CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

A SPECIFICATION FOR ELECTRONIC DEVICE COMMUNICATION IN MARINE ENVIRONMENTS

A thesis submitted in partial fulfillment of the requirements for the
degree of Master of Science in
Computer Science

by

Anthony J Arnold

May 2011

The thesis of Anthony J Arnold is approved:

John Noga, Ph.D	Date
Robert McIlhenny, Ph.D	Date
Jeff Wiegley, Ph.D, Chair	Date

California State University, Northridge

Dedication

This thesis is dedicated to my amazing wife and my awe-inspiring son. Without their understanding and total support I would not have been able to complete this thesis work. They made all of the tireless nights and busy weekends worth the sacrifice.

Table of Contents

List of Tables

List of Figures

List of Listings

ABSTRACT

A SPECIFICATION FOR ELECTRONIC DEVICE COMMUNICATION IN MARINE ENVIRONMENTS

By

Anthony J Arnold

Master of Science in

Computer Science

The aim of this thesis project, "OpenMECS" (Open Marine Electronics Communication Standard), is to produce a specification defining the protocol governing communication between electronic devices in marine environments and to produce a prototype system to prove the efficacy of the protocol specification. The specification will define data formats and communication sequencing for exchanging data between marine electronic devices. The prototype system will prove the viability of the message formats and communication sequences with two example marine electronic devices communicating with a data display using the OpenMECS standard.

Chapter 1
Introduction

The objective of OpenMECS is to provide a free and easy to use protocol for sharing data between electronic devices in marine environments. The idea is that device and boat manufacturers, as well as hobbyists, can use and modify part or all of OpenMECS as an integral component of their systems.

One of the main goals of OpenMECS is to provide a low cost, yet high performance and complete alternative to NMEA-0183 [1] and NMEA-2000 [2]. To support this goal OpenMECS itself will be free. The OpenMECS specification is licensed as follows: Anyone may use or modify the contents of this specification at their will. The reference code implementation will be licensed under the BSD license.

OpenMECS will also support low cost development by being based as much as possible on already existing open standards and technologies. I have chosen Ethernet and TCP/IP to serve as the backbone of OpenMECS. Both are freely available standards that have existed for a long time. Inexpensive development is easy to achieve with Ethernet and TCP/IP. I also plan to utilize higher level protocols, such as DHCP, to achieve the goals of OpenMECS.

At the end of my thesis I want to achieve a complete specification that anyone can utilize without modification. I also want to demonstrate the viability of OpenMECS by implementing a prototype system using the OpenMECS protocol. I plan to develop the software for the prototype system with the goal of releasing it as a free and public OpenMECS library.

The primary focus of this thesis is to define messaging formats and protocols which will allow for easy and timely production of electronic devices which can communicate with each other in marine environments, primarily watercraft such as boats. These formats and protocols will be defined in a formal and complete manner which will allow producers of commercial quality marine electronics to focus on device quality and features without having to worry about interoperability with other manufacturers. This will provide a level playing field on which any company can compete. Because the specification will be open and based on Ethernet, development equipment is inexpensive and easy to obtain.

Hobbyists are an additional audience of OpenMECS. OpenMECS will allow any person who has interest in developing their own devices the means to do so with the knowledge that what they create will work with the other devices in their possession. The specification will be simple and understandable enough that the average hobbyist with basic programming and electronic skills will have a fast learning curve to implement the protocol.

1.1 Background

There are two well-known standards for marine environment electronics communication. The proprietary standards NMEA-0183 [1] and NMEA-2000 [2] define communication formats, protocols, and electrical specifications for electronic devices in marine environments. These standards are published by NMEA, the National Marine Electronics Association. At the time of this writing the cost for complete documentation of NMEA-0183 is $340, and complete documentation for NMEA-2000 is $4999. Because the price of these standards is prohibitive, very few specific or technical references to these standards will be included in my thesis. There are sources online which claim to provide some amount of reverse engineering of the NMEA standards. When needed these sources will be listed. They will be used sparingly as for the most part they are the work of hobbyists and not peer reviewed. NMEA-0183 is severely out-of-date and provides limited bandwidth and wiring options for deployment. NMEA-0183 is based on RS-422 with a very low baud rate of 4800 [3]. One good aspect of NMEA-0183 is that it uses the well known 8-bit serial standard, which is easy to develop for. Development hardware is inexpensive as any modern personal computer has the necessary hardware. RS-422 serial is also simple to work with, so software development costs are minimal.

NMEA-2000 is based on CAN [4] and provides for data speeds of up to 250kbit/sec. This speed is roughly 52 times the speed available with NMEA-0183. Although NMEA-2000 provides for higher bandwidth than NMEA-0183 it does not approach the 100Mbit/s or 1000Mbit/s available bandwidths for Ethernet. One major drawback of NMEA-2000 is the very high development costs associated with the CAN bus. Not only does the NMEA-2000 standard cost thousands of dollars, but so do hardware and software development tools. Hardware for a single commercial development station allowing communication on one CAN bus can cost well into the $20,000 range. The structure of CAN data is also limited. With NMEA-0183 the data can be arbitrary in length. A single CAN message is limited to a maximum of 8 bytes of data.

I will show that although Ethernet based communication is more complicated than RS-422 or CAN, it will still be easy to implement as evidenced by my prototype system. Ethernet is also very common, and hardware and software development tools are available very inexpensively or free. Most PCs sold today have Ethernet jacks that support 10/100/1000 Mbit/s speeds. There are a myriad of free bus monitoring tools for Ethernet based networks. One such tool that I employed during the development of my prototype systems is Wireshark [10].

One aspect of marine communication that will not be addressed in this thesis is the electrical specifications of Ethernet. Typically there are special grounding and other various

electrical requirements that are considered for electronics in potentially wet environments. This thesis will leave these details to be defined at a later date.

Chapter 2
Devices

The first step was to identify the types of devices which are intended to be supported by OpenMECS. Although they do vary, they all share the common purpose of being used in a marine environment. OpenMECS was not designed to support entertainment devices (at least not this version). It is possible that support for entertainment types of devices will be added to OpenMECS as a follow up effort to the thesis.

The first task was to list as many marine electronic devices as I could think of and then to group them logically by function. The following list is not exhaustive of all marine devices.

- GPS receiver
- Compass
- Anemometer
- Knotmeter
- Thermometer
- Wind Vane
- Sonar
- Camera
- Engine Monitor
- Radar
- Barometer
- Humidity Meter
- Powerplant

I immediately discarded the extremely high bandwidth devices such as camera, sonar and radar. They were also excluded from being considered as I have no domain knowledge of their data interchange formats and protocols.

Next I researched the NMEA-0183 and NMEA-2000 protocols to figure out which kinds of devices they supported. I was unable to find any information about the supported devices of NMEA-2000. I decided to assume NMEA-2000 would at least support a super-set of the devices supported by NMEA-0183. In summary, most of the information available for NMEA-0183 points only towards its support of Navigational devices, GPS in particular. Glenn Baddeley's website [7] was particularly useful. It lists a cornucopia of different types of data that GPS devices can transmit. The vast majority of the information was well beyond my comprehension of navigation domain. I decided to support only a few basic types of information formats based on GPS navigation: Latitude/Longitude, Bearing, Altitude and Velocity.

This website [6] also added some useful data formats such as Rudder Angle and Rate of Turn. Neither website is official or credentialed so I used their data only as informational and not as authoritative resources.

Chapter 3
OpenMECS Protocol

This document identifies OpenMECS protocol version **1**. Defining a version of the protocol allows forward compatibility with future versions of the protocol. The protocol is self identifying as shown in section 3.4.2. Being self identifying allows the nodes on the network to know which version of the protocol they are receiving. Nodes that implement future versions of the protocol can also implement older versions to allow nodes constructed to use an older version of the protocol to be used as long as possible.

The standard TCP/IP network model has four layers: The Link layer, the Internet layer, the Transport layer, and the Application layer. The Link layer is standard Ethernet protocol with no modifications. Most modern operating systems include support for Ethernet natively. Libraries for embedding Ethernet communication into small devices are also available freely. The MPLab Suite that came with the PIC32 Starter Kits came bundled with a complete TCP/IP stack which included Ethernet. The Internet layer will be standard IP version 4 with no modifications. IP Addressing is covered in section 3.3. The Transport layer can either be UDP or TCP. The Transport layer is covered in section 3.2. The OpenMECS protocol itself entirely defines the Application Layer, as shown in figure 3.1

3.1 Physical bus

To increase openness and portability of nodes between applications, the OpenMECS standard assumes a default physical layer of Copper Category X cable. This also assumes that each node will connect to the cable by using an RJ-45 jack. Users of the OpenMECS protocol are free to implement any physical layer they please, however, interoperability will be harder to ensure between nodes and applications.

3.1.1 Topology

One of the drawbacks of both NMEA-0183 and NMEA-2000 is topology. Topology is the physical layout of the network that connects the devices which speak each of those protocols. NMEA-0183 is a simple point-to-point bus. Each device can only talk with one other device. This means that a Master node that speaks NMEA-0183 must have a physical connection for every device it gets data from. This increases the complexity of the master device. It also increases the amount of wiring to connect multiple NMEA-0183 devices together.

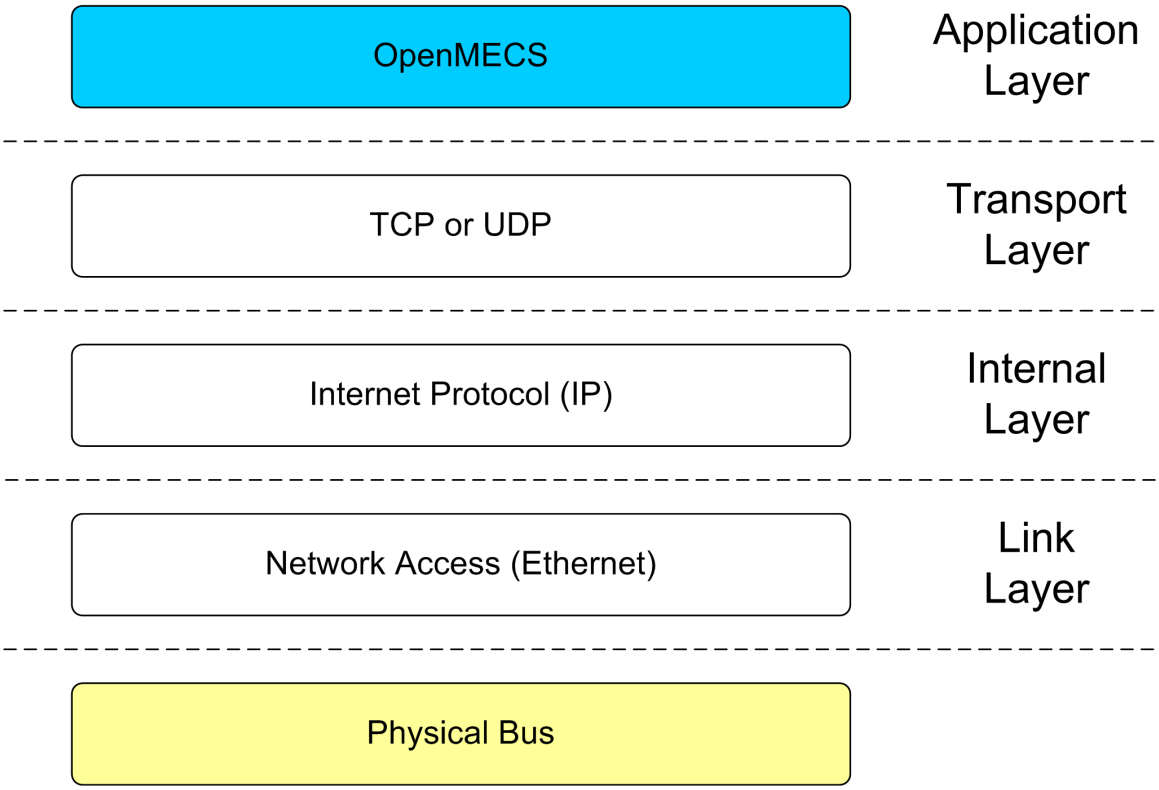

Figure 3.1: TCP/IP Model

NMEA-2000 is a networked bus. All devices on the network share the exact same two wires. This allows the wiring to be somewhat simpler than a NMEA-0183 “network.” Although each device shares the same two wires, the NMEA-2000 standard allows for stubs that splice into the main set of wires. The stubs are limited in length, reducing their effectiveness for large physical networks.

The OpenMECS protocol network, like all application layers that depend on TCP/IP, will have a ”star” topology. This means that every node on the network will have a wired connection to a centralized point. There are numerous variations on what the centralized point may be. The centralized point may be the Master node of the network. This will probably not be the case as it would mean that the Master node would need to have many cable jacks. The most likely scenario would be for the central point in the network to be a hub or switch. With this configuration each node would have a cable running from itself to the switch. This would include the Master node. It is also possible with the ”star” topology to have multiple switches connected to each other. This would reduce the number of cables required to connect separate parts of the ship that are far from each other. Care must be taken with this approach as every switch added between an End node and the Master node

adds latency to the communication between the nodes. This may be an important factor depending on the data being transmitted. Figure 3.2 depicts an example of the topology of an OpenMECS network deployed on a ship:

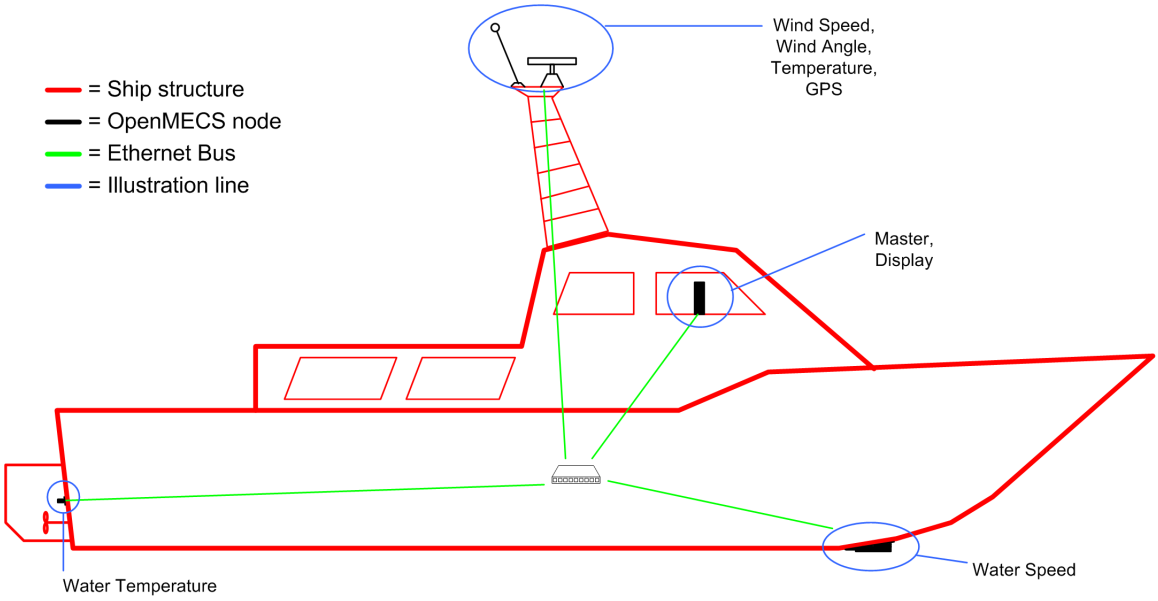

Figure 3.2: "Star" Topology

3.1.2 Bandwidth

Assuming Copper Category X cable or higher from section 3.1 is used to connect the OpenMECS network, the minimum supported speed for an OpenMECS network is 10Mb/S. Manufacturers of OpenMECS compliant devices should strive for backwards compatibility down to this speed. It is recommended that device manufactures support a maximum bandwidth of at least 1000Mb/S (1000BASE-T). Each application of an OpenMECS network will need a bandwidth analysis performed to ensure that the capacity of the OpenMECS network is not exceeded. In general, a spare bandwidth capacity of 30% is recommended for all applications. Thirty percent spare is taken from my experience in the Commercial Aerospace industry.

3.2 Transport Layer

The OpenMECS messages define the Application Layer of the TCP/IP model. Either UDP or TCP can be used as the Transport Layer protocol. The following details are defined for the use of UDP:

- Port: 57460 for initiated mode communication
- Port: 57461 for automatic mode
- Port: 57465 for broadcast data

The following details are defined for the OpenMECS use of TCP:

- Port: 57462 for all modes
- Push Flag: Always set to 1 (ensures timely transmission of TCP packets)

3.3 Addressing

This version of OpenMECS is designed for IP version 4. Future versions may support IP version 6. Each node on an OpenMECS network needs an IPv4 address. DHCP is used as the address assignment protocol. The following DHCP details are defined for the OpenMECS use of DHCP:

- This version of the protocol requires that all OpenMECS nodes must support DHCP, and receive their IP address through DHCP
- All OpenMECS End nodes must support the following standard DHCP Options:
  - 1: Subnet Mask
  - 53 : DHCP Message Type
  - 51: IP Address Lease Time (End nodes may assume a minimum of 1 hour)
- Although the DHCP server may offer additional options than those listed above, no OpenMECS node will rely on any option not listed.
- The DHCP server does not need to be the Master node.

The End nodes will decode the "Gateway" data field within the DHCP Acknowledgement message. They will assume that the Gateway address is either the Master node, or can forward their data to the Master node.

3.4 Application Layer

3.4.1 Communication Sequencing

Once any End node starts up it will immediately start the DHCP process to receive its IP address. Once an End node has its IP address the first message it transmits will be the Device Identification message to the Master node. This effectively "registers" the End node with the Master node. This has two uses. It allows the Master node to request data from the End node if it did not know it would be on the network. It also allows the Master node to check a preconfigured data table of expected End nodes to ensure the health of the entire system. Once the initial Device Identification message has been transmitted the End node is free to send and receive messages in either initiated or automatic mode. The exception being the Device Identification message. Each End node must resend this message to the Master node at least once every ten seconds. This provides a "heartbeat" signal that allows the Master node to check on network health. It also allows the network to recover gracefully from power interruptions.

There are two modes of communication, initiated and automatic. For most data the intended communication mode is automatic. In this mode the End nodes will start transmitting their Data Sets automatically to the Master node as soon as they are ready. It is up to the manufacturer of the End node to pick an appropriate transmission rate. This mode allows for less overhead on the network because it cuts out a request message for every message containing data. This reduction in overhead will be important for data that is going to be transmitted very frequently. Although this mode allows for more efficient use of the bus, care must be taken to ensure that the available bandwidth of the bus is not exceeded. Whomever is selecting nodes for installation in an OpenMECS network should perform an analysis of bus bandwidth based on the stated data rates of the node manufacturers. Figure 3.3 depicts an example automatic mode communication sequence. In the sequence shown the End node receives its IP address using DHCP. Then it transmits the Device Identification message to the Master node as its first message. After the Device Identification message, the End node transmits the Bearing message to the Master node at a rate of .5 seconds, repeating the Device Identification every 10 seconds or less.

Initiated mode works in a traditional command-response format. The master node may request data from any of the End nodes. When an End node receives a request from the master node, the dataRequest flag will be set to true in the OpenMECS message. The message will also contain the list of Data Sets the Master node is requesting from the End node. The End node may transmit a response with any number of the requested Data Sets back to the Master node. The response from the End node will set the dataRequest flag to false.

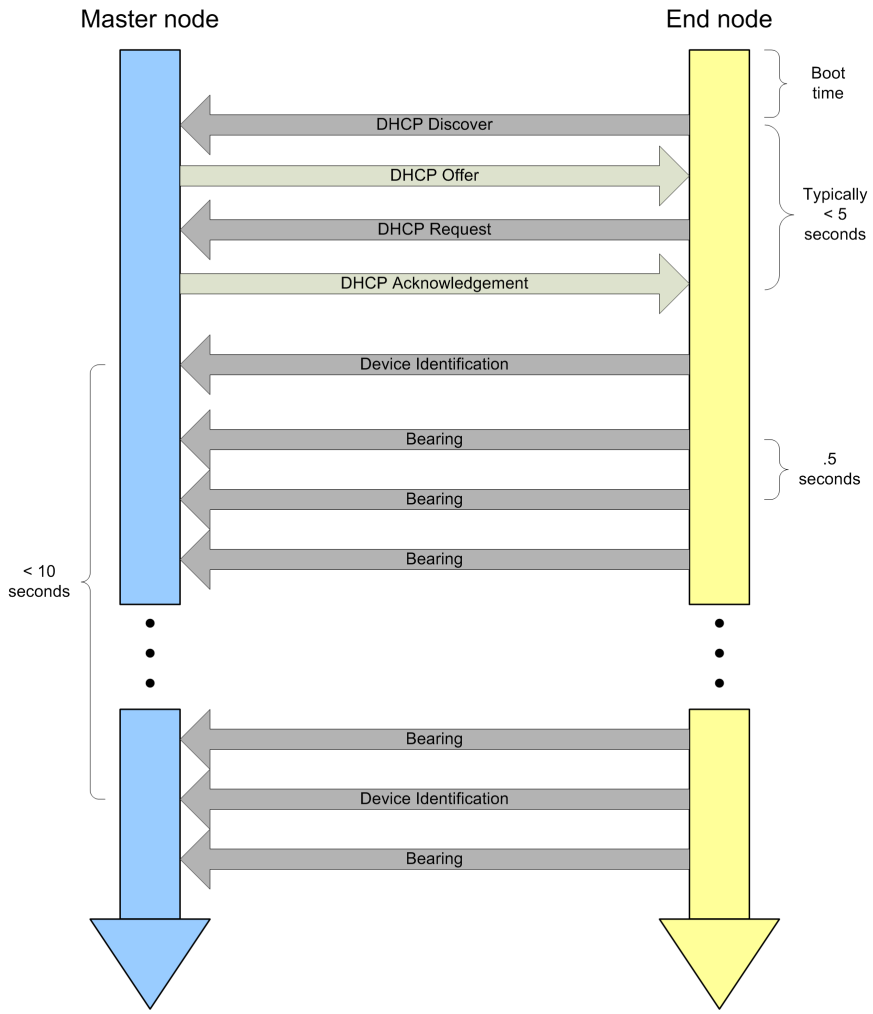

Figure 3.3: Automatic Mode Communication Sequence

This mode of communication is mostly useful for data that is not needed continuously or that is very infrequently needed. Figure 3.4 depicts an example initiated mode communication sequence. The End node in this example transmits the Air Temperature message upon request from the Master node. The Master node is requesting the Air Temperature from the End node every 5 seconds.

3.4.2 Application Layer Data Format

JSON (JavaScript Object Notation) is used as the underlying generic data exchange format for OpenMECS messages. JSON is a textual data format that allows for numerical and textual data to be represented in a structured and hierarchical format. The structure and hierarchy for the OpenMECS usage of JSON is defined in the following sections. Each

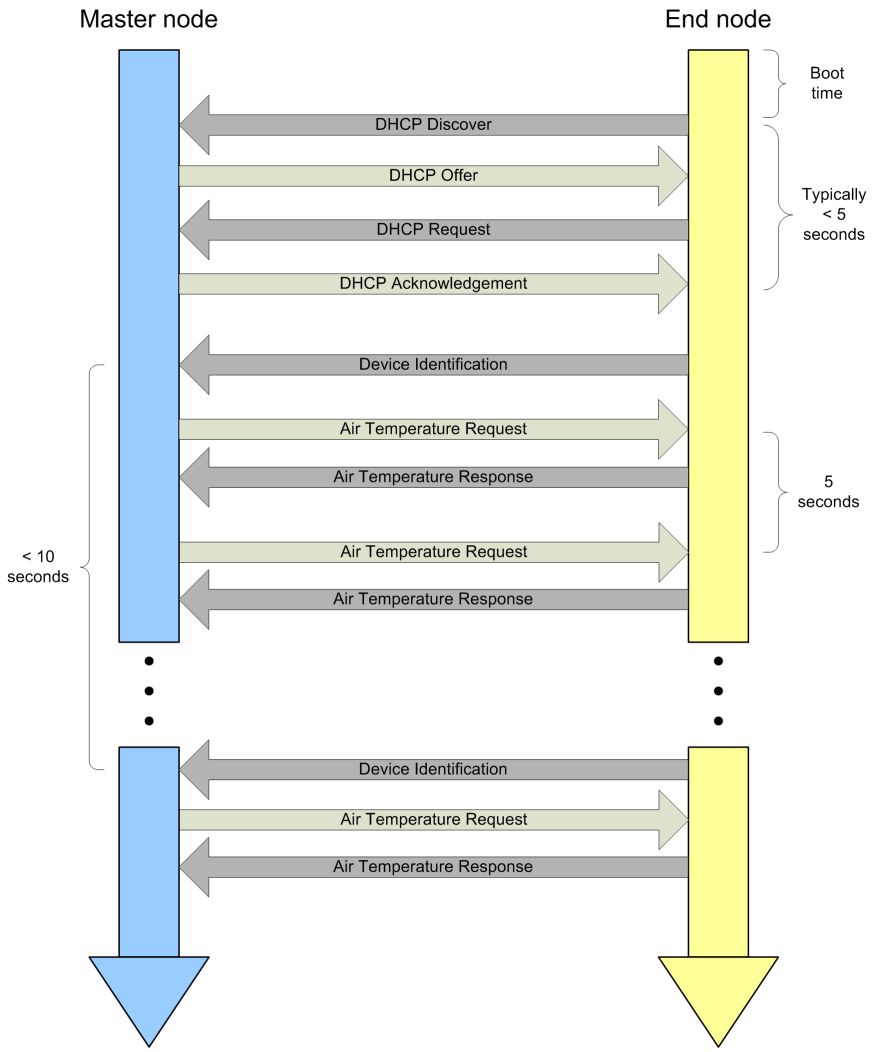

Figure 3.4: Initiated Mode Communication Sequence

message will have the following general format at the top most level:

{

“content”:”OpenMECS”,

“protocolVersion”:P,

“dataRequest”:true—false,

“dataSets”:

[Data Set 1, Data Set 2, Data Set N],

“numDataSets”:N,

“validity”: V

}

Whitespace characters (space, tab, carriage return, line feed, etc) have no meaning and

will always be ignored. However they will be used during the calculation of the validity value of the message. The order of the key:value pairs within the JSON object does not matter.

The following example, less the quotes, depicts a complete OpenMECS message string as it might be transmitted on the network. This sample is for version 1 of the protocol, and consists of two Data Sets, namely Bearing and Altitude. The Validity value is shown as a variable as it was not calculated for this example. As stated above the whitespace is ignored. Any whitespace shown in the example below is for readability only.

```
``{content:OpenMECS,
   protocolVersion:1,
   dataRequest:false,
   dataSets:
        [{dataType:12,dataContents:
              [23:15:40:123,12.34,A,13.45,1.11,E,A]},
         {dataType:11,dataContents:
              [23:15:40:189,234,171,F,E]}],
   numDataSets:2,
   validity:VVVVVVVV}''
```

3.4.3 Field definitions

3.4.3.1 “content”:”OpenMECS”

This field defines the message as belonging to the OpenMECS protocol. It is the first field that will be checked for message validity in nodes adhering to the OpenMECS protocol.

3.4.3.2 “protocolVersion”:P

This field identifies which version of the protocol the particular instance of an OpenMECS message was designed for. This allows nodes using OpenMECS to ensure that correct data format and syntax is used. Evolution of the protocol will mean that the version number will be incremented. Nodes using OpenMECS may support multiple protocol versions to enhance compatibility. P is a string that represents the version of the protocol.

3.4.3.3 “dataRequest”:true—false

This field denotes whether the OpenMECS transmission is a request for data, or contains data. The Master node within an OpenMECS network will transmit messages with this

field set to true. The End nodes on the network will transmit messages with this field set to false.

3.4.3.4 "dataSets":[Data Set 1, Data Set 2, Data Set N]

This field contains the functional data (or requests) used by the Master node. This field can consist of more than one piece of data. Each piece of data is referred to as a Data Set. Each Data Set will have the following format at its top most level:

{
"dataType": DT,
"dataContents":[Data Content]
}

3.4.3.4.1 Data Set field definitions

3.4.3.4.1.1 "dataType": DT

This field indicates the type of data within the Data Set. Each Data Set contains only one type and one instance of data. For example a Data Set may contain one Latitude/Longitude only. It would not also contain a heading or a second Latitude/Longitude. DT is defined below for each Data Set.

3.4.3.4.1.2 dataContents: [Data Content]

If "dataRequest" is set to true, this array will be null. If "dataRequest" is false, this field will contain the actual data from the End node. The detailed definitions of the dataContents field for each Data Set is defined in section 3.5

3.4.3.5 "numDataSets":N

This field indicates the number of Data Sets that are included in the OpenMECS message. Used by the node to ensure all data was received. If not all Data Sets were received, the entire message will be ignored by the node.

N is an integer.

3.4.3.6 "validity": V

This field contains the validity check value for the message. The validity check value is an IEEE 802.3e 32-bit CRC calculated over the entire message text, excluding the following:

Data Type Definitions		
Category	Data Type(DT)	Definition
Utility	1	Device Identification
	2	System Time
Navigation	10	Latitude/Longitude
	11	Altitude
	12	Bearing
	13	Velocity
	14	Rate of Turn
Weather	30	Air Temperature
	31	Water Temperature
	32	Wind Speed
	33	Wind Angle
Mechanical	50	Fuel Level
	51	Battery Level
	52	Engine Revolutions
	53	Rudder Angle
	54	Power Plant Operational Status
Reserved	0	
	3 - 9	
	15 - 29	
	34 - 49	

Table 3.1: Available Data Sets

- The opening brace
- The “,” character at the end of numDataSets
- The “validity”: V field
- The closing brace

3.5 Data Sets

Each Data Set defines one atomic piece of data. Each Data Set has a uniquely identifying attribute called “Data Type.” The Data Type will be used by the Master node to decode the received Data Sets. Table 3.1 summarizes the Data Sets that have been defined.

3.5.1 Utility

3.5.1.1 Device Identification

This Data Set is not optional. All OpenMECS compliant nodes must support this message.

3.5.1.1.1 Data Type

DT = “1”

3.5.1.1.2 Data Contents Format

[
IPAddress ,
MACAddress ,
Make,
Model,
SerialNum,
PartNum,
NumDeviceCapabilities,
[
DataType1,
,
DataTypeN
]
]

3.5.1.1.3 Field definitions

3.5.1.1.3.1 IPAddress

The IP version 4 address of the node transmitting this message. String in a dotted decimal formation. Example: “192.69.0.65”.

3.5.1.1.3.2 MACAddress

The MAC address of the node transmitting this message. String in a colon separated hex formation. Example: “6C:F0:48:01:B3:82”.

3.5.1.1.3.3 Make

The Make of the node transmitting this message. String of maximum length of 255 characters.

3.5.1.1.3.4 Model

The Model of the node transmitting this message. String of maximum length of 255 characters.

3.5.1.1.3.5 SerialNum

The Serial number of the node transmitting this message. String of maximum length of 255 characters.

3.5.1.1.3.6 PartNum

The Part number of the node transmitting this message. String of maximum length of 255 characters.

3.5.1.1.3.7 NumDeviceCapabilities

The number of distinct Data Set Data Types that this node is capable of providing. Must match the number of items in the array which follows this field.

3.5.1.1.3.8 [DataType1, , DataTypeN]

This is an array of Data Types that this node is capable of providing. For a device supporting the Data Set Types of Bearing(12) and Velocity(13) this field would be: [12,13].

3.5.1.2 System Time

All OpenMECS compliant nodes that report any Data Sets with a time stamp must support reception of this message. This message is intended to be transmitted by the Master node. The End nodes should receive this message and synchronize their internal clocks accordingly. This message was chosen to be part of the protocol instead of relying on a service such as NTP, because this message is simple. There should be little network overhead for this message. It requires that the Master node have a built-in, accurate time mechanism, or receive its time from a GPS node.

3.5.1.2.1 Data Type

DT = “2”

3.5.1.2.2 Data Contents Format

[
Date,
Time,
“A” | “N” | “S”
]

3.5.1.2.3 Field definitions

3.5.1.2.3.1 Date

This field indicates the current date. It has the following string format:
“YYYY-MM-DD”
Where:
YYYY = years
MM = months
DD = days

3.5.1.2.3.2 Time

This field indicates the current time. It has the following string format:
“HH:MM:SS:NNN”
Where:
HH = Hours

MM = minutes

SS = seconds

NNN = milliseconds

3.5.1.2.3.3 “A” | “N” | “S”

This field indicates the quality of the Time measurement.

“A” = Active

“N” = Not Valid

“S” = Simulated

Only a quality of “A” should be trusted for normal operation.

3.5.2 Navigation

All of the Data Sets below are optional. Each node manufacturer is free to implement as few or as many Data Set capabilities as they please.

3.5.2.1 Latitude/Longitude

3.5.2.1.1 Data Type

DT = “10”

3.5.2.1.2 Data Contents Format

[
Time,
Latitude,
“N”| “S”,
Longitude,
“E” | “W”,
“A” | “D” | “E” | “N” | “S”
]

3.5.2.1.3 Field definitions

3.5.2.1.3.1 Time

This field indicates what time the Latitude and Longitude measurements were taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.2.1.3.2 Latitude

This field contains the Latitude of the End node. It has the following string format: “1212.123”. This example equates to 12 degrees 34.123 minutes.

3.5.2.1.3.3 “N” | “S”

This field denotes whether the Latitude is North “N” or South “S” of the equator.

3.5.2.1.3.4 Longitude

This field contains the Longitude of the End node. It has the following string format: “12012.123”. This example equates to 12 degrees 12.123 minutes.

3.5.2.1.3.5 “E” | “W”

This field denotes whether the Longitude is East “E” or West “W” of the Prime Meridian (0 degrees).

3.5.2.1.3.6 “A” | “D” | “E” | “N” | “S”

This field indicates the quality of the Latitude and Longitude measurements.

“A” = Autonomous

“D” = Differential

“E” = Estimated (Dead Reckoning)

“N” = Not Valid

“S” = Simulated

Only a quality of “A” or “D” should be trusted for normal operation.

3.5.2.2 Altitude

3.5.2.2.1 Data Type

DT = “11”

3.5.2.2.2 Data Contents Format

[
Time,
Altitude,
HeightOfGeoid,
“M” | “F”,
“A” | “E” | “N” | “S”
]

3.5.2.2.3 Field definitions

3.5.2.2.3.1 Time

This field indicates what time the Altitude measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.2.2.3.2 Altitude

This field contains the Altitude of the End node. It has the following numerical format: 01234.12. This example equates to 1,234.12 units.

3.5.2.2.3.3 HeightOfGeoid

This field contains the height of the Geoid (mean sea level) above WGS84 ellipsoid. It has the following numerical format: 01234.12.

3.5.2.2.3.4 “M” | “F”

This field indicates the units used to measure altitude.
“M” = Meters
“F” = Feet

3.5.2.2.3.5 “A” | “E” | “N” | “S”

This field indicates the quality of the Altitude measurement.

“A” = Active

“E” = Estimated (Dead Reckoning)

“N” = Not Valid

“S” = Simulated

Only a quality of “A” or “E” should be trusted for normal operation.

3.5.2.3 Bearing

3.5.2.3.1 Data Type

DT = “12”

3.5.2.3.2 Data Contents Format

[
Time,
BearingTrue,
“A” | “N” | “S”,
BearingMagnetic,
MagneticVariation,
“E” | “W”,
“A” | “N” | “S”
]

3.5.2.3.3 Field definitions

3.5.2.3.3.1 Time

This field indicates what time the Heading measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.2.3.3.2 BearingTrue

This field contains the True Bearing of the End node in degrees. It has the following numerical format: 012.34.

3.5.2.3.3.3 “A” | “N” | “S”

The first A|N|S field indicates the quality of the True Bearing measurement.
“A” = Active
“N” = Not Valid
“S” = Simulated
Only a quality of “A” should be trusted for normal operation.

3.5.2.3.3.4 BearingMagnetic

This field contains the Magnetic Bearing of the End node in degrees. It has the following numerical format: 123.45.

3.5.2.3.3.5 MagneticVariation

This field (also known as declination) contains the Magnetic Variation of the End node in degrees. It is the difference between Magnetic and True bearing for the current position of the End node. It has the following numerical format: 123.45.

3.5.2.3.3.6 "E" | "W"

This field indicates the direction of Magnetic Variation of the End node in degrees.

"E" = East

"W" = West

3.5.2.3.3.7 "A" | "N" | "S"

The second A|N|S field indicates the quality of the Magnetic Bearing measurement.

"A" = Active

"N" = Not Valid

"S" = Simulated

Only a quality of "A" should be trusted for normal operation.

3.5.2.4 Velocity

3.5.2.4.1 Data Type

DT = “13”

3.5.2.4.2 Data Contents Format

[
Time,
Velocity,
A | E | N | S,
]

3.5.2.4.3 Field definitions

3.5.2.4.3.1 Time

This field indicates what time the Velocity measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.2.4.3.2 Velocity

This field contains the Velocity of the End node in knots. It has the following numerical format: 012.34.

3.5.2.4.3.3 “A” | “E” | “N” | “S”

This field indicates the quality of the Velocity measurement.
“A” = Active
“E” = Estimated (Dead Reckoning)
“N” = Not Valid
“S” = Simulated
Only a quality of “A” should be trusted for normal operation.

3.5.2.5 Rate of Turn

3.5.2.5.1 Data Type

DT = “14”

3.5.2.5.2 Data Contents Format

[
RateOfTurn,
A | N | S,
]

3.5.2.5.3 Field definitions

3.5.2.5.3.1 RateOfTurn

This field indicates at what rate the vessel is turning. It is measured in degrees per minute. This field is a number. If positive it indicates a turn towards starboard (right), negative indicating a turn towards port (left).

3.5.2.5.3.2 “A” | “E” | “N” | “S”

This field indicates the quality of the Velocity measurement.
“A” = Active
“N” = Not Valid
“S” = Simulated
Only a quality of “A” should be trusted for normal operation.

3.5.3 Weather

3.5.3.1 Air Temperature

3.5.3.1.1 Data Type

DT = “30”

3.5.3.1.2 Data Contents Format

[Time,
Temperature,
“C” | “F”,
“A” | “N” | “S”
]

3.5.3.1.3 Field definitions

3.5.3.1.3.1 Time

This field indicates what time the Air Temperature measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.3.1.3.2 Temperature

This field contains the Temperature of the End node in degrees. It has the following numerical format: 012.34.

3.5.3.1.3.3 “C” | “F”

This field indicates the measurement units for Temperature.
“C” = Celsius
“F” = Fahrenheit

3.5.3.1.3.4 “A” | “N” | “S”

This field indicates the quality of the Air Temperature measurement.
“A” = Active
“N” = Not Valid

“S” = Simulated

Only a quality of “A” should be trusted for normal operation.

3.5.3.2 Water Temperature

3.5.3.2.1 Data Type

DT = “31”

3.5.3.2.2 Data Contents Format

[
Time,
Temperature,
“C” | “F”,
“A” | “N” | “S”
]

3.5.3.2.3 Field definitions

3.5.3.2.3.1 Time

This field indicates what time the Temperature measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.3.2.3.2 Temperature

This field contains the Temperature of the End node in degrees. It has the following numerical format: 012.34.

3.5.3.2.3.3 “C” | “F”

This field indicates the measurement units for Temperature.
“C” = Celsius
“F” = Fahrenheit

3.5.3.2.3.4 “A” | “N” | “S”

This field indicates the quality of the Water Temperature measurement.
“A” = Active
“N” = Not Valid
“S” = Simulated

Only a quality of “A” should be trusted for normal operation.

3.5.3.3 Wind Speed

3.5.3.3.1 Data Type

DT = “32”

3.5.3.3.2 Data Contents Format

[
Time,
WindSpeed,
“K” | “N” | “M”,
“A” | “N” | “S”
]

3.5.3.3.3 Field definitions

3.5.3.3.3.1 Time

This field indicates what time the Wind Speed measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.3.3.3.2 WindSpeed

This field indicates the speed of the Wind. It has the following numerical format: 012.34.

3.5.3.3.3.3 “K” | “N” | “M”

This field indicates the measurement units for Wind Speed.
“K” = Knots per hour
“N” = Nautical miles per hour
“M” = Kilometers per hour

3.5.3.3.3.4 “A” | “N” | “S”

This field indicates the quality of the Wind Speed measurement.
“A” = Active
“N” = Not Valid
“S” = Simulated

Only a quality of “A” should be trusted for normal operation.

3.5.3.4 Wind Angle

3.5.3.4.1 Data Type

DT = “33”

3.5.3.4.2 Data Contents Format

[
Time,
WindAngle,
“R” | “T”,
“A” | “N” | “S”
]

3.5.3.4.3 Field definitions

3.5.3.4.3.1 Time

This field indicates what time the Heading measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.3.4.3.2 WindAngle

This field indicates the angle of the Wind in degrees. It has the following numerical format: 012.34.

3.5.3.4.3.3 “R” | “T”

This field indicates the measurement reference for Wind Angle.
“R” = Relative
“T” = True

3.5.3.4.3.4 “A” | “N” | “S”

This field indicates the quality of the Wind Angle measurement.
“A” = Active
“N” = Not Valid
“S” = Simulated

Only a quality of “A” should be trusted for normal operation.

3.5.4 Mechanical

3.5.4.1 Fuel Level

3.5.4.1.1 Data Type

DT = “50”

3.5.4.1.2 Data Contents Format

[
Time,
FuelLevel,
“A” | “N” | “S”
]

3.5.4.1.3 Field definitions

3.5.4.1.3.1 Time

This field indicates what time the Fuel Level measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.4.1.3.2 FuelLevel

This field indicates the percentage of fuel remaining. It has the following numerical format: 100.00. It has a range of 0.00 to 100.00

3.5.4.1.3.3 “A” | “N” | “S”

This field indicates the quality of the Fuel Level measurement.
“A” = Active
“N” = Not Valid
“S” = Simulated
Only a quality of “A” should be trusted for normal operation.

3.5.4.2 Battery Level

3.5.4.2.1 Data Type

DT = “51”

3.5.4.2.2 Data Contents Format

[
Time,
BatteryLevel,
“A” | “N” | “S”
]

3.5.4.2.3 Field definitions

3.5.4.2.3.1 Time

This field indicates what time the Battery Level measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.4.2.3.2 BatteryLevel

This field indicates the percentage of battery remaining. It has the following numerical format: 100.00. It has a range of 0.00 to 100.00

3.5.4.2.3.3 “A” | “N” | “S”

This field indicates the quality of the Battery Level measurement.
“A” = Active
“N” = Not Valid
“S” = Simulated
Only a quality of “A” should be trusted for normal operation.

3.5.4.3 Engine Revolutions

3.5.4.3.1 Data Type

DT = “52”

3.5.4.3.2 Data Contents Format

[
Time,
EngineRevs,
“EngineID”,
“A” | “N” | “S”
]

3.5.4.3.3 Field definitions

3.5.4.3.3.1 Time

This field indicates what time the Engine Revolutions measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.4.3.3.2 EngineRevs

This field indicates the number of revolutions at which the engine is rotating. It has the following numerical format: 01234.

3.5.4.3.3.3 “EngineID”

This field identifies the engine to which this message applies. It is a string of up to 64 characters.

3.5.4.3.3.4 “A” | “N” | “S”

This field indicates the quality of the Engine Revolutions measurement.
“A” = Active
“N” = Not Valid
“S” = Simulated
Only a quality of “A” should be trusted for normal operation.

3.5.4.4 Rudder Angle

3.5.4.4.1 Data Type

DT = “53”

3.5.4.4.2 Data Contents Format

[
Time,
RudderAngle,
“A” | “N” | “S”
]

3.5.4.4.3 Field definitions

3.5.4.4.3.1 Time

This field indicates what time the Rudder Angle measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: “23:15:40:123”. This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.4.4.3.2 RudderAngle

This field indicates the angle of the Rudder in degrees, in relation to the craft. It has the following numerical format: 012.30. It has a range of 0.00 to 359.99

3.5.4.4.3.3 “A” | “N” | “S”

This field indicates the quality of the Rudder Angle measurement.
“A” = Active
“N” = Not Valid
“S” = Simulated
Only a quality of “A” should be trusted for normal operation.

3.5.4.5 Power plant Operational Status

This message defines a very generic notion of a power plant. It is designed to be able to identify any number of power plant types such as gasoline engines, diesel engines, electric Azimuth thrusters. As well as locomotive power sources this message also supports power plants that might be used purely to generate electricity or heat. Generality is achieved by having the "PowerPlantID" field, which is a string. Each power plant manufacturer can identify their product as they please.

3.5.4.5.1 Data Type

DT = "54"

3.5.4.5.2 Data Contents Format

[
Time,
"PowerPlantID",
"R"| "N" | "I",
"A" | "N" | "S"
]

3.5.4.5.3 Field definitions

3.5.4.5.3.1 Time

This field indicates what time the Power Plant Operational Status measurement was taken. This data enables the Master node to ensure that the End node is operating on current data. This field has the following string format: "23:15:40:123". This example equates to 23 hours, 15 minutes, 40 seconds, and 123 milliseconds UTC.

3.5.4.5.3.2 "PowerPlantID"

This field identifies the power plant to which this message applies. It is a string of up to 64 characters.

3.5.4.5.3.3 "R" | "N" | "I"

This field indicates the operational status of the power plant.
"R" = Running

“N” = Not Running

“I” = Inoperable, indicates damage or maintenance

Only a quality of “A” should be trusted for normal operation.

3.5.4.5.3.4 “A” | “N” | “S”

This field indicates the quality of the Power Plant Operational Status measurement.

“A” = Active

“N” = Not Valid

“S” = Simulated

Only a quality of “A” should be trusted for normal operation.

3.5.5 Undefined Data Sets

The following Data Sets are listed here as potential Data Sets to implement in the future. I do not have enough domain knowledge and/or could not find enough information on the NMEA-0183 equivalent to provide a detailed definition of these Data Sets.

- Waypoint Alarm
- Position Accuracy
- GPS satellites in view
- Humidity
- Precipitation condition
- Precipitation chance
- Sunny/Cloudy
- Swell

Chapter 4
Project Demonstration

One of the main goals of this project is to show that the OpenMECS protocol is feasible. To support this goal, two real-world demonstration examples were created. Both were created using PIC32 Ethernet Starter Kits from Microchip®. The Starter Kits contained a PIC32 starter board which has a PIC32 microcontroller, on-board USB debugging, an RJ-45 jack, and some switches and LED. The USB cable is used for debugging, Flash programming and powering the PIC32 Starter kit. The kits also contain the free IDE MPLab, which includes the compiler, debugger, and libraries for getting started writing code that takes advantage of the included TCP/IP stack. The MPLab IDE supports the C programming language. The Microchip® PIC32 I/O expansion board was added to the starter boards. The combination of the PIC32 Ethernet starter board and expansion board allowed me to create OpenMECS applications that read sensors and reported the data back to the Master node. The two demonstration nodes and their applications are detailed below. I chose to implement a fuel level sensor and an anemometer for the demonstration. The entire demonstration setup is shown in Figure 4.1.

Refer to the schematic in Figure 4.1 while reading sections 4.2 and 4.3

4.1 PIC32 Source Code Project

I started with the Microchip® example project "Ethernet - TCPIP-BSD - HTTP Server Demo" which ships with the installation CD that comes with the PIC32 Starter kit. This example project contains source code which initializes the hardware including the PIC32 microcontroller and the Ethernet chip. It also initializes the TCP/IP stack. It uses a basic system scheduler from the Microchip® "system_services.c" API which has a configurable "tick." The tick is initialized based on a desired value and is achieved using an interrupt available on the PIC32 microcontroller Each tick of the scheduler calls the TCP/IP stack function. Each time the TCP/IP stack function is called data packets are either received or transmitted from/to the TCP/IP packet buffers in RAM. The example project also contains a "while" loop which executes forever as fast as possible. Essentially the system "tick" can be thought of as the "foreground" task, while the "while" loop can be thought of as the "background" task.

I developed one project that contained the source code for both the Fuel Level Sensor and the Anemometer. My modifications to the existing example project were all performed in the file "main.c." Since my project did not require any HTTP functionality I removed the calls to the HTTP Initialization and Application functions. The HTTP initialization func-

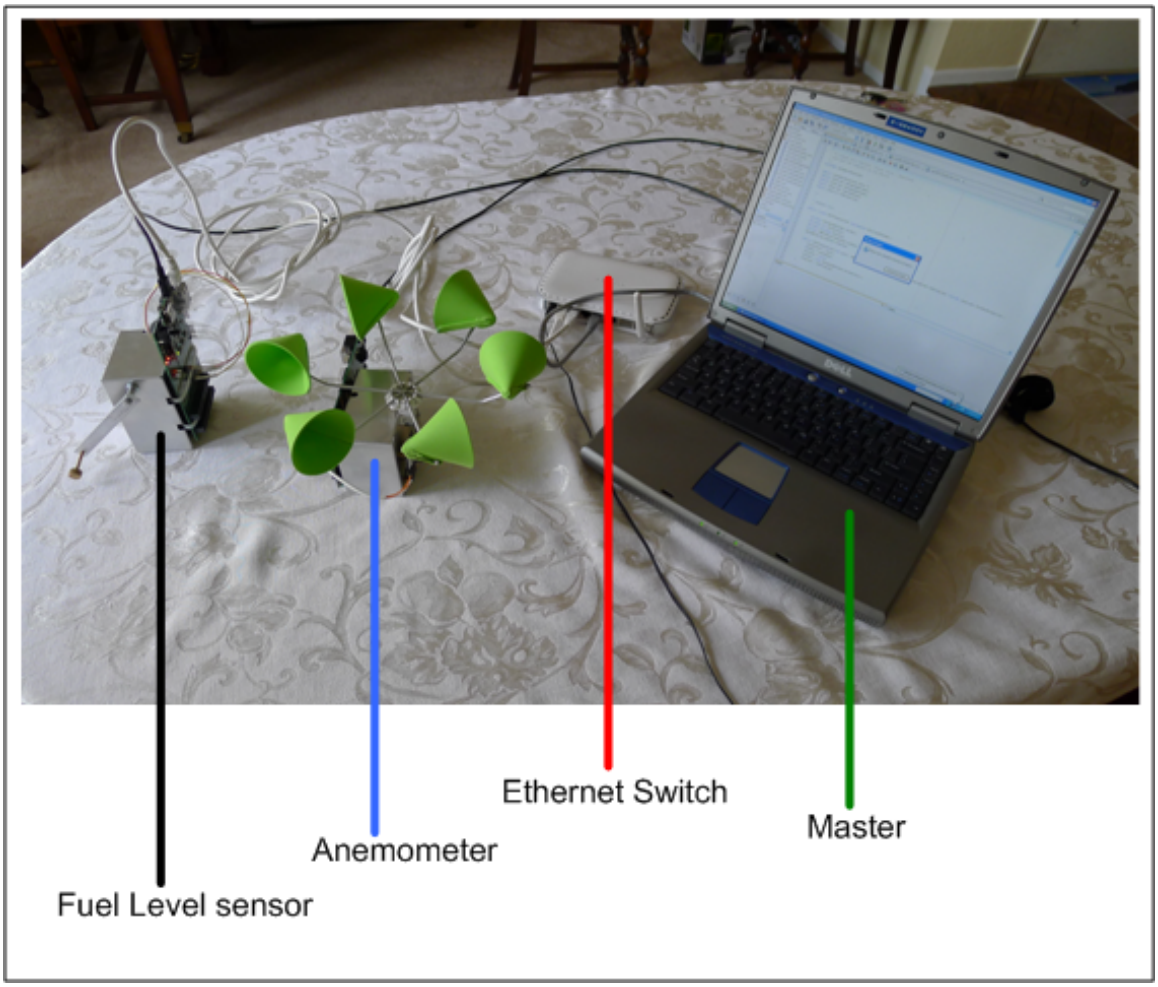

Figure 4.1: Demonstration Setup

tion was called just once in the beginning of function main(). The HTTP Application function was called once on every iteration of the "while" loop. I added code into the "while" loop to check if the DHCP functionality had received an IP address from the DHCP server. Once the IP address is received the OpenMECS initialization function OMECS_ClientInit() is called once from the "while" loop with values for the Server (DHCP Gateway) IP address and the IP address received for the PIC32 board. After the initialization function has been called the End node application function OMECS_Client() is called once every iteration of the "while" loop.

I defined a "#define" called WINDSPEED in the omecs_client.c file. If WINDSPEED has a value of "1" the code for the Anemometer is compiled into the executable and the code for the Fuel Level sensor is not. If WINDSPEED has a value of "0" the reverse is true. This allowed me to share much of the same code for the End node application between the

two different devices. The device specific implementations are detailed in sections 4.2 and 4.3 below.

For both the Fuel Level Sensor and the Anemometer the ADC on the PIC32 was used to read in the analog value from the sensor or motor. Both devices use physical pin AN4 which correspond to ADC channel 4. Channel 4 of the ADC is configured in OMECS_ClientInit() to automatically sample the value of the pin in the background. This means that the application code does not have to request a conversion of the data. The conversion results will be waiting for the application code. Channel 4 is also configured to reference external 5Volt and Ground signals fed into the VREF and AVSS pins on the microcontroller. Channel 4 is configured to perform two samples in a double buffered setup, which ensures no data overwriting when a conversion results is being read by the application code. OMECS_ClientInit() also calculates how many system ticks are in a one second period by calling the API function provided by the Microchip® System Services API.

OMECS_Client() is called by the "while" loop as fast as can be. This means the timing for the End node application had to be derived in this function. OMECS_Client() reads the ticks per second variable that was setup by OMECS_ClientInit() and packs and transmits the "Device ID" message once every 10 seconds. The only difference is that the "dataSets" field in the message is changed based on if the device is an Anemometer (WINDSPEED == 1) or a Fuel Level Sensor (WINDSPEED == 0).

The other common functionality between the two devices is the reception and unpacking of the "System Time" OpenMECS message from the Laptop PC. The function OMECS_Client() manually unpacks the OpenMECS data in the "SystemTime" message into a global variable kept for the system time. This was done by manually unpacking the data instead of using the OpenMECS library code to show the flexibility of using JSON as the basic data layer. If a developer wished to perform basic string manipulation to pack and unpack the OpenMECS data instead of using the OpenMECS library code there would be no functional problem.

One area of difficulty that I ran into was RAM utilization. Numerous times during development I would compile and execute the project with no bus output as expected. When I tried to debug the problem the debugger would get "lost" and not know which line of C code was executing. I had to resort to debugging the disassembled object code. I figured out that the dynamic memory usage of calling malloc() was overwriting the stack area of RAM. This was causing the code to get "lost" by performing jumps to random address locations and confusing the debugger. The solution was to call the free() function every iteration of OMECS_Client() to free up RAM for other functionality when the OpenMECS messages did not need to be packed and transmitted on an iteration. My original implementation

kept a copy of the OpenMECS message structure in RAM at all times and just updated the data within that structure. By calling free() on the OpenMECS message structure, the structure had to be rebuilt from scratch on every iteration in which it would be transmitted. Another possible solution could have been to modify the CRC32 code. I chose a table based CRC32 method. This method keeps a table in RAM that contains a 256 byte array of pre-calculated CRC32 values for all possible bit combinations in a byte of data. This table is generated during initialization when OMECS_ClientInit() calls the function OMECS_Init() from the OpenMECS code library. I could have stored this table as a constant in Non-volatile memory instead of generating the table into RAM. This would have freed up 256 bytes of RAM. However it would have also had the consequence of making the CRC32 algorithm slower as memory reads from Non-volatile memory are slower than reads from RAM.

4.2 Fuel Level Sensor

I wanted to start with an easy real-world demonstration to which anyone could relate. I picked a Fuel Level Sensor. The premise of this sample is to measure the level of fuel available to the power plant. I have approximated a float-type liquid level sensor. The float in my demonstration is replaced by a knob to allow a human to actuate the arm. The knob is attached to an arm which rotates a potentiometer. The potentiometer essentially acts as a voltage divider circuit. The potentiometer has three wires. Two wires are the reference voltage inputs, the third wire is the divided voltage output. One voltage wire was connected to a 5 volt supply pin on the I/O expansion board. The other voltage wire was connected to a ground pin on the I/O expansion board. The voltage output wire was connected to the ADC channel 4 on pin AN4 of the PIC32 microcontroller. A voltage between 0 and 5 Volts is fed into the ADC as the potentiometer is turned by rotating the arm connected to it. The OMECS_Client() function requests the conversion results from the ADC every 500 milliseconds. The results are stored in a two position array. The results from the very first read are duplicated into both positions of the array. The values in both positions are averaged together to smooth out any big spikes in the data. More averaging could have been used if required. Empirical testing showed that the data was clean enough to used based on a two sample average. The ADC returns a 10 bit result. This means a range of 0 to 1023. The averaged result value is scaled down by a factor of 4 to derive the final "level" of fuel available and transmits that data in an OpenMECS message to the Master node. The result value is checked to ensure that it is below or equal to the maximum value of 100 (for percentage) is used in case the ADC is inaccurate. The PIC32 Starter board contains a physical switch that is connected to pin RD7 of the PIC32 microcontroller. When packing

the wind speed data into the OpenMECS message, the state of pin RD6 is read. If RD6 is low a value of “S” is packed into the OpenMECS message, indicating simulated data. If RD6 is high, a value of ”A” is packed, indicating active data. The system time received is also packed into the message containing the fuel level. The Fuel Level sensor is pictured in Figure 4.3.

4.3 Anemometer

Wind Speed is a crucial component of maritime navigation. An anemometer is the device that is commonly used to measure wind speed. The anemometer proved to be more difficult than the Fuel Level Sensor in terms of the hardware build. I found a board called the Peppermill Power Board offered by Microsoft® Research. It converts the voltage generated by a DC motor into signals which indicate direction and rate of rotation of the motor. The DC motor has two wires. Both wires are connected to the spring-loaded motor terminals on the Peppermill board. The Speed output signal from the Peppermill board is connected to pin AN4 of the PIC32 microcontroller. It was possible to connect the output from the DC motor directly to the ADC of the PIC32 microcontroller, however the DC motor is capable of generating more than 5 volts if spun fast enough. This could have damaged the ADC on the microcontroller. The Peppermill board limits the voltage output of the Speed signal to a maximum of 5 volts. The Ground signal on the Peppermill board was connected to a ground pin on the I/O expansion board. When the hub connected to the DC motor is rotated a signal ranging from 0 volts to 5 volts is fed into the ADC. The OMECS_Client() function requests the conversion results from the ADC every 500 milliseconds. A result calculated from two averaged samples was also used in the Anemometer application. The averaged result value is directly as the “speed” of wind and transmitted in an OpenMECS message to the Master node. The result value is checked to ensure that it is below or equal to a relatively reasonable maximum value of 999. The Anemometer is not calibrated to actual read world wind speed. It is only meant to show a simulation of how a real anemometer device would possible function. Pin RD6 is also used in the Anemometer to pack either “S” or “A” in the same manner as the Fuel Level Sensor. In the Anemometer a second switch connected to pin RD7 is also used. If RD7 is low a value of ”K” is packed into the OpenMECS message, indicating the units used are Knots/h. If RD7 is high a value of ”N” is packed, indicating the units used are Nautical miles/h. The system time received is also packed into the message containing the wind speed. The anemometer node is pictured in Figures 4.4 and 4.5.

4.4 Master node

The master node and the display are one in the same for my project demonstration. The Master node is my laptop PC. Both the Fuel Level Sensor and the Anemometer will receive their IP address from the master node. The Master node runs free DHCP server software named The DHCP server [8]. The display of the data received is being done through a custom Java application I wrote. It handles displaying the fuel level as well as the wind speed. It also has a configuration page with an option of whether or not to send the 'System Time' message, as well as the rate at which to send it if it is enabled. The application also logs which OpenMECS End nodes are connected to the system. A log of every message is displayed with the ability to save the log to a file. All errors that are decipherable are also logged. Figures 4.6, 4.7, and 4.8 depict the operation of the Java application representing the Master node.

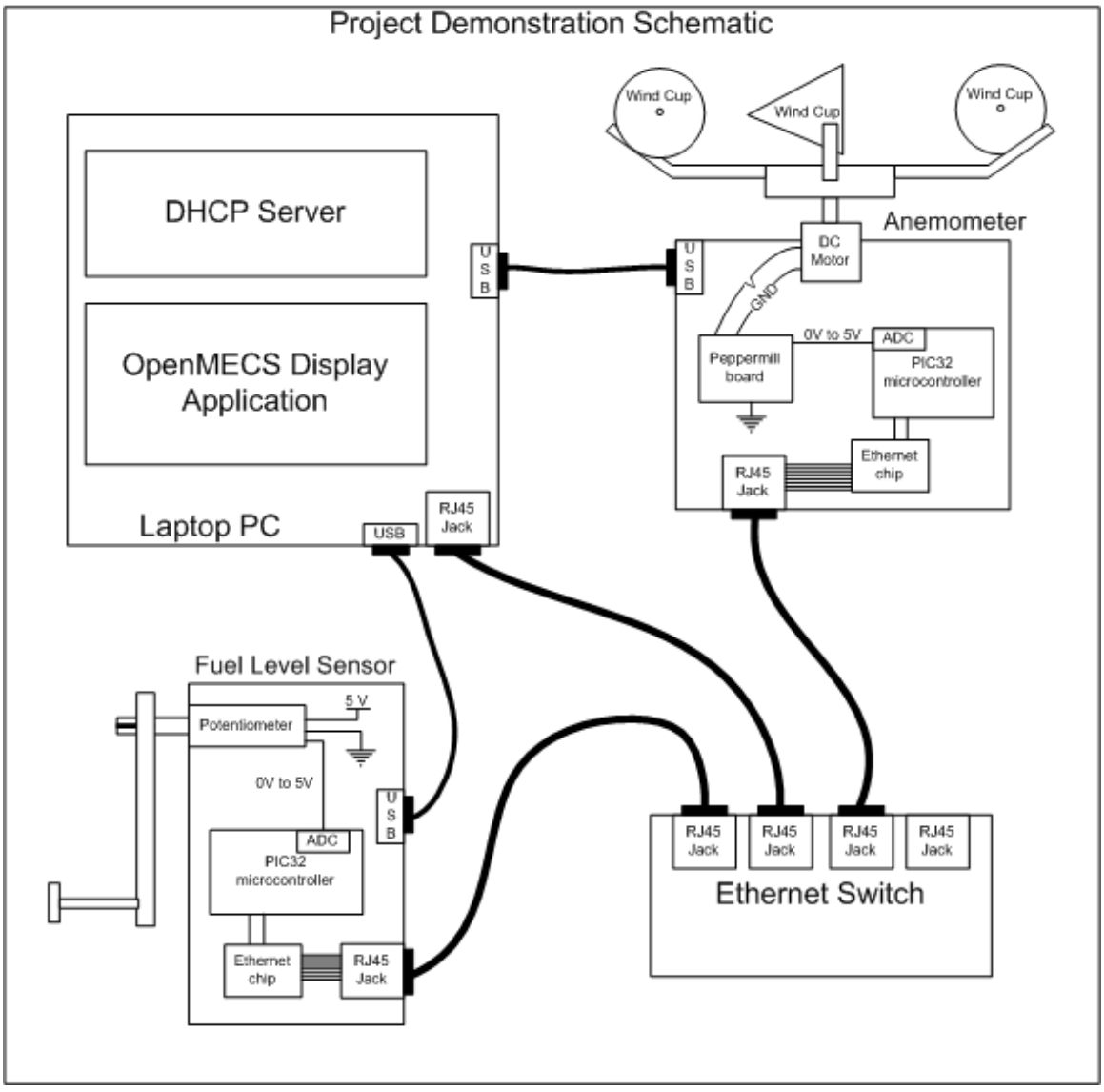

Figure 4.2: Project Demonstration Schematic

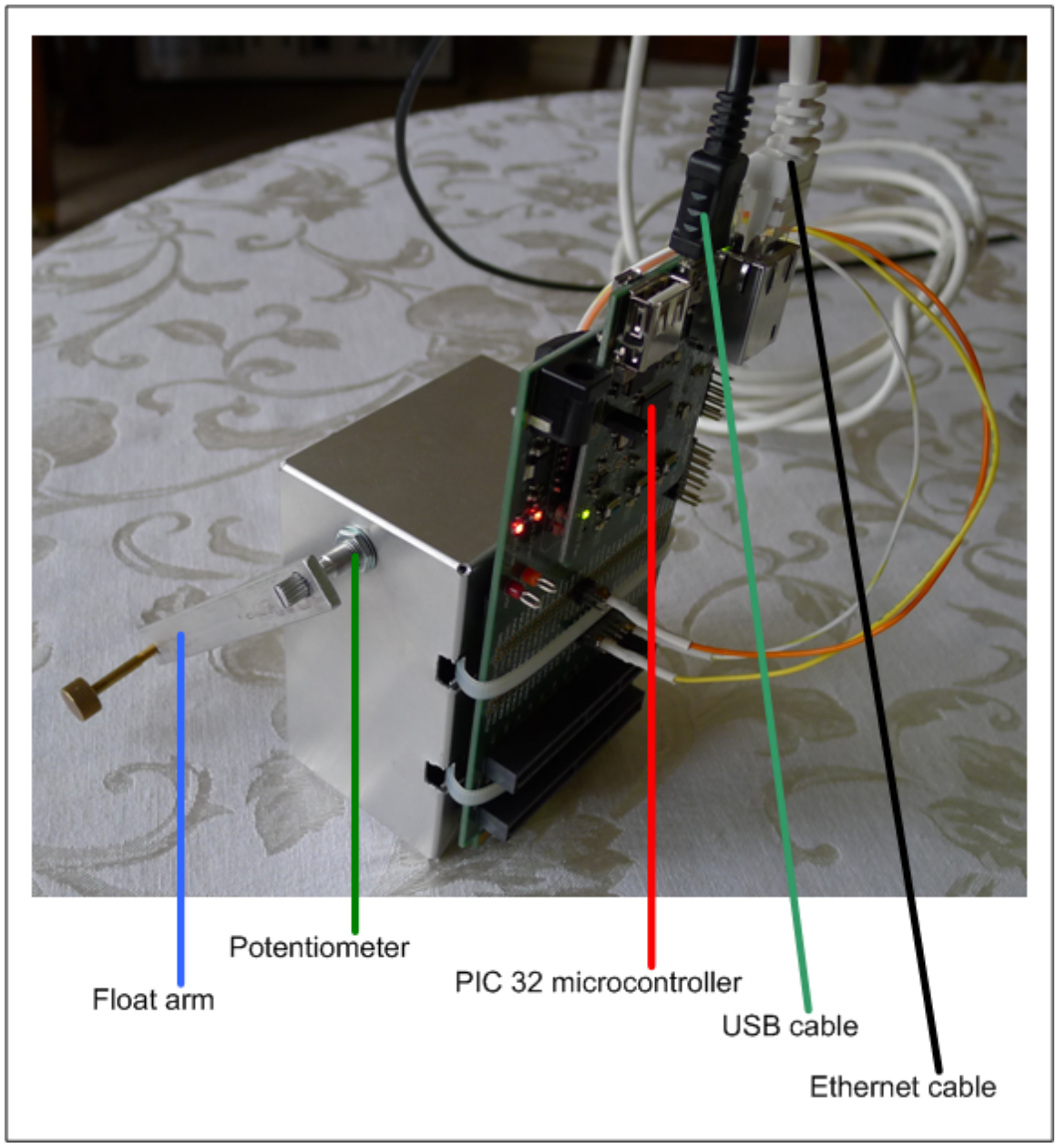

Figure 4.3: Fuel Level Sensor

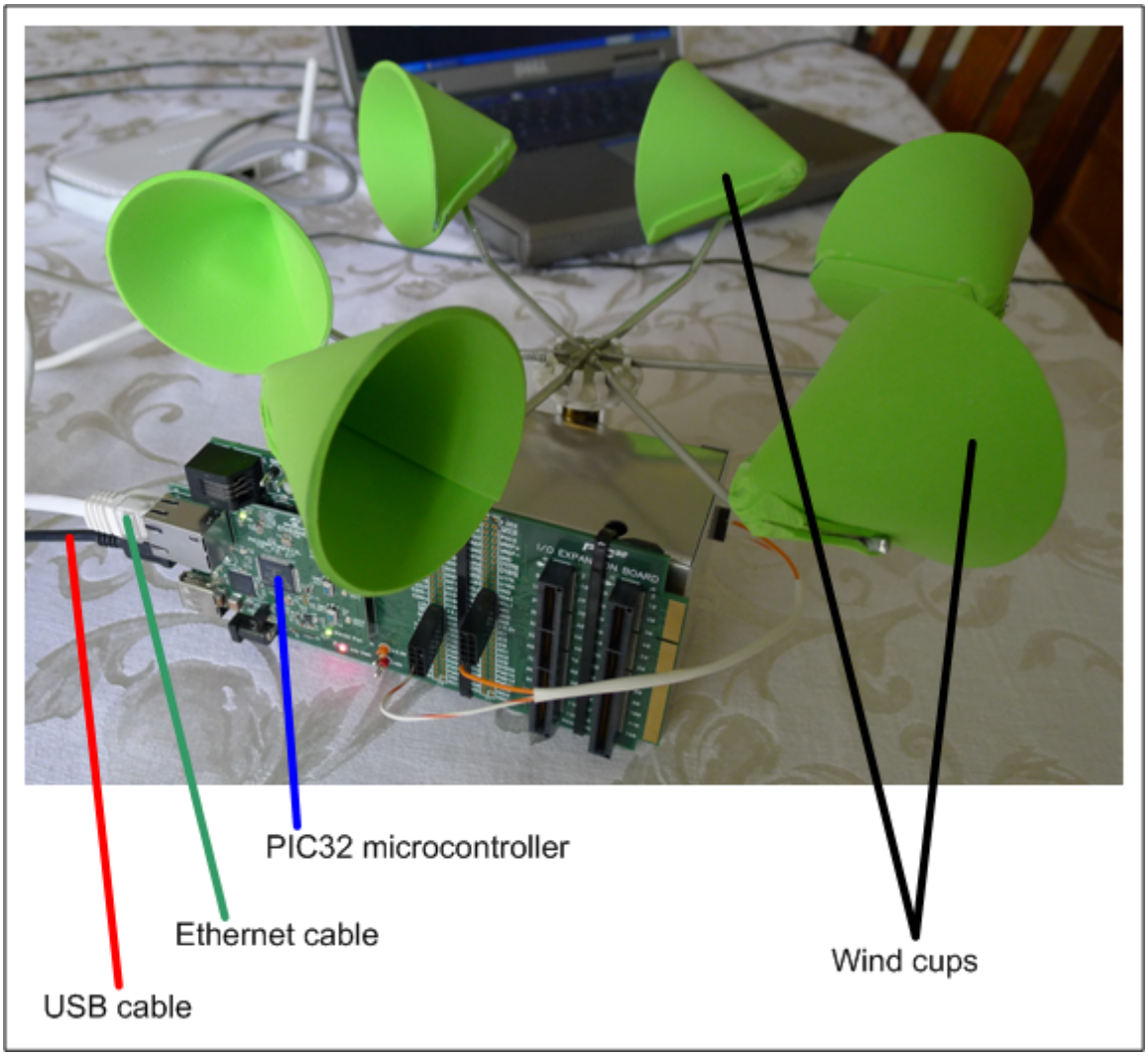

Figure 4.4: Anemometer - Front view

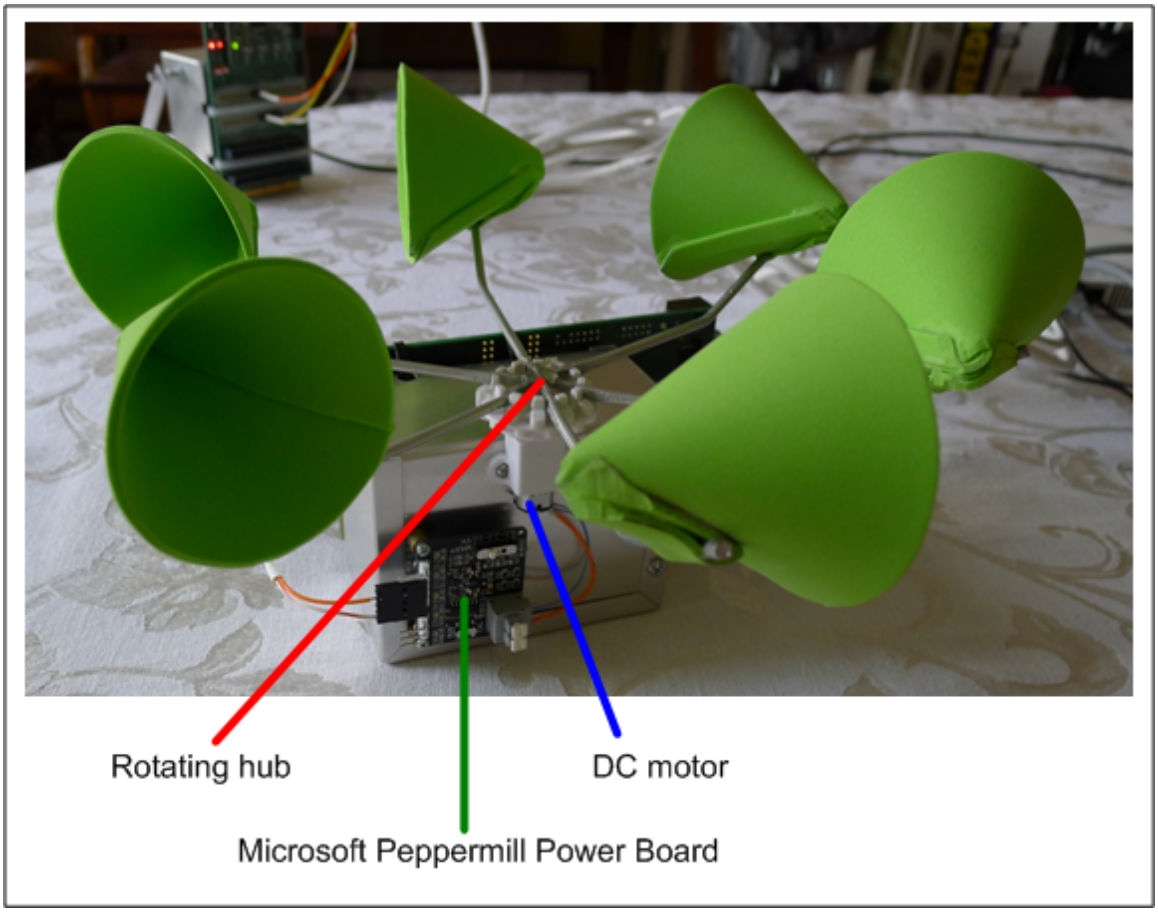

Figure 4.5: Anemometer - Back view

Figure 4.6: Data Display

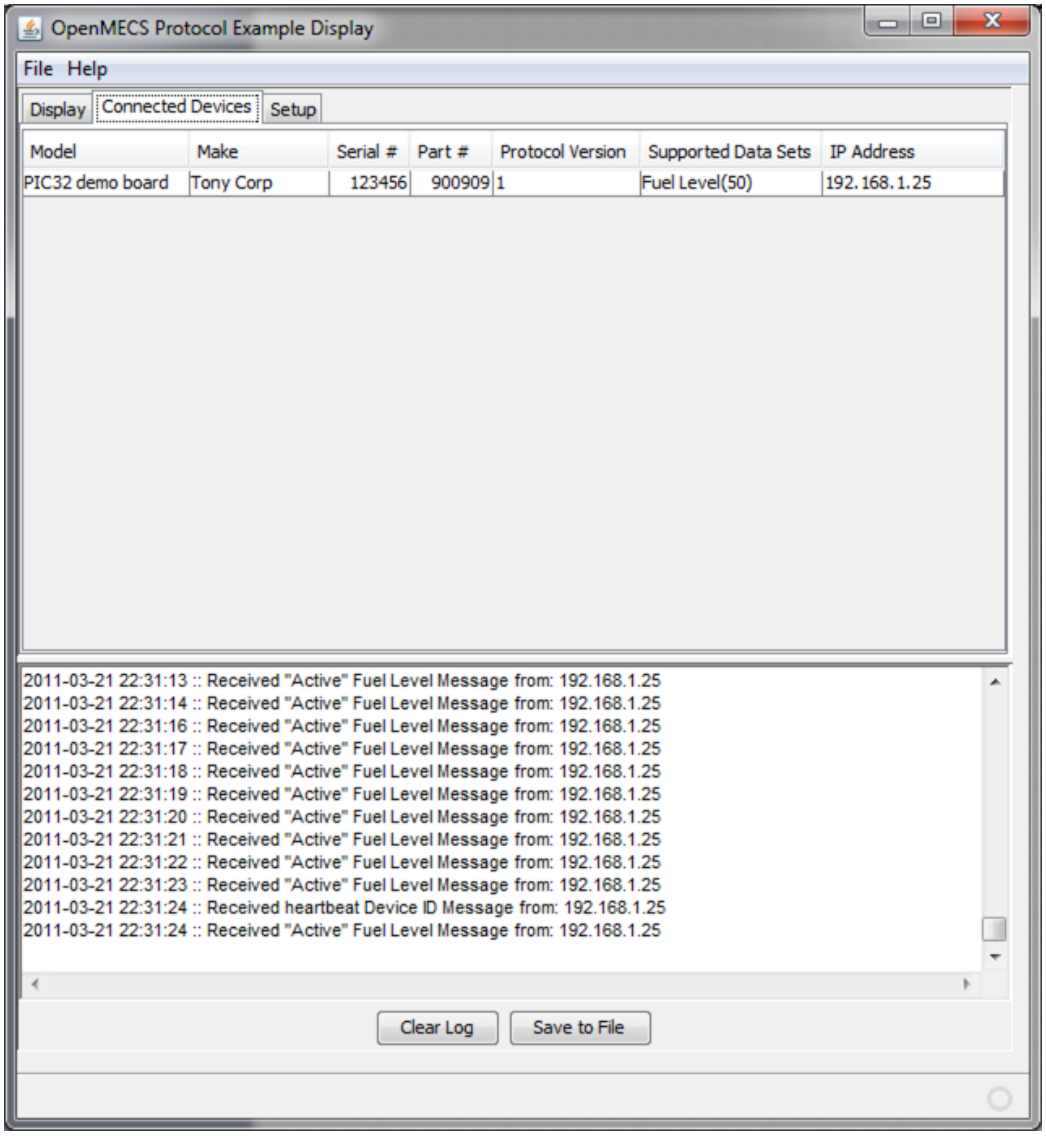

Figure 4.7: Connected Devices

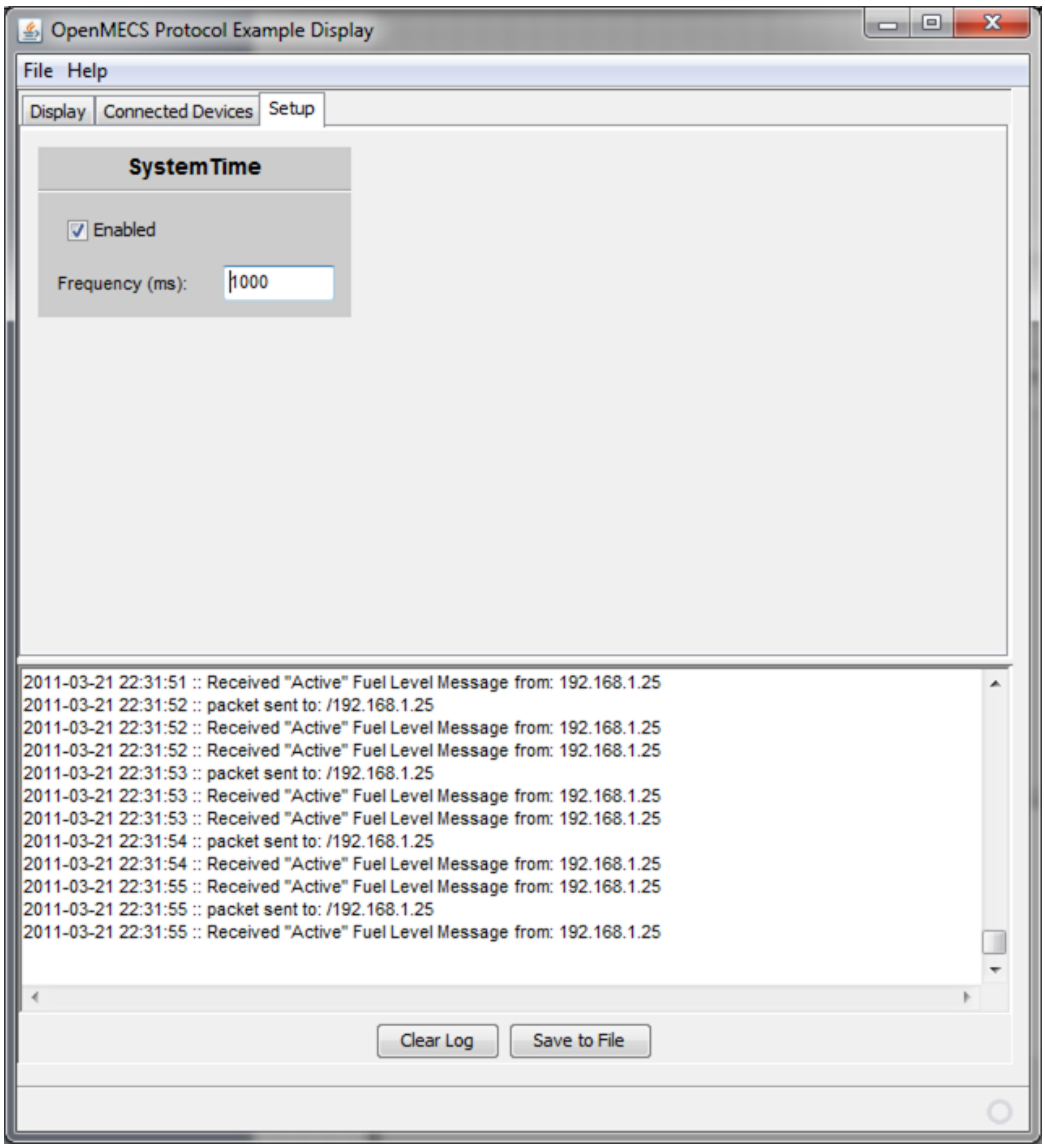

Figure 4.8: Master Setup

Chapter 5
Conclusion

I set out to show that it is possible to create a communication specification that allows electronic devices in marine environments to communicate effectively with each other using inexpensive and readily available technologies. I have shown that not only is it possible, but quite effective. I was able to bring two real world prototypes to working order in a very short time with a minimal budget and little domain knowledge. One of the salient aspects for me was that, just by following the specification as I wrote it, the technologies seemed to disappear from the development. I mean to say that the technology of the communication medium in an of itself was not a hurdle in any way in the development of my prototype nodes. By relying on available and proven technologies, and the simple specification outlined in this paper I was able to concentrate foremost on engineering the electronic and mechanical portions of the sensing nodes. This goes to show just how easy development using the OpenMECS standard on top of TCP/IP can be. Not only was the development easy in terms of technical difficulty, it was inexpensive. The majority of my development was carried out on my existing home PC and networking equipment. The software tools I used to develop the demonstration prototypes, including the IDE and the network analysis software was all free. The hardware was very inexpensive. Each Starter Kit with I/O expansion board ran a total of $144. The remaining parts to complete the electronics and mechanical packages cost roughly $50.

After all is said and done I believe I have created a specification that will truly give hobbyists and device manufacturers the ability to create their products easily, inexpensively, and most importantly without worrying about compatibility. I truly hope that after my work is complete this standard will be adopted by the community and furthered to its utmost potential.

5.1 Future Research

Although I was able to achieve the goals I set for myself, there are areas of the specification that can benefit from further research and development.

5.1.1 Peer to Peer Communication

Along with Master node to End node communication, peer to peer communication would make a strong addition to OpenMECS. This would effectively remove the idea of a Master node being required in an OpenMECS network. The idea is basically to allow any node

to request data from any other node. There are some complexities that would have to be worked out such as how does any single node know what other nodes are supposed to be available on the network?

There are some possibilities such as a discovery protocol. With a discovery protocol a node may broadcast an initial request to query possible providers of the data it seeks. Nodes that have the capability to send the data requested would then send a response alerting the requesting node that they can provide the data requested. Based on the nodes that respond, the requesting node may request the actual data from one or more of those nodes. Another possible scenario is to make further use of the Device identification message. If the Device identification message were to be broadcast by all nodes, then every node on the network would come to know what data is available for them to request. A potential downfall to the discovery method is that it could take a large amount of time for the entire network of OpenMECS nodes to complete the discovery phase and start requesting the data they need. Add this to the already somewhat long process of DHCP and the entire network could take a long time to come online. Longer than might be desirable for a ship operator.

Another possibility is that the list of nodes available is pre-programmed into each End node. In this scenario each End node would also have to have its IP Address pre-programmed into its memory. DHCP would not be needed in this case. A side effect of having the IP Address pre-programmed into the node is that the startup time for the node could be reduced by several seconds.

5.1.2 Security

One aspect of networking that was not discussed in my thesis is security. Although it is tempting to think that security would not need to be considered in an isolated network like OpenMECS, recent news regarding hacking of automobile electronics systems [5] leads me to believe this area would have to be addressed eventually. Some cursory advice might be to not share an OpenMECS network with any network that has Internet access. There are no security provisions built into this version of the protocol. Future versions of the protocol may allow shared networks with well thought out and provisioned network hardware such as smart switches, and software features such as encryption.

References

[1] NMEA-0183 . url http://www.nmea.org/content/nmea_standards/nmea_083_v_400 .asp , 2004.

[2] NMEA-2000 . url http://www.nmea.org/content/nmea_standards/nmea_2000_ed2 _10.asp , 2005.

[3] NMEA 0183 - Wikipedia, the free encyclopedia . url http://en.wikipedia.org/wiki/NMEA_0183 , 2009.

[4] NMEA 2000 - Wikipedia, the free encyclopedia . url http://en.wikipedia.org/wiki/NMEA_2000 , 2009.

[5] Andrew Moseman . . url http://blogs.discovermagazine.com/80beats/2010/05/18 /forget-car-jacking-car-hacking-is-the-crime-of-the-future , 2010.

[6] anonymous . NMEA 0183 Datensatze . url http://www.nmea.de/nmea0183datensaetze.html , 2009.

[7] Glenn Baddeley . Glenn Baddeley - GPS - NMEA sentence information . url http://home.pacific.net.au/ gnb/gps/nmea.html , 2009.

[8] Uwe Ruttkamp . DHCP Server for Windows . url http://ruttkamp.gmxhome.de/dhcpsrv/dhcpsrv.htm , 2011.

[9] various . Introducing JSON . url http://www.json.org , 2010.

[10] various . Wireshark . url http://www.wireshark.org/ , 2011.

Appendix A
Acronyms and Definitions

Table A.1: Acronyms and Definitions

Term \Acronym	**Definition**
ADC	Analog to Digital Converter
API	Application Programming Interface
CAN	Controller Area Network - protocol by Bosch company
CRC	Cyclic Redundancy Check
DHCP	Dynamic Host Configuration Protocol - address assignment
End node	The class of all devices which are responsible for gathering actual data and responding to commands/requests from the Master node.
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment, consists of editor, compiler, debugger, and project setup
JSON	Javascript Object Notation [9]
Master node	The device which requests and processes the OpenMECS data. Only one of these nodes is allowed per OpenMECS network.
NMEA	National Marine Electronics Association
NTP	Network Time Protocol
node	Any electronic device on the OpenMECS network. Can be either a Master node or an End node.
OpenMECS	Open Marine Electronics Communication Specification
RAM	Random Access Memory
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol
USB	Universal Serial Bus
UTC	Coordinated Universal Time
V	Volts or Voltage

Appendix B
Source Code

Part of the OpenMECS standard is a reference implementation of the OpenMECS protocol in the C programming language. C was chosen because it is popular in embedded programming, especially when the embedded equipment consists of low-cost microcontrollers. It is also embeddable in C++ applications, covering another popular embedded programming language. Most low-cost microcontrollers are available with cheap or free programming, compiling, and debugging environments for the C language. All efforts have been made to make the OpenMECS library as cross-platform compatible as possible. While developing the library, the microcontroller family used to test the library was the PIC32 family from Microchip®. The PIC32 is a 32-bit microcontroller family based on the MIPS4000 series processors.

The source code comprising the OpenMECS API as well as the application code from the PIC32 projects may be downloaded from the following location:
http://sourceforge.net/projects/openmecs/files/

B.1 JSON parser

The base format for all OpenMECS messages is JSON. cJSON [9] was chosen as the reference JSON parser. cJSON was chosen for two reasons:

- it is small, fitting in a single C source file combination (.c/.h)
- it is available under the open source MIT license

B.2 OpenMECS API

The following functions are provided as a reference implementation of OpenMECS in the C language. For the most part they add a small wrapper layer around cJSON.

B.2.1 File: omecs_message.c

B.2.1.1 OMECSInit()

void OMECSInit(): initilizes the CRC32 table and the cJSON library.

B.2.1.2 createOMECSMessage()

omecs_object* createOMECSMessage(omecs_bool isDataRequest): Creates a new JSON object that represent a blank OpenMECS message.

B.2.1.3 createOMECSDataSet()

omecs_object* createOMECSDataSet(omecs_data_set_type DataType): Returns a pointer to an array which is an empty Data Set.

B.2.1.4 addToDataSetContentsArray()

void addToDataSetContentsArray(omecs_object *Array, omecs_object *newData): Appends data 'newData' to the end of the Data Set array 'Array'.

B.2.1.5 addContentToOMECSDataSet()

void addContentToOMECSDataSet(omecs_object *DataSet, omecs_object *DataContents): Adds the existing Contents 'DataContents' to the Data Set 'DataSet'.

B.2.1.6 addDataSetToOMECSMessage()

void addDataSetToOMECSMessage(omecs_object *OMECSMessage, omecs_object * DataSet): Adds the existing Data Set 'DataSet' to the message 'OMECSMessage'.

B.2.1.7 finalizeOMECSMessage()

void finalizeOMECSMessage(omecs_object *OMECSMessage): Creates the vailidity value of the message 'OMECSMessage' and appends it to the message. The value is the CRC32 value calculated on the message in string format.

B.2.1.8 OMECSMessageToString()

const char* OMECSMessageToString(omecs_object *OMECSMessage, omecs_bool formatted): Returns a char pointer to the string representation of the message 'OMECSMessage'. 'formatted' indicates whether or not whitespace should be added into the string for readability purposes. This function is used to create the raw text data which will be transmitted across the network.

B.2.1.9 ParseOMECSMessageText()

omecs_object* ParseOMECSMessageText(char * OMECSMessageText): Returns a pointer to an OpenMECS object which is an in-memory object oriented representation of the string 'OMECSMessageText'. This function is used to parse the text of a received OpenMECS message.

B.2.1.10 ValidateOMECSMessage( )

omecs_bool ValidateOMECSMessage(omecs_object *OMECSMessage): Returns a boolean value indicating whether the contents of the message 'OMECSMessage' check against the validity value transmitted with the message.

B.2.1.11 isOMECSMessageRequest()

omecs_bool isOMECSMessageRequest(omecs_object *OMECSMessage): Returns a boolean value indicating whether the message 'OMECSMessage' is a request, or contains data.

B.2.1.12 numDataSetsInOMECSMessage()

int numDataSetsInOMECSMessage(omecs_object *OMECSMessage): Returns a count of the number of Data Sets in the message 'OMECSMessage'.

B.2.1.13 getOMECSDataSetArray()

omecs_object* getOMECSDataSetArray(omecs_object *OMECSMessage): Returns a pointer to the array containing all Data Sets int the message 'OMECSMessage'.

B.2.1.14 getDataSetFromOMECSDataSetArray()

omecs_object* getDataSetFromOMECSDataSetArray(omecs_object *DataSetArray, int which-DataSet): Returns a pointer to the Data Set at index 'WhichDataSet' within the array 'DataSetArray'.

B.2.1.15 deleteOMECSObject()

void deleteOMECSObject(omecs_object *Object): Frees up all memory associated with the object pointed to by 'Object'.

The source in file omecs_message.c is listed as follows:

```
1  /* LICENCE
2  Copyright (c) 2010, Anthony Joseph Arnold Jr
3  All rights reserved.
4
5  Redistribution and use in source and binary forms, with or without
6  modification, are permitted provided that the following conditions are met:
7  * Redistributions of source code must retain the above copyright
8     notice, this list of conditions and the following disclaimer.
9  * Redistributions in binary form must reproduce the above copyright
10    notice, this list of conditions and the following disclaimer in the
11    documentation and/or other materials provided with the distribution.
12 * The name of Anthony Joseph Arnold Jr may not be used to endorse or promote products
13    derived from this software without specific prior written permission.
14
15 THIS SOFTWARE IS PROVIDED BY Anthony Joseph Arnold Jr ''AS IS'' AND ANY
16 EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
17 WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
18 DISCLAIMED. IN NO EVENT SHALL Anthony Joseph Arnold Jr BE LIABLE FOR ANY
19 DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
20 (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
21 LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
22 ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
23 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
24 SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
25 */
26
27
28 #include <stdio.h>
29 #include <stdlib.h>
30 #include <string.h>
31 #include "omecs.h"
32
33 /**
34  * Initializes the OpenMECS API
35  */
36 void OMECSInit(void)
37 {
38     // initialize the CRC32 mechanism
39     gen_crc_table();
40
41     // initialize cJSON, by giving a NULL value, cJSON will use standard malloc, realloc
          and free
42     cJSON_InitHooks(NULL);
43 }
44
45
46 /**
47  * Creates a new JSON object that represent a blank OpenMECS message
48  * @returns pointer to the omecs_object instance created
49  * @param dataRequest indicates whether the message is a data request
50  */
51 omecs_object* createOMECSMessage(omecs_bool isDataRequest)
52 {
53     omecs_object *OMECSMessage;
```

```
54
55      // create base JSON object
56      OMECSMessage = cJSON_CreateObject();
57
58      if(OMECSMessage)
59      {
60          // add fixed fields to OpenMECS message, including an empty Data Sets array to be
61          // filled at a later time
62          cJSON_AddItemToObject(OMECSMessage, "content", cJSON_CreateString("OpenMECS"));
63          cJSON_AddItemToObject(OMECSMessage, "protocolVersion", cJSON_CreateString(
                OMECS_PROTOCOL_VERSION));
64          cJSON_AddItemToObject(OMECSMessage, "dataRequest", ( (isDataRequest == omecs_true)
                ? cJSON_CreateTrue() : cJSON_CreateFalse() ) );
65          cJSON_AddItemToObject(OMECSMessage, "dataSets", cJSON_CreateArray() );
66      }
67      return OMECSMessage;
68  }
69
70
71  /**
72   * Creates a new JSON object that represents a blank OpenMECS Data Set
73   * @returns the omecs_object where the Data Set was created
74   * @param DataType indicates what data will be in the Data Set
75   */
76  omecs_object* createOMECSDataSet(omecs_data_set_type DataType)
77  {
78      omecs_object *NewDataSet;
79
80      //create an empty Data Set with the provided DataType
81      NewDataSet = cJSON_CreateObject();
82
83      if(NewDataSet)
84      {
85          cJSON_AddItemToObject(NewDataSet, "dataType", cJSON_CreateNumber(DataType));
86      }
87
88      return NewDataSet;
89  }
90
91
92  /**
93   * Adds data into an OpenMECS Data Set "dataContents" array
94   * @param Array the array where the data will be appended
95   * @param newData the data to append to the array
96   */
97  void addToDataSetContentsArray(omecs_object *Array, omecs_object *newData)
98  {
99      //always replace existing array
100     cJSON_AddItemToArray(Array, newData);
101 }
102
103
104 /**
105  * Creates a new JSON object that represents a blank OpenMECS Data Set
106  * @param DataSet the Data Set where the contents will be added
```

```
107  * @param DataContents the actual data
108  */
109 void addContentToOMECSDataSet(omecs_object *DataSet, omecs_object *DataContents)
110 {
111     //always replace existing array
112     cJSON_AddItemToObject(DataSet, "dataContents", DataContents);
113 }
114
115
116 /**
117  * Adds a finished Data Set to an OpenMECS Message
118  * @param OMECSMessage the OpenMECS message to add the Data Set into
119  * @param DataSet the Data Set being added to the OpenMECS message
120  */
121 void addDataSetToOMECSMessage(omecs_object *OMECSMessage, omecs_object * DataSet)
122 {
123     omecs_object *DataSetArray;
124
125     DataSetArray = cJSON_GetObjectItem(OMECSMessage, "dataSets");
126
127     //add data set only if a valid pointer was recieved
128     if(DataSetArray)
129     {
130         cJSON_AddItemToArray(DataSetArray, DataSet);
131     }
132 }
133
134
135 /**
136  * Finalizes the remaining fields in the OpenMECS message: numBytes and
137  * Validity.
138  * @param OMECSMessage the OpenMECS message to finalize
139  */
140 void finalizeOMECSMessage(omecs_object *OMECSMessage)
141 {
142     char* messageText = "test text, ignore this";
143     omecs_object *DataSetArray;
144     unsigned long messageLength = 0, temp;
145     unsigned long arrayLen = 0;
146     int crc32 = (int)crc32_first_last_xor;
147
148     // Get the total number of Data Sets in the message
149     DataSetArray = cJSON_GetObjectItem(OMECSMessage, "dataSets");
150     arrayLen = cJSON_GetArraySize(DataSetArray);
151     cJSON_AddItemToObject(OMECSMessage, "numDataSets", cJSON_CreateNumber(arrayLen));
152
153     // Get the message in text form
154     messageText = OMECSMessageToString(OMECSMessage, omecs_false);
155
156     // Get the length of the message text, adjusted to remove first an last chars
157     messageLength = strlen(messageText);
158     messageLength -=2;
159
160     // feed the string into the CRC32 functions, beginning with the byte after the opening
          brace
```

```
161     // and ending with the byte before the ending brace
162     temp = update_crc(crc32_first_last_xor, (messageText + 1), messageLength);
163     crc32 = (int)temp;
164     crc32 ^= crc32_first_last_xor;
165     //printf("CRC: %d\n", temp);
166 
167     // free up the memory in messageText
168     free(messageText);
169 
170     // Add crc32 to the OpenMECS message
171     cJSON_AddItemToObject(OMECSMessage, "validity", cJSON_CreateNumber(crc32));
172 }
173 
174 
175 /**
176  * creates a string representing the entire omecs_object
177  * @returns a string representing the entire OpenMECS message
178  * @param OMECSMessage the OpenMECS message to finalize
179  * @param formatted true indicates that a formatted (with white space) message should
180  *          be returned, as opposed to unformatted
181  */
182 const char* OMECSMessageToString(omecs_object *OMECSMessage, omecs_bool formatted)
183 {
184     return (formatted == omecs_true)? cJSON_Print(OMECSMessage):cJSON_PrintUnformatted(
        OMECSMessage);
185 }
186 
187 
188 /**
189  * Creates an OpenMECS object from a JSON text string
190  * @returns a pointer to the OpenMECS object that was created
191  * @param the JSON text string to parse
192  */
193 omecs_object* ParseOMECSMessageText(char * OMECSMessageText)
194 {
195     return cJSON_Parse(OMECSMessageText);
196 }
197 
198 
199 /**
200  * Validates an OpenMECS message by checking the CRC32 against the
201  * contents of the message
202  * @returns a boolean indicating validity of the message
203  * @param the OMECSMessage to validate
204  */
205 omecs_bool ValidateOMECSMessage(omecs_object *OMECSMessage)
206 {
207     char *text, *ptr, *newstr;
208     int len=0;
209     int validity = cJSON_GetObjectItem(OMECSMessage, "validity")->valueint;
210     int crc32 = crc32_first_last_xor;
211 
212     // get message text and peel off opening brace peel off opening brace
213     text = OMECSMessageToString(OMECSMessage, omecs_false);
214 
```

```
215     // peel off everything past validity
216     ptr = strstr(text, "validity");
217     len = ptr - text - 3;
218
219     newstr = malloc(len+1);
220     newstr[len] = 0;
221     memcpy(newstr, text+1, len);
222
223     //printf("Validating: \n\n%s\n\n", newstr);
224
225     // feed the string into the CRC32 functions, beginning with the byte after the opening
            brace
226     // and ending with the byte before the ending brace
227     crc32 = (int)update_crc(crc32_first_last_xor, newstr, len);
228     crc32 ^= crc32_first_last_xor;
229
230     return (validity == crc32)? omecs_true: omecs_false;
231 }
232
233
234 /**
235  * Indicates whether a message is a request, or contains data
236  * @returns a boolean indicating true if message is a request
237  * @param the OMECSMessage to check
238  */
239 omecs_bool isOMECSMessageRequest(omecs_object *OMECSMessage)
240 {
241     omecs_object *request = cJSON_GetObjectItem(OMECSMessage, "dataRequest");
242
243     return (request->type == cJSON_True)?omecs_true:omecs_false;
244 }
245
246
247 /**
248  * Indicates the number of Data sets contained in the message
249  * @returns the number of Data sets in the message
250  * @param the OMECSMessage to check
251  */
252 int numDataSetsInOMECSMessage(omecs_object *OMECSMessage)
253 {
254     return cJSON_GetObjectItem(OMECSMessage, "numDataSets")->valueint;
255 }
256
257
258 /**
259  * Get the pointer to the array containing all Data Sets
260  * @returns pointer to the array
261  * @param the OMECSMessage to get array from
262  */
263 omecs_object* getOMECSDataSetArray(omecs_object *OMECSMessage)
264 {
265     return cJSON_GetObjectItem(OMECSMessage, "dataSets");
266 }
267
268
```

```
269 /**
270  * Get the pointer to the array containing all Data Sets
271  * @returns pointer to the array
272  * @param the OMECSMessage to get array from
273  */
274 omecs_object* getDataSetFromOMECSDataSetArray(omecs_object *DataSetArray, int whichDataSet
        )
275 {
276     return cJSON_GetArrayItem(DataSetArray, whichDataSet);
277 }
278
279
280
281
282 void deleteOMECSObject(omecs_object *Object)
283 {
284         cJSON_Delete(Object);
285 }
```

B.2.2 File: omecs.h

In addition to type definitions, this header also contains the following function macros:

B.2.2.1 createOMECSdataContentsArray()

#define createOMECSdataContentsArray(): Creates an empty array to hold the Data Contents portion of a Data Set. Returns a pointer to the empty array.

B.2.2.2 createOMECSstring()

#define createOMECSstring(string): Creates a string that can be used in an OpenMECS message. Returns a char pointer to the string.

B.2.2.3 createOMECSboolean()

#define createOMECSboolean(bool): Creates a boolean value that can be used in an OpenMECS message. Returns the created boolean.

B.2.2.4 createOMECSnumber()

#define createOMECSnumber(number): Creates a number, real or integer, that can be used in an OpenMECS message. Returns the created number.

This file expects the cJSON library to be in a folder one level deeper than the this file. It also expects the user to provide a crc32.h header file which defines the following two functions:

- void gen_crctable();
- unsigned long update_crc(unsigned long crc_accum, char *data_blk_ptr, int data_blk_size);

The function gen_crc_table() should generate a table of all possible CRC32 calculations for a data space of 8 bits. The function update_crc() should have access to the above table and use it to calculate the CRC32 value of 'data_blk_ptr' with a size of 'data_blk_size' and a starting CRC32 value of 'crc_accum'. Many examples of a table based CRC32 algorithm are availble online. I found one at the following location: http:www.packet.ccfilesCRC32-code.html

The source in file omecs.h is listed as follows:

```
1  /* LICENCE
2  Copyright (c) 2010, Anthony Joseph Arnold Jr
3  All rights reserved.
4
5  Redistribution and use in source and binary forms, with or without
6  modification, are permitted provided that the following conditions are met:
7  * Redistributions of source code must retain the above copyright
8     notice, this list of conditions and the following disclaimer.
9  * Redistributions in binary form must reproduce the above copyright
10    notice, this list of conditions and the following disclaimer in the
11    documentation and/or other materials provided with the distribution.
12 * The name of Anthony Joseph Arnold Jr may not be used to endorse or promote products
13    derived from this software without specific prior written permission.
14
15 THIS SOFTWARE IS PROVIDED BY Anthony Joseph Arnold Jr ''AS IS'' AND ANY
16 EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
17 WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
18 DISCLAIMED. IN NO EVENT SHALL Anthony Joseph Arnold Jr BE LIABLE FOR ANY
19 DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
20 (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
21 LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
22 ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
23 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
24 SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
25 */
26
27 /*
28
29         This is the root API file for the OpenMECS library. All public
30         OpenMECS function prototypes and type definitions are located in
31         this header file. Other header files contain private function
32         prototypes and type definitions.
33
34         To use the OpenMECS library copy the omecs source folder into your
35         source tree. Then include this header file in your C Source code as
36         follows:
37
38         #include "omecs\omecs.h"
39
```

```
40 */
41
42 #ifndef _OMECS_API_HEADER_
43 #define _OMECS_API_HEADER_
44
45 // ********** Version identification **********
46
47 #define OMECS_VERSION "0.0.1"
48
49 #define OMECS_PROTOCOL_VERSION "1"
50
51
52
53 // ********** Header includes **********
54
55 // Using cJSON library
56 #include "cJSON\cJSON.h"
57
58 // CRC32 library
59 #include "crc32.h"
60
61
62 // ********** Preprocessor Definitions ********
63
64 #define omecs_object cJSON
65
66 #define crc32_first_last_xor 0xFFFFFFFF
67
68
69
70 // ********** Data Type Definitions **********
71
72 //Make sure OpenMECS boolean definitsion are not colliding with other definitions
73 typedef enum
74 {
75     omecs_false,
76     omecs_true
77 } omecs_bool;
78
79
80 typedef enum
81 {
82         omecs_data_type_none=0,
83     omecs_data_type_DeviceIdentification=1,
84     omecs_data_type_SystemTime=2,
85     omecs_data_type_LatitudeLongitude=10,
86     omecs_data_type_Altitude=11,
87     omecs_data_type_Bearing=12,
88     omecs_data_type_Velocity=13,
89     omecs_data_type_RateOfTurn=14,
90     omecs_data_type_AirTemperature=30,
91     omecs_data_type_WaterTemperature=31,
92     omecs_data_type_WindSpeed=32,
93     omecs_data_type_WindAngle=33,
94     omecs_data_type_FuelLevel=50,
```

```
95      omecs_data_type_BatteryLevel=51,
96      omecs_data_type_EngineRevolutions=52,
97      omecs_data_type_RudderAngle=53,
98      omecs_data_type_PowerplantOperationalStatus=54
99  } omecs_data_set_type;
100
101
102
103
104
105
106 // ********** API Function prototypes **********
107 void OMECSInit();
108
109 omecs_object* createOMECSMessage(omecs_bool isDataRequest);
110
111 omecs_object* createOMECSDataSet(omecs_data_set_type DataType);
112
113 void addToDataSetContentsArray(omecs_object *Array, omecs_object *newData);
114
115 void addContentToOMECSDataSet(omecs_object *DataSet, omecs_object *DataContents);
116
117 void addDataSetToOMECSMessage(omecs_object *OMECSMessage, omecs_object * DataSet);
118
119 void finalizeOMECSMessage(omecs_object *OMECSMessage);
120
121 const char* OMECSMessageToString(omecs_object *OMECSMessage, omecs_bool formatted);
122
123 omecs_object* ParseOMECSMessageText(char * OMECSMessageText);
124
125 omecs_bool ValidateOMECSMessage(omecs_object *OMECSMessage);
126
127 omecs_bool isOMECSMessageRequest(omecs_object *OMECSMessage);
128
129 int numDataSetsInOMECSMessage(omecs_object *OMECSMessage);
130
131 omecs_object* getOMECSDataSetArray(omecs_object *OMECSMessage);
132
133 omecs_object* getDataSetFromOMECSDataSetArray(omecs_object *DataSetArray, int whichDataSet
        );
134
135 void deleteOMECSObject(omecs_object *Object);
136
137 #define createOMECSdataContentsArray() cJSON_CreateArray()
138
139 #define createOMECSstring(string) cJSON_CreateString(string)
140
141 #define createOMECSboolean(bool) ((bool==omecs_true)?cJSON_CreateTrue():cJSON_CreateFalse
        ()
142
143 #define createOMECSnumber(number) cJSON_CreateNumber(number)
144
145
146 #endif
```

B.3 Source Code for PIC32 Specific Implementation

B.3.1 File: omecs_client.c

This is the main source file for the End node application. It impements both the Anemometer and Fuel Level sensor functionality.

```
1  /*
2   *
3   * OMECS Client
4   */
5
6  #include <stdio.h>
7  #include <plib.h>
8  #include <stdlib.h>
9  #include <assert.h>
10 #include <string.h>
11 #include <ctype.h>
12 #include <time.h>
13 #include <TCPIP~1\tcpip_bsd.h>
14 #include "omecs.h"
15 #include "omecs_client.h"
16
17 // private function prototypes:
18 void ProcessRecvdMessage(omecs_object * theObject);
19
20
21 // IOPORT bit masks can be found in ports.h
22 #define CONFIG          (CN_OFF)
23 #define PINS            (0)
24 #define PULLUPS         (CN15_PULLUP_ENABLE | CN16_PULLUP_ENABLE)
25 #define INTERRUPT       (CHANGE_INT_ON | CHANGE_INT_PRI_2)
26
27 // temp define for which board config this is
28 #define WINDSPEED               1
29
30
31 SOCKADDR_IN serverIPAddress;
32 SOCKADDR_IN myIPAddress;
33 char myIPAddressStr[16] = "";
34 BOOL OMECS_initialized = FALSE;
35 SOCKET mySocket;
36 UINT ticksPerQuartSecond;
37 BOOL quarterSecond = FALSE;
38 UINT deviceIDCount = 9;
39 UINT dataCount;
40 UINT ticks;
41 omecs_object *deviceIDMessage;
42 omecs_object *deviceIDDataSet;
43 omecs_object *deviceIDDataSetContents;
44 omecs_object *deviceIDDataSetContents1;
45 omecs_object *recvdMessage;
46 char * deviceIDMessageStr;
47
```

```
48  omecs_object *dataMessage;
49  omecs_object *dataDataSet;
50  omecs_object *dataDataSetContents;
51  char * dataMessageStr;
52  UINT fuelLevel=0;
53  double windSpeed=0.0;
54
55  char systemTime[13]="99:99:99:999";
56  unsigned int channel4[2];// conversion result as read from result buffer
57  unsigned int channel5;  // conversion result as read from result buffer
58  unsigned int offset;    // buffer offset to point to the base of the idle buffer
59  unsigned int analogTemp;  // used for temporary storage during calculations based on
        analog data
60  unsigned char resultsIndex;
61
62  void OMECS_ClientInit(IP_ADDR serverAddress, DWORD myAddress)
63  {
64          int i = 0;
65          int offset = 0;
66          char a[4],b[4],c[4],d[4];
67          int err;
68
69          serverIPAddress.sin_port                        = C_OMECSPORT;
70          serverIPAddress.sin_family                      = AF_INET;
71          serverIPAddress.sin_addr.S_un.S_un_b.s_b1    =serverAddress.v[3];
72          serverIPAddress.sin_addr.S_un.S_un_b.s_b2    =serverAddress.v[2];
73          serverIPAddress.sin_addr.S_un.S_un_b.s_b3    =serverAddress.v[1];
74          serverIPAddress.sin_addr.S_un.S_un_b.s_b4    =serverAddress.v[0];
75
76
77          myIPAddress.sin_port                      = C_OMECSPORT;
78          myIPAddress.sin_family                    = AF_INET;
79          myIPAddress.sin_addr.S_un.S_un_b.s_b1     = (myAddress & 0x000000FF);
80          myIPAddress.sin_addr.S_un.S_un_b.s_b2     = (myAddress & 0x0000FF00) >> 8;
81          myIPAddress.sin_addr.S_un.S_un_b.s_b3     = (myAddress & 0x00FF0000) >> 16;
82          myIPAddress.sin_addr.S_un.S_un_b.s_b4     = (myAddress & 0xFF000000) >> 24;
83
84
85          // create the socket on the OpenMECS port
86          mySocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
87
88          // bind socket to received DHCP addresses....
89          err = bind(mySocket, (LPSOCKADDR)&myIPAddress, sizeof(struct sockaddr));
90
91          // check if bind was successfull
92          if(err == 0) OMECS_initialized = TRUE;
93
94          // set up counter variables for timing purposes
95          ticksPerQuartSecond = SystemTickGetResolution() * 315;
96          ticks = 0;
97          quarterSecond = 0;
98          deviceIDCount = 0;
99          dataCount = 0;
100
101         /* string representation of the boards IP address
```

```
102             */
103             itoa(myIPAddress.sin_addr.S_un.S_un_b.s_b1, a, 10);
104             itoa(myIPAddress.sin_addr.S_un.S_un_b.s_b2, b, 10);
105             itoa(myIPAddress.sin_addr.S_un.S_un_b.s_b3, c, 10);
106             itoa(myIPAddress.sin_addr.S_un.S_un_b.s_b4, d, 10);
107             strcat(myIPAddressStr, a);
108             strcat(myIPAddressStr, ".");
109             strcat(myIPAddressStr, b);
110             strcat(myIPAddressStr, ".");
111             strcat(myIPAddressStr, c);
112             strcat(myIPAddressStr, ".");
113             strcat(myIPAddressStr, d);
114
115             /* init OMECS API
116             */
117             OMECSInit();
118
119
120
121             // set up the Switch pins as inputs
122             // PORTD.RD6, RD7 as inputs
123             // could also use mPORTDSetPinsDigitalIn(BIT_6 | BIT_7);
124             PORTSetPinsDigitalIn(IOPORT_D, BIT_6 | BIT_7);
125
126             // Enable change notice, enable discrete pins and weak pullups
127             mCNOpen(CONFIG, PINS, PULLUPS);
128
129             // configure and enable the ADC
130             CloseADC10();   // ensure the ADC is off before setting the configuration
131
132             // define setup parameters for OpenADC10
133             //                              Turn module on | ouput in integer | trigger mode
                    auto | enable autosample
134             #define PARAM1  ADC_FORMAT_INTG | ADC_CLK_AUTO | ADC_AUTO_SAMPLING_ON
135
136             // define setup parameters for OpenADC10
137             //                              ADC ref external    | disable offset test    |
                    disable scan mode | perform 2 samples | use dual buffers | use alternate mode
138             #define PARAM2  ADC_VREF_AVDD_AVSS | ADC_OFFSET_CAL_DISABLE | ADC_SCAN_OFF |
                    ADC_SAMPLES_PER_INT_2 | ADC_ALT_BUF_ON | ADC_ALT_INPUT_ON
139
140             // define setup parameters for OpenADC10
141             //                                 use ADC internal clock | set sample time
142             #define PARAM3  ADC_CONV_CLK_INTERNAL_RC | ADC_SAMPLE_TIME_15
143
144
145             // define setup parameters for OpenADC10
146             //               set AN4 and AN5 as analog inputs
147             #define PARAM4  ENABLE_AN4_ANA | ENABLE_AN5_ANA
148
149
150             // define setup parameters for OpenADC10
151             // do not assign channels to scan
152             #define PARAM5  SKIP_SCAN_ALL
153
```

```
154         // use ground as neg ref for A | use AN4 for input A      | use ground as neg ref
                for A | use AN5 for input B
155
156         // configure to sample AN4 & AN5
157         SetChanADC10( ADC_CH0_NEG_SAMPLEA_NVREF | ADC_CH0_POS_SAMPLEA_AN4 |
                ADC_CH0_NEG_SAMPLEB_NVREF | ADC_CH0_POS_SAMPLEB_AN5); // configure to sample
                AN4 & AN5
158         OpenADC10( PARAM1, PARAM2, PARAM3, PARAM4, PARAM5 ); // configure ADC using the
                parameters defined above
159
160         EnableADC10(); // Enable the ADC
161
162 }
163
164
165         char  *data;
166
167 void OMECS_Client()
168 {
169         // local data
170         int len,sent,recvFlag,i=0;
171         char *ptr1,*ptr2;
172         char test[20];
173         static UINT firstAnalogRead = 1;
174
175         // update ticks
176         ticks ++;
177         if(ticks >= ticksPerQuartSecond)
178         {
179                 quarterSecond = TRUE;
180                 ticks = 0;
181         }
182
183         // ensure initialization was successful before doing anything
184         if(OMECS_initialized)
185         {
186
187                 if(quarterSecond)
188                 {
189                         deviceIDCount++;
190                         dataCount++;
191                         quarterSecond = FALSE;
192                 }
193
194                 // Send Device ID once every 10 seconds
195                 if(deviceIDCount >= C_DEVICE_ID_PERIOD)
196                 {
197                         /* init Device ID packet to send to Master node
198                         */
199                         deviceIDMessage = createOMECSMessage(omecs_false);
200                         deviceIDDataSet = createOMECSDataSet(
                                omecs_data_type_DeviceIdentification);
201                         deviceIDDataSetContents = cJSON_CreateArray();
202                         deviceIDDataSetContents1 = cJSON_CreateArray();
203                         addToDataSetContentsArray(deviceIDDataSetContents,
```

```
                                createOMECSstring(myIPAddressStr));
204                         addToDataSetContentsArray(deviceIDDataSetContents,
                                createOMECSstring(C_MACADDR));
205                         addToDataSetContentsArray(deviceIDDataSetContents,
                                createOMECSstring(C_MAKE));
206                         addToDataSetContentsArray(deviceIDDataSetContents,
                                createOMECSstring(C_MODEL));
207                         addToDataSetContentsArray(deviceIDDataSetContents,
                                createOMECSnumber(C_SERIAL));
208                         addToDataSetContentsArray(deviceIDDataSetContents,
                                createOMECSnumber(C_PART));
209                         addToDataSetContentsArray(deviceIDDataSetContents,
                                createOMECSnumber(1));
210
211             // pack the "supported data sets" based on which board this is
212 #if WINDSPEED == 1
213                         addToDataSetContentsArray(deviceIDDataSetContents1,
                                createOMECSnumber(omecs_data_type_WindSpeed));
214 #else
215                         addToDataSetContentsArray(deviceIDDataSetContents1,
                                createOMECSnumber(omecs_data_type_FuelLevel));
216 #endif
217                         addToDataSetContentsArray(deviceIDDataSetContents,
                                deviceIDDataSetContents1);
218                         addContentToOMECSDataSet(deviceIDDataSet, deviceIDDataSetContents)
                                ;
219                         addDataSetToOMECSMessage(deviceIDMessage, deviceIDDataSet);
220                         finalizeOMECSMessage(deviceIDMessage);
221                         deviceIDMessageStr = OMECSMessageToString(deviceIDMessage,
                                omecs_false);
222                         deleteOMECSObject(deviceIDMessage);
223
224                         // send packet
225                         len = strlen(deviceIDMessageStr);
226                         sent = sendto(mySocket, deviceIDMessageStr, len, 0,(LPSOCKADDR)&
                                serverIPAddress, sizeof(struct sockaddr));
227
228                         free(deviceIDMessageStr);
229
230                         if(sent == len)
231                         {
232                                 len = 0;
233                         }
234                         deviceIDCount = 0;
235                 }// End device ID message
236
237                 if(dataCount >= C_MESSAGE_PERIOD)
238                 {
239
240 #if WINDSPEED == 1
241                         // make wind speed message
242
243
244                         // Read the ADC
245                         offset = 8 * ((~ReadActiveBufferADC10() & 0x01));  // determine
```

```
                    which buffer is idle and create an offset
246                 channel4[resultsIndex] = ReadADC10(offset);             // read
                    the result of channel 4 conversion from the idle buffer
247                 channel5 = ReadADC10(offset + 1);       // read the result of
                    channel 5 conversion from the idle buffer
248
249                 // if this is the very first read, assign the read data into all
                    array positions
250                 if(firstAnalogRead)
251                 {
252                         channel4[1] = channel4[0];
253                         firstAnalogRead = 0;
254                 }
255
256                 // updated results array index (array used to smooth the analog
                    data)
257                 if(resultsIndex < 1)
258                 {
259                         resultsIndex++;
260                 } else
261                 {
262                         resultsIndex = 0;
263                 }
264
265                 // average analog data
266                 analogTemp = (channel4[0] + channel4[1])/2;
267                 // check range of read data
268                 if(analogTemp <= 999)
269                 {
270                         windSpeed = analogTemp;
271                 } else
272                 {
273                         windSpeed = 999;
274                 }
275
276
277
278
279                 dataMessage = createOMECSMessage(omecs_false);
280                 dataDataSet = createOMECSDataSet(omecs_data_type_WindSpeed);
281                 dataDataSetContents = cJSON_CreateArray();
282                 addToDataSetContentsArray(dataDataSetContents, createOMECSstring(
                    systemTime));
283                 addToDataSetContentsArray(dataDataSetContents, createOMECSnumber(
                    windSpeed));
284
285                 // base the K/N/M flag on the position of SW2 on the board
286                 if(PORTDbits.RD7 == 0)
287                 {
288                         addToDataSetContentsArray(dataDataSetContents,
                                createOMECSstring("K"));
289                 }
290                 else
291                 {
292                         addToDataSetContentsArray(dataDataSetContents,
```

```
                createOMECSstring("N"));
293             }
294
295             // base the A/N/S flag on the position of SW1 on the board
296             if(PORTDbits.RD6 == 0)
297             {
298                     addToDataSetContentsArray(dataDataSetContents,
                            createOMECSstring("S"));
299             }
300             else
301             {
302                     addToDataSetContentsArray(dataDataSetContents,
                            createOMECSstring("A"));
303             }
304
305             addContentToOMECSDataSet(dataDataSet, dataDataSetContents);
306             addDataSetToOMECSMessage(dataMessage, dataDataSet);
307             finalizeOMECSMessage(dataMessage);
308             dataMessageStr = OMECSMessageToString(dataMessage, omecs_false);
309
310     #else
311             // make fuel level message
312             // Read the ADC
313             offset = 8 * ((~ReadActiveBufferADC10() & 0x01)); // determine
                    which buffer is idle and create an offset
314             channel4[resultsIndex] = ReadADC10(offset);             // read
                    the result of channel 4 conversion from the idle buffer
315             channel5 = ReadADC10(offset + 1);       // read the result of
                    channel 5 conversion from the idle buffer
316
317             // if this is the very first read, assign the read data into all
                    array positions
318             if(firstAnalogRead)
319             {
320                     channel4[1] = channel4[0];
321                     firstAnalogRead = 0;
322             }
323
324             // updated results array index (array used to smooth the analog
                    data)
325             if(resultsIndex < 1)
326             {
327                     resultsIndex++;
328             } else
329             {
330                     resultsIndex = 0;
331             }
332
333             // average analog data
334             analogTemp = (channel4[0] + channel4[1])/2;
335             // check range of read data
336             if((analogTemp/4) <= 100)
337             {
338                     fuelLevel = analogTemp / 4;
339             } else
```

```
340                         {
341                                 fuelLevel = 100;
342                         }
343
344                         dataMessage = createOMECSMessage(omecs_false);
345                         dataDataSet = createOMECSDataSet(omecs_data_type_FuelLevel);
346                         dataDataSetContents = cJSON_CreateArray();
347                         addToDataSetContentsArray(dataDataSetContents, createOMECSstring(
                                systemTime));
348                         addToDataSetContentsArray(dataDataSetContents, createOMECSnumber(
                                fuelLevel));
349
350                         // base the A/N/S flag on the position of SW1 on the board
351                         if(PORTDbits.RD6 == 0)
352                         {
353                                 addToDataSetContentsArray(dataDataSetContents,
                                        createOMECSstring("S"));
354
355                         }
356                         else
357                         {
358                                 addToDataSetContentsArray(dataDataSetContents,
                                        createOMECSstring("A"));
359                         }
360
361                         addContentToOMECSDataSet(dataDataSet, dataDataSetContents);
362                         addDataSetToOMECSMessage(dataMessage, dataDataSet);
363                         finalizeOMECSMessage(dataMessage);
364                         dataMessageStr = OMECSMessageToString(dataMessage, omecs_false);
365 #endif
366
367                         // send packet containing Fuel level/Wind Speed Message
368                         len = strlen(dataMessageStr);
369                         sent = sendto(mySocket, dataMessageStr, len, 0,(LPSOCKADDR)&
                                serverIPAddress, sizeof(struct sockaddr));
370
371                         // free memory
372                         deleteOMECSObject(dataMessage);
373                         free(dataMessageStr);
374
375                         dataCount = 0;
376
377                 }// End Fuel Level/Wind Speed message
378
379                 // check for received OpenMECS messages once every 250ms, but to conserve
                        memory only receive
380                 // on cycles where the fuel level message is not being formed
381
382                 else if(ticks == (ticksPerQuartSecond-5))
383                 {
384                         // reserve some memory to receive into
385                         data = malloc(1548);
386
387                         // try to receive from socket
388                         recvFlag = recv(mySocket, data, 1548, 0);
```

```
389
390                     switch(recvFlag)
391                     {
392                             case 0: break;// no data, do nothing
393                             case -1: break;// Error receving from socket
394                             default: // something was received, process the data
395                             {
396                                     // make sure a terminated string is received
397
398
399                                     // try to form an OpenMECS message from the
                                            received data
400                                     // recvdMessage = ParseOMECSMessageText(data);
401                                     // NOT WORKING, manually look for system time
402                                     ptr1 = strstr(data, "\"dataType\":2");
403                                     memcpy(test, ptr1, 19);
404                                     // if ptr1 is not NULL, then this is system time
                                            message
405                                     if(ptr1)
406                                     {
407                                             // look for time portion, start at data
                                                    contents
408                                             // the time portion will be location
                                                    between third and fourth quotes
409                                             ptr1 = strstr(data, "\"dataContents\"");
410                                             memcpy(test, ptr1, 19);ptr1++;
411                                             ptr2 = strstr(ptr1, "\"");ptr2++; ptr1 =
                                                    strstr(ptr2, "\"");
412                                             memcpy(test, ptr1, 19);ptr1++;
413                                             ptr2 = strstr(ptr1, "\"");ptr2++; ptr1 =
                                                    strstr(ptr2, "\"");
414                                             memcpy(test, ptr1, 19);ptr1++;
415                                             ptr2 = strstr(ptr1, "\"");*ptr2='\0';
416
417                                             // at this point ptr2 points to begining
                                                    of time, ptr1 to end
418                                             // store time in global for later use.
                                                    terminate the string after ptr1
419                                             memcpy(test, ptr1, 19);
420                                             strcpy(systemTime,ptr1);
421                                     }
422
423                                     // if pointer is not NULL validate the message
424                                     if(recvdMessage != NULL)
425                                     {
426                                             // the message is valid process the
                                                    message
427                                             //if(0)//ValidateOMECSMessage(recvdMessage
                                                    ))
428                                             //{
429                                                     ProcessRecvdMessage(recvdMessage);
430                                             //}
431                                     }
432                                     else
433                                     {
```

```
434                                             // do something for debug
435                                             dataCount+=recvFlag;
436
437                                         }
438
439                                         // free the memory from the receved message
440                                         deleteOMECSObject(recvdMessage);
441
442                                     }// End default case
443                             }// End switch
444
445                             // free up the memory
446                             free(data);
447
448
449                     } // End recieve OpenMECS messages
450
451
452             }
453
454
455
456 }
457
458
459 void ProcessRecvdMessage(omecs_object * theObject)
460 {
461         if(theObject != NULL)
462         {
463                 dataCount+=30;
464         }
465
466
467 }
```

B.3.2 File: main.c

The following file is the "main.c" file that is provided by Microchip® with the PIC32 "Ethernet - TCPIP-BSD - HTTP Server Demo." I have modifed main.c from the original Microchip® provided content for the purposes of the OpenMECS End node application. Section 4.1 describes the modifications necessary to tailor the file for the OpenMECS application.

```
1 /*********************************************************************
2  *
3  *                Example application for the Microchip BSD stack HTTP Server
4  *
5  *********************************************************************
6  * FileName:        main.c
7  * Company:         Microchip Technology, Inc.
8  *
```

```
 9  * Software License Agreement:
10  *
11  * The software supplied herewith by Microchip Technology Incorporated
12  * (the Company) for its dsPIC30F and PICmicro Microcontroller is intended
13  * and supplied to you, the Companys customer, for use solely and
14  * exclusively on Microchip's dsPIC30F and PICmicro Microcontroller products.
15  * The software is owned by the Company and/or its supplier, and is
16  * protected under applicable copyright laws. All rights are reserved.
17  * Any use in violation of the foregoing restrictions may subject the
18  * user to criminal sanctions under applicable laws, as well as to
19  * civil liability for the breach of the terms and conditions of this
20  * license.
21  *
22  * THIS SOFTWARE IS PROVIDED IN AN AS IS CONDITION. NO WARRANTIES,
23  * WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED
24  * TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
25  * PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT,
26  * IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR
27  * CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
28  *
29  *
30  ********************************************************************/
31 #include <plib.h>
32 #include "tcpip_bsd_config.h"
33 #include <TCPIP-BSD\tcpip_bsd.h>
34 #include "OpenMECS\omecs.h"
35 #include "OpenMECS\omecs_client.h"
36 #include "system_services.h"
37
38
39 #if (( (__PIC32_FEATURE_SET__ < 500) || (__PIC32_FEATURE_SET__ > 799) || !defined (_ETH)
        || !defined(MAC_EMBEDDED_PIC32) ))
40         #error "This demo is supossed to run on PIC32MX5-7 family with embedded Ethernet
               Controller!"
41 #endif
42
43 #if !defined (ETH_STARTER_KIT)
44         #error "This demo is supposed to run on an Ethernet STarter Kit board. Define the
               ETH_STARTER_KIT symbol!"
45 #endif
46
47 #include "hardware_profile.h"
48 #include "system_services.h"
49
50
51 // notification function
52 void     TCPIPEventCB(eTCPIPEvent event);
53
54 void     ProcessTCPIPError(eTCPIPEvent errEvent);
55 void     ProcessLinkError(void);
56
57 volatile int     newStackEvent=0;
58 volatile int     totStackEvents=0;
59
60 int              stackUsePolling=0;                    // instead of notification events.
```

```
    No good reason.
61                                                          // always use notification
                                                                , if available
62
63  int             ethErrorEventCnt=0;                     // number of errors that occurred
64  int     ethErrorEvent=0;            // the errors that occurred
65  int     ethRxOvflCnt=0;             // RX overflow count: the most important error; shows
        if the PIC32 system can keep up with the incoming data flow
66
67  int             ethLinkDownCnt=0;                       // link down counter
68  int     ethLinkWasUp=0;             // flag to tell if the link was detected up (we might
        start with the link down or negotiation not performed)
69
70  int             stackHttpTest=0;            // use the http server: always for this demo
71  int             stackUseDhcp=1;             // use the DHCP client to get a dynamic IP
        address
72
73  // OMECS data
74  BOOL initialized = FALSE;
75
76  IP_ADDR curr_ip;
77  IP_ADDR serverIP;
78  DWORD myIP;
79  int count;
80  int lastTick;
81
82  //////////////////////////////////////////////////
83  int main()
84  {
85          unsigned int    sys_clk, pb_clk;
86
87          sys_clk=GetSystemClock();
88          pb_clk=SYSTEMConfigWaitStatesAndPB(sys_clk);
89
90          // Turn on the interrupts
91          INTEnableSystemMultiVectoredInt();
92
93      //Turn ON the system clock
94      SystemTickInit(sys_clk, TICKS_PER_SECOND);
95
96      // Initialize the TCP/IP
97          TCPIPSetDefaultAddr(DEFAULT_IP_ADDR, DEFAULT_IP_MASK, DEFAULT_IP_GATEWAY,
                DEFAULT_MAC_ADDR);
98
99          if(!TCPIPInit(sys_clk))
100         {
101                 return 0;
102         }
103
104         if(stackUseDhcp)
105         {
106                 DHCPInit();
107         }
108
109         curr_ip.Val = 0;
```

```
110
111         // OpenMECS
112         // removed initialization of HTTP server, it is not needed by OpenMECS application
113
114         if(!stackUsePolling)
115         {
116                 TCPIPEventSetNotifyHandler(TCPIPEventCB);
117                 TCPIPEventSetNotifyEvents(TCPIP_EV_STACK_PROCESSED|TCPIP_EV_ERRORS);
118         }
119
120         // OpenMECS
121         // removed all HTTP server code from while loop
122         while(1)
123         {
124         int linkUpdated=0;
125                 int     linkOk=0;
126
127                 if(stackUsePolling==0 && newStackEvent)
128                 {
129                         eTCPIPEvent     activeEvent;
130
131                         newStackEvent=0;
132
133                         activeEvent=TCPIPEventGetPending();
134
135                         if(activeEvent&TCPIP_EV_STACK_PROCESSED)
136                         {
137                                 linkOk=TCPIPEventProcess(activeEvent);
138                 linkUpdated=1;
139                         }
140
141                         if(activeEvent&TCPIP_EV_ERRORS)
142                         {       // some error has occurred
143                                 ProcessTCPIPError(activeEvent);
144                         }
145
146                 }
147                 else if(stackUsePolling)
148                 {       // polling
149                         linkOk=TCPIPEventProcess(0);
150             linkUpdated=1;
151                 }
152
153         if(linkUpdated)
154         {
155             if(linkOk)
156             {
157                 ethLinkWasUp=1;
158             }
159             else if(ethLinkWasUp)
160             {
161                 ProcessLinkError();     // link is actually down only if it was up before
162             }
163         }
164
```

```
165                 // OpenMECS
166                 // Initialize Open MECS once an IP address is received, read server IP
                        address and "my" IP address
167                 serverIP = getDHCPServerID();
168                 myIP = TCPIPGetIPAddr();
169
170                 // OpenMECS
171                 // Check to see if the DHCP process is finished. Compare server IP address
                        againt default value
172                 if((serverIP.v[0] != NULL) && (serverIP.v[0] != '.') && (initialized ==
                        FALSE) && (count > 50000))
173                 {
174                         // as soon as we have an IP address and the Server IP address,
                                initialize the OpenMECS protocol code
175                         OMECS_ClientInit(serverIP, myIP);
176                         initialized = TRUE;
177                 }
178                 // OpenMECS
179                 // Arbitrarily wait until a count of 5000 is reached to ensure the DHCP
                        process can finish
180                 else if ((serverIP.v[0] != NULL) && (serverIP.v[0] != '.') && (initialized
                        == FALSE))
181                 {
182                         count++;
183                 }
184
185                 // OpenMECS
186                 // PROCESS OpenMECS messages, both transmit and receive
187                 if(initialized)
188                 {
189                         // OpenMECS
190                         //  |
191                         //  V   The actual OpenMECS "End node" application
192                         OMECS_Client();
193                 }
194
195                 if(stackUseDhcp)
196                 {
197                         IP_ADDR ip;
198                         DHCPTask();
199
200                     if(curr_ip.Val != (ip.Val = TCPIPGetIPAddr()))
201                     {   // board has changed the address
202                         curr_ip.Val = ip.Val;
203                         }
204                 }
205         }
206
207     return 0;
208 }
209
210 // TCPIP event notification handler
211 void TCPIPEventCB(eTCPIPEvent event)
212 {
213         newStackEvent++;
```

```
214         totStackEvents++;
215
216 }
217
218 void ProcessTCPIPError(eTCPIPEvent errEvent)
219 {
220         // some error has occurred...
221         // take a snapshot of the ETh Controller status
222
223         sEthDcptQuery    rxQuery, txQuery;
224
225         ethRxOvflCnt+=EthStatRxOvflCnt();
226
227         EthDescriptorsQuery(ETH_DCPT_TYPE_RX, &rxQuery);
228         EthDescriptorsQuery(ETH_DCPT_TYPE_TX, &txQuery);
229
230         if(errEvent&TCPIP_EV_ERRORS)
231         {
232         ethErrorEvent|=errEvent;
233                 ethErrorEventCnt++;
234         TCPIPEventAck(errEvent);
235     }
236         // just ignore for now
237 }
238
239
240 void ProcessLinkError(void)
241 {
242         // link is down...
243         ethLinkDownCnt++;
244         return;          // just ignore it for now
245 }
```