Technology Transfer of Observatory Software

2010 MBARI Summer Internship Project

IP PUCK implementation
with OGC SensorML and Sensor Interface Descriptors

Daniel Mihai Toma

Technical University of Catalonia

Mentors: Thomas C. O'Reilly, Kent Headley

Document revision history

24 Aug 2010: Some grammar and spelling corrections (O'Reilly)
30 Aug 2010: Grammar and spelling revision (Headley)
16 Sep 2010: Text revision

CONTENTS

Keywords: IP PUCK; Zeroconf; Sensor Modeling Language (SensorML); Sensor Interface descriptor (SID); Sensor Networks; Ocean Observing Systems

ABSTRACT

Ocean observing systems like MARS, NEPTUNE, OOI RSN and ESONET include cabled networks providing high-speed network and power connections on the sea floor to enable remotely operated and coordinated experiments to be conducted *in situ*. These observing systems may support many different kinds of instruments comprising complex experiments, and need to be capable of connecting to other observing systems and sharing data with them. Therefore, the software infrastructure for these systems must be highly scalable, flexible and interoperable, and must support network-enabled instruments.

During a ten week summer program at MBARI, intern Daniel Toma from the Polytechnic University of Catalonia (UPC) implemented PUCK protocol over TCP/IP and prepared a demonstration of IP PUCK together with Zeroconf and SID (Sensor Interface Descriptor), a proposed extension to OGC SensorML. This work provides a needed reference implementation of PUCK over TCP/IP, and suggests the potential of a set of protocols and standards that could realize true end to end "Plug and Work" capability for sensor networks, and virtually eliminate the need to write new instrument drivers each time a sensor is integrated into a new sensor network.

INTRODUCTION

Interoperability between observing systems is enabled by common interfaces between systems and their sensors and data handling systems. Scalability is enabled by automation, making the integration of sensors and the configuration of platforms and data systems transparent to system users and operators. One can imagine a scenario in which it would be possible for many types of users - from individual researchers on their desktops to web services to other instruments in distant sensor networks - to make use of new data streams simply by plugging an instrument into a seafloor observatory.

There are a number of existing standards for describing and managing geo-referenced data streams. For example the OGC Sensor Web Enablement suite of standards includes SensorML, Web Notification Service, Observations and Measurements, and Sensor Observation Service. These do much to enable interoperability between observing systems. Currently, most

oceanographic instruments have a serial communications interface and are not designed for network operation. In general, instrument control protocols and data exchange formats implemented for oceanographic instruments are not standardized. Integrating new sensors into observing systems typically requires significant effort: it is often the case that driver software must be written for each new sensor, and the task of configuring host platform and data handling system requires a number of manual steps. Each of these tasks introduces expense and significant risk of errors, making the system less reliable. In the context of ocean observing systems, these errors can be extremely costly to address, since it may require the deployment of expensive assets like ships and remotely operated vehicles.

Automatic integration of new sensors and configuration of platforms and data systems can do much to address these issues, and standardizing system interfaces enables automation and interoperation with other systems. While protocols such as IEEE 1451.2 standardize configuration and control interfaces at the instrument level, it is not clear that there are sufficient economic incentives for instrument manufacturers to do so (particularly in the oceanographic instrument sector, where companies are generally small). Moreover, candidate standards like IEEE1451 are somewhat complex and lack reference implementations.

Software frameworks can be used to create a common (if not necessarily standard) interface to web services and other clients, and to obscure the many proprietary instrument interfaces behind them. They do this by generalizing (abstracting) sensor operations. But frameworks still require that a translation occur through specialized instrument drivers. Even if somewhat simplified by the framework, driver software development often requires significant effort.

Another approach is to instead abstract the protocol (in general, a finite state machine) rather than the operations, and write software engines (interpreters) to dynamically implement the protocols. These interpreters could be implemented on many types of platforms, and the abstracted protocol, expressed in a portable form (e.g. XML) could be reused on any platform. Such protocol abstractions would effectively be a universal instrument driver – a single interpreter implementation could communicate with any instrument for which there is a protocol description. Once a protocol description is written (perhaps by the instrument manufacturer), the driver could be reused from system to system, greatly reducing the effort of integrating the sensor. Existing standards and/or software frameworks could encapsulate the standardization of

instrument operations. Combined, the potential exists to create an end-to-end software stack that almost fully automates and standardizes the integration of sensors and their data.

INTERNSHIP PROJECT

Devices enabled with MBARI PUCK protocol have an amount of persistent storage space and use a simple protocol for storing and retrieving resources from this storage space. Instruments and observing systems can use PUCK protocol to enable self-configuration and interoperability by using this capability to provide resources needed to configure and control the instrument and its data in the context of different observing systems. PUCK protocol was initially defined over serial (EIA-232, EIA-422) communications connections. There is an Open Geospatial Consortium (OGC) PUCK standard working group whose objective is to introduce PUCK into OGC's Sensor Web Enablement (SWE) standards suite.

The OGC PUCK working group recommends extending PUCK protocol to include TCP/IP networks, and seeks to demonstrate useful interconnection with other SWE standards. In May 2010, PUCK v1.4 which includes "IP PUCK" was drafted by the OGC PUCK working group and the Smart Ocean Sensors Consortium. The primary objective of the 2010 summer internship project was to provide a reference implementation of IP PUCK protocol, and to demonstrate this in conjunction with other OGC standards.

Project Concept

The figure below illustrates the basic architecture and major components of the project demonstration:

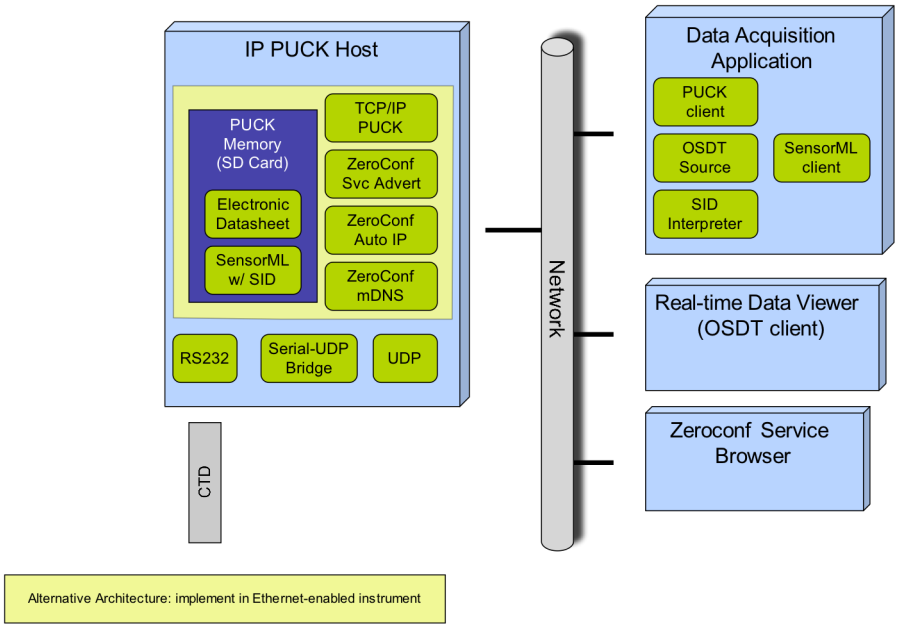


FIG. IP PUCK DEMONSTRATION COMPONENTS

The demonstration architecture is patterned after a network that uses a lightweight host to interface to COTS serial instruments (which were conveniently available for the project). A low power micro controller (Luminary) acts as a **IP Puck host** to one or more COTS oceanographic instruments (here, a **CTD** with an RS232 interface). The IP Puck host also implements **IP PUCK Protocol**, and advertises the instrument service to **Clients** (Data Acquisition Application, Open Source Data Turbine, Service Browser) on the **Network** using **Zeroconf** (service discovery protocol). In addition to service discovery, the Zeroconf implementation we use includes two other core components. The first of these is automatic link-local address arbitration, used here to dynamically negotiate an IP address without the need for a DHCP server. The second component is multicast Domain Name Service (mDNS), that is used to dynamically associate a network name with a device's IP address without the need for a conventional DNS.

The **data acquisition application** uses IP PUCK protocol to retrieve resources (an electronic datasheet and a SensorML/SID document) used to discover, configure and control the instrument and its data.

Other architectures that do not require intermediate host hardware are also possible, and would require that various functions be partitioned between (TCP/IP) instruments and other hosts and clients in the system.

The demonstration flow shows how, *the IP Puck host (with the instrument), when it is connected to the network, may be automatically discovered and used by information clients to acquire samples and display data, all without prior knowledge of the instrument or it's data and control protocols*.

Demonstration Details

In this section, the demonstration flow is presented in greater detail.

Instrument Plug-In

Initially, the instrument (IP Puck host) is not connected to the network, and the instrument is connected to the IP Puck host on an RS232 port. For this project, the IP Puck host has a configuration map associating the PUCK-enabled instrument to a particular serial port, though it could also use a serial monitor program to discover when a serial instrument has been plugged into a serial port and detect that it is PUCK-enabled.

The IP Puck host PUCK memory contains a payload that includes an electronic datasheet and a SensorML document with the SID extension. The PUCK payload is implemented as files on an SD flash memory card. In our case, the electronic datasheet is the simple datasheet defined by the PUCK specification, though it could also include others as well.

PUCK Payload

As described in the previous section, the PUCK payload implemented here by the

host includes a simple electronic datasheet and a SensorML document describing the instrument data stream and control protocol. The PUCK datasheet is 96 bytes of information that uniquely identify the instrument:

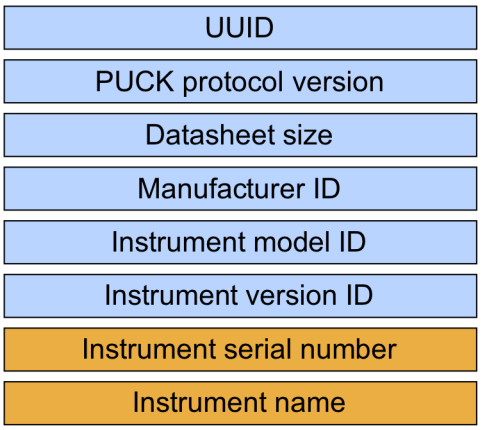


FIG. IP PUCK DATASHEET USED FOR MDNS

Each of these fields are described in more detail in the PUCK specification. As we see in the following section, the serial number and name are used to dynamically define a network name for the instrument's PUCK and data interfaces. The SensorML document specifies the instrument's data outputs, and also includes a description of the instrument control protocol in the form of a Sensor Interface Descriptor (SID) markup, a proposed extension to SensorML. The SID description used here is a minimal subset of the proposed standard, and simply defines the command required to acquire a sample from the instrument.

Negotiating a Network Address

To be truly "Plug and Work" it should possible to for the host/instrument to dynamically and automatically perform network configuration. When the IP Puck host is connected to the network, it must negotiate an IP address, with which it presents two interfaces on its TCP/IP port: one for PUCK protocol and one for instrument control. For the demonstration, the IP Puck host uses Zeroconf's Auto-local IP component to associate an IP address with the host Ethernet interface.

Dynamic Instrument Naming

Although the IP Puck host has established a network address, the address is not convenient for (human or software) clients because it is likely to change if the host/instrument are relocated, and is difficult for human users to manage. For this reason, it is preferable for the instrument TCP/IP interface to be given a unique name that can always be used to refer to that specific instrument.

For this project, PUCK protocol and Multicast Domain Name Service (MDNS) are used to create this association. The host reads the PUCK datasheet and concatenates the Instrument Name and Instrument Serial Number fields to create a name that is both unique and human-readable, for example “RBR-XR420_SN0012”.

Service Discovery via Zeroconf

Once the IP Puck host has established an IP address and well-known name for the instruments interfaces, it advertises the IP PUCK and instrument interfaces by registering them with Zeroconf (which in this case is running on the IP Puck host controller). Zeroconf may then advertise these instrument (service) interfaces for clients on the network to discover. For this project, a simple Zeroconf browser was implemented that shows the information available to clients; it shows that the CTD’s IP address on the network, with a PUCK protocol interface on a TCP/IP port and that data is

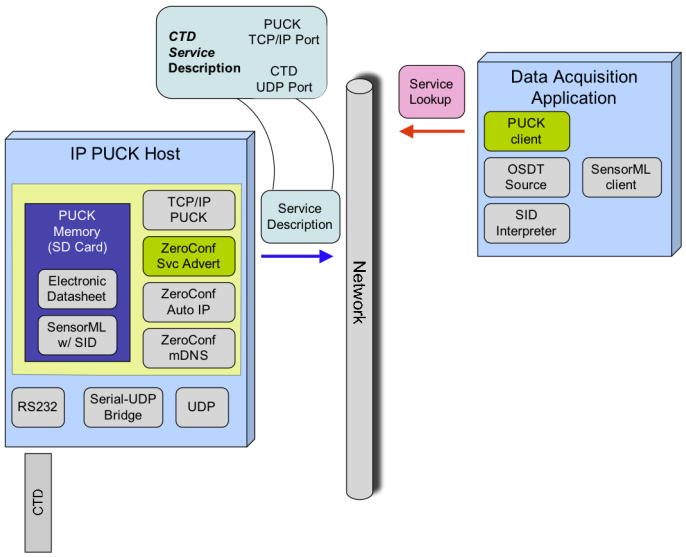


FIG. DATA ACQUISITION APPLICATION DISCOVERS PUCK VIA ZEROCONF

available via UDP on a second port. It could easily advertise specific capabilities that clients may require. *Any client that is Zeroconf-aware can find this instrument, and any client that is PUCK protocol enabled can obtain and use the PUCK payload(s), remotely and without any prior knowledge about the instrument or host.*

Obtaining PUCK payloads: SensorML with SID

For our demonstration, a data acquisition application was created that is capable of using PUCK protocol and includes a SensorML parser and SIDS interpreter. When the data acquisition service is started, it looks to Zeroconf and discovers the CTD service entry that indicates a PUCK protocol service on a specified TCP/IP port. The data acquisition application uses this information to obtain the SensorML payload (along with any other payloads that that it needs) provided by the instrument:

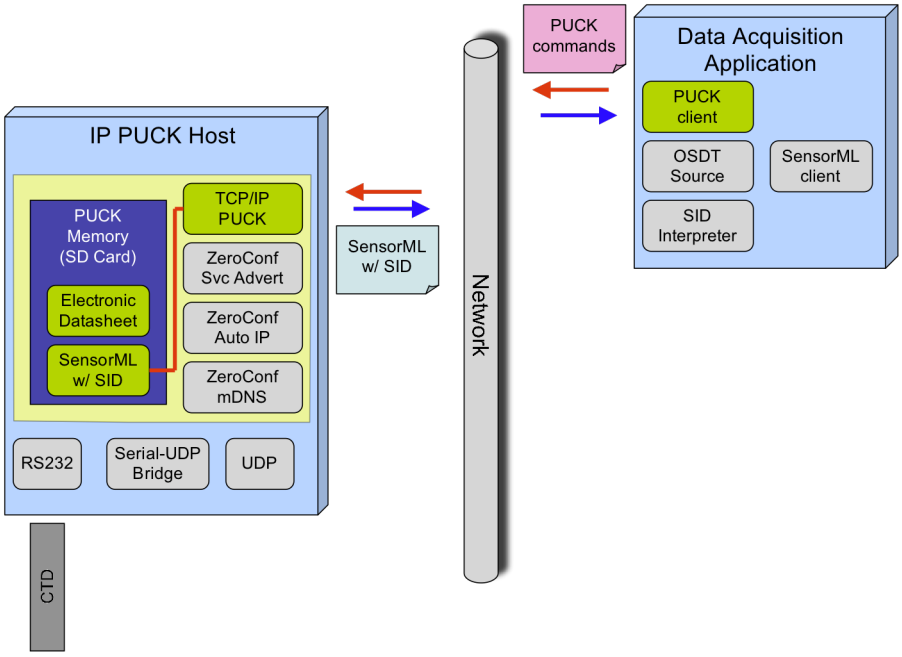


FIG. DATA ACQUISITION APPLICATION OBTAINS SENSORML PUCK PAYLOAD

Data Acquisition using SID

The data acquisition application uses its SensorML/SID parser to interpret the

payload; the SID content in the SensorML document describes the instrument's control protocols sufficiently to enable the SID interpreter to configure the instrument and begin collecting data. Now the data acquisition service can connect to the instrument via the advertised UDP port and begin collecting data.

Other portions of the SensorML document (or perhaps separate Observation and Measurement PUCK payloads) also describes the instrument data stream, allowing the data acquisition application to autonomously parse the data and format to provide data to the Open Source Data Turbine (OSDT) data source.

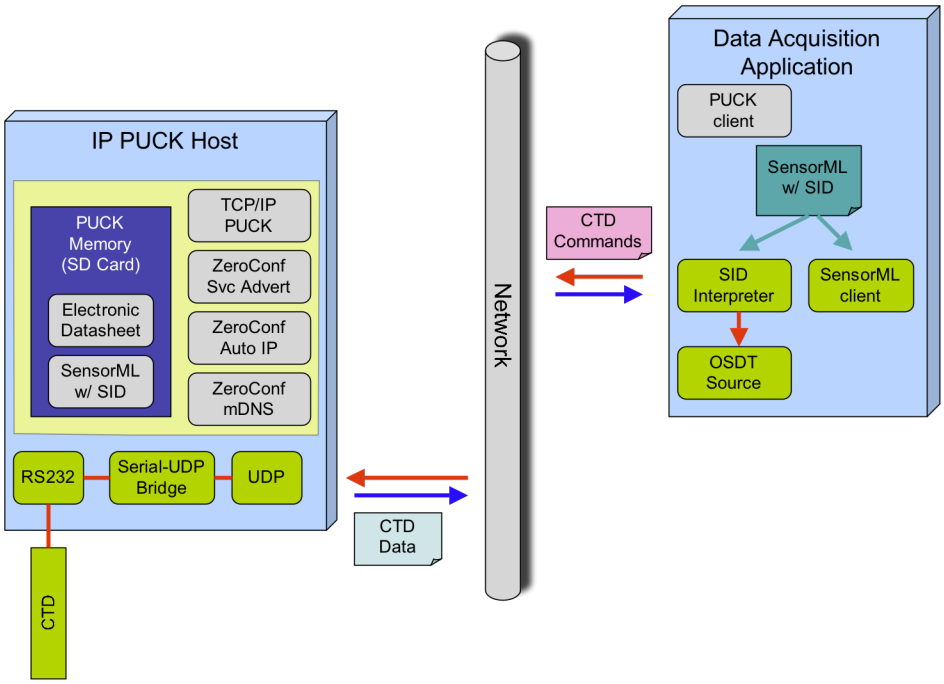


FIG. DATA ACQUISITION APPLICATION USES SID INTERPRETER TO COLLECT DATA

Data Visualization via OSDT and Real-time Data Viewer

The OSDT data source buffers data in a ring buffer (which may be on disk, in memory, or both), which it can stream to clients in real-time (or play back an archived buffer). The Real-time Data Viewer (RDV) is an OSDT client that enables data visualization services in the form of real-time XY plots.

When RDV is started, it may be connected to an OSDT data source by specifying the host name and TCP/IP port associated with one or more OSDT data streams (ring buffers). We can specify

the well-known host name and TCP/IP port used for OSDT by the data acquisition application. RDV automatically detects when streams appear and disappear from the specified source: once the host controller has been attached to the network and the data acquisition application begins publishing data, the CTD data channels (conductivity, temperature, depth) automatically appear in RDV and may be selected for plotting.

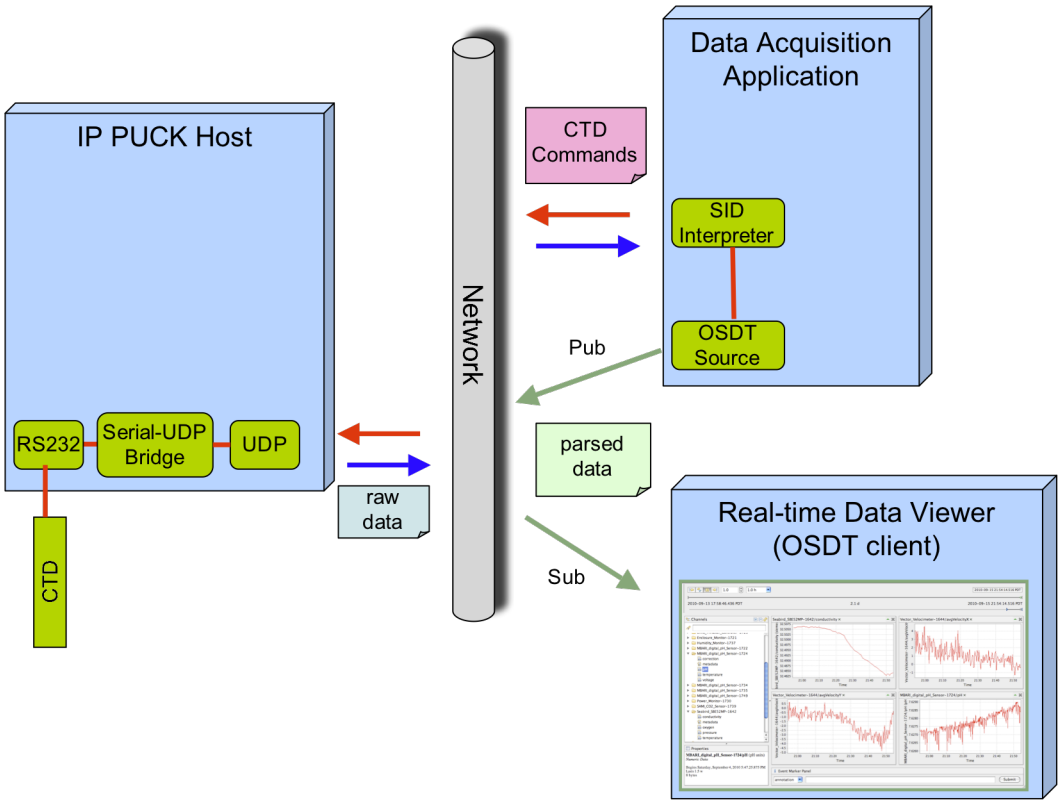


FIG. REAL-TIME DATA VIEWER DISPLAYS DATA FROM OPEN SOURCE DATA TURBINE DATA SOURCE

MATERIALS AND METHODS

This section provides technical details of the hardware and software components used and developed for the 2010 MBARI summer internship project.

IP PUCK Host Controller: Stellaris Luminary Evaluation Board

Fig Stellaris LM3S9B96 Microcontroller Development Kit (DK-LM3S9B96)

This smart Ethernet instrument is implemented on the Stellaris Luminary LM3S9B96 microcontroller. This microcontroller is based on the AR Cortex™-M3 controller core operating at up to 80 MHz, with 256 kB flash, 96 kB SRAM and ROM. In order to address the increasing need for energy conservation in marine instrumentation, the Cortex-M3 processor supports extensive clock gating and integrated sleep modes. Enabled by these features, the processor delivers a power consumption of just 4.5mW and a silicon footprint of 0.30 mm^2 when implemented at a target frequency of 50MHz on the TSMC 0.13G process using ARM Metro™ standard cells. Also the Stellaris family offers Thumb-2 instruction set that combines both 16-bit and 32-bit instructions to deliver the best balance of code density and performance. Thumb-2 uses 26 percent less memory than pure 32-bit code to reduce system cost while delivering 25 percent better performance.

The Cortex-M3 processor has been designed to be fast and easy to program, with the users not required to write any assembler code or have extensive knowledge of the architecture to create

simple applications. The processor has a simplified stack-based programmer's model, which still maintains compatibility with the traditional ARM architecture but is analogous to the systems employed by legacy 8- and 16-bit architectures, making the transition to 32-bit easier. Additionally a hardware based interrupt scheme means that writing interrupt service routines (handlers) becomes trivial, and that start-up code is now significantly simplified as no assembler code register manipulation is required (Sadasivan, 2010).

The LM3S9B96 microcontroller also offers hardware-assisted support for synchronized industrial networks utilizing the IEEE 1588 Precision Time Protocol (PTP). The PTP IEEE1588v2 clock synchronization is based on the Zurich University of Applied Sciences development (Sciences, 2007). Having this support, the instruments based on this platform, can be synchronized underwater with precision of tens of nanoseconds.

Microcontroller interface:

- 3 serial ports (RS232, RS422, RS485) which can be used for the scientific instrument connected to the platform
- Bus: CAN, I2C, SPI, GPIO, USB 2.0 On the Go (OTG), the board can be used as a main controller in a new instrument design.

Because the power consumption in marine instrumentation is a very important factor it is necessary for the microcontroller running with Ethernet interface to have very low power consumption. The current consumption of the LM3S9B96 microcontroller running with all the peripherals on, and different sleeping modes are presented in the following table.

Table Preliminary Current Consumption

Parameter	Parameter Name	Condition	Nom	Max	Unit
I_{DD_RUN}	Run mode 1(Flash loop)	V_{DD} = 3.3 V Code= while(1){} executed in Flash Peripherals = All ON System Clock = 50 MHz (with PLL) Temp = 25°C	80	-	mA
I_{DD_SLEEP}	Sleep mode	VDD = 3.3 V Peripherals = All clock gated System Clock = 50 MHz (with PLL) Temp = 25°C	8	-	mA
$I_{DD\_DEEPSLEEP}$	Deep-sleep mode	Peripherals = All OFF 550 - µA System Clock = IOSC30KHZ/64 Temp = 25°C	550	-	uA

The Stellaris LM3S9B96 Microcontroller Development Kit (DK-LM3S9B96) is a full-featured development kit for LM3S9000 series devices. The LM3S9B96 development board has a maximum set of peripherals to demonstrate the microcontroller's capabilities and provides maximum flexibility with breakout jumpers for all I/O. Also the LM3S9B96 development board provides a platform for evaluating memory-demanding applications.

IP PUCK

PUCK is a protocol specification that enables automatic configuration and interoperability. PUCK-enabled devices must provide some amount of persistent data storage ("PUCK memory") and implement a simple command protocol for storing and retrieving resources (which we refer to interchangeably as data, information or payloads) in the storage. The stored information could be an instrument description (metadata), driver code, or any other resources deemed relevant by the observing systems in which it operates (O'Reilly, 2009).

The original implementation of PUCK protocol was defined only for serial (point-to-point, e.g. RS-232) connections. In May 2010 a new version of the PUCK specification known as "PUCK v1.4" was drafted in collaboration with the Smart Ocean Sensors Consortium and the OGC PUCK standard working group. PUCK v1.4 includes specification of "IP PUCK" for TCP/IP connections. An implementation of the draft PUCK v 1.4 protocol was implemented as part of this project.

When an IP PUCK-enabled device is plugged into local area network (LAN) it can be interrogated by an application that is running in the same LAN. The application can use PUCK protocol to retrieve the relevant payloads from the device and use the information appropriately. For example, the application may install and execute instrument driver code that has been retrieved from the instrument or use a description of the data stream to automatically configure the data handling system. We refer to this automated configuration process as *plug-and-work*.

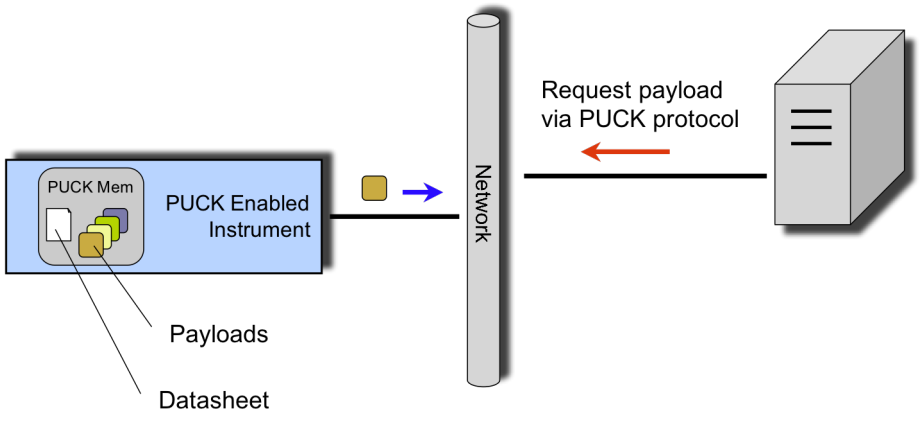


FIG APPLICATION RETRIEVE THE IP PUCK PAYLOAD

The IP PUCK protocol uses simple ASCII command-response sequences to store and retrieve information from the PUCK memory. The command set and semantics for IP PUCK is virtually

identical to the serial PUCK protocol, except that commands that do not apply in the context of TCP/IP connection are not implemented, for example the commands to set baud rate and to enter or exit PUCK mode. The IP PUCK commands are summarized in the table below.

Command	Description
PUCKRM	Read from PUCK memory
PUCKWM	Write to PUCK memory
PUCKFM	End PUCK write session
PUCKEM	Erase PUCK memory
PUCKGA	Get address of PUCK internal memory pointer
PUCKSA	Set address of PUCK internal memory pointer
PUCKSZ	Get the size of the PUCK memory
PUCKTY	Query PUCK type
PUCKVR	Get PUCK protocol version string
PUCK	Null command

Table IP PUCK Command Summary

There is no concurrency model for IP PUCK; as with point-to-point serial links, only one client is allowed access to the device's PUCK interface at any given time.

lwIP TCP/IP Stack

The IP PUCK TCP protocol is developed on the lwIP TCP/IP Stack. The lwIP is a small independent implementation of the TCP/IP protocol suite. lwIP is an implementation of a small TCP/IP, small enough to be used in minimal systems (Dunkels, 2004).

LwIP features include:

- IP (Internet Protocol) including packet forwarding over multiple network interfaces
- ICMP (Internet Control Message Protocol) for network maintenance and debugging
- UDP (User Datagram Protocol) including experimental UDP-lite extensions
- TCP (Transmission Control Protocol) with congestion control, RTT estimation and fast recovery/fast retransmit
- Specialized raw API for enhanced performance
- Optional Berkeley-alike socket API
- DHCP (Dynamic Host Configuration Protocol)
- PPP (Point-to-Point Protocol)

- ARP (Address Resolution Protocol) for Ethernet

The IP transport layer used for IP PUCK protocol is TCP, which provides reliable, ordered delivery of a stream of bytes from a networked device to another networked device.

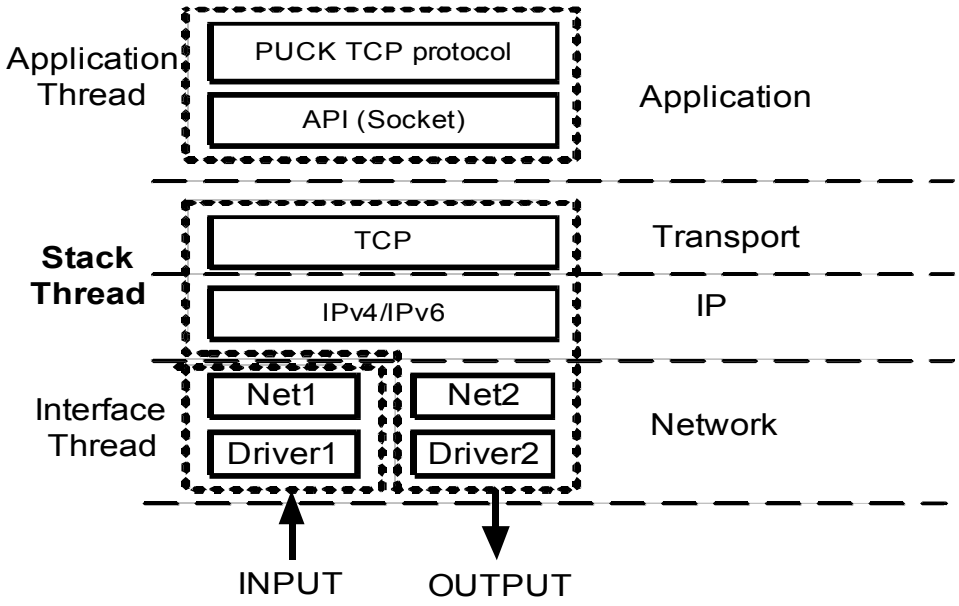


FIG IP PUCK API ON LWIP TCP/IP STACK

FatFs File System

The PUCK memory is implemented using the FatFs file system. FatFs is a generic FAT file system module for small embedded systems. The FatFs is written in compliance with ANSI C and completely separated from the disk I/O layer. Therefore it is independent of hardware architecture. It can be incorporated into low cost microcontrollers, such as AVR, 8051, PIC, ARM, Z80, 68k and etc., without any change (Chan, 2010).

Features of the FatFs include:

- Windows compatible FAT file system.
- Platform independent. Easy to port.
- Very small footprint for code and work area.
- Various configuration options:

- Multiple volumes (physical drives and partitions).
- Multiple ANSI/OEM code pages including DBCS.
- Long file name support in ANSI/OEM or Unicode.
- RTOS support.
- Multiple sector size support.
- Read-only, minimized API, I/O buffer

In the figure below is presented the PUCK memory implementation based on the generic FatFs file system.

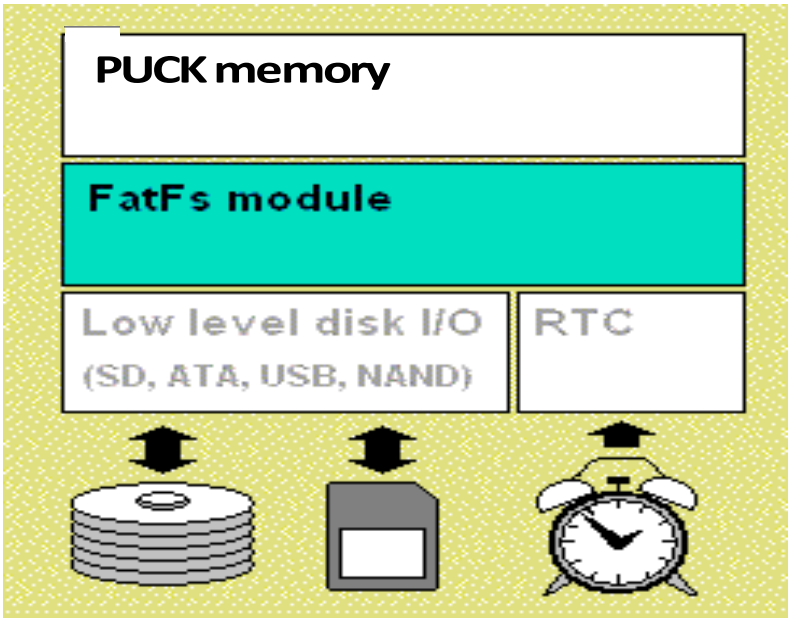


FIG FATFS IMPLEMENTATION OF PUCK MEMORY

The FatFs filesystem is used to implement the IP PUCK memory for this project. On plug-in, after the IP address has been negotiated, the file system code will first check to see if an SD card has been plugged into the microSD slot. If puckdata.dat file exists on the SD card, it will be used as PUCK memory, containing the PUCK datasheet and any payloads. If puckdata.dat file is not found, the file is created.

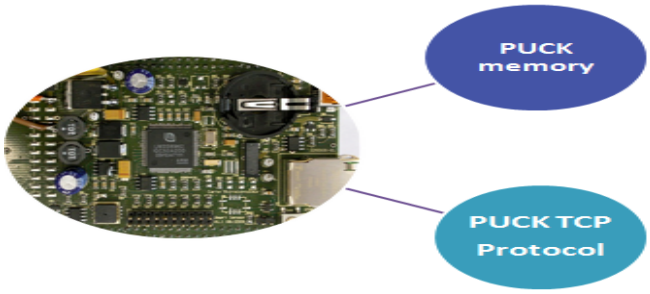


Fig IP PUCK Implementation on Luminary microcontroller

Zero Configuration Networking

Zero configuration networking (Zeroconf), is a protocol that automatically creates a usable Internet Protocol (IP) network without manual operator intervention or special configuration servers.

Zeroconf protocol is able to operate correctly in the absence of configured information from either a user or infrastructure services such as conventional DHCP [RFC2131] or DNS [RFC1034][RFC1035] servers (Williams, 2002). The benefits of Zeroconf protocols over existing configured protocols are an increase in the ease-of-use for end-users and a simplification of the infrastructure necessary to operate protocols.

Compliant Zeroconf implementations include three primary components:

- IPv4 Automatic Link-Local Addressing: Allocate addresses without a DHCP server
- Multicast DNS (mDNS): Translate between names and IP addresses without a DNS server
- DNS Service Discovery: Find services, like printers, without a directory server

In general, networks are dynamic; for example, instruments may be added and removed, network segments may be re-arranged, and devices may change names or run different services. In a configured network an administrator ensures that protocol parameters are updated to reflect these changes and is responsible for ensuring network consistency.

In contrast, Zeroconf enables dynamic configuration under changing network conditions. Zeroconf is able to resolve conflicts and return the network to a consistent state after changes in network topology or other events occur.

For our project, Zeroconf was implemented with a UDP transport layer using the lwIP TCP/IP Stack.

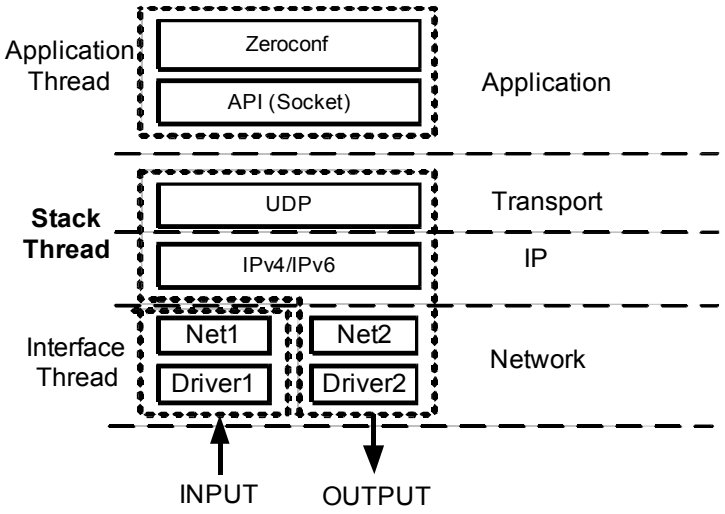


FIG ZEROCONF API ON LWIP TCP/IP STACK

Zeroconf AutoIP Automatic Link-local IP Configuration

If the DHCP client failed to get a response from any DHCP server, it would simply make up a fake response containing a random 169.254.x.x address. If the ARP module reported a conflict for that address, then the DHCP client would try again, making up a new random 169.254.x.x address as many times is necessary until it succeeded.

Zeroconf Multicast DNS (mDNS)

After an IP address was successfully assigned to the device, Zeroconf uses PUCK protocol to get the PUCK datasheet. The instrument name and the serial number are used to form a unique network name for the device. Any whitespace characters in the name are replaced with an underscore character, since mDNS names cannot contain whitespace. Next, the mDNS name is published on the local network.

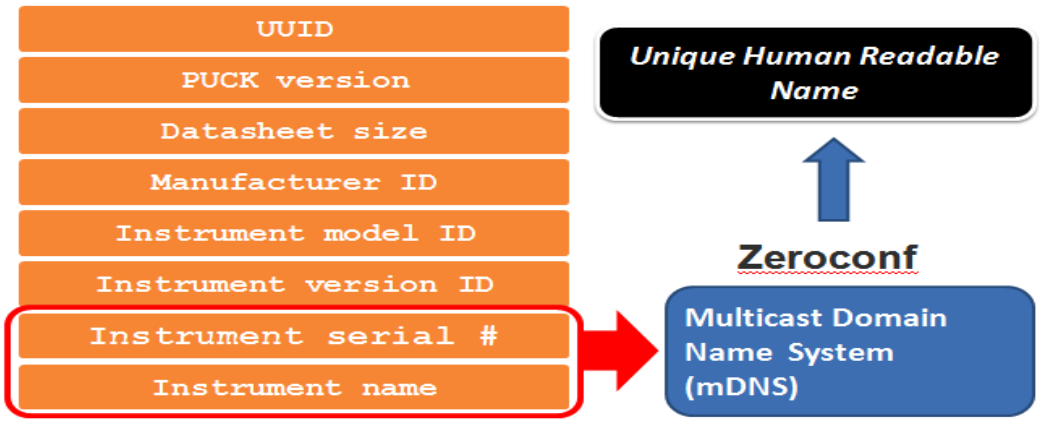


FIG MULTICAST DOMAIN NAME SERVICE FOR PUCK INSTRUMENTS

It is important that PUCK datasheet information is correct, as duplicate name and serial numbers will result in network conflicts, making it impossible to uniquely identify those instruments.

Zeroconf Service Advertisement

After publishing the name to the network, the PUCK protocol interface and CTD data interface services are registered for Zeroconf service discovery. The services are advertised using the mDNS name. The service registry entry contains also other useful information like the service expiration time (time to live), the TCP port for the IP PUCK protocol and a short description of the puck service. Once registered, client applications (e.g. the Data Acquisition Service) may discover and use the PUCK and data interfaces.

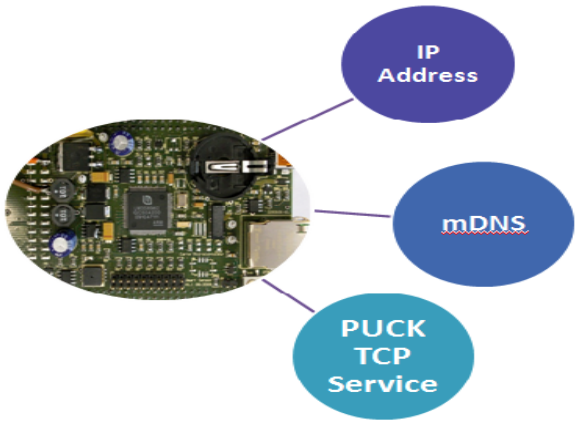


Fig Zeroconf Implementation on Luminary microcontroller

SensorML and Sensor Interface Descriptors (SID)

Interoperability at the sensor interface level is essential to achieving end-to-end "plug and work" capability. Without standardization the interface between observing systems and instruments, integrating instruments and their data streams in new observing systems is expensive and error prone, since the code for this is not reusable. Standardization at the instrument firmware level is difficult, largely because there are insufficient incentives for oceanographic instrument manufacturers to develop and implement standards. Another approach is to standardize a description of instrument protocols and data streams, then build tools around these standards that are widely available and compatible with observing system infrastructure. This approach places less burden on instrument vendors, and the tools are more easily adopted and managed by builders of observing systems. SensorML provides standard models and an XML encoding for describing sensors and measurement processes.

The Sensor Interface Descriptor (SID) schema enables the declarative description of sensor interfaces, including the definition of the communication protocol, sensor commands, processing steps and metadata association. This schema is designed as a profile and extension of SensorML. Based on this schema, SID interpreters can be implemented, independently of particular sensor technology. Services running a SID interpreter can establish the connection to a sensor and are able to communicate with it by using the sensor protocol definition of the SID. SID instances for particular sensor types can be reused in different scenarios and can be shared among user communities (Broering & Stefan, 2010).

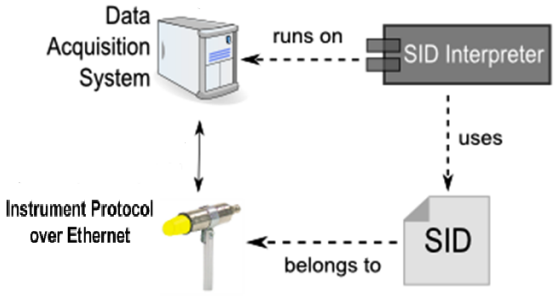


FIG USING SID IN A DATA ACQUISITION APPLICATION

Project Code Organization for Luminary Firmware

The firmware of the Smart Ethernet Instrument Platform was developed using the CodeSourcery tool chain, an Eclipse-based development tool for use with Stellaris CPUs.

Sourcery G++ includes StellarisWare, TI's library containing code for using all the many peripherals (CAN, Ethernet, UART, and many more) present on Stellaris devices.

The C project structure is described in the figure and table below.

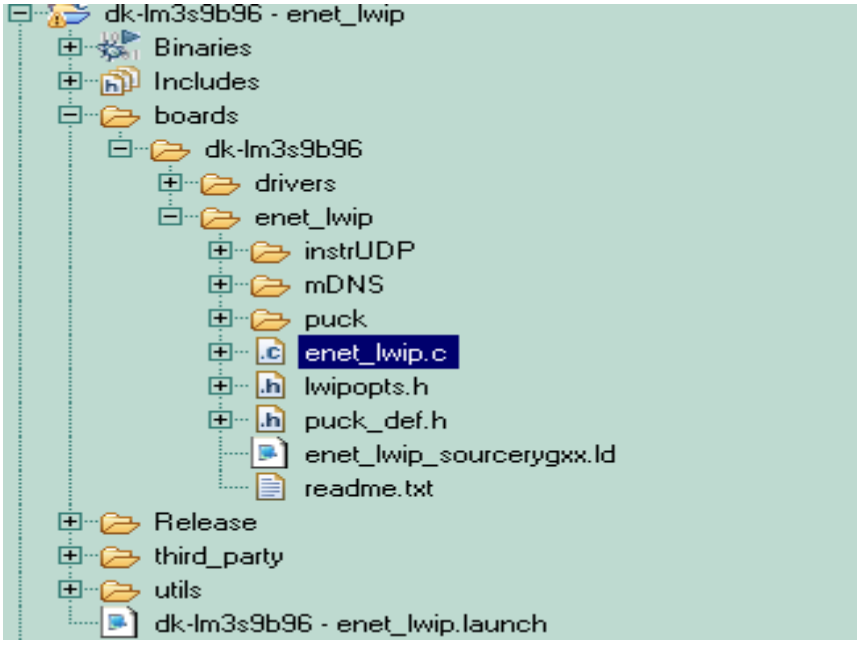


FIG CODESOURCERY PROJECT FOR SMART ETHERNET INSTRUMENT PLATFORM

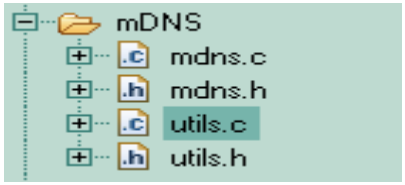


Fig mDNS implementation sources

Folder	Contents
Release	• Binary image used to program the LM3S9B96 microcontroller • lwIP binary image (enet_lwip.bin)
third_party, utils	• libraries for the lwIP stack • FatFs implementation for the LM3S9B96 microcontroller
boards/dk-lm3s9b9z	• modules for the Smart Instrument Platform specific to the LM3S9B96 microcontroller
Drivers, enet_lwip	• Implementation of IP PUCK • Zeroconf • Serial-UDP bridge • C main file, • enet_lwip.c, • Project variable definitions (PUCK port, instrument port, etc.) • interrupt handlers used by enet_lwip_sourcerygxx.ld, for system tick, Ethernet and seria1 peripherals

TABLE FIRMWARE PROJECT DIRECTORY CONTENTS

Project Code Organization for IP PUCK Implementation

In the figure below shows how the IP PUCK implementation source and header files are organized.

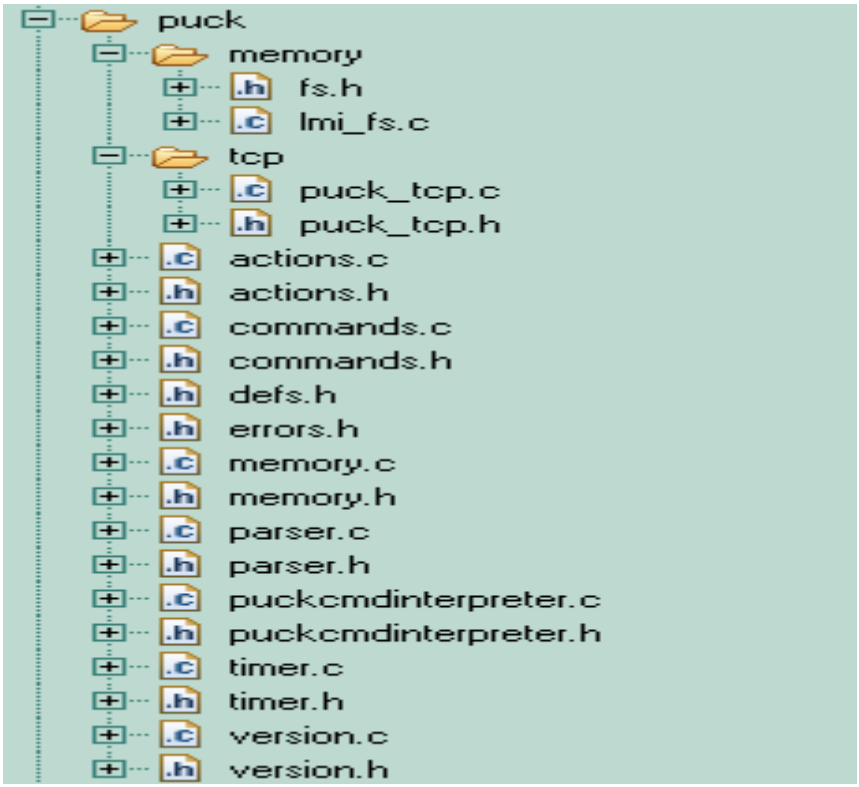


FIG IP PUCK PROJECT DIRECTORY STRUCTURE

Folder	Contents
memory	• lmi_fs.c file, containing the methods for reading, writing, initialization, etc. of the puck memory using the FatFs file system
tcp	• implementation of the TCP communication for the PUCK protocol. Contains method to initialize a TCP socket with a user-defined port number
mDNS	• Multicast DNS implementation
instrUDP	• Serial-UDP bridge implementation

TABLE IP PUCK PROJECT DIRECTORY CONTENTS

FIG SERIAL-UDP BRIDGE PROJECT DIRECTORY STRUCTURE

Project Code Organization for Data Acquisition Application

The data acquisition application is organized in a Java Eclipse project. The projet (and application) runs on a laptop connected to the instrument network. The project contains APIs for Zeroconf and IP PUCK, a SID interpreter and Data Turbine. The com.strangeberry.jmdns.tools package contains the data acquisition application application Java main class. The application command line requires an argument indicating the IP address of the IP Puck host computer (e.g. –i 196.254.0.1). The application runs a Zeroconf Service Discovery browser, which displays all the Zeroconf services on the IP Puck host computer's local area network. When the demo instrument is discovered, an IP PUCK extractor is runs, and obtains the PUCK datasheet and sensorML payloads.

The org.mbari.puck package contains the Java classes for the IP PUCK utilities. The Java implementation of the SID interpreter is in org.mbari.interpreter, and the UDP client and server for the instrument communication are in org.mbari.udp package.

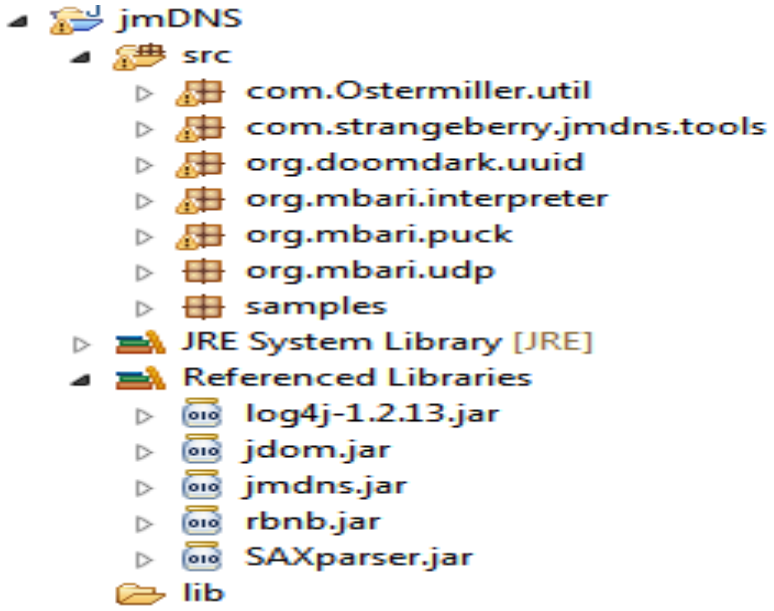


FIG JAVA PROJECT FOR DATA ACQUISITION APPLICATION

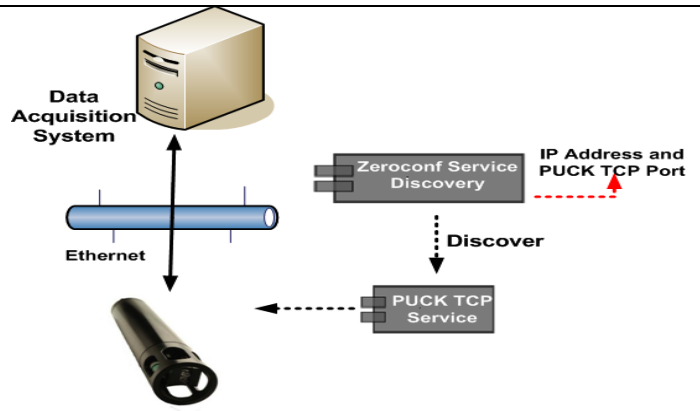
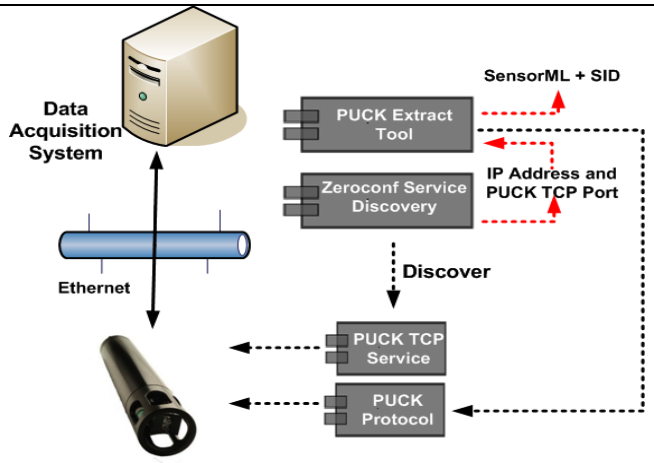
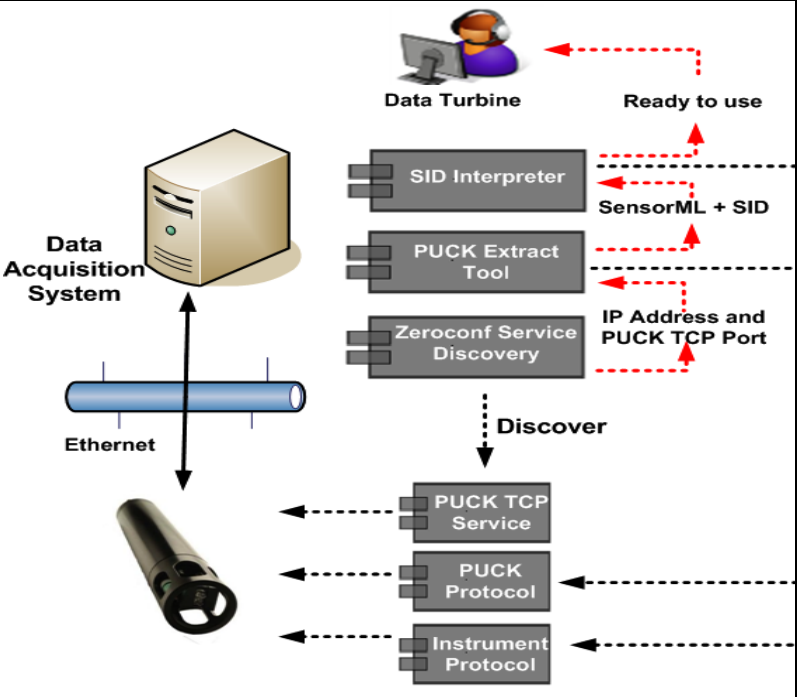
RESULTS

During the 10-week 2010 MBARI summer internship program, SARTI-UPC collaborator Daniel Toma authored an implementation of IP PUCK protocol, extending PUCK capabilities to network enabled devices. In addition, an implementation of Zeroconf, a SensorML parser and other utilities were made to demonstrate a concept for end-to-end plug and work using a SID-based "universal driver". ***This demonstration is important because it shows a potential technology path that uses a completely standards-based software infrastructure to enable much more interoperable and scalable sensor networks by significantly reducing the effort needed to integrate new sensors through standardization and automation.***

For the demonstration, the software developed was used on a Luminary microprocessor development kit to create a network-enabled front end for a COTS CTD made by RBR (del Rio, et al., 2009). When the resulting network-enabled instrument is connected to the network, Zeroconf is used to negotiate an IP address and name using information supplied in the IP PUCK memory, and then advertise the CTD's PUCK protocol and data interfaces in the Zeroconf service registry. Next, a data acquisition client automatically discovers the CTD service, and uses IP PUCK protocol to get a SensorML payload that includes SID markup describing the instrument's command protocol. The data acquisition service interprets the SID information and uses it to collect data from the instrument, *all without using any instrument driver or having prior knowledge of the CTD's proprietary instrument protocol*.

Implementation of PUCK v1.4 and Zeroconf on the Luminary board was straightforward, requiring about four weeks. This relatively modest effort may be attractive to instrument manufacturers, especially since we plan to make the implementaton code available through MBARI's PUCK reference development kit.

Finally, the data acquisition service used the SensorML PUCK payload to parse the CTD data, then published it to an Open Source Data Turbine data source, where RDV (an open source OSDT client) automatically displays the data in real-time.

IP Address Assignment and Service Advertisement	
Service Discovery and PUCK content retrieval	
Using SensorML and SID to acquire, parse data Data Visualization using Open Source Data Turbine and RDV	

Conclusions and Next Steps

The approach demonstrated is potentially a very powerful one. Standardizing the representation of protocols (rather than the protocols themselves) in a platform-neutral way makes it easier to ingrate new sensors into observing systems. It balances the effort required by instrument vendors and observing system integrators in a way that creates incentives and reduces barriers to adopt such standards.

There are still many critical details to work through; the process of representing instrument protocols using the SID standard needs to be rigorously demonstrated, and implementations of parsers and interpreters need to be build that match the computing capability of real observing systems. A critical mass of instrument vendors must be entrained to invest in the approach.

We plan to continue working with collaborators at SARTI-UPC, 52 North, ESONET and OGC along with commercial partners to further develop these concepts as we continue to move PUCK protocol through the OGC standards process. We plan to develop demonstrations with increasingly robust versions of these technologies and present the results. Hopefully, we will see (and perhaps contribute to) the development of reference implementations of SID parsers and interpreters.

ACKNOWLEDGEMENTS

I am grateful to Kent Headley, Duane Edgington, Joaquín del Río, Antoni Manuel, and especially Tom O´Reilly, whose encouragement, supervision and support for the duration of the program enabled me to develop the project.

I would also like to offer my thanks to my coordinators, George Matsumoto and Linda Kuhnz. Without them this research would not have been possible, and their tolerance, good humor, and insight added much.

I am grateful to the many interns who participated in the MBARI 2010 summer internship.

Lastly, I offer my regards to all of those who supported me in any respect during the completion of the project.

REFERENCES

Broering A & Below, S. (2010). *Sensor Interface Descriptors.* OpenGIS Discussion Paper.

Chan. (2010). *FatFs Generic File System Module*. Retrieved from http://elm-chan.org/fsw/ff/00index_e.html

del Rio, J., Auffret, Y., Toma D. M., Shariat, S., et al. (2009). Smart sensor interface for sea bottom observatories. *Instrumentation Viewpoint* , 96-98.

Dunkels, A. (2004). *SICS.* Retrieved from Swedish Institute of Computer Science: http://www.sics.se/~adam/lwip/doc/lwip.pdf

K., W., & Edward, N. (2009). Coupling Wireless Sensor Networks and the Sensor Observation Service. *Bridging the Interoperability Gap* .

S.S., I., & R.R., B. (2005). *"An overview" in Distributed Sensor Networks.* Chapman &Hall/CRC Ed.

Sadasivan, S. (2010). *ARM* . Retrieved 2010, from http://www.arm.com/files/pdf/IntroToCortex-M3.pdf

Sciences, T. Z. (2007). *The Institute of Embedded Systems*. Retrieved from http://www.ines.zhaw.ch/en/engineering/ines/ieee-1588.html

Song, E., & Lee, K. (2007). Smart Transducer Web Services Based on the IEEE 1451.0 Standard. *Sensor Systems* .

O'Reilly et al (2009). *MBARI*. Retrieved from http://www.mbari.org/pw/puck.htm

Williams, A. (2002). *Zeroconf*. Retrieved from Zero Configuration Networking: http://files.Zeroconf.org/draft-ietf-Zeroconf-reqts-12.txt

APPENDIX A

Level of effort

The implementation of the IP PUCK on the Luminary microcontroller based on the lwIP stack and FatFs file system took approximately 2.5 weeks to complete

The implementation of the mDNS part of the Zeroconf protocol took approximately 2 weeks to complete