.

Ishikawa, K. and A. Mita. 2008. Time synchronization of a wired sensor network for structural health monitoring. *Smart Materials and Structures* 17 (1): 15016–15021.

Iyengar, S. S., D. N. . Jayasimha, and D. Nadig. 1994. A versatile architecture for the distributed sensor integration problem. *IEEE Transactions on Computers* 43 (2): 175–185.

Jaman, G. and S. Hussain. 2007. Structural monitoring using wireless sensors and controller area network. In *Proceedings of the Fifth Annual Conference on Communication Networks and Services Research*, *14-17 May, 2007*, *Fredericton, Canada*, 26–34. IEEE.

Jayasimha, D., S. Iyengar, and R. Kashyap. 1991. Information integration and synchronization in distributed sensor networks. *IEEE Transactions on Systems, Man, and Cybernetics* 21 (5): 1032–1043.

Jeong, J. and C. Ee. 2006. Forward error correction in sensor networks. University of California, Berkeley.

Jones, N. 2002. Introduction to MISRA C. *Embedded Systems Programming* 15 (7): 55–56.

Kim, S., S. Son, J. Stankovic, S. Li, and Y. Choi. 2003. SAFE: A data dissemination protocol for periodic updates in sensor networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*, *19-22, May 2003*, *Providence, RI, USA*, 228–234.

Klein, L. A. 1999. *Sensor and Data Fusion Concepts and Applications*. 2nd ed. Bellingham, WA, USA: Society of Photo-Optical Instrumentation Engineers (SPIE).

Klues, K., V. Handziski, J.-H. Hauer, and P. Levis. 2008. TEP 115: Power management of non-virtualised devices. http://www.tinyos.net/tinyos-2.x/doc/html/tep115.html, (accessed 26 October 2009).

Korber, H. J., H. Wattar, G. Scholl, and W. Heller. 2005. Embedding a Microchip PIC18F452 based commercial platform into TinyOS. In *Workshop on Real-World Wireless Sensor Networks*, *20-21 June, 2005*, *Stockholm, Sweden*.

Krishnamachari, B., D. Estrin, and S. Wicker. 2002. The impact of data aggregation in wireless sensor networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, *2-5 July, 2002*, *Vienna, Austria*, 575–578. IEEE.

Krishnamachari, B. and S. Iyengar. 2004. Distributed bayesian algorithms for fault-tolerant event region detection in wireless sensor networks. *IEEE Transactions on Computers* 53 (3): 241–250.

Kulkarni, S. and M. Arumugam. 2006. Infuse: A TDMA based data dissemination protocol for sensor networks. *International Journal of Distributed Sensor Networks* 2 (1): 55–78.

Kumar, R., M. Wolenetz, B. Agarwalla, J. S. Shin, P. Hutto, A. Paul, and U. Ramachandran. 2003. Dfuse: a framework for distributed data fusion. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, *5-7 November, 2003*, *Los Angeles, CA, USA*, 114–125. ACM.

Kuris, B. and T. Dishongh. 2006. SHIMMER - sensing health with intelligence, modularity, mobility, and experimental reusability: Hardware guide. http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/SHIMMER_HWGuide_REV1P3.pdf, (accessed 23 October 2009).

Lange, D. B. and M. Oshima. 1999. Seven good reasons for mobile agents. *Communications of the ACM* 42 (3): 88–89.

Lee, J. and D. Stinson. 2004. Deterministic key predistribution schemes for distributed sensor networks. In *Proceedings of the 11th International Workshop on Selected Areas in Cryptography*, *9-10, August, 2004*, *Waterloo, Canada*, 294–307. Springer-Verlag.

Lentsch, T. 2005. The eyesifx platform. http://www.tinyos.net/ttx-02-2005/platforms/ttx2005-eyesIFX.ppt, (accessed 21 February 2009).

Leopold, M. 2008. TEP 131: Creating a new platform for TinyOS 2.x. http://www.tinyos.net/tinyos-2.x/doc/html/tep131.html, (accessed 26 October 2009).

Levis, P. 2006. TinyOS programming.

Levis, P. 2007a. TEP 1: TEP structure and keywords. http://www.tinyos.net/tinyos-2.x/doc/html/tep1.html, (accessed 23 October 2009).

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

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

Levis, P. and D. Gay. 2009. TinyOS programming.

Li, E. L. and J. Y. Halpern. 2001. Minimum-energy mobile wireless networks revisited. In *IEEE International Conference on Communications*, Volume 1, *11-14 June, 2001*, *Helsinki, Finland*, 278–283. IEEE.

Li, L., R. Maunder, B. Al-Hashimi, and L. Hanzo. 2009. An energy-efficient error correction scheme for IEEE 802.15.4 wireless sensor networks. *IEEE Transactions on Circuits and Systems II*.

Li, Q., R. Peterson, M. DeRosa, and D. Rus. 2003. Reactive behavior in self-reconfiguring sensor networks. *ACM Mobile Computing and Communications Review* 7 (1): 56–58.

Lian, F., J. Moyne, D. Tilbury, et al.. 2002. Network design consideration for distributed control systems. *IEEE Transactions on Control Systems Technology* 10 (2): 297–307.

Lim, A. 2001. Distributed services for information dissemination in self-organizing sensor networks. *Journal of the Franklin Institute* 338 (6): 707–727.

Liu, D., P. Ning, and R. Li. 2005. Establishing pairwise keys in distributed sensor networks. *ACM Transactions on Information and System Security* 8 (1): 41–77.

Luo, H., F. Ye, J. Cheng, S. Lu, and L. Zhang. 2005. TTDD: Two-tier data dissemination in large-scale wireless sensor networks. *Wireless Networks* 11 (1): 161–175.

Lynch, C. and F. O. Reilly. 2005. Pic-based TinyOS implementation. In *2nd European Workshop on Wireless Sensor Networks*, *31 January - 2 February 2005*, *Istanbul, Turkey*, 378–385.

Lynch, J. and K. Loh. 2006. A summary review of wireless sensors and sensor networks for structural health monitoring. *Shock and Vibration Digest* 38 (2): 91–130.

Marcondes, H., A. Hoeller, L. Wanner, and A. Frohlich. 2006. Operating systems portability: 8 bits and beyond. In *IEEE Conference on Emerging Technologies and Factory Automation*, *20-22 September, 2006*, *Prague, Czech Republic*, 124–130. IEEE.

Maróti, M., B. Kusy, G. Simon, and Á. Lédeczi. 2004. The flooding time synchronization protocol. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, *3-5 November, 2004*, *Baltimore, MD, USA*, 39–49. ACM.

Micrium. 2009. μC/OS-II Kernel. http://micrium.com/page/products/rtos/os-ii, (accessed 22 October 2009).

Moss, S., C. Phoumsavanh, M. Konak, K. Tsoi, N. Rajic, S. Galea, I. Powlesland, and P. McMahon. 2009. Design of the acoustic electric feedthrough demonstrator MK II. *Materials Forum* 33.

Moteiv Corporation. 2004. Telos mote. http://www.ece.osu.edu/~bibyk/ee582/telosMote.pdf, (accessed 21 February 2009).

Nachman, L., R. Kling, R. Adler, J. Huang, and V. Hummel. 2005. The Intel® mote platform: a Bluetooth-based sensor network for industrial monitoring. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks*, *25-27 April, 2005*, *Los Angeles, CA, USA*, 61. ACM.

Paek, J., K. Chintalapudi, J. Cafferey, R. Govindan, and S. Masri. 2005. A wireless sensor network for structural health monitoring: Performance and experience. In *Proceedings of the Second IEEE Workshop on Embedded Networked Sensors*, *30-31 May, 2005*, *Sydney, Australia*, 30–31. IEEE.

Pastor Llorens, E., J. López Rubio, and P. Royo Chic. 2007. UAV payload and mission control hardware/software architecture. *IEEE Aerospace and Electronic Systems Magazine* 22 (6): 3–8.

Patel, R. and D. Jain. 2009. Energy efficient data diffusion in wireless sensor networks. In *Proceedings of the International Conference on Advances in Computing, Communication and Control*, *23-24 January, 2009*, *Mumbai, India*, 643–648. ACM.

P&E Micro. 2006. Coldfire USB BDM (USB-ML-CF) and GDB? http://www.pemicro.com/forums/index.php?showtopic=933, (accessed 26 October 2009).

P&E Micro. 2009. USB-ML-12 + GNU Toolchain + Linux. http://www.pemicro.com/forums/index.php?showtopic=2810, (accessed 26 October 2009).

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

Polastre, J., R. Szewczyk, and D. Culler. 2005. Telos: Enabling ultra-low power wireless research. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks*, *25-27 April, 2005*, *Los Angeles, CA, USA*, 48. ACM.

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.

Qi, H., X. Wang, S. Iyengar, and K. Chakrabarty. 2001. Multisensor data fusion in distributed sensor networks using mobile agents. In *Proceedings of the 4th International Conference on Information Fusion*, *7-10 August, 2001*, *Montreal, Canada*, 11–16. International Society of Information Fusion.

Ricadela, A. 2005. Sensors everywhere. *InformationWeek* (1023).

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

Sabbineni, H. and K. Chakrabarty. 2005. Location-aided flooding: an energy-efficient data dissemination protocol for wireless-sensor networks. *IEEE Transactions on Computers* 54 (1): 36–46.

Sarris, Z. 2001. Survey of UAV applications in civil markets (June 2001).

Schneider, W. 2004. Unmanned aerial vehicles and uninhabited combat aerial vehicles. *Study by the Defense Science Board, Office of the Under Secretary of Defense, Washington, DC, USA*.

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

Shenker, S., S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. 2003. Data-centric storage in sensornets. *ACM SIGCOMM Computer Communication Review* 33 (1): 137–142.

Shih, E., S. Cho, N. Ickes, R. Min, A. Sinha, A. Wang, and A. Chandrakasan. 2001. Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, *16-21 July, 2001*, *Rome, Italy*, 272–287. ACM.

Sichitiu, M. and C. Veerarittiphan. 2003. Simple, accurate time synchronization for wireless sensor networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, *16-20 March, 2003*, *New Orleans, LA, USA*, 16–20. IEEE.

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.

Sivrikaya, F. and B. Yener. 2004. Time synchronization in sensor networks: A survey. *IEEE Network* 18 (4): 45–50.

Sohrabi, K., J. Gao, V. Ailawadhi, and G. J. Pottie. 2000. Protocols for self-organization of a wireless sensor network. *IEEE Personal Communications* 7 (5): 16–27.

Solis, I. and K. Obraczka. 2004. The impact of timing in data aggregation for sensor networks. In *Proceedings of the IEEE International Conference on Communications*, *20-24 June, 2004*, *Paris, France*. IEEE.

Sourceforge.NET. 2009. Index of /tinyos-2.x/tos/platforms. http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/platforms/?hideattic=0, (accessed 21 February 2009).

Szewczyk, R., P. Levis, M. Turon, L. Nachman, P. Buonadonna, and V. Handziski. 2007. TEP 112: Microcontroller power management. http://www.tinyos.net/tinyos-2.x/doc/html/tep112.html, (accessed 17 October 2009).

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 26 September 2009).

Trigoni, N., Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. 2004. WaveScheduling: Energy-efficient data dissemination for sensor networks. In *Proceedings of the 1st Workshop on Data Management for Sensor Networks*, *30 August, 2004*, *Toronto, Canada*, 48–57. ACM.

UC Berkley. n.d. TinyOS Community Forum || Mission Statement. http://tinyos.net/special/mission, (accessed 22 October 2009).

UCLA NESL. 2008. SOS 2.x Home Page. https://projects.nesl.ucla.edu/public/sos-2x/doc/, (accessed 22 October 2009).

van der Velden, S. and I. Powlesland. 2008. Interconnecting a flexible sensing array. In *Second Asia-Pacific Workshop on Structural Health Monitoring*, *2-4 December, 2008*, *Melbourne, Australia*.

van der Velden, S., I. Powlesland, and J. Singh. 2007. A proposed robust distributed control system architecture for mini/micro UAVs. In *22nd International Unmanned Air Vehicle Systems Conference*, *16-18 April, 2007*, *Bristol, UK*.

van Greunen, J. and J. Rabaey. 2003. Lightweight time synchronization for sensor networks. In *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications*, *19 September, 2003*, *San Diego, CA, USA*, 11–19. ACM.

Vieira, M., C. Coelho Jr, D. da Silva Jr, and J. da Mata. 2003. Survey on wireless sensor network devices. In *Emerging Technologies and Factory Automation*, 16–19.

Vuran, M. and I. Akyildiz. 2006. Cross-layer analysis of error control in wireless sensor networks. In *Proceedings of IEEE International Conference on Sensor and Ad-hoc Communications and Networks*, *25-28 September, 2006*, *Reston, VA, USA*, 585–594. IEEE.

Walters, J., Z. Liang, W. Shi, and V. Chaudhary. 2007. Wireless sensor network security: A survey. In *Security in Distributed, Grid, and Pervasive Computing*, ed. Y. Xiao, 367–410. Boca Raton, FL, USA: Auerback Publications.

Wang, X. and S. Wang. 2007. Collaborative signal processing for target tracking in distributed wireless sensor networks. *Journal of Parallel and Distributed Computing* 67 (5): 501–515.

Wesson, R., F. Hayes-Roth, J. Burge, C. Stasz, and C. Sunshine. 1981. Network structures for distributed situation assessment. *IEEE Transactions on Systems, Man and Cybernetics* 11 (1): 5–23.

Wiki, T. D. 2008. Platforms. http://docs.tinyos.net/index.php/Platforms, (accessed 26 October 2009).

Wiki, T. D. 2009. Installing TinyOS 2.1. http://docs.tinyos.net/index.php/Installing_TinyOS_2.1, (accessed 25 October 2009).

Woo, A., T. Tong, and D. Culler. 2003. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, *5-7 November, 2003*, *Los Angeles, CA, USA*, 14–27. ACM.

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 30 August 2009).

Ye, F., G. Zhong, S. Lu, and L. Zhang. 2005. Gradient broadcast: A robust data delivery protocol for large scale sensor networks. *Wireless Networks* 11 (3): 285–298.

Younis, O. and S. Fahmy. 2004. HEED: a hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. *IEEE Transactions on Mobile Computing* 3 (4): 366–379.

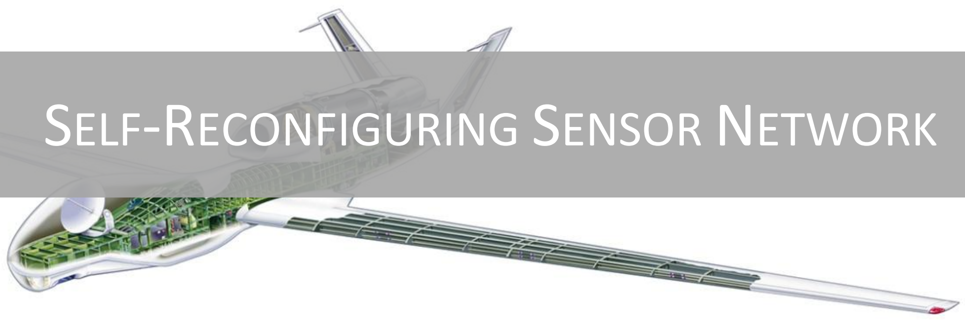
Yu, Y., R. Govindan, and D. Estrin. 2001. Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. UCLA Computer Science Department Technical Report, UCLA-CSD TR-01-0023.

Zou, L., M. Lu, and Z. Xiong. 2005. A distributed algorithm for the dead end problem of location based routing in sensor networks. *IEEE Transactions on Vehicular Technology* 54 (4): 1509–1522.

Zurich, E. 2007. Welcome to the BTnode platform. http://www.btnode.ethz.ch/, (accessed 21 February 2009).

Appendix A: System requirements specification

23/02/2009



CURTIN UNIVERSITY

SELF-RECONFIGURING SENSOR NETWORK

Project Specification | Robert Moore

Table of Contents

Table of Figures

1 Introduction

1.1 Purpose

The purpose of this document is to propose a set of requirements for the development of a self-reconfiguring sensor network control system. It will explain the purpose and desired features of the system, as well as a structured set of phases that the system will be implemented in. This document is intended for both the stakeholders and developers of the system.

1.2 Scope

The system in question is attempting to implement the system described in (van der Velden & Powlesland 2008). More specifically, the system will be designed to integrate with particular MEMs sensors currently being developed by the authors of that article.

The system will consist of a number of "nodes". Each node will comprise a processing element connected to a number of multi-sensing MEMs sensors. Each node will also be connected to a number of other nodes, forming an overall distributed sensing network.

The goal of the system is for it to be robust to damage, sensitive of, and adaptable to limited power levels and effective at distributing accurate, fresh sensor information for all the different possible information types sensed by the system according to a number of utility functions.

It is proposed that the system be implemented in a number of stages, starting at the most basic possible functioning system and then adding on more complicated parts in an iterative manner. In saying that, the initial phases will need to be implemented with the final goals of the system in mind. In order to outline the desired overall function of the system, the "ideal", final system is described thus:

The system being described will comprise the following modes:

- Initialisation Mode: All nodes will perform time synchronisation and probe their network connections to see which neighbours are alive and perform any network topology computations / communications required.
- Normal Mode: All nodes will be in normal operation and sensing / sending / receiving data.

Normal Mode would ideally consist of the following operations:

- Data Fusion: A data fusion algorithm will be used to combine the information from the different local sensors into a single, accurate value for each information type. A believability descriptor will also result from this algorithm. The algorithm will include some sort of fault tolerance / detection algorithm to detect and disregard faulty sensors.
- Network Protocol: Each node will have a data-link layer network protocol that will integrate with the proposed physical layer. This data-link layer will allow nodes to both send and receive sensor data and other communication packets across the network.
- Data Aggregation: A data aggregation algorithm will be used to combine the local sensor data with the different global sensor data received from local nodes within the network. This algorithm will make use of the believability descriptors as well as some sort of fault

detection scheme to ensure data from a faulty node is identified and ignored. The algorithm will include logic to determine when to send out / who to send aggregated values to neighbouring nodes.

- Network Layer: The system will have some sort of network layer protocol to ensure the nodes organise themselves in an efficient and redundant manner to ensure that all nodes receive all appropriate data, yet network traffic is minimised and the system won't fail if nodes are damaged (possibly through reconfiguration).
- Power Management: Various power management schemes will need to be incorporated into the system to ensure that power is conserved where possible and the operation of each node is sensitive to how much power that node has available to it at any time.
- Data Access API: An API will exist so that higher level layers on each node (e.g. actuator code, control code etc.) can access the current sensor information.
- Nodes should have the ability to make requests for data they need, and for other nodes to provide it.

Portions of the system should possibly be implemented in VHDL to improve speed, power use and portability.

This document outlines two main phases and a number of extensions as a structure of implementation for the system. It is proposed that the two main phases will be designed and implemented as part of the initial project with the possibility of implementing one or more of the extensions if time permits.

1.3 Glossary

Term	Definition
AI	Artificial intelligence (AI) is both the intelligence of machines and the branch of computer science which aims to create it.
Bayesian inference	Statistical use of observations to infer probability about a hypothesis
Believability	Credibility or trustworthiness
Distributed control	A system whereby control processing is decentralised and independent of a central computer
FPGA	Field Programmable Gate Arrays (FPGA) are semiconductor devices that contain programmable logic components
Fuzzy logic	Multi-valued logic using approximate reasoning
HDL	Hardware Description Languages (HDL) allow formal description of electronic circuits
Neural networks	Pattern-recognition processes that search for a solution using a network of "neuron-like" structures
Self-reconfiguring	The ability of a system to change its configuration based on its environment
Structural Health Monitoring (SHM)	Observation of structures over time for health and damage information
VHDL	VHSIC (Very-High-Speed Integrated Circuits) HDL (VHDL) is a hardware description language used to design field FPGAs and application specific integrated circuits (ASICs)

1.4 References

Crossbow (n.d.)-a, *IRIS Datasheet*, Retrieved: 21 February 2009, from http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/IRIS_Datasheet.pdf.

Crossbow (n.d.)-b, *MICA2 Datasheet*, Retrieved: 21 February 2009, from http://www.xbow.com/products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf.

Crossbow (n.d.)-c, *MICAZ Datasheet*, Retrieved: 21 February 2009, from http://www.xbow.com/products/Product_pdf_files/Wireless_pdf/MICAZ_Datasheet.pdf.

Culler, D. E. 2006, 'TinyOS: Operating System Design for Wireless Sensor Networks', *Sensors,* vol. 23, no. 5, p. 14. http://www.sensorsmag.com/sensors/article/articleDetail.jsp?id=324975.

Klein, L. A. 1999, *Sensor and Data Fusion Concepts and Applications*, Society of Photo-Optical Instrumentation Engineers (SPIE) Bellingham, WA, USA.

Korber, H. J., Wattar, H., Scholl, G. & Heller, W. 2005, 'Embedding a Microchip PIC18F452 based commercial platform into TinyOS', in *Workshop on Real-World Wireless Sensor Networks (REALWSN'05)*, Stockholm, Sweden.

Lentsch, T. 2005, *The eyesIFX platform*, Retrieved: 21 February 2009, from http://www.tinyos.net/ttx-02-2005/platforms/ttx2005-eyesIFX.ppt.

Levis, P. 2003, *Simulating TinyOS Networks*, Retrieved: 21 February 2009, from http://www.cs.berkeley.edu/~pal/research/tossim.html.

Levis, P. 2006, *TinyOS Programming*, Retrieved: 21 February 2009, from http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf.

Lynch, C. & Reilly, F. O. 2005, 'Pic-based TinyOS implementation', in *2nd European Workshop on Wireless Sensor Networks, EWSN*, Istanbul, Turkey, pp. 378–385.

Miyashita, M. 2002, *TinyOS Overview*, Retrieved: 21 February 2009, from http://deneb.cs.kent.edu/~mikhail/classes/es.u02/Research/Tiny_OS_Intro.ppt.

Moteiv Corporation 2004, *Telos Mote*, Retrieved: 21 February 2009, from http://www.ece.osu.edu/~bibyk/ee582/telosMote.pdf.

Sourceforge.NET 2009, *Index of /tinyos-2.x/tos/platforms*, Retrieved: 21 February 2009, from http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/platforms/?hideattic=0.

tinyos-help (n.d.), Retrieved: 21 February 2009, from http://www.mail-archive.com/tinyos-help@millennium.berkeley.edu/info.html.

TinyOS Documentation Wiki 2009a, *Getting Started*, Retrieved: 21 February 2009, from http://docs.tinyos.net/index.php/Getting_started.

TinyOS Documentation Wiki 2009b, *TinyOS 2.x index of contributed code*, Retrieved: 21 February 2009, from http://docs.tinyos.net/index.php/Contributing_Code_to_TinyOS.

TinyOS Documentation Wiki 2009c, *TinyOS Tutorials*, Retrieved: 21 February 2009, from http://docs.tinyos.net/index.php/TinyOS_Tutorials.

UC Berkeley 2004, *TinyOS*, Retrieved: 21 February 2009, from http://www.tinyos.net/.

van der Velden, S. & Powlesland, I. 2008, 'Interconnecting a Flexible Sensing Array', *Second Asia-Pacific Workshop on Structural Health Monitoring*.

1.5 Overview

The next section Phase 1: Basic System (section 2) gives an overview of the most basic system that could be implemented (the proposed first phase). Section 3 (Phase 2: Multi-level System) outlines a second phase system that is more sophisticated and incorporates more of the desired elements of the system. A range of extensions to the phase 2 system that will bring to system closer to the "ideal" system are discussed in section 4 (Extensions). Finally, a number of open questions not answered by this specification are covered in section 5 (Open Questions).

2 Phase 1: Basic System

2.1 Purpose

The purpose of the basic system is to provide a platform that fulfils the most basic of needs from the system and will be easily and quickly implemented.

The system will need to be implemented with the goals of future phases in mind in order to ensure that it is flexible enough to be extended.

2.2 Block Diagram

Figure 2.1 depicts the external environment of the control system.

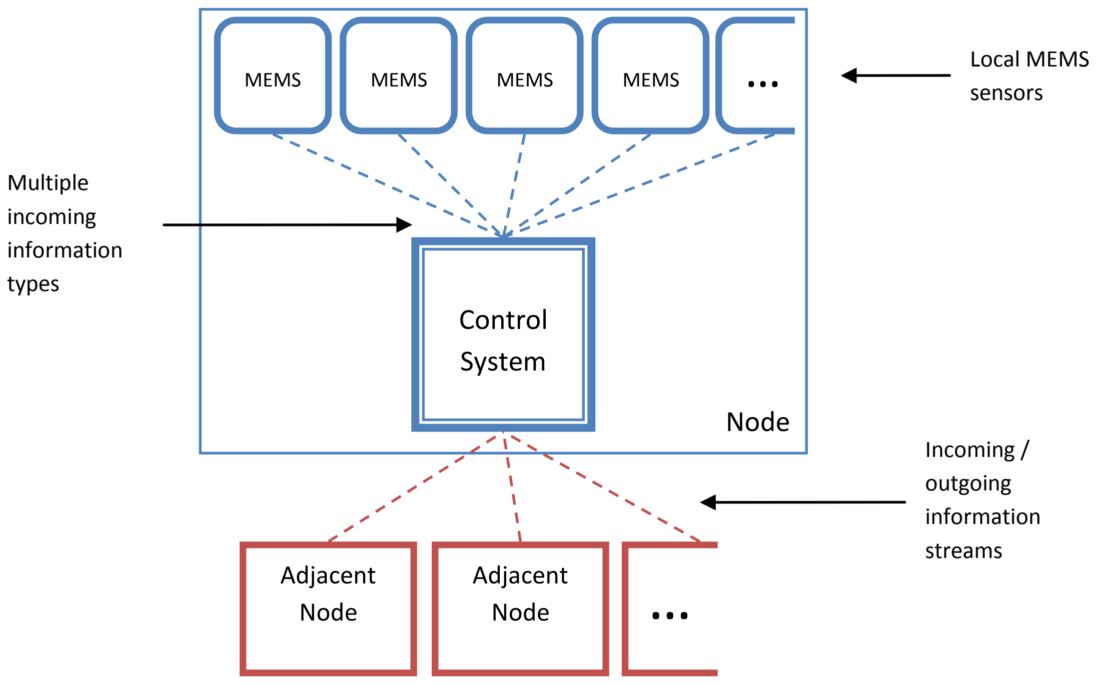


Figure 2.1 - Control System External Environment

2.3 Description

2.3.1 System Overview

The Basic System will comprise of a network of distributed nodes. Each node will be connected to a number of local MEMs sensors. Each node will be in one of two modes:

- Initialisation Mode: All nodes will perform time synchronisation and probe their network connections to see which neighbours are alive (apart from time synchronisation, the end result of this process is that each node will know how many neighbours they have, on which ports and the delay in each connection).
- Normal Mode: All nodes will be in normal operation and sensing / sending / receiving data.

Normal Mode will consist of the following operations:

- Data Fusion: All the local sensor information will be fused into a distinct average value for each of the different types of information being sensed. Erroneous values will be detected and omitted from the average calculation. This will happen at a regular time period appropriate to each information type; this time period will be a multiple of the basic time period *p*.
- Data Sending: Each node will send all of its fused values through the network in a broadcast manner. Each transmission will consist of the following values:

○ Average Value	
○ Minimum Value	} *Primitive believability*
○ Maximum Value	} *descriptors*
○ Number of sensors used to generate the average	} *Accuracy descriptor*
○ Information Type	
○ Timestamp	} *Freshness descriptor*
○ Source address	} *Optional?*

- Data Receiving: Each node will listen on its connections to neighbouring nodes for information and when it receives data, if it is more recent than the value it currently has for that information type then it will update the value it has stored for that information type. When it updates its local value, if the change in value is significant it will also forward the packet to its neighbours, this should prevent network overload. This technique means that a data aggregation algorithm is not required and is a simple way of ensuring that a given packet doesn't circulate a network indefinitely when there are loops in the network topology. It does mean however, that the newest information value is simply taken, rather than being combined with other sources to improve overall accuracy.
- Data Access API: An API will exist so that higher level layers on each node (e.g. actuator code, control code etc.) can access the current sensor information. The API will allow a particular type of information to be requested, and the returned value (if that data is available) will consist of the average value, the minimum and maximum value and how fresh the data is (current time – timestamp of data).

2.3.2 Time Synchronisation

The time synchronisation algorithm should be accurate to within *z*ms and should take a minimum amount of time to complete to minimise power use. If possible the algorithm should ensure that time synchronisation remains within *a*ms at all times, and perform corrective action if this constraint is broken.

2.3.3 Packet Structure

The packet structure will need to support the following:

- Multiple packet types; two main types (data and system), with system consisting of at least 3 subtypes; time sync, probe, probe acknowledgement, with the flexibility to extend this for future systems
- The data packet will need the largest amount of storage so this specification will focus on this packet type in order to determine the maximum packet size, and assume that this leaves enough room to store the data for the different system packet types (the exact structure of each will be left as a design-time decision). The data packet:

- Will need the ability to store the average, minimum and maximum values for the sensor data as well as the maximum possible number of sensors connected to a single node, as well as the timestamp and type of data
- Should fit the message size of the chosen network protocol
- Should possibly also include an identification number for the source of the packet (i.e. will need to fit the maximum number of nodes)
- Should be flexible enough to scale as the system is further developed

In order to quantify the size of the individual parts of the packet, the following metrics need to be determined:

1. Number of packet types
2. Resolution of data
3. Maximum possible number of local sensors
4. Resolution of timestamps
5. Maximum number of information types
6. Maximum number of nodes in the network

2.3.3.1 Number of packet types

To support the basic system, 1 bit is needed to differentiate between a data packet and system packet and another 2 bits would be needed for sub-identification of the system packet. In order to preserve flexibility for extension of the system, **4** bits of information will be allocated to the packet type number for system packets, allowing a maximum of 16 system packet types.

2.3.3.2 Resolution of data

According to Appendix A, the preliminary information suggests that up to 8 bit resolution will be needed for almost all of the information types. Thus a logical choice would be to choose 8 bits to represent the average / max / min value of the data, leaving **24** bits in total.

2.3.3.3 Number of sensors

The number of sensors each node can be connected to will be determined by the MEMs sensor hardware currently being developed (i.e. how many sensors can the hardware support?). This number is currently unknown. Furthermore, the number of sensors that the packet will need to support will probably increase for the phase 2 system as it will include data aggregation, resulting in the combination of local sensors from more than one node. Thus the maximum number of sensors that needs to be supported is more like:

*max_number_of_local_sensors * max_number_of_nodes*

The number of bits required for this will be prohibitively large and the usefulness of counting every single sensor will have less meaning the more sensors there are. Thus it is proposed that a scale is used instead. There are two options:

- One bit is used to indicate whether the accuracy is based on the number of sensors directly or on aggregations of sensors with a minimum count of *n*. The value of *n* will be a system constant, which can be adjusted such that the number of sensors in the system divided by *n* will fit into the number of bits set aside for this value.

- An abstract accuracy scale could be used that spans the range of numbers afforded by the resolution of this value. The value on the scale would be calculated by some formula or function in the system possibly based on more than just the number of sensors. The phase 1 system will not implement this scale.

A reasonable number of bits to allocate is **4** bits, leaving 16 levels of accuracy rating.

2.3.3.4 Resolution of timestamps

If an absolute timestamp is used then the full timestamp resolution of the system will be needed which will take up a lot of room in the packet. An alternative solution is to store a relative timestamp (e.g. how far back from current time was the measurement taken) and assuming that each node knows the delay in the network link to its neighbours (it should know this from initialisation mode), it can extrapolate the absolute timestamp of the data. Furthermore, it is conceivable that relative timestamp is more useful than absolute timestamp as an indicator of freshness. If a relative timestamp is used, the maximum value of the timestamp will be determined by the maximum delay network-wide.

Another possible method of restricting the resolution of the transmitted timestamp is to have an abstract freshness scale of the same sort as the abstract accuracy scale described above in section 2.3.3.3. The phase 1 system will not have such a scale; it will transmit a relative timestamp.

Thus the value for the timestamp resolution **can't be determined at this stage** since it is dependent on the maximum delay in the system. See section 2.3.3.6 for more information about this.

2.3.3.5 Number of information types

According to Appendix A, preliminary information suggests that there will be at least 13 information types need to be supported. Thus to allow some flexibility, the system will support 16 information types, requiring **4** bits.

2.3.3.6 Number of nodes

In order to maximise the number of nodes restricted by the system, the maximum number of nodes in the network should be determined by three factors:

a) The number of bits left in the maximum message size allowed by the chosen network protocol after the required size for the other parts of the packet are taken into consideration
b) The maximum desired delay in the network (the number of maximum nodes should be determined via quantitative testing of the delay performance of the network)
c) The fact that timestamp resolution is dependent on maximum delay (at least in the phase 1 system)

Thus the value for the timestamp resolution **can't be determined at this stage** as it depends on design decisions.

2.3.3.7 Conclusion

Thus the data packet structure will need to support **33 bits** (4 bytes + 1 bit) as well as the size of the relative timestamp and source address (to be determined at design time).

3 Phase 2: Multi-level System

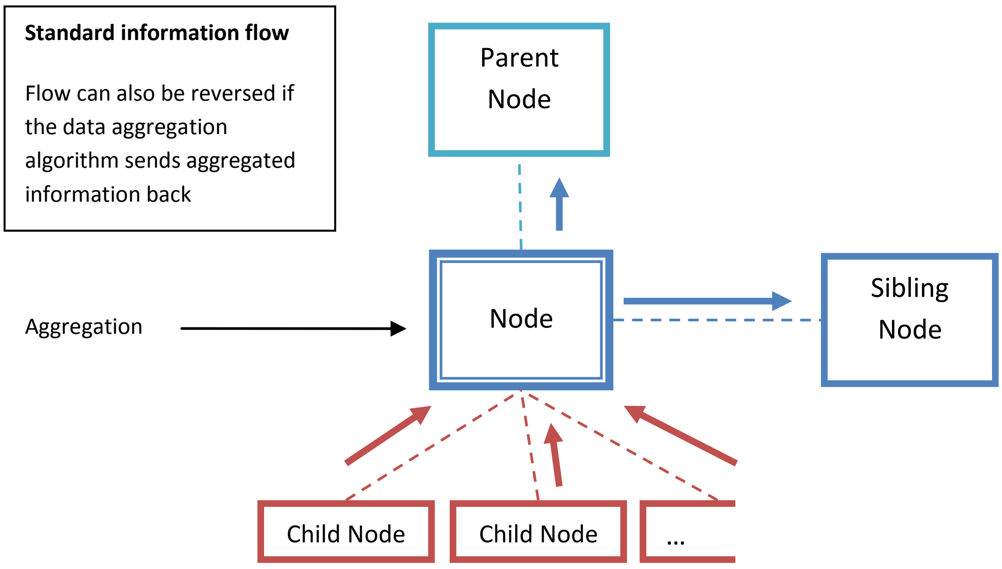
3.1 Purpose

The phase 2 system aims to increase the scalability of the control system, reduce the amount of network traffic (and thus power use) and improve the accuracy of the system (by aggregating data from multiple nodes rather than blindly accepting data as soon as it is received). This will be achieved by incorporating another three elements to the phase 1 system:

- A network layer protocol so the system organises itself into some sort of hierarchy
- A data aggregation algorithm to aggregate data from multiple nodes and improve accuracy
- A data dissemination strategy to intelligently send aggregated data through the network in a way that minimises network traffic

3.2 Block Diagram

Figure 3.1 depicts the standard information flow of the phase 2 system.



Standard information flow
Flow can also be reversed if the data aggregation algorithm sends aggregated information back
Aggregation
Parent Node
Node
Sibling Node
Child Node
Child Node
...

Figure 3.1 - Standard information flow for the phase 2 system

3.3 Description

3.3.1 Network Layer

Organising the network into a hierarchical or clustered structure has a number of advantages:

- It allows the deployment of data aggregation algorithms to improve the accuracy and intelligence of the system
- In combination with appropriate data dissemination strategies, it can increase the efficiency of network traffic

- Routing can be more intelligent: for instance, parent nodes can forward an information request only to the child(ren) that has/have the type of information requested (not relevant to the phase 2 system, relevant to section 4.3)

The network protocol for this system will need to support the following properties:

- High robustness to damage (of links and nodes)
- Low overhead / power use

3.3.2 Data Aggregation

One of the main disadvantages of the phase 1 system is the fact that data is "blindly" accepted and adopted as the current value for that information type. This means that only the redundant nature of multiple nodes is used, rather than the increase in accuracy and reliability of information. Incorporating a data aggregation algorithm allows the system to leverage accuracy increases by combining information streams from multiple nodes.

The following factors will need to be taken into consideration when choosing / designing an aggregation algorithm:

- The data must be kept fresh, if the data changes suddenly then that might be the new value, or it could be that the node is damaged and sending incorrect data. Some sort of combination of averaging / voting would need to be used to improve the data accuracy / identify errant / changing data.
- It should be sensitive of the power level remaining and possibly able to be configured to have different accuracy vs power levels (not relevant for the phase 2 system, more relevant to section 4.5)

3.3.3 Data Dissemination

The effectiveness of the data aggregation depends significantly on how that information is disseminated throughout the network. Factors that should be considered are:

- Do any of the child nodes have data that is incorrect? If so they should possibly be informed
- Has the aggregated value changed significantly? Is there a need to retransmit the value to the parent node(s)
- Should the information be transmitted to sibling nodes – do they need it / can they confirm its accuracy based on aggregation from its children?
- Power should be conserved where possible by minimising network traffic, in particular if power availability is low
- Which nodes need the value being aggregated (not relevant to the phase 2 system, relevant to section 4.3)

4 Extensions

4.1 Overview

This section outlines the other characteristics of the system that have been identified as desirable. These characteristics may or may not be implemented as part of this project depending on the availability of time.

Each different extension is outlined in (high-to-low) priority order, along with its subsection number in the list below:

1. Re-configurability / Utility Functions (4.2)
2. Speculative Requests / Advertisements (4.3)
3. Fault Tolerance / Detection (4.4)
4. Power Management (4.5)
5. Sophisticated Data Fusion / Aggregation (4.6)
6. Hardware Implementation (4.7)

4.2 Re-configurability / Utility Functions

The robustness, accuracy and usefulness of the system will be greatly increased if it is possible to specify utility functions against each information type that ensures the data in the system is accurate. It is envisaged this would take the form of statements such as "acceleration needs at least 16 sensors from 3 nodes to get an accurate reading". If such utility functions are specified then the system can reconfigure itself in a manner that maximised these functions. E.g. if there are not enough nodes sensing acceleration then one or more of them can re-task their sensors from sensing a parameter which is more than meeting its utility function, to sense acceleration. (van der Velden & Powlesland 2008)

4.3 Speculative Requests / Advertisements

The goal of minimising network traffic (and thus power use) and also improving the robustness of the system would be improved if speculative requests and advertisements are introduced.

Speculative requests allow nodes to request information they need, for instance if the higher level layer requests this information and it doesn't have a fresh value for it, or if a node detects that the utility function for a given information type isn't met and it wants more sources. This would both allow network traffic to be more intelligently routed and improve robustness. (van der Velden & Powlesland 2008)

Similarly, speculative advertisements allow nodes to advertise when they find or receive a given information type. This could also be used to inform the network when a node reconfigures its nodes to sense different information types. This will help improve the robustness of the system. (van der Velden & Powlesland 2008)

4.4 Fault Tolerance / Detection

Fault tolerance and detection has been touched on in a number of sections of this specification, namely sections 3.3.2, 3.3.3, 4.2 and 4.3. In order to improve the intelligence of the system, fault tolerance and detection algorithms can be introduced, and integrated with the other algorithms present e.g. data dissemination, speculative advertisements etc.

There are three main areas where faults can occur, namely:

- Physical layer – link(s) go down
- Power loss / damage to node(s) – node(s) go down
- Damage to sensor(s) – sensor(s) go down

These faults can result in either a lack of information flow, or erroneous information flow. Algorithms to combat and respond to both of these situations for all three levels should be introduced to the system.

4.5 Power Management

One of the largest concerns of the sensor network is power use and conservation. A common requirement among the components of the system is that they have minimalist power and memory use. The system in question will have access to power information (e.g. how much power is left) and power control (e.g. different power states at a hardware level) (van der Velden, S. 2009, pers. comm., January 30).

Thus, whilst the individual components that make up the system will be power conscious, it is still necessary to provide some sort of higher-level power management algorithm to switch the node into different power modes based on power available, and possibly integrating this with the different components.

4.6 Sophisticated Data Fusion / Aggregation

The data fusion and aggregation algorithms outlined for the phase 1 and phase 2 systems are deliberately primitive to simplify initial implementation. The intelligence of the system, and possibly the accuracy / fault detection of the system could be improved by introducing more sophisticated fusion / aggregation algorithms. There are a range of relevant AI topics such as believability, voting, Bayesian inference, fuzzy logic and neural networks that may be worth investigating (Klein 1999).

4.7 Hardware Implementation

When speed, memory and power use become constraining factors, it sometimes becomes useful to implement all or part of the system in digital logic. The easiest way to do this is using a HDL, such as VHDL and then deploying to an FPGA or similar.

Once the software has been finalised, it should be analysed to see if any benefit can be gained from implementing any portion of it in hardware.

5 Open Questions

5.1 Base data fusion time period

In order to ensure the regularity of data fusion, a basic time period *p* will need to be defined (see section 2.3.1) and all data fusion will occur at regular time periods that are integer multiples of the base time period.

5.2 Source address

For the phase 1 system, should the source address be included in the packet structure (see section 2.3.3)?

5.3 Time synchronisation constants

The constants *a* and *z* as mentioned in section 2.3.2 need to be determined.

5.4 Maximum number of local sensors

In order to provide a better idea of how well the algorithm described in section 2.3.3 on representing the accuracy of data will work, it would be useful to know the maximum possible number of local sensors that a node can have.

5.5 Maximum delay / nodes

As discussed in sections 2.3.3.6 and 2.3.3.4, the maximum delay of the system, the maximum number of nodes and the resolution of the timestamp portion of the packet structure need to be decided.

5.5.1 Network Protocol

To support the goals of the system, the network protocol used will need the following properties:

- Some sort of MAC protocol to overcome interference issues (one possible physical layer the system will work with is via acoustic waves and can have interference (van der Velden, S. 2009, pers. comm., 30 January))
- Minimalist
- Low power
- Supports the maximum number of nodes the system will have (possibly not relevant since point-to-point links will be used)
- Supports the control system packet structure size
- Code / hardware implementations are available
- Compatible with a distributed network

In order to ensure flexibility, the implementation of the network protocol will need to be abstracted from the rest of the system via a well defined interface. This allows the system to be network protocol agnostic, and the network protocol can be changed if the requirements of the system change in a way that invalidates the use a selected protocol (e.g. the packet structure size might increase when the system is extended past phase 1 / 2).

5.6 Operating System

A decision needs to be made about whether the control system should be implemented around a particular operating system or if it can be implemented independent of an operating system. If it is implemented using an operating system then the operating system will need to have the following properties:

- Minimalist
- Low overhead
- Portable to a variety of computing elements
- Good documentation

If an operating system is chosen then this will also likely tie the higher level layers (actuator code, control code etc.) to using that operating system as well.

A possible operating system that could be used, that has been designed specifically for sensor networks is TinyOS, which was developed at Berkeley University (UC Berkeley 2004). This operating system has a number of advantages, including:

- It has good documentation, including a wiki containing getting started guides and tutorials, and a detailed programming manual (Levis 2006; TinyOS Documentation Wiki 2009c; TinyOS Documentation Wiki 2009a)
- There is an established community using it, including thousands of developers. There are a range of user contributed plugins for it and an email discussion list (*tinyos-help* (n.d.); Culler 2006; TinyOS Documentation Wiki 2009b)
- It is lightweight, it only has a single stack and is designed for low memory environments (Culler 2006)
- There are a range of development tools, including IDE plugins and a VMware image with the development environment already set up (TinyOS Documentation Wiki 2009a; TinyOS Documentation Wiki 2009b)
- There is a graphical simulation tool that can simulate an entire sensor network (Levis 2003; TinyOS Documentation Wiki 2009b)
- There is a unit testing framework available for it (TinyOS Documentation Wiki 2009b)
- It is portable, with implementations for the Intel Mote (Intel XScale PXA), Mica2/MicaZ/Iris/BTnode Motes (Atmel ATmega128) and Telos/eyesIFX Motes (Ti MSP430) (Crossbow (n.d.)-c; Crossbow (n.d.)-a; Crossbow (n.d.)-b; Lentsch 2005; Moteiv Corporation 2004; Sourceforge.NET 2009). Some work has been put into porting TinyOS to Microchip PIC microcontrollers as well (Korber et al. 2005; Lynch & Reilly 2005). The nesC language that TinyOS is implemented in is portable, it simply needs a GNU toolchain for the processor to be able to be compiled to it.
- It has power management built-in (Culler 2006)
- It uses a component based architecture which lends nicely to the abstraction requirement of the network protocol and interface to higher layers (as mentioned in sections 2.3.1 and 5.5.1 respectively)

There are however a number of disadvantages in using this operating system apart from the aforementioned problem of tying the system to a particular implementation:

- There will almost certainly need to be some sort of port to make the TinyOS compatible with the hardware this system will run on (however, this code will need to be written anyway if an OS isn't used)
- There is a learning curve for nesC and TinyOS (it is based on C and there is good documentation though, so the impact of this is lessened).
- The operating system is very focussed on wireless communication, which is not strictly the environment that this system will be running on. This shouldn't be too big a problem though, the wireless components can simply be ignored. This does make it unclear how much of the built-in networking code can be used with the hardware for this system however.
- According to (Culler 2006) the nested event driven execution means that small changes may require changes throughout the codebase, making reuse and incremental development difficult.
- No real-time guarantees (Miyashita 2002)
- Only FIFO non-preemptive scheduling (suited to I/O bound tasks rather than CPU bound tasks) (Miyashita 2002)

Appendices

Appendix A Preliminary Information Type Information

The following table summarises the preliminary information about the information types that can be sensed by the MEMs sensors currently being developed that this system is being designed to integrate with (van der Velden, S. 2009, pers. comm., 2 February). It should be noted that these are initial estimates, and as such are subject to change:

Type	Unit	Range	Accuracy	Points	Resolution
Latitudinal Acceleration	m/s^2	-50 – 50	0.5	200	7.6
Longitudinal Acceleration	m/s^2	-50 – 50	0.5	200	7.6
Normal Acceleration	m/s^2	-50 – 100	0.75	200	7.6
Roll Rate	$^{o}/s$	-100 – 100	1	200	7.6
Pitch Rate	$^{o}/s$	-50 – 50	0.5	200	7.6
Yaw Rate	$^{o}/s$	-50 – 50	0.5	200	7.6
Static Pressure	kPa	70 – 105	0.1	350	8.5
Dynamic Pressure	kPa	0 – 5	0.025	200	7.6
Temperature	^{o}C	-60 – 60	0.5	240	7.9
Latitudinal Magnetic Field	Micro Tesla	-200 – 200	4	100	6.6
Longitudinal Magnetic Field	Micro Tesla	-200 – 200	4	100	6.6
Normal Magnetic Field	Micro Tesla	-200 – 200	4	100	6.6
Audio	dB	60 – 120	3	20	4.3

Appendix B: Reading list

The literature outlined in this reading list serves as a base for anyone continuing the work in developing a control system for the sensor network outlined in section 2.0 of this thesis. These references can be used as starting points for initial research into the system as a whole, as well as for research into each individual aspect of the system that is considered. They represent an exposure to the potential algorithms and ideas that can be utilised to realise the final system, and thus serve as a trigger for further research.

The reading list is presented in the following pages in table format using the Chicago Manual of Style (2003) referencing style. Each reference is given a relevance weighting and is categorised according to the aspect of the system for which they are relevant. The categorisation and relevance were created by skim reading the material and as such are subjective. The Sensor Networks category relates to general information, or information that comes under multiple categories. The meaning of the relevance weightings is as follows:

Relevance Weighting	**Description**
Low	Probably not useful
Medium	Might be useful when designing the aspect of the system indicated by the category
Medium-High	Likely to be useful with initial research into the aspect of the system indicated by the category
High	Very likely to be useful during initial research of the system as a whole

It should be noted that some of the references contained in the References section of the thesis are also contained within this reading list so they could be included in the categorisation and relevance weighting. This thesis and the documents contained within the appendices are not included in the reading list, but are also part of the recommended reading.

Reference	**Category**	**Relevance**
Prat, N., and S. Madnick. 2008. Measuring data believability: a provenance approach. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, *Waikoloa, HI, USA*, *7-10 January, 2008* , 393. IEEE.	Believability	Low
Qi, H., X. Wang, S. S. Iyengar, and K. Chakrabarty. 2001. Multisensor data fusion in distributed sensor networks using mobile agents. In *Proceedings of the 4th International Conference on Information Fusion*, *Montreal, Canada*, *7-10 August, 2001*, 11-16. International Society of Information Fusion.	Data Aggregation	Low
Dasgupta, K., K. Kalpakis, and P. Namjoshi. 2003. An efficient clustering-based heuristic for data gathering and aggregation in sensor networks. In *Proceedings of the IEEE Wireless Communication and Networking Conference*, *New Orleans, LA, USA*, *16-20 March 2003*, 16–20. IEEE.	Data Aggregation	Medium
Kalpakis, K., K. Dasgupta, and P. Namjoshi. 2003. Efficient algorithms for maximum lifetime data gathering and aggregation in wireless sensor networks. *Computer Networks* 42 (6): 697–716.	Data Aggregation	Medium
Madden, S., M. J. Franklin, J. M. Hellerstein, and W. Hong. 2002. TAG: a Tiny AGgregation service for ad-hoc sensor networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, *Boston, MA, USA*, *9-11 December, 2002*. USENIX Association.	Data Aggregation	Medium
Boulis, A., S. Ganeriwal, and M. B. Srivastava. 2003. Aggregation in sensor networks: an energy-accuracy trade-off. *Ad hoc networks* 1 (2-3): 317-331.	Data Aggregation	Medium-High
Guestrin, C., P. Bodik, R. Thibaux, M. Paskin, and S. Madden. 2004. Distributed regression: an efficient framework for modeling sensor network data. In *Proceedings of the Third International Symposium on Information Processing in Sensor Networks*, *Berkeley, CA, USA*, *26-27 April, 2004*, 1-10. ACM.	Data Aggregation	Medium-High
Solis, I., and K. Obraczka. 2004. The impact of timing in data aggregation for sensor networks. In *Proceedings of the IEEE International Conference on Communications*, *Paris, France*, *20-24 June, 2004*. IEEE.	Data Aggregation	Medium-High
Luo, H., F. Ye, J. Cheng, S. Lu, and L. Zhang. 2005. TTDD: Two-tier data dissemination in large-scale wireless sensor networks. *Wireless Networks* 11 (1): 161-175.	Data Dissemination	Low
Sabbineni, H., and K. Chakrabarty. 2005. Location-aided flooding: an energy-efficient data dissemination protocol for wireless-sensor networks. *IEEE Transactions on Computers* 54 (1): 36-46.	Data Dissemination	Low

continued on next page...

...continued from previous page

Reference	**Category**	**Relevance**
Ye, F., G. Zhong, S. Lu, and L. Zhang. 2005. Gradient broadcast: A robust data delivery protocol for large scale sensor networks. *Wireless Networks* 11 (3): 285-298.	Data Dissemination	Low
Hui, J. W., and D. Culler. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, *Baltimore, MD, USA*, *3-5 November, 2004*, 81-94. ACM.	Data Dissemination	Medium
Kim, S., S. Son, J. Stankovic, S. Li, and Y. Choi. 2003. SAFE: A data dissemination protocol for periodic updates in sensor networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*, *Providence, RI, USA*, *19-22, May 2003*, 228-234.	Data Dissemination	Medium
Trigoni, N., Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. 2004. WaveScheduling: Energy-efficient data dissemination for sensor networks. In *Proceedings of the 1st Workshop on Data Management for Sensor Networks*, *Toronto, Canada*, *30 August, 2004*, 48-57. ACM.	Data Dissemination	Medium
Coffin, D. A., D. J. Van Hook, S. M. McGarry, and S. R. Kolek. 2000. Declarative ad-hoc sensor networking. In *Proceedings of the Conference on Integrated Command Environments*, *San Diego, CA, USA*, *31 July, 2000*, 109-120. SPIE.	Data Dissemination	Medium-High
Shenker, S., S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. 2003. Data-centric storage in sensornets. *ACM SIGCOMM Computer Communication Review* 33 (1): 137-142.	Data Dissemination	Medium-High
Yu, Y., R. Govindan, and D. Estrin. 2001. Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. UCLA Computer Science Department Technical Report, UCLA-CSD TR-01-0023.	Data Dissemination	Medium-High
Parker, J. R. 1999. Multiple sensors, voting methods and target value analysis. In *Signal processing, sensor fusion, and target recognition VIII*, *Orlando, FL, USA*, *5-7 April, 1999*, 330–335. Society of Photo-Optical Instrumentation Engineers (SPIE).	Data Fusion	Low
Klein, L. A. 1999. *Sensor and Data Fusion Concepts and Applications*, Series *Sensor and Data Fusion Concepts and Applications*. Bellingham, WA, USA: Society of Photo-Optical Instrumentation Engineers (SPIE).	Data Fusion	Medium
Hall, D. L., and J. Llinas. 1997. An introduction to multisensor data fusion. *Proceedings of the IEEE* 85 (1): 6–23.	Data Fusion	Medium-High
Mitchell, H. B. 2007. *Multi-sensor data fusion*, Series *Multi-sensor data fusion*. Belin, Germany: Springer-Verlag.	Data Fusion	Medium-High
Krishnamachari, B., D. Estrin, and S. Wicker. 2002. The impact of data aggregation in wireless sensor networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, *Vienna, Austria*, *2-5 July, 2002*, 575-578. IEEE.	Data-centric Networking	Medium-High

continued on next page...

...continued from previous page

Reference	Category	Relevance
Koh, B. H., Z. Li, P. Dharap, S. Nagarajaiah, and M. Q. Phan. 2005. Actuator failure detection through interaction matrix formulation. *Journal of Guidance, Control, and Dynamics* 28 (5): 895–901.	Fault Tolerance	Low
Benítez-Pérez, H., and F. García-Nocetti. 2005. *Reconfigurable Distributed Control*, Series *Reconfigurable Distributed Control*. London, UK: Springer-Verlag.	Fault Tolerance	Medium
Krishnamachari, B., and S. Iyengar. 2004. Distributed bayesian algorithms for fault-tolerant event region detection in wireless sensor networks. *IEEE Transactions on Computers* 53 (3): 241-250.	Fault Tolerance	Medium
Hoblos, G., M. Staroswiecki, and A. Aitouche. 2000. Optimal design of fault tolerant sensor networks. In *Proceedings of the IEEE International Conference on Control Applications*, *Anchorage, AK, USA*, *25-27 September, 2000*, 467–472. IEEE.	Fault Tolerance	Medium-High
Braginsky, D., and D. Estrin. 2002. Rumor routing algorthim for sensor networks. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, *Atlanta, GA, USA*, *28 September, 2002*, 22-31. ACM.	Network Routing	Medium
Chu, M., H. Haussecker, and F. Zhao. 2002. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. *International Journal of High Performance Computing Applications* 16 (3): 293.	Network Routing	Medium
Heinzelman, W. R., A. Chandrakasan, and H. Balakrishnan. 2000. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, *Wailea Maui, HI, USA*, *4-7 January, 2000*, 3005-3015. IEEE.	Network Routing	Medium
Zou, L., M. Lu, and Z. Xiong. 2005. A Distributed Algorithm for the Dead End Problem of Location Based Routing in Sensor Networks. *IEEE Transactions on Vehicular Technology* 54 (4): 1509-1522.	Network Routing	Medium
Akkaya, K., and M. Younis. 2005. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks* 3 (3): 325-349.	Network Routing	Medium-High
Heinzelman, W. R., J. Kulik, and H. Balakrishnan. 1999. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, *Seattle, WA, USA*, *15-19 August, 1999*, 174-185. ACM.	Network Routing	Medium-High
Intanagonwiwat, C., R. Govindan, and D. Estrin. 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, *Boston, MA, USA*, *6-11 August, 2000*, 56-67. ACM.	Network Routing	Medium-High

continued on next page...

...continued from previous page

Reference	Category	Relevance
Sohrabi, K., J. Gao, V. Ailawadhi, and G. J. Pottie. 2000. Protocols for self-organization of a wireless sensor network. *IEEE Personal Communications* 7 (5): 16-27.	Network Routing	Medium-High
Gupta, G., and M. Younis. 2003. Fault-tolerant clustering of wireless sensor networks. In *Proceedings of the IEEE Wireless Communication and Networking Conference*, *New Orleans, LA, USA*, *16-20 March 2003*. IEEE.	Network Structure	Medium
Iyengar, S. S., D. N. Jayasimha, and D. Nadig. 1994. A Versatile Architecture for the Distributed Sensor Integration Problem. *IEEE Transactions on Computers* 43 (2): 175-185.	Network Structure	Medium
Younis, O., and S. Fahmy. 2004. HEED: a hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. *IEEE Transactions on Mobile Computing* 3 (4): 366-379.	Network Structure	Medium
Jia, W., and W. Zhou, eds. 2005. *Distributed network systems*. Edited by D. Du and C. Raghavendra. Vol. 15, *Network Theory and Applications*. Boston, MA, USA: Springer Science + Business Media.	Networking Protocols	Low
Li, Q., R. Peterson, M. DeRosa, and D. Rus. 2003. Reactive behavior in self-reconfiguring sensor networks. *ACM Mobile Computing and Communications Review* 7 (1): 56-58.	Networking Protocols	Medium
Vuran, M. C., and I. F. Akyildiz. 2006. Cross-layer analysis of error control in wireless sensor networks. In *Proceedings of IEEE International Conference on Sensor and Ad-hoc Communications and Networks*, *Reston, VA, USA*, *25-28 September, 2006*, 585-594. IEEE.	Networking Protocols	Medium
Akyildiz, I. F., W. Su, Y. Sankarasubramaniam, and E. Cayirci. 2002. A survey on sensor networks. *IEEE communications magazine* 40 (8): 102-114.	Networking Protocols	Medium-High
Rabaey, J., J. Ammer, J. L. da Silva Jr, and D. Patel. 2000. PicoRadio: Ad-hoc wireless networking of ubiquitous low-energy sensor/monitor nodes. In *Proceedings of the IEEE Computer Society Workshop on VLSI*, *Orlando, FL, USA*, *27-28 April, 2000*, 9–12. IEEE.	Networking Protocols	Medium-High
Woo, A., and D. E. Culler. 2001. A transmission control scheme for media access in sensor networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, *Rome, Italy*, *16-21 July, 2001*, 221–235. ACM.	Networking Protocols	Medium-High
Woo, A., T. Tong, and D. Culler. 2003. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, *Los Angeles, CA, USA*, *5-7 November, 2003*, 14-27. ACM.	Networking Protocols	Medium-High

continued on next page...

...continued from previous page

Reference	Category	Relevance
Moss, S., C. Phoumsavanh, M. Konak, K. Tsoi, N. Rajic, S. Galea, I. Powlesland, and P. McMahon. 2009. Design of the Acoustic Electric Feedthrough Demonstrator MK II. *Materials Forum* 33.	Physical Layer	High
Sinha, A., A. Chandrakasan, and C. Mit. 2001. Dynamic power management in wireless sensor networks. *IEEE Design & Test of Computers* 18 (2): 62–74.	Power Management	High
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.	Sensor Networks	Medium
Shen, C. C., C. Srisathapornphat, and C. Jaikaeo. 2001. Sensor information networking architecture and applications. *IEEE Personal Communications* 8 (4): 52–59.	Sensor Networks	Medium
Ilyas, M., I. Mahgoub, and L. Kelly, eds. 2005. *Handbook of sensor networks: compact wireless and wired sensing systems*. Boca Raton, FL, USA: CRC Press LLC.	Sensor Networks	Medium-High
Iyengar, S. S., and R. R. Brooks, eds. 2005. *Distributed sensor networks*. Edited by S. Sahni, *Computer and Information Science Series*. Boca Raton, FL, USA: Chapman & Hall / CRC.	Sensor Networks	Medium-High
Stojmenovi, I., ed. 2005. *Handbook of sensor networks: algorithms and architectures*. Hoboken, NJ, USA: Wiley-Blackwell.	Sensor Networks	Medium-High
Dai, H., and R. Han. 2004. TSync: a lightweight bidirectional time synchronization service for wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review* 8 (1): 125-139.	Time Syncronisation	Low
Elson, J., and D. Estrin. 2001. Time Synchronisation for Wireless Sensor Networks. In *IPDPS Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, *San Francisco, CA, USA*, *27 April, 2001*.	Time Syncronisation	Low
Elson, J., L. Girod, and D. Estrin. 2002. Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS Operating Systems Review* 36: 147-163.	Time Syncronisation	Low
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.	Time Syncronisation	Medium
Ganeriwal, S., R. Kumar, and M. B. Srivastava. 2003. Timing-sync protocol for sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, *Los Angeles, CA, USA*, *5-7 November, 2003*, 138-149. ACM.	Time Syncronisation	Medium-High
Hu, A., and S. D. Servetto. 2003. Asymptotically optimal time synchronization in dense sensor networks. In *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications*, *San Diego, CA, USA*, *19 September, 2003*, 10. ACM.	Time Syncronisation	Medium-High

continued on next page...

...continued from previous page

Reference	**Category**	**Relevance**
Sichitiu, M. L., and C. Veerarittiphan. 2003. Simple, accurate time synchronization for wireless sensor networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, *New Orleans, LA, USA*, *16-20 March, 2003*, 16-20. IEEE.	Time Syncronisation	Medium-High
Sivrikaya, F., and B. Yener. 2004. Time synchronization in sensor networks: A survey. *IEEE Network* 18 (4): 45-50.	Time Syncronisation	Medium-High
van Greunen, J., and J. Rabaey. 2003. Lightweight time synchronization for sensor networks. In *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications*, *San Diego, CA, USA*, *19 September, 2003*, 11-19. ACM.	Time Syncronisation	Medium-High
van der Velden, S., and I. Powlesland. 2008. Interconnecting a Flexible Sensing Array. In *Second Asia-Pacific Workshop on Structural Health Monitoring*, *Melbourne, Australia*, *2-4 December, 2008*.	UAV Sensor Network	High
van der Velden, S., I. Powlesland, and J. Singh. 2007. A Proposed Robust Distributed Control System Architecture For Mini/Micro UAVs. In *22nd International Unmanned Air Vehicle Systems Conference*, *Bristol, UK*, *16-18 April, 2007*.	UAV Sensor Network	High

Appendix C: nesC component syntax examples

The following code snippet demonstrates the syntax for singleton configurations and modules, the code snippets were copied from the `tos/system/LedsC.nc`, `tos/system /LedsP.nc` and `tos/interfaces/Leds.nc` files contained in TinyOS 2.1.0. The comments in the original code were removed and then comments were added to describe the code, the ...'s indicate code portions left out for brevity:

```
// TinyOS 2.1.0 Leds Interface
interface Leds {
  async command void led0On();
  async command void led0Off();
  async command void led0Toggle();
  async command void led1On();
  async command void led1Off();
  async command void led1Toggle();
  async command void led2On();
  async command void led2Off();
  async command void led2Toggle();
  async command uint8_t get();
  async command void set(uint8_t val);
}

// TinyOS 2.1.0 LedsC Configuration: Wire platform leds to Leds
configuration LedsC {
  // This configuration provides a single interface
  provides interface Leds;
}
implementation {
  // This configuration uses the LedsP and
  //    PlatformLedsC singleton components
  components LedsP, PlatformLedsC;

  // Assign the LedsP component as the provider
  //    of the Leds interface provided by this configuration
  Leds = LedsP;

  // Wire the init interface used by PlatformLedsC to
  //    the one provided by LedsP
  LedsP.Init <- PlatformLedsC.Init;
  // Wire the Led0-2 interfaces used by LedsP to
  //    the ones provided by PlatformLedsC
  LedsP.Led0 -> PlatformLedsC.Led0;
  LedsP.Led1 -> PlatformLedsC.Led1;
  LedsP.Led2 -> PlatformLedsC.Led2;
}

// TinyOS 2.1.0 LedsP Module: Uses platform leds to implement Leds
module LedsP {
  provides {
    interface Init;
    interface Leds;
  }
  uses {
    // Interfaces can be re-named using the as keyword
    interface GeneralIO as Led0;
    interface GeneralIO as Led1;
    interface GeneralIO as Led2;
  }
}
implementation {
  command error_t Init.init() {
    ...
    return SUCCESS;
  }
  async command void Leds.led0On() {
    // Call a command in the Led0 interface being used by
    //  this module using the call keyword
    call Led0.clr();
  }
  ...
}
```

The following code snippet demonstrates the syntax for generic components, the Notify and Queue interfaces and QueueC module were copied from the `tos/interfaces /Notify.nc`, `tos/interfaces/QueueC` and `tos/system/QueueC.nc` files contained in TinyOS 2.1.0 respectively. The comments and other components were created for illustration:

```
// Notify the values in a queue as tasks are popped from
//  the scheduler, has no real application, it just serves as
//  an example of a range of nesC syntax.

// TinyOS 2.1.0 Notify Interface
interface Notify<val_t> {
    command error_t enable();
    command error_t disable();
    event void notify(val_t val);
}

// TinyOS 2.1.0 Queue Interface
interface Queue<t> {
    async command void push(t x);
    async command t pop();
    async command bool empty();
    async command bool full();
}

// TinyOS 2.1.0 QueueC Module
generic module QueueC(typedef queue_t, uint8_t queueSize) {
    provides interface Queue<queue_t>;
}
// Example Configuration to demonstrate instantiation of
//  a generic component
configuration ExampleConfiguration {
    provides interface Notify<uint8_t> as Notification;
}
implementation {
    // Instantiate a component to encapsulate a length 12
    //  queue of 8-bit unsigned integers
    components new QueueC(uint8_t, 12) as MyQueue;
    // Reference a singleton component (definition below)
    //  to use the queue and provide the notification signal
    components ExampleModule;

    // Wire MyQueue to the Notify interface
    Notification = MyQueue;
}

// Example Module to demonstrate event signalling, task posting and
//  use of generic module parameters
module ExampleModule {
    provides interface Notify<uint8_t> as Notification;
    uses interface Queue(uint8_t) as MyQueue;
}
implementation {
    // Private state variable to store status
    //  of the module
    bool enabled = FALSE;
    // enable command from Notification interface;
    //  posts notify task to scheduler, setting enabled
    //  to true
    command error_t Notification.enable() {
        atomic {
            if (post notify()) {
                enabled = TRUE;
            }
        }
        return enabled?SUCCESS:FAIL;
    }
    // disable command from Notification interface;
    //  sets enabled to false
    command error_t Notification.disable() {
        enabled = FALSE;
        return SUCCESS;
    }
    // Task that signals the notification with the top
    //  value on the queue
    task void notify() {
        if (enabled) {
            if (! call MyQueue.empty()) {
                signal Notification.notify(call MyQueue.pop());
            }
            // Re-post this task
            post notify();
        }
    }
}
```

Appendix D: SensorOS programming manual

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.

```
/**
 * Example Class definition file
 *
 * The example class prints out hello world n number
 *  of times when the helloWorld() method is called.
 * This class illustrates various points from the
 *  SensorOS coding conventions
 *
 * @author Robert Moore <rob@mooredesign.com.au>
 * @version 1.0.0
 * @date 2009-08-30
 */

#ifndef _EXAMPLE_CLASS_H
#define _EXAMPLE_CLASS_H

class ExampleClass {

    public:
        ExampleClass(uint8_t numTimes);
        ~ExampleClass();
        void helloWorld(void);
    private:
        static const char HELLO_WORLD_STRING[15];
        uint8_t numTimes;
};

#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

```
/**
 * Example class implementation file
 *
 * @author Robert Moore <rob@mooredesign.com.au>
 * @version 1.0.0
 * @date 2009-08-30
 */

// Include main.h for SensorOS core files
#include <main.h>

const char ExampleClass::HELLO_WORLD_STRING[] = "Hello World!\r\n";

/**
 * Example class constructor
 *
 * @param numTimes The number of times to print the
 *  hello world message when helloWorld() is called
 */
ExampleClass::ExampleClass(uint8_t numTimes) : numTimes(numTimes) {}

/**
 * Example class destructor
 */
ExampleClass::~ExampleClass() {}

/**
 * Print hello world message numTimes times
 */
void ExampleClass::helloWorld() {
    uint8_t i;
    for(i=0; i<numTimes; ++i) {
        Debug::print(HELLO_WORLD_STRING);
    }
}
```

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:

```
~/sensoros $ make
g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
    sensoros/applications/null/ -c os/main.cpp -o os/main.o
g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
    sensoros/applications/null/ -c os/network.cpp -o os/network.o
g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
    sensoros/applications/null/ -c os/scheduler.cpp -o os/scheduler.o
g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
    sensoros/applications/null/ -c os/signal_router.cpp -o os/signal_router.
    o
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
g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
    sensoros/applications/null/ -c platforms/dragon12/debug.cpp -o platforms
    /dragon12/debug.o
g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
    sensoros/applications/null/ -c platforms/dragon12/platform.cpp -o
    platforms/dragon12/platform.o
g++  -g -Wall -I~/sensoros/platforms/dragon12/ -I~/sensoros/os/ -I ~/
    sensoros/applications/null/ -c applications/null/application.cpp -o
    applications/null/application.o
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:

```
$PROJ_DIR$\src
$PROJ_DIR$\..\..\..\os
$PROJ_DIR$\..\..\..\platforms\dragon12
$PROJ_DIR$\..\..\..\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:

```
###
# Edit Below Here
###

# Set platform
PLATFORM = dragon12
# Set application
APPLICATION = null
# Set compiler
COMPILER = g++
# Set compiler flags
FLAGS = -g -Wall

###
# Don't Edit Below Here
###
```

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.

```
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.

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

Code Listing 4: Posting a static function to the scheduler

```
class MyClass {
public:
    static void myTask(void);
}
static const VoidStaticFunctor myTaskFunctor(&MyClass::myTask);
...
scheduler->postTask(&myTaskFunctor);
```

Code Listing 5: Posting a static method to the scheduler

```
class MyClass {
public:
    MyClass();
    void myTask(void);
}
static MyClass myClass();
static const VoidInstanceFunctor<MyClass> myTaskFunctor(&myClass, &MyClass
    ::myTask);
...
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

```
/**
 * Returns the singleton instance of the Scheduler class
 * @return The instance
 */
Scheduler* Scheduler::getInstance()

/**
 * Pushes the given task onto the end of the task queue
 * Contains an atomic section
 * @param task The task to push onto the queue
 * @return The status of the operation (either SUCCESS or EFULL)
 */
error_t Scheduler::pushTask(task_t task)

/**
 * Runs the task at the head of the queue and returns whether or
 *  not there is a task left on the queue
 * Contains an atomic section
 *  @return Whether or not there is a task left on the queue
 */
bool Scheduler::runNextTask()

/**
 * Enters an infinite loop that runs the next task on the queue and
 *  sends the processor into sleep mode when the queue is exhausted
 * Calls Platform::sleep()
 */
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.

```
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).

```
// signal_t Type
typedef enum {
    SIG_SOFTWARE_INIT=0,
    SIG_SYSTEM_BOOTED,
    #include <platform_signals.h> // Can define a max of 127 signals
    #include <application_signals.h> // Can define a max of 127 signals
    NUM_SIGNALS // Must be last!
} signal_t;
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.

```
// mainFunctor is already defined as a const VoidFunctor
// signalRouter is already defined as a reference to the signal router
//  see code listing 8
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.

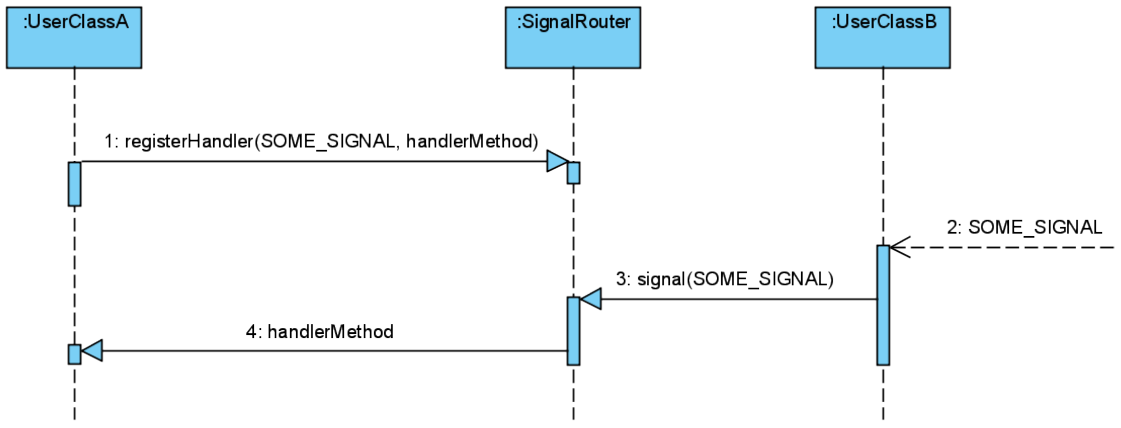


Figure 2.1: Sequence diagram illustrating how events are signalled and handled in SensorOS

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

```
/**
 * Returns the singleton instance of the SignalRouter class
 * @return The instance
 */
SignalRouter* SignalRouter::getInstance()

/**
 * Registers a functor to be called when a particular event occurs
 * Dynamically allocates memory, either explicitly for a list or implicitly
 *  via List::push_back()
 */
void SignalRouter::registerHandler(signal_t signal, VoidFunctor const*
   handler)

/**
 * Signals to the signal router that a particular event
 *  has occurred and any registered event handlers for
 *  that signal should be notified
 * @param signal The signal that has occurred
 */
void SignalRouter::signal(signal_t signal)

/**
 * SignalRouter destructor: deletes dynamically allocated handler lists
 */
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.

```
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.

```
static MyNetworkLayer myNetworkLayer();
...
// network is already defined as a reference to the network
//  see code listing 12
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.

```
// network is already defined as a reference to the network
//  see code listing 12
// dataLinkId is predefined (of type link_id_t) as the id
//  of the data link you want to reference
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

```
/**
 * Returns the singleton instance of the Network class
 * @return The instance
 */
Network* Network::getInstance()

/**
 * Registers a data link layer with the network
 * Will only add the data link layer if it is added
 *  in order (e.g. link id 1 first, then 2 etc.) and the link id
 *  is less than the NUM_DATA_LINKS platform defined constant
 * @param link A reference to the data link layer to register
 */
void Network::assignLink(DataLinkLayer* link)

/**
 * Returns a reference to the data link with the given id
 * @param linkId The id of the data link to return
 * @return A reference to the data link layer
 */
DataLinkLayer* Network::getLink(link_id_t linkId)

/**
 * Registers the network layer with the network
 * @param A reference to the network layer to register
 */
void Network::assignNetworkLayer(NetworkLayer* networkLayer)

/**
 * Returns a reference to the network layer
 * @return A reference to the network layer
 */
NetworkLayer* Network::getNetworkLayer()

/**
 * Returns the number of data links registered with the network
 * @return The number of data links
 */
link_id_t Network::getNumLinks()

/**
 * Returns the network address of the current node
 * @return The node's network address
 */
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

```
/**
 * Gives a packet to send across the link
 *
 * @param destination The destination to send to, NULL if not applicable
 *  or if broadcasting
 * @param msg A pointer to a message buffer containing the packet in the
 *  data field
 * @param length The length of the packet to be sent
 * @param priority The local priority of the message, smaller is greater
 *  set to NULL if not applicable
 *
 * @return The success of the sending operation
 *  SUCCESS If queuing the message for sending was successful
 *  ESIZE If length is greater than the value returned from
 *      maxPayloadLength()
 *  EFULL If the send queue is full
 *  EINVAL If length is 0
 *  FAIL If there was some other error sending
 */
virtual error_t downFromNetwork(node_address_t destination, message_t* msg,
                                data_length_t length, priority_t priority)

/**
 * Requests that the message pending sending in the given
 *  message buffer not be sent
 *
 * @param msg Pointer to the message buffer with the message to cancel
 *  the sending of
 *
 * @returns The status of the cancel attempt:
 *  SUCCESS if the message was still pending
 *  FAIL if the message had already been sent
 *  EINVAL if the message buffer wasn't recognised
 */
virtual error_t cancel(message_t* msg)

/**
 * Returns the link id of this data link (which then allows it to be
 *  referenced from the Network class via the getLink method)
 *
 * @return The id of this data link
 */
virtual link_id_t getLinkId()

/**
 * Returns the maximum length of the payload field
 *  i.e. MTU - header - footer
 *
 * @return The maximum payload field length
 */
virtual data_length_t maxPayloadLength()

/**
 * Returns the length of the payload field in the given message buffer
 *
```

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

```
 */
virtual data_length_t getPayloadLength(message_t* msg)

/**
 * Returns the timestamp in the given message buffer
 *
 * @return The timestamp in the given message buffer
 */
virtual timestamp_t getTimestamp(message_t* msg)

/**
 * Returns the source address in the given message buffer
 *
 * @return The source address in the given message buffer
 */
virtual node_address_t getSource(message_t* msg)

/**
 * Returns the destination address in the given message buffer
 *
 * @return The destination address in the given message buffer
 */
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

```
/**
 * Passes a filled buffer with received data to the network layer
 * Called within an atomic section to preserve integrity of message
 *  buffer; this method must only perform short computation(s) and then
 *  return ASAP
 *
 * @param linkLayer A pointer to the link layer that received the data
 * @param msg A pointer to the message buffer with the received data
 * @param length The length of the received payload (for convenience,
 *  same value can be gotten with linkLayer->getPayloadLength(msg)
 *
 * @return A pointer to a valid message buffer that the link layer
 *  can use for the next received packet
 */

virtual message_t* upFromDataLink(DataLinkLayer* linkLayer, message_t* msg,
                            data_length_t length)
/**
 * Allows release of the buffer pointed to by msg and informs
 *  network layer of the status of sending of the message in that
 *  buffer
 *
 * @param msg A pointer to the message buffer that can be freed
 * @param status The status of sending the message in msg:
 *  SUCCESS if the message was successfully sent
 *  FAIL if the message failed to send
 *  ECANCEL if the message was cancelled by a call to DataLinkLayer::cancel
 *   ()
 */
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.

```
// Define function
void func(void);
// Initialise VoidStaticFunctor with function pointer
static const VoidStaticFunctor myFunctor(&func);
// Call the functor (calls the func() function)
myFunctor();
```

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

```
// Define class with static method
class MyClass {
public:
    static void func(void);
}
// Initialise VoidStaticFunctor with method pointer
static const VoidStaticFunctor myFunctor(&MyClass::func);
// Call the functor (calls the MyClass::func() method)
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.

```cpp
// Define class with instance method
class MyClass {
public:
    MyClass();
    void func(void);
}
// Initialise instance of MyClass
static MyClass myClass();
// Initialise VoidInstanceFunctor with method pointer
static const VoidInstanceFunctor<MyClass> myFunctor(&myClass, &MyClass::
    func);
// Call the functor (calls the myClass.func() method)
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

```
// Define test class (MyClass) and some instances of it
class MyClass {
public:
    MyClass(uint8_t value) : value(value) {};
    uint8_t value;
};
MyClass one(1);
MyClass two(2);
MyClass three(3);
MyClass four(4);
MyClass five(5);
// Create a list, this is dynamic to match the usage in
//  SignalRouter, but this could also be a static instantiation
List<MyClass>* myList = new List<MyClass>();
// Add one to the back of the list
myList->push_back(&one);
// Get an iterator pointing to the current front of the list
// Uses the constructor of the iterator to assign the iterator
//  to an item
List<MyClass>::iterator i(myList->begin());
// Add two to the front of the list
myList->push_front(&two);
// Add three to the back of the list
myList->push_back(&three);
// Insert four after the item pointed to by iterator i
myList->insert(i, &four);
// Add five to the front
myList->push_front(&five);
// Iterate through the list with a new iterator, j
// Uses assignment to assign the iterator to an item
List<MyClass>::iterator j;
for (j=myList->begin(); j != myList->end(); ++j) {
    // Dereference the iterator to get the class
    //  reference, then print out the value in the class
    //  using the Debug class
    Debug::printf("%d\r\n", (*j).value);
}
// Note: Incrementing of the iterator MUST be preincrement
//  i.e. ++j instead of j++. Post increment is NOT supported
//  because it is less efficient (requires object copy)
// Note also: Only != and == are available for the iterator
//  and can be used to compare the iterator with an item in
//  the list or another iterator
```

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

```
/**
 * List destructor
 *
 * Deletes all the link objects allocated in this class
 */
template<class TClass> List<TClass>::~List()

/**
 * Adds the given item to the end of the list
 *
 * @param obj A reference to a TClass object to
 *  be added to the list
 */
template<class TClass> void List<TClass>::push_back(TClass* obj)

/**
 * Adds the given item to the start of the list
 *
 * @param obj A reference to a TClass object to
 *  be added to the list
 */
template<class TClass> void List<TClass>::push_front(TClass* obj)

    /**
 * Inserts the given item to the list after the item
 *  pointed to by the given iterator
 *
 * @param iter An iterator pointing to an item in the list
 * @param obj A reference to a TClass object to
 *  be added to the list
 */
template<class TClass> void List<TClass>::insert(List<TClass>::iterator&
    iter, TClass* obj)

/**
 * Returns the first item in the list
 *
 * @return The first item in the list
 */
template<class TClass> ListLink<TClass>* List<TClass>::begin() const

/**
 * Returns the last item in the list
 *
 * @return The last item in the list
 */
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

```
/**
 * Initialises the hardware at a very basic level
 *  e.g. CPU / memory mode selection
 */
void Platform::bootstrap()

/**
 * Initialise the hardware platform
 * Can post tasks to be run after the method is finished
 *  if there are order dependant tasks
 */
void Platform::init()

/**
 * Puts the hardware into a low power mode when there are no
 *  more tasks to run. It should exit this mode either after
 *  a specified amount of time, or preferably when there is
 *  an interrupt that requires the platform to perform some
 *  action, e.g. network activity
 */
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

```
/**
 * Prints a static string
 *
 * Prints the given string to Serial Communications Interface (SCI) 0
 *
 * @param string The string to print
 */
void Debug::print(const char* string)

/**
 * Prints a formatted string
 *
 * Prints the given format string to Serial Communications Interface (SCI)
     0
 * Uses the same syntax as the C stdio printf function
 *
 * @param format The format string
 * @param ... Any values referenced in the format string
 */
void Debug::printf(const char* format, ...)

/**
 * Lights a number of LEDs based on an integer value
 *
 * If the integer given is say 6, then LED 1 and 2 will be lit
 *  but LED 0 won't be
 *
 * @param ledsValue The value to set the leds to
 */
void Debug::leds(uint8_t ledsValue)

/**
 * Lights (or unlights) a specific LED
 *
 * @param ledValue Whether to light (true) or unlight
 *  (false) the LED
 * @param led The LED to modify (0-7)
 */
void Debug::leds(bool ledValue, uint8_t led)

/**
 * Toggles a specific LED on or off
 *
 * @param led The LED to toggle (0-7)
 */
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

```
/**
 * Initialises the application before interrupts are enabled
 */
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:

```
Application
```

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

```
========

null

Description
===========

SensorOS application that simply prints when it initialises and
when the system is booted.

Author
======

[Robert Moore](mailto:rob@mooredesign.com.au)

Date
====

2009-09-27

Version
=======

1.0.0

Platforms
=========

This application should be compatible with all platforms, at present
it has been successfully used with the following platforms:

* dragon12

Notes
=====

This is a simple example application that can be used when
initially developing platforms to ensure they compile.

License
=======

MIT
---

Copyright (c) 2009 Robert Moore

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
```

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.

```
#ifdef PLATFORM_DRAGON12
    // Use CAN module 0 as data link 1
    #define MSCAN0 1
    // Use CAN module 3 as data link 2
    #define MSCAN3 2
#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.

```
/**
 * Main system function (first function the system loads)
 *
 * @returns The status of the program, should never actually return
 *  due to infinite scheduler loop
 */
int main() {

    // Platform bootstrap
    Platform::bootstrap();
    // Initialise the scheduler
    Scheduler* scheduler = Scheduler::getInstance();
    // Initialise the platform
    Platform::init();
    // Run any tasks
    while (scheduler->runNextTask());
    // Initialise the signal router
    SignalRouter* signalRouter = SignalRouter::getInstance();
    // Send a software initialisation signal
    // Allows the platform initialisation to post event handlers
    //  for this signal if they need to be deferred until after
    //  initialisation
    signalRouter->signal(SIG_SOFTWARE_INIT);
    // Run any tasks
    while (scheduler->runNextTask());
    // Initialise the application
    Application::init();
    // Run any tasks
    while (scheduler->runNextTask());
    // Enable interrupts
    __enable_interrupts();
    // Call the system booted event
    signalRouter->signal(SIG_SYSTEM_BOOTED);
    // Spin in the scheduler loop
    scheduler->taskLoop();
    // Include for compiler, shouldn't ever be reached
    return -1;
}
```

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`

```
class MyClass: implements MyInterface {
public:
    MyClass();
    void someMethod(uint8_t someValue);
    void someMethod2(uint8_t someOtherValue);
}
```

Code Listing 31: Implementing an Interface

```
Atomic
    // Race condition vulnerable code ...
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).

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

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

```
{
    Atomic
    // ...
}
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.

```
              14 Bytes        MAX_DATA_BYTES Bytes       4B   7 Bytes
          +--------------+------------------------------+----+-------+
message_t |    header    |            data              |foot| meta  |
          +--------------+------------------------------+----+-------+

          +--------------+------------------------------+    +-------+
Ethernet  |    header    |            data              |    | meta  |
          +--------------+------------------------------+    +-------+

                      +--+--------+                     +----+
CAN                   |hd|  data  |                     |foot|
                      +--+--------+                     +----+

                      +--+--------+----+
CAN (contiguous)      |hd|  data  |foot|
                      +--+--------+----+
```

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:

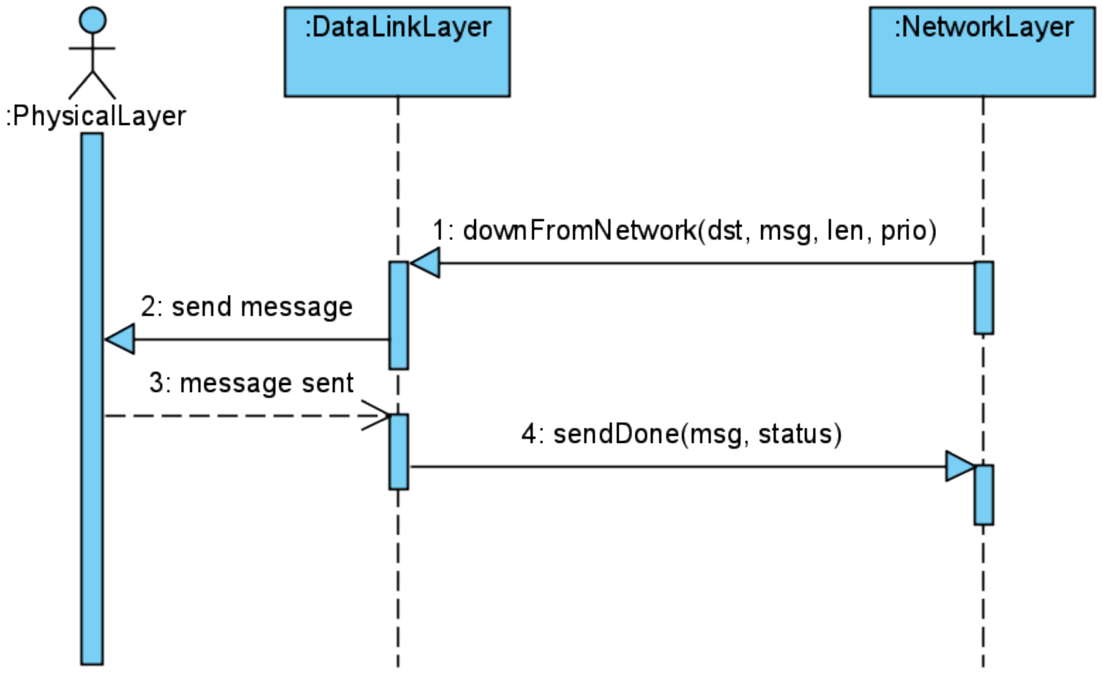


Figure 2.3: Sequence diagram illustrating how messages are sent in SensorOS

- **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->`

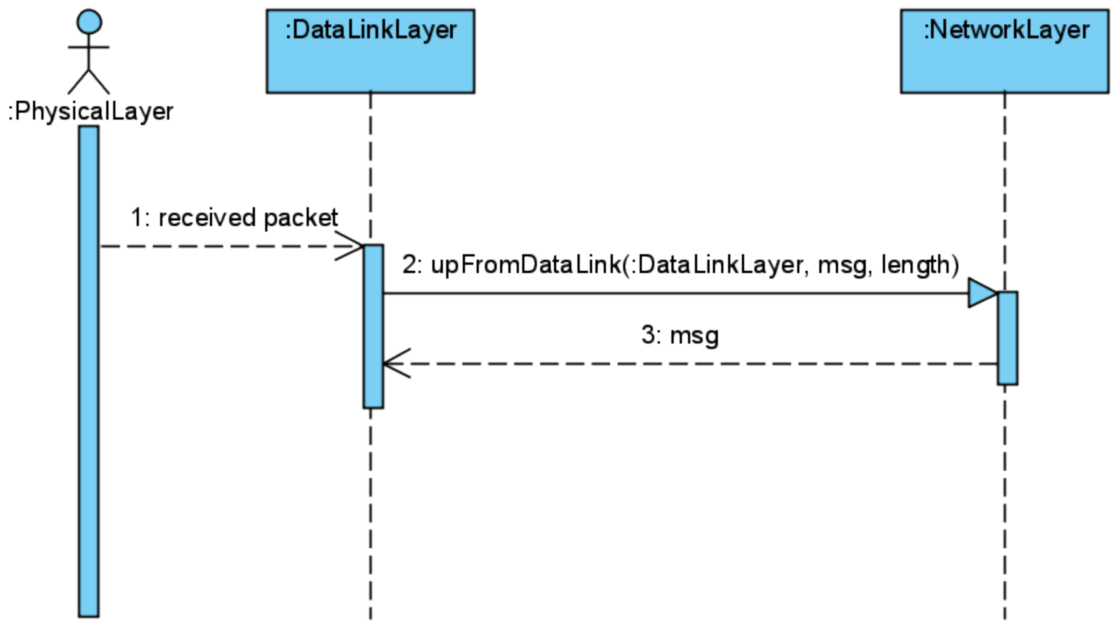


Figure 2.4: Sequence diagram illustrating how messages are received in SensorOS

`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.

```
class TestNetwork: implements NetworkLayer {
public:
    TestNetwork() {
        // Set all buffers as unused
        used = 0xFF;
    };
    message_t* upFromDataLink(DataLinkLayer* linkLayer, message_t* msg,
                              data_length_t length);
    void sendDone(message_t* msg, error_t status) {
        // Free the buffer used by msg
        ptrdiff_t bufferId = (ptrdiff_t)(buffers - msg);
        if (bufferId >=0 && bufferId <= 7) {
            Atomic
                if (bufferId == 0) {
                    used |= 0x1;
                } else {
                    used |= 0x1 << bufferId;
                }
            EndAtomic;
        }
        // ...
    };
private:
    // Array of message buffers
    message_t buffers[8];
    // Bit mask, a 1 indicates an unused buffer
    uint8_t used;
    /**
     * Returns a pointer to a free message buffer
     *
     * The message buffer returned is reserved atomically
     *  so it can't be used elsewhere
     *
     * @returns The message buffer pointer, or NULL if no
     *  free buffers
     */
    message_t* getFreeBuffer() {
        message_t* buffer = NULL;
        uint8_t lowestFreeBuffer = 0;
        // Extract lowest set bit
        // http://realtimecollisiondetection.net/blog/?p=78
        uint8_t lowestSetBit = used&-used;
        uint8_t bit = lowestSetBit;
        Atomic
            // If there are any unused buffers
            if (used > 0) {
                // Find log2 of lowestUnused to get the buffer number
                // Takes advantage of the fact that if only one
                //  bit is set in the integer, if the integer is 4 or less
                //  the log2 of it is itself shifted right one
                // For an 8-bit number, this loop will enter a maximum of
                //   two times
                while (bit > 0x4) {
                    lowestFreeBuffer += 0x3;
                    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.

```
// MSCAN Message Buffer
typedef struct {
    node_address_t source;
    node_address_t destination;
} mscan_header_t;
typedef struct {
    data_length_t length;
    priority_t priority;
    timestamp_t timestamp;
} mscan_footer_t;
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.

```
// All Message Buffers (Used in conjunction with sizeof() in
//  message_t definition)
typedef union message_header {
    mscan_header_t mscan;
} message_header_t;
typedef union message_footer {
    mscan_footer_t mscan;
} message_footer_t;
typedef union message_metadata {
    mscan_metadata_t mscan;
} 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:

```
Platform
========

dragon12

Description
===========

SensorOS platform for the [Wytec Dragon12 Development
Board](http://www.evbplus.com/dragon12_hc12_68hc12_9s12_hcs12.html)

Author
======

[Robert Moore](mailto:rob@mooredesign.com.au)

Date
====

2009-09-27

Version
=======

1.0.0
```

```

Processor
=========

[Freescale 9S12/HCS12](http://www.freescale.com/webapp/sps/site/
overview.jsp?nodeId=06258A25802570256D)

Compilers
=========

IAR Embedded Workbench (EW) and GCC compatible

When compiling with IAR EW, the following files need to be added
to the project:

* sensoros/os/main.cpp
* sensoros/os/network.cpp
* sensoros/os/scheduler.cpp
* sensoros/os/signal_router.cpp
* sensoros/os/signal_router.cpp
* sensoros/platforms/dragon12/can_link.cpp
* sensoros/platforms/dragon12/debug.cpp
* sensoros/platforms/dragon12/platform.cpp
* sensoros/platforms/dragon12/iar.s12
* Any application source files

And "Additional include directories" need to be added (Project >
C/C++ Compiler > Preprocessor) as (assuming the project resides in
a subdirectory of the application's directory (e.g. sensoros/
applications/<application_name>/iar/)):

* $PROJ_DIR$\..\..\..\os
* $PROJ_DIR$\..\..\..\platforms\dragon12
* $PROJ_DIR$\..

Notes
=====

There wasn't any macros defined by IAR EW, so in platform.h
__IAR__ is set if the compiler is not GCC. If the platform is
extended to include any more compilers then the setting of __IAR__
will need to be modified.

If an application wants to use any of the 5 CAN modules then it
should define CAN<X> (where X is 0 - 4, and refers to CAN module
X) as the link id number that the corresponding CAN module should
be. These definitions need to go in application_network_info.h and
should obviously be used as a conditional compilation for this
platform only, e.g.:

        #ifdef PLATFORM_DRAGON12
```

```
                #define MSCAN0 1 // Use CAN module 0 as data link 1
                #define MSCAN3 2 // Use CAN module 3 as data link 2
        #endif

Note: you must set the data link ids in ascending, incremental
order or the assignment of the data link ids to the network will fail.

The debug class will print strings to SCI0 at a baud of 19200 with 1
start bit, 8 data bits, 1 stop bit and no parity bits.

License
=======

MIT
---

Copyright (c) 2009 Robert Moore

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
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

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

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

```
#ifdef __IAR__
    #pragma vector=MSCAN4TX_VECTOR
#endif
interrupt_t can4TXISR(void) {
    // Signal a CAN 4 Send event
    SignalRouter::getInstance()->signal(SIG_CAN4_SENT);
    // Call the sendDone method for this link
    can4Link.sendDone();
}
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.

```
#define MESSAGE_HEADER(msg, type) ((type*)((msg)->data-sizeof(type)))
#define MESSAGE_FOOTER(msg, type) ((type*)((msg)->footer))
#define MESSAGE_FOOTER_CONTIGUOUS(msg, type) ((type*)((msg)->footer)-
    getPayloadLength(msg))
#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.

```
/**
 * Returns the source address of the message in the given message buffer
 * @return Message source
 */
node_address_t CanLink::getSource(message_t* msg) {

    return MESSAGE_HEADER(msg, mscan_header_t)->source;
}
```

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.

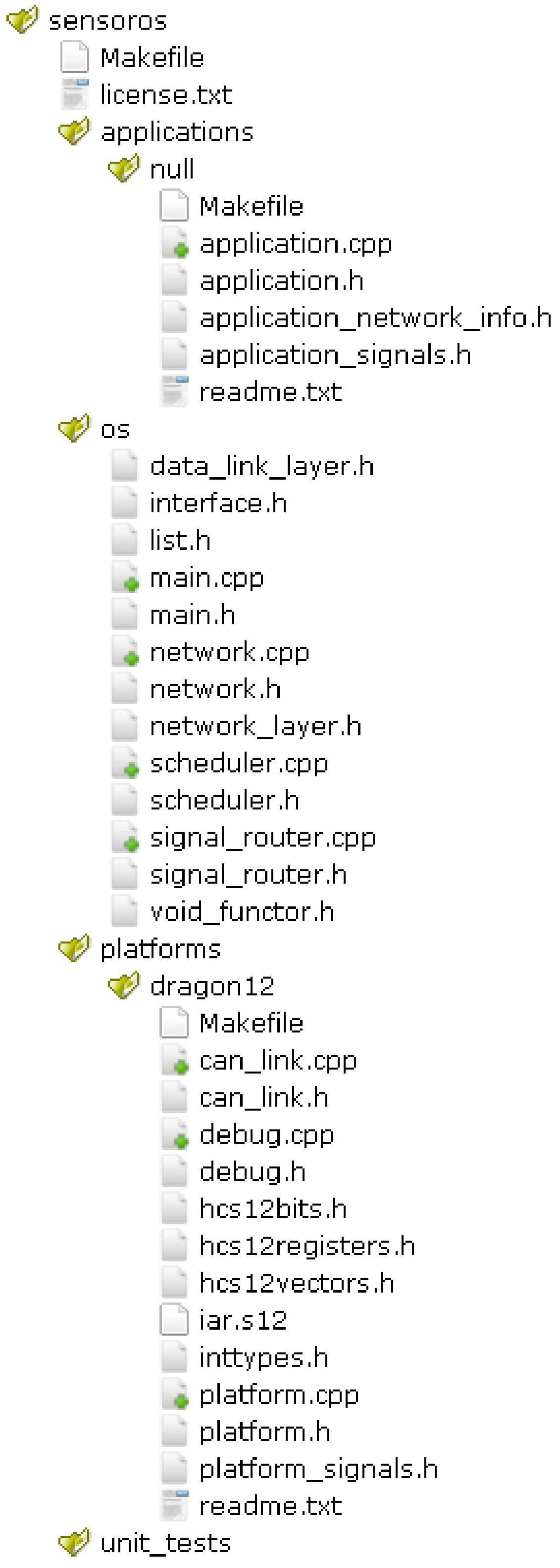
Tolle, G., P. Levis, and D. Gay. 2008. TEP 114: SIDs: Source and sink independent drivers. http://www.tinyos.net/tinyos-2.x/doc/html/tep114.html, (accessed September 26, 2009).

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).

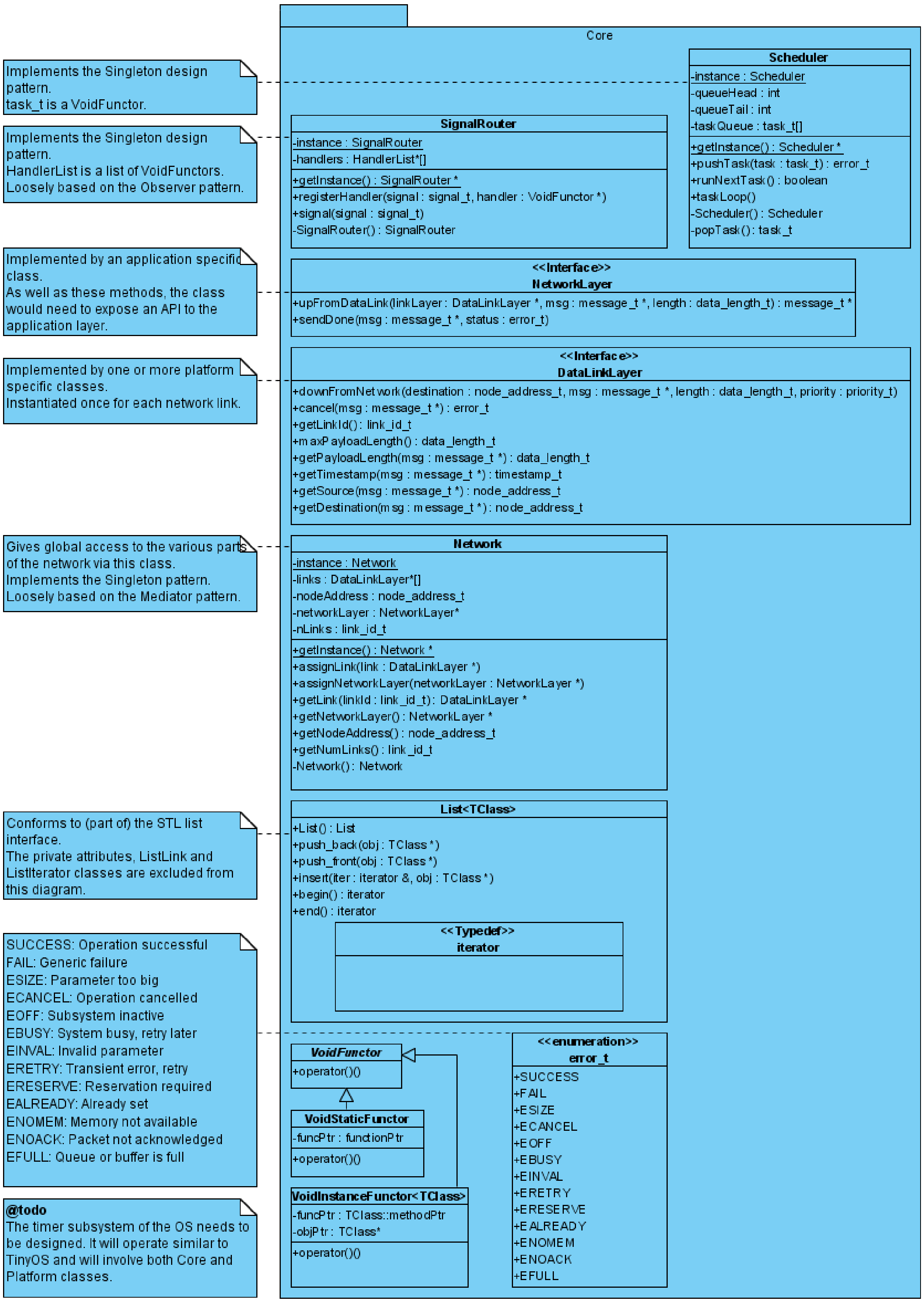
Appendix A: File Structure

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



```
sensoros
    Makefile
    license.txt
    applications
        null
            Makefile
            application.cpp
            application.h
            application_network_info.h
            application_signals.h
            readme.txt
    os
        data_link_layer.h
        interface.h
        list.h
        main.cpp
        main.h
        network.cpp
        network.h
        network_layer.h
        scheduler.cpp
        scheduler.h
        signal_router.cpp
        signal_router.h
        void_functor.h
    platforms
        dragon12
            Makefile
            can_link.cpp
            can_link.h
            debug.cpp
            debug.h
            hcs12bits.h
            hcs12registers.h
            hcs12vectors.h
            iar.s12
            inttypes.h
            platform.cpp
            platform.h
            platform_signals.h
            readme.txt
    unit_tests
```

Appendix B: SensorOS.Core - UML Class Diagram


Implements the Singleton design pattern.
task_t is a VoidFunctor.

Implements the Singleton design pattern.
HandlerList is a list of VoidFunctors.
Loosely based on the Observer pattern.

Implemented by an application specific class.
As well as these methods, the class would need to expose an API to the application layer.

Implemented by one or more platform specific classes.
Instantiated once for each network link.

Gives global access to the various parts of the network via this class.
Implements the Singleton pattern.
Loosely based on the Mediator pattern.

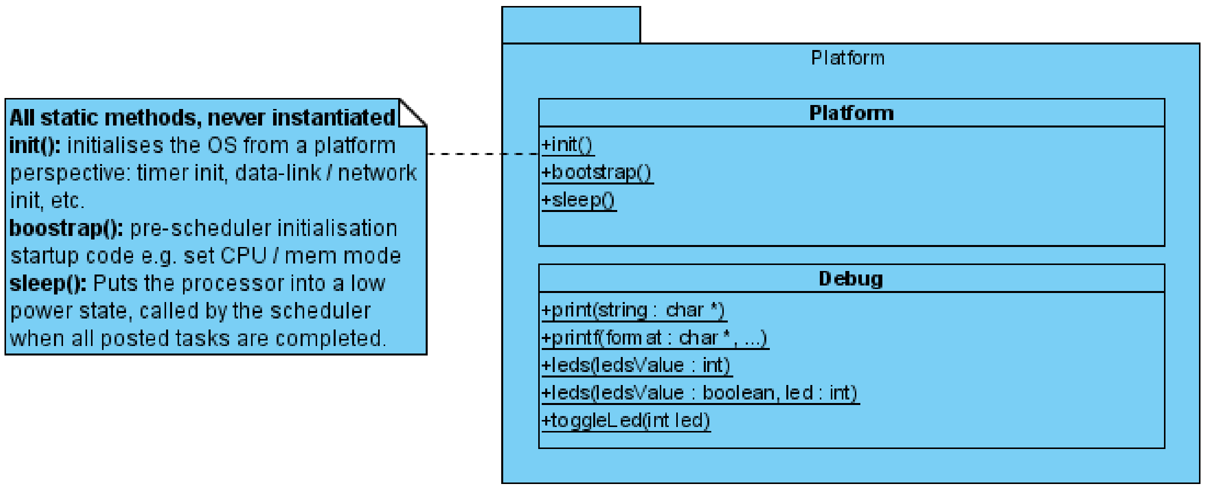
Conforms to (part of) the STL list interface.
The private attributes, ListLink and ListIterator classes are excluded from this diagram.

SUCCESS: Operation successful
FAIL: Generic failure
ESIZE: Parameter too big
ECANCEL: Operation cancelled
EOFF: Subsystem inactive
EBUSY: System busy, retry later
EINVAL: Invalid parameter
ERETRY: Transient error, retry
ERESERVE: Reservation required
EALREADY: Already set
ENOMEM: Memory not available
ENOACK: Packet not acknowledged
EFULL: Queue or buffer is full

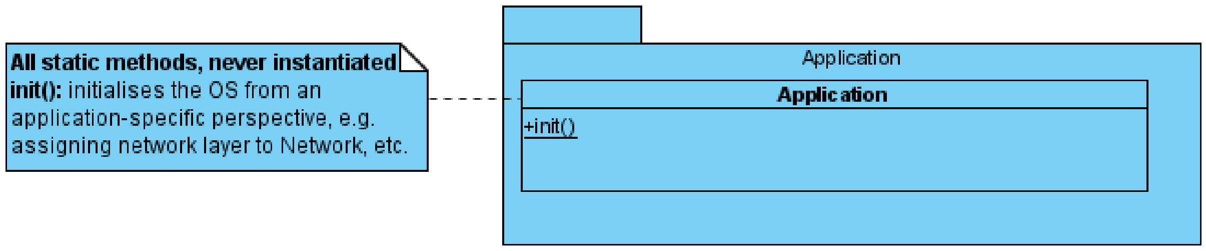
@todo
The timer subsystem of the OS needs to be designed. It will operate similar to TinyOS and will involve both Core and Platform classes.

Core
Scheduler
-instance : Scheduler
-queueHead : int
-queueTail : int
-taskQueue : task_t[]
+getInstance() : Scheduler *
+pushTask(task : task_t) : error_t
+runNextTask() : boolean
+taskLoop()
-Scheduler() : Scheduler
-popTask() : task_t
SignalRouter
-instance : SignalRouter
-handlers : HandlerList*[]
+getInstance() : SignalRouter *
+registerHandler(signal : signal_t, handler : VoidFunctor *)
+signal(signal : signal_t)
-SignalRouter() : SignalRouter
<<Interface>>
NetworkLayer
+upFromDataLink(linkLayer : DataLinkLayer *, msg : message_t *, length : data_length_t) : message_t *
+sendDone(msg : message_t *, status : error_t)
<<Interface>>
DataLinkLayer
+downFromNetwork(destination : node_address_t, msg : message_t *, length : data_length_t, priority : priority_t)
+cancel(msg : message_t *) : error_t
+getLinkId() : link_id_t
+maxPayloadLength() : data_length_t
+getPayloadLength(msg : message_t *) : data_length_t
+getTimestamp(msg : message_t *) : timestamp_t
+getSource(msg : message_t *) : node_address_t
+getDestination(msg : message_t *) : node_address_t
Network
-instance : Network
-links : DataLinkLayer*[]
-nodeAddress : node_address_t
-networkLayer : NetworkLayer*
-nLinks : link_id_t
+getInstance() : Network *
+assignLink(link : DataLinkLayer *)
+assignNetworkLayer(networkLayer : NetworkLayer *)
+getLink(linkId : link_id_t) : DataLinkLayer *
+getNetworkLayer() : NetworkLayer *
+getNodeAddress() : node_address_t
+getNumLinks() : link_id_t
-Network() : Network
List<TClass>
+List() : List
+push_back(obj : TClass *)
+push_front(obj : TClass *)
+insert(iter : iterator &, obj : TClass *)
+begin() : iterator
+end() : iterator
<<Typedef>>
iterator
VoidFunctor
+operator()()
VoidStaticFunctor
-funcPtr : functionPtr
+operator()()
VoidInstanceFunctor<TClass>
-funcPtr : TClass::methodPtr
-objPtr : TClass*
+operator()()
<<enumeration>>
error_t
+SUCCESS
+FAIL
+ESIZE
+ECANCEL
+EOFF
+EBUSY
+EINVAL
+ERETRY
+ERESERVE
+EALREADY
+ENOMEM
+ENOACK
+EFULL

Appendix C: SensorOS.Platform - UML Class Diagram


All static methods, never instantiated
init(): initialises the OS from a platform perspective: timer init, data-link / network init, etc.
boostrap(): pre-scheduler initialisation startup code e.g. set CPU / mem mode
sleep(): Puts the processor into a low power state, called by the scheduler when all posted tasks are completed.
Platform
Platform
+init()
+bootstrap()
+sleep()
Debug
+print(string : char *)
+printf(format : char *, ...)
+leds(ledsValue : int)
+leds(ledsValue : boolean, led : int)
+toggleLed(int led)

Appendix D: SensorOS.Application - UML Class Diagram


All static methods, never instantiated
init(): initialises the OS from an application-specific perspective, e.g. assigning network layer to Network, etc.
Application
Application
+init()

Appendix E: Digital content

E.1 Overview

This section outlines the digital content that is distributed with this thesis be it on DVD when this thesis was submitted, on the Curtin Department of Electrical and Computer Engineering web server, or on the SensorOS Website. The following subsections each describe the content contained within each root level directory.

E.2 BibTeX to Endnote Conversion

One of the useful things that was learnt while writing this thesis was how to convert between BibTeX and Endnote references and vice versa. In order to share this knowledge, the `bibtex-endnote-conversion` directory contains a number of files to facilitate these conversions. The `BibTeX_Export.ens` file is an Endnote style (created with version X1) for exporting to BibTeX format that is much better than the one that comes with Endnote. To be used it needs to be placed in the Styles directory within the Endnote installation directory. The `bibtex_to_endnote.txt` file gives instructions on how to convert from BibTeX to Endnote.

E.3 LaTeX Thesis Template

This thesis was typeset using the LaTeX typesetting engine. The LaTeX code that sets up the template for the thesis was deliberately made flexible enough to be easily reused with the goal of sharing the code with future students. The `curtin-ece-latex-thesis-template` directory contains a blank template and usage instructions for someone creating a Curtin Department of Electrical and Computer Engineering Honours Thesis in LaTeX.

E.4 SensorOS Programming Manual LaTeX Code

The LaTeX code and corresponding files and documents that were used to create the SensorOS Programming Manual (Appendix D) are contained in the `manual` directory.

E.5 Research Documents

The documents written by the DSTO that this project was based off are contained in `research/dsto` for reference, since they don't appear to be readily available on the Internet. Technical documentation for the HCS12 and Dragon12 board that were collected from various sources are also included for reference (in `research/hcs12`).

E.6 Seminar

The PowerPoint files that were created for the Project Seminar and a presentation at the 2009 West Australian Institution of Engineering & Technology (IET) Present Around The World (PATW) student paper competition are included in the `seminar` directory.

E.7 SensorOS

The files that make up the SensorOS operating system are included in the `sensoros` directory.

E.8 Specification

The Word document, Endnote reference file and Endnote output style that were used to generate the requirements specification in Appendix A are contained in the `specificat ion` directory.

E.9 Thesis LaTeX Code

The LaTeX code and corresponding files and documents that were used to create this thesis are contained in the `thesis` directory.

E.10 TinyOS

A copy of TinyOS 2.1.0 is contained in the `tinyos` directory. The following files have been added as part of the initial port to the HCS12 / Dragon12 hardware platform (as described in section 5.4.2):

- `support/make/dragon12.target`
- `support/make/hcs12/*`
- `tos/chips/hcs12/*`
- `tos/platforms/dragon12/*`

E.11 UML

The Visual Paradigm files that were created to generate the UML 2.0 diagrams shown in the SensorOS Programming Manual (Appendix D) are contained in the `uml` directory.

E.12 Website

The files that make up the SensorOS Website are contained in the `website` directory.