1st International Conference on Sensing Technology
November 21-23, 2005 Palmerston North, New Zealand

# Software Architecture to Integrate Sensors and Controllers

Ryan D. Weir, G. Sen Gupta, Donald G. Bailey
Institute of Information Sciences and Engineering,
Massey University,
Palmerston North, New Zealand
Ryan.Weir.1@uni.massey.ac.nz, G.SenGupta@massey.ac.nz, D.G.Bailey@massey.ac.nz

## Abstract

A system architecture is proposed for integrating sensors, controllers, actuators and instrumentation within a common framework. The goal is to provide a flexible and scalable system. Extending the system, by adding additional components such as sensors or actuators, does not increase the overheads and is achieved seamlessly with minimal modification of the core controller program. The architecture is generic, though it has been proposed to implement a prototype system on a small form factor PC to remote control an autonomous vehicle. The architecture will find application in many other spheres such as home, office and factory automation, process and environmental monitoring, surveillance and robotics.

**Keywords**: sensor, flexible architecture, controller, actuator, scalable system

## 1 Introduction

Sensors are used in a variety of environments, with varying platforms. In the real world, sensory data comes from multiple sensors of different modalities in either a distributed or centralised location. There are enormous challenges in collecting data, evaluating the information, performing decision-making, formulating meaningful user displays and actuating alarms or other controls.

In the last decade, tremendous research efforts have been concentrated in the area of intelligent networked sensors and wireless network sensors addressing issues of self-organisation, scalability and data flow management [1-3]. A system architecture for monitoring habitat has been proposed in [1]. A tiered architecture comprising sensor nodes, connected to a base station via transit networks, Internet connectivity to data services, and user interface for data display has been developed. Their architecture does not incorporate any actuators. Moreover, the emphasis is on network architecture, rather than the controller architecture. [2] presents protocol architectures for scalable self-assembly of sensor networks. [3] makes a case for a data-centric network in which data is named by attributes. This approach decouples data from the sensor that produces it allowing for more robust application design. The CODGER system [4], developed for large remote vehicles, addresses the overall architecture of a sensor system. The manager keeps a list of which tasks are ready to execute, and in its central loop selects one task to execute. Each task has a pre-determined pattern of data that must be satisfied before it can be considered for execution. Tasks have access to data at various levels and each task is a separate, continuously running program.

While there is enough work that has been done on sensor fusion, no evidence is available that leads us to believe that attempts have been made to design a flexible architecture to integrate sensors, actuators, controller and instrumentation within a single framework. In a scalable system, it should be easy to add components, such as sensors and actuators, seamlessly as this happens frequently. In most systems, this would entail considerable system re-configuration, resource re-mapping and possibly change of control methods. This has motivated the authors to undertake this work and we believe that a generic integrated sensor and controller architecture would be beneficial in many applications.

The proposed architecture uses a data driven approach. The data is passed from one task (module) to another, with each module processing the data in some way. Each processing module is independent. There is separate processing for data acquisition, communication and actuation. This makes modifying or adding further processing blocks, for example to add new features when required, quite straightforward.

In this paper we present the details of the integrated sensor and controller architecture using the control of a remote vehicle from a base station as an example. This configuration is illustrated in Figure 1 where a remote model helicopter is controlled for surveillance purposes. However, since this architecture is flexible, scalable and extensible it could be used for a variety of other applications such as home, office and factory

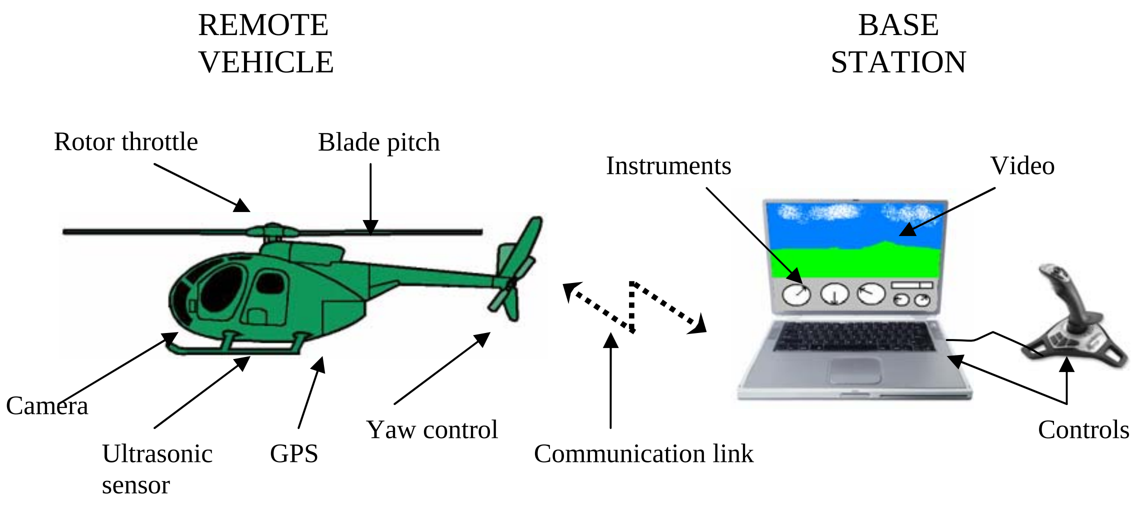


**Figure 1:** Typical system configuration for remote vehicle control illustrating use of sensors, instruments, controls and actuators.

automation, process and environmental monitoring, surveillance and robotics.

## 2 System Architecture

To simplify the design, a common architecture is used for both the remote vehicle and the base station. The architecture has a number of desired features. To be general purpose, it must be able to take readings from a variety of sensors. These sensors can be on different buses with varying protocols. It may need to perform custom processing or manipulation of the data obtained. This includes converting or stripping the raw information from the sensor to get information that can be used. Another key function of the architecture is to facilitate the timely communication of sensor and control data between the remote and base station.

The communication channel could be wired or wireless and based on a number of different protocols. The display shows processed data on virtual instruments. For remote vehicle control, the base station may be configured as a virtual cockpit. The architecture accepts control input from the base station. It then communicates this control information to the remote platform. The proposed architecture also allows for easy integration of autonomous control operations.

These requirements suggest that a data-driven approach is most appropriate for the system architecture. In this context, it is observed that every device may be thought of as a data source or data sink as shown in the basic system structure in Figure 2. Sensors and controls are both data sources. Actuators and virtual instruments are data sinks. Data conversion and autonomous operations are both sinks and sources, with the sink corresponding to the input to the function and the processed result or control output being a data source. Each direction of the communications channel may be considered a data sink at the transmitting end, and a data source at the receiver.

The core of the system is therefore a data manager that coordinates the data received from the sources, and ensures that it gets passed to the corresponding sinks.

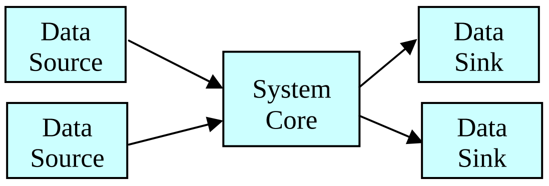
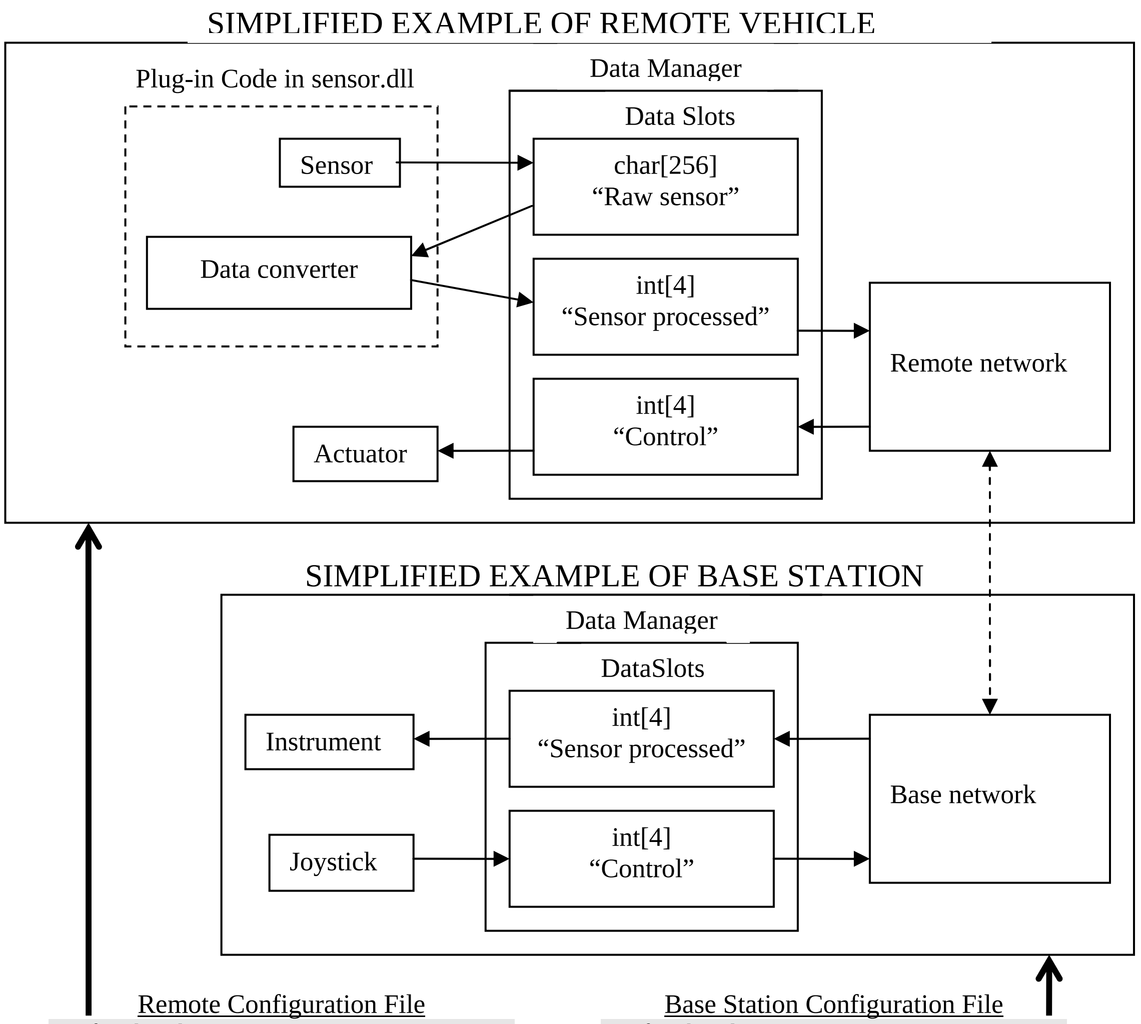

$$\begin{tikzcd}
\text{Data Source} \arrow[dr] & & \text{Data Sink} \\
& \text{System Core} \arrow[ur] \arrow[dr] & \\
\text{Data Source} \arrow[ur] & & \text{Data Sink}
\end{tikzcd}$$

**Figure 2:** Basic structure of the system where incoming data is considered a data source, and outgoing data a data sink.

### 2.1 System Operation

To maintain maximum flexibility, it is important to keep the operations performed on the data separate from the sensing and actuation processes. This may be accomplished by facilitating communication between the various modules through a set of data buffers. These buffers may be thought of as a little like pigeon-holes in a filing clerk’s office.

As new data becomes available at a source (either because a sensor reading is available, or a control input has changed) the new data is placed into a corresponding data slot within the manager. The manager then notifies all of the data sinks that depend on the new data so that they may begin their processing, transmitting, or displaying depending on their particular function. The data processing is



Remote Configuration File

```
// Define data slots
DATA = "\char[256]"
        NAME = "Raw sensor"
DATA = "\int[4]"
        NAME = "Sensor processed"
        NETID = 1
DATA = "\int[4]"
        NAME = "Control"
        NETID = 2

// Define sources – sensor.
DEVICE = "sensor.dll\setupSensor"
        NAME = "Sensor"
        DATAOUT = "Raw sensor"
        PORT = "COM1"
        // String for setting RS-232 comms
        MODE = "9600, N, .."

// Data Converter.  Extracting useful information from
data received from the sensor.
DEVICE = "sensor.dll\setupConverter"
        NAME = "Data converter"
        DATAIN = "Raw Sensor"
        DATAOUT = "Sensor processed"

// Actuator
DEVICE = "\Servo"
        NAME = "Actuator"
        DATAIN = "Control"

// Network
DEVICE = "\Network"
        NAME = "Remote network"
        DATAIN = "Sensor processed"
        DATAOUT = "Control"
```

Base Station Configuration File

```
// Define data slots
DATA = "\int[4]"
        NAME = "Sensor processed"
        NETID = 1
DATA = "\int[4]"
        NAME = "Control"
        NETID = 2

// Network
DEVICE = "\Network"
        NAME = "Base network"
        DATAOUT = "Sensor processed"
        DATAIN = "Control"

DEVICE = "\Display"
        NAME = "Instrument"
        POSITION = (200,300)
        SIZE = (200,200)
        DATAIN = "Sensor processed"

DEVICE = "\Gameport"
        NAME = "Joystick"
        DATAOUT = "Control"
```

**Figure 3:** Flow of information and configuration files in a simplified example. This shows how sensors, data converters, bidirectional communication, instruments and controls are all integrated in the framework.

facilitated by a task queue. Every data sink must register itself with a data slot that contains its input data. When new data becomes available, the manager adds each of the data sinks that have registered to process the new data into a task queue, where each is called in turn to process the data.

Some sensors may require a periodic trigger to acquire data. Within the proposed architecture, this is accomplished by using a timer. The timer also works through the data slot mechanism, in that when a timer times out, this is indicated through a data slot. To be triggered, a sensor only needs to be registered as a sink on the timer data slot. When the timer times out, then the sensor trigger is queued, and when executed, the data capture phase begins.

Keeping data sources and handlers separate in this way enhances flexibility and greatly simplifies reconfiguration. Data dependencies are implicit through the data slots, and this dependency is specified in a configuration file.

## 2.2 System Initialisation

To maintain flexibility, a particular application is set up using a configuration file. This is loaded into the program upon start up. A simple example is shown in Figure 3. There are two sections to the file.

The first section defines the data slots. A data slot is identified with the keyword “DATA”. It specifies the type of data contained (char, int), the name, the size and network identification code if it is needed. The name of a data slot must be unique as it is used to identify the data buffer to any devices that may use data from, or produce data for that slot. If the data slot is a timer, there is a field which shows how often that data slot is activated. There is also the facility of creating a custom data slot if needed. Data slots are derived from a common C++ base class that performs the management operations such as signalling when new data is available, and queuing data sinks on the task queue. A custom data slot allows more complex data structures to be implemented. In the example in Figure 3, there are three data slots created in the remote system, and two in the local system. The first is a character buffer that can hold string of up to 256 characters. All of the other slots are arrays of 4 integers.

A device is any data source or sink and is defined by the keyword “DEVICE”. The second section of the configuration file specifies the devices. The device keyword specifies the location of the constructor for the C++ class that will handle the device. This may either be a built-in device or contained within a DLL. By making use of DLL plug-ins, whenever a new device is added, the code for handling that device can be easily provided without having to rebuild the complete system.

The name attribute of the device is not actually required by the system, but provides documentation, and can be used to identify a device in error messages when something goes wrong. The other attributes for each device depend on what type of device it is. Those that are data sources specify the data slot that the new data is to be written to. The configuration file therefore associates a source with the data slot. If a device is a data sink, the configuration file indicates where the data is to come from. This enables the manager to register the device as a listener on the data slot. Many of the remaining attributes are device dependent, and specify additional setup information required by the device handler, such as what port or protocol to use to gather the data from a sensor, or the location of a virtual instrument on the screen.

Devices may also specify a task queue to use instead of the default. This will be described in more detail later.

A data converter is just another type of device, which processes the data to extract useful or required information into a more convenient form. For example to convert a string to an integer, or extracting latitude and longitude information from a NMEA string produced by a GPS device [5]. The DATAIN attribute points to the data slot containing the raw information from the sensor, and the DATAOUT attribute points to the data slots being written to with the processed data. Data converters may also be used for providing data compression (for example of speech or video signals). Keeping the converter separate from the source enables different algorithms to be slotted in and out with relative ease.

Autonomous control operations may also be implemented as data converters. In some applications, critical control loops may become unstable with the communication delays between the remote and base station. In such situations, one may decide to implement a local control loop on the remote platform. The sensor data will be used to directly control the actuators, thereby creating an autonomous remote station.

## 2.3 Data Communication

Data communication between the remote and base station may be achieved by any available form of communication. In the prototype, we are using a TCP/IP network.

Network communication starts with the communication module registering itself as a sink on all of the data slots that contain data to be communicated between stations. Each communication data slot is supplied with a network ID used to identify the data to the receiver. When new data is available and the transmit task is triggered, it sends a data packet to the receiver. When data is received, the packet identification is used to determine which data slot the data is placed in at the

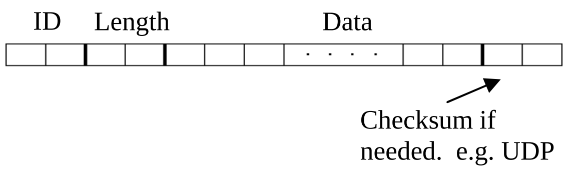


**Figure 4:** Packet structure for communication between base station and remote stations. Each box represents one byte.

destination. This activates dependent data sinks at the base station (such as virtual instruments). The same mechanism is used for control upstream, from the base station to the remote vehicle.

In the data driven approach, the data from each slot is communicated as a network packet, as shown in Figure 4. The structure of the packets has been defined so that the network ID is at the start of the packet. This is two bytes in length, allowing up to 65,536 separate data slots to be linked between the remote and base stations. The length field is next having two bytes. This indicates the length of the data field. Lastly there is the data field which can be up to 65,535 bytes in length. The data format is application defined. For this reason it is important that both remote and base stations have the same representation for integers (big endian / little endian), floating point, etc. Some communication channels, such as TCP, include error checking and correction, so will not need any checksums. However, some communication protocols, such as UDP, will need this. In such cases, a CRC-16 checksum will be added to the end of the packet.

Upon initial connection of the network, network control packets will pass between the remote and base station. The purpose of this is to check that the data slots at each end of the link are of the same type and have the same name and network identification. Any errors on either end are reported to the base station for the user.

## 3 Implementation Issues

### 3.1 Class Structure and use of DLLs

The software has been implemented as a collection of C++ classes with the hierarchy as shown in Figure 5. All classes are derived from the base class called Element. The hierarchical class structure simplifies the parsing of the configuration file. While parsing the configuration file, each attribute string is passed to a method of the current DEVICE or DATA being initialised. This is then passed to the parent in the class hierarchy and propagated up to the base Element class. If the attribute keyword is recognised, it is parsed; otherwise the parent class returns False. In this way, attributes common to groups of data slots or devices only need to be parsed in one place and device/data slot specific attributes need to be parsed by the called class.

The hierarchical class structure also allows extension by defining new operations without having to re-implement common operations. Methods of DataSlot perform data management while the methods of Processor perform task queuing.

In keeping with the flexible architecture, the system has been designed with minimal code in the exe file. Most of the program is contained in dynamically linked libraries (DLLs). Many of these DLLs contain plug-ins, providing the code necessary for a specific device. This allows for 'hot swapping' or dynamic reconfiguration. The benefits of this are that sensors may be added or removed without the need to shut down the program.

### 3.2 Multithreading and Task Queues

In certain parts of the program it is essential to have minimal interference between tasks and have quick throughput of data. This has been achieved by implementing multiple task queues. Each task queue has a certain priority set depending upon the type of

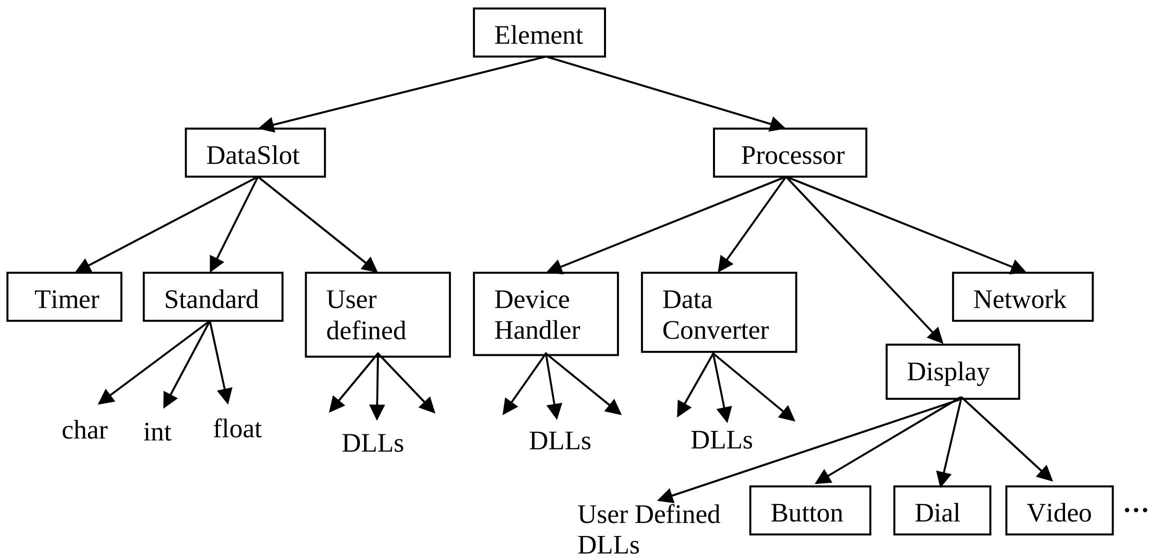


**Figure 5:** Class Hierarchy showing how the various entities of the framework relate to one another.

tasks handled by the queue. For example, setting the priority of the queue, which handles network communication tasks, higher than the queue which handles data converters will mean fewer peaks in the network usage and possibly minimal or no loss of data packets.

While running a single threaded program, a process may get stuck in a loop trying to access a device. In this situation a time-out will need to be generated. However, a lot of time may get wasted in just waiting for a sensor to respond or a time-out to be called. To overcome this problem and also to make the architecture flexible and error resilient, multithreading has been used. The implication of this is mutual exclusion is needed to make sure two threads do not access a device at the same time. Also if two threads attempt to access some common data, it must be ensured that a deadlock situation does not happen.

The performance of this source/sink system ultimately depends on the production and consumption of messages, which is related to the speed of the CPU and the number of sensors. So to prevent an excess of messages being produced, only one task is queued per item. For example if a GPS device was on a queue to indicate that data is available, if the same task wanted to be put on the queue again it would be blocked. This limits buffers to a certain size. So this system degrades gracefully, unlike an interrupt driven system, which could lose important information.

### 3.3 Communications

In a remote vehicle application, the operator at the base station views the situation of the remote vehicle through the video displayed. His or her inputs are accepted and fed through the network to the remote vehicle. Because of the way the system is set up, other people can be connected to the remote vehicle, after authentication. This would allow other users to view the information from the remote vehicle. While in some applications, distributed control may be an advantage, in controlling a remote vehicle it makes sense to have only one “driver”. This could be accomplished by disabling the control inputs on the other interfaces. The architecture easily allows a simple chat (or even voice) connection between users. This would enable a viewer to direct the driver to perform certain actions. For example the controller could be a person with skills of how to control a remote vehicle. The viewer could be a farmer wanting to check the far regions of his property.

If this architecture is used in a laboratory situation with static sensors, the network section does not have to be used. So after the processed data is collected, the information just could be displayed locally.

This architecture could be used in a wireless sensor network as well. Instead of going through a bus to communicate with the sensors, communication would use the network. Here you would request the data, or the smart sensor would send the information automatically. This would allow a higher level of interface instead of talking directly to the sensors in their protocol.

## 4 Summary

In this paper we have presented an architecture that integrates sensors, actuators, instrumentation and controller into one common framework. The framework is flexible, scalable and reconfigurable. Components are registered in the framework using a configuration file which defines data slots and devices. Changes can be made to the system by modifying the configuration file alone. The data manager will parse the configuration file and modify the system accordingly. Sensors and actuators can thus be added or removed seamlessly without having to rebuild the system.

Tasks are managed in separate queues with set priorities. A multi-threaded program architecture ensures task exclusivity, high data throughput and prevents system delays.

A hierarchical class structure ensures code reusability through inheritance. Newer components may be added by inheriting from established classes and adding required functionality at the derived class level.

The proposed framework will have wide ranging applications in the areas of home and office automation, process monitoring and control, surveillance and robotics.

## 5 Reference

[1] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, J. Anderson, “Wireless Sensor Networks for Habitat Monitoring”, Wireless Sensor Networks and Application, WSNA’02, Atlanta, USA, September 2002, pp. 88-97

[2] K. Sohrabi, W. Merrill, J. Elson, L. Girod, F. Newberg , W. Kaiser, “Methods for Scalable Self-Assembly of Ad Hoc Wireless Sensor Networks”, IEEE Transactions on Mobile Computing, Vol 3, No. 4, 2004, pp 317- 331

[3] D. Estrin, R. Govindan, J. Heidemann, S. Kumar, “Next Century Challenges: Scalable Coordination in Sensor Networks”, Mobicom ’99, Seattle, USA, 1999, pp. 263-270

[4] Shafer, S.A., Stentz, A., Thorpe, C.E., “An Architecture for Sensor Fusion in a Mobile Robot”, *Robotics and Automation*, pages 2002 – 2011, 1986

[5] National Marine Electronics Association, “NMEA 0183 Standard”, http://www.nmea.org/pub/0183/index.html, visited on 15/7/05