Grand Valley State University
ScholarWorks@GVSU

Technical Library | School of Computing and Information Systems

2011

ADLib: An Arduino Communication Framework for Ambient Displays

Russ Shearer
Grand Valley State University

Follow this and additional works at: https://scholarworks.gvsu.edu/cistechlib

ScholarWorks Citation
Shearer, Russ, "ADLib: An Arduino Communication Framework for Ambient Displays" (2011). *Technical Library*. 141.
https://scholarworks.gvsu.edu/cistechlib/141

This Project is brought to you for free and open access by the School of Computing and Information Systems at ScholarWorks@GVSU. It has been accepted for inclusion in Technical Library by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.

ADLib: An Arduino Communication Framework for Ambient Displays

By

Russ Shearer
April, 2011

ADLib: An Arduino Communication Framework for Ambient Displays

By
Russ Shearer

A project submitted in partial fulfillment of the requirements for the degree of

Master of Science in

Computer Information Systems

at

Grand Valley State University

April, 2011

D. Robert Adams **April, 2010**

Table of Contents

Abstract

As computers become more and more a part of our everyday lives, the need to change the way in which people interact with them is also evolving. Ambient displays provide an effective way to move computers away from our main focus and into the periphery.

ADLib is a small communication framework that aims to simplify the construction of ambient displays built using the Arduino prototyping platform. The ADLib framework provides an easy-to-use library for communicating with an Arduino, allowing the user to focus on the construction and development of the display.

The framework consists of three main components:

- A protocol for encoding information to be sent from a host computer to the Arduino
- An Arduino library for receiving and parsing incoming data
- A desktop application for sending data to the Arduino

Introduction

"Ubiquitous computing is the method of enhancing computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user." [Mark Weiser 1993] Ambient displays, a subset of Ubiquitous computing, are devices that are “distinguished from more typical informational displays in that they are designed to be minimally attended and perceivable primarily from outside of a person's direct focus of attention, providing pre-attentive processing without being unnecessarily distracting.” [Hazelwood et al. 2008]. In short, an ambient display is an electronic device that allows the user to obtain and perceive information quickly and easily with a single glance.

An ambient display is a type of information display, but there is more to an ambient display than just the physical implementation used to communicate information, and more than what is simply seen by the user. Ambient displays are typically made up of three parts:

1. The physical device to display the necessary information (data sink) which allows the user to quickly and easily view and discern the required information
2. A data source that gathers and processes information (for example an application that grabs the current value of the Dow Jones Industrial Average)
3. A protocol to transfer data between the physical device and the data source

Ambient displays vary greatly in their shape, size and function. One of the earliest examples is known as the "Dangling String" at Xerox's Palo Alto Research Center. Dangling string is literally a piece of string hanging from the ceiling that is connected to a stepper motor. The stepper motor rotates the string faster or slower according to the amount of traffic on the network [Weiser et al. 1995]. Another example is a display that tracks popular bus lines with a mobile [Mankoff et al. 2003]. The mobile indicates the distance of each individual bus from its respective stop using the vertical position of the mobile. Once a bus is within twenty-five minutes of its stop, the mobile associated with it begins to rise from behind a white sheet, and continues upward as the bus approaches. With a quick glance, a rider can estimate how far away the bus is from its stop. Probably the most commercially successfully example of an ambient display is a device known as the "Ambient Orb" [Ambient Devices, 2010], a glass dome that changes color based on different information as configured by the user. The Ambient Orb interfaces with the Ambient Orb web site where a user can choose from a list of data sources and configure details and thresholds. It can be used to show current weather conditions, stock market trends, traffic conditions, etc.

Building ambient displays can be quite challenging. The required knowledge needed spans many different disciplines such as hardware, electronics, protocols, communication methods and software development.

Working with hardware requires knowledge of the specific platform chosen such as the Arduino prototyping platform [Arduino 2009a], and in all but the simplest scenarios, knowledge of electronics such as resisters, capacitors, power supplies, will be required to interface the hardware with components of the display such as LEDs, electric motors, etc.

Communicating with the display from a host computer system requires knowledge of designing protocols for encoding data and communication methods such as Ethernet, Wireless or serial connections for sending the encoded data. Even with libraries for handling most of the low level communication, having the knowledge of how they work helps in designing and debugging protocols.

Development skills are needed for both the host computer and the hardware platform. Often the two systems are very different. The host computer is often a personal computer running a modern operating system like Windows, Linux or Mac OSX, and there are modern development tools and libraries that make retrieving and parsing data fairly easy by screen scraping web sites, consuming RSS feeds, etc. The hardware platform is often a low power embedded microcontroller which is developed in a lower level language such as C and even Assembly language. They have limited computing power and memory with limited or no local storage.

ADLib Overview

As described above, when building ambient displays that use information from an external source, the implementer is often required to spend a significant amount of time designing protocols for data transmision, writing code for encoding and sending the data from a host computer like a PC or server, and writing code for receiving and parsing data on an embedded device such as the Arduino.

The goal of ADlib is to design a framework that simplifies the process of constructing an ambient display by providing an easy-to-use method of transmitting data to the newly designed Arduino-based ambient display. This allows the implementer to effectively utilize his or her valuable time on actually constructing the ambient display, instead of spending unnecessary hours on the details of data transmission. The hope is that a library such as this will help reduce time and increase the innovation in ambient displays.

The ADLib framework consists of three part (Figure 1); a protocol for encoding the data to be sent, an application for performing the actual encoding and transmitting of the data from a host computer over a serial or Ethernet link, as well as an Arduino library that handles the receiving and parsing of data and transferring control to the user's code running on the Arduino. All three combine to create a framework that eases the construction of ambient displays.

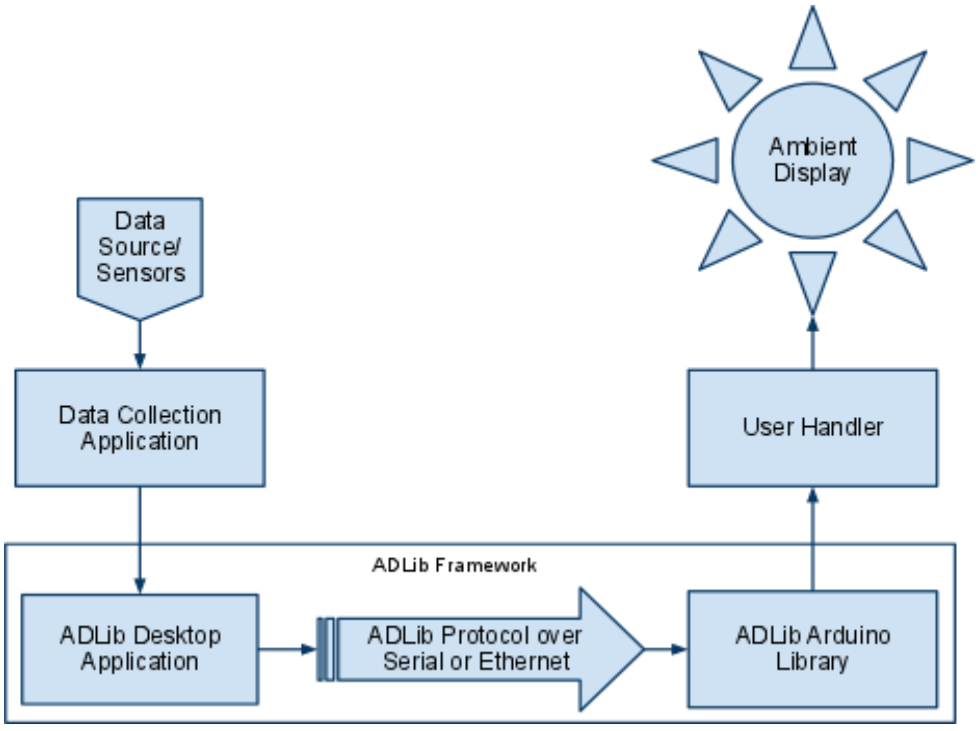

Figure 1: Data flow of the ADLib framework

Background and Related Work

The options for communicating with an Arduino are incredibly varied, and although most have fully implemented libraries providing easy Arduino use, significant effort must still be expanded to design and implement a protocol for encoding data.

The Arduino prototyping platform includes native support for serial communications, allowing a user to easily communicate with an Arduino using their computer or some other device with serial support using nothing more than the proper cable. The Arduino prototyping platform also has the concept of "shields", which are "boards that can be plugged on top of the Arduino PCB extending its capabilities" [Arduino 2009b]. There is an impressive variety of shields available for adding functionality. There are shields that support 802.11 wireless standards that allow an Arduino to become a node on a wireless network, as well as Bluetooth shields, shortwave radio, shields that can drive electric motors or receive GPS signals. There are also Ethernet shields that when plugged into an Arduino provide full Ethernet support allowing an Arduino communicate with other devices using TCP/IP, be a web server or web client, the possibilities are almost limitless.

Beyond the use of shields for hardware support, several libraries exists that aim at easing communication. The serial [Arduino 2009c] and Ethernet [Arduino 2009d] libraries are well-supported by Arduino although they do not provide and mechanisms for encoding (they both send simple strings). The Messenger library [Arduino 2009e] provides the ability to receive a string and split on a single character delimiter. While this is very useful, in all but the simplest of designs, the designer must still develop the details on what each field contains, how to handle the data, and possibly write an application for encoding and sending data. Messenger library is also currently limited to serial communications only. Another well-known option is the AJSON library [Interactive Matter. 2010]. The AJSON library provides JavaScript object notation (JSON) support for Arduino. JSON is widely supported standard, and as such, the available libraries simplify the process of sending object data as JSON strings. The designer is still required to write code that will create JSON objects and code to work with a communication device such as a serial port or Ethernet network.

ADLib Implementation

The ADLib framework defines a protocol for encoding the data to be sent, an application for performing the actual encoding and transmitting of the data from a host computer over a serial or Ethernet link. The high-level requirements were defined as:

- Define a message-based protocol for encoding data. Four types of messages were defined:

Data Type	Description
Error	Used to send error text
Text	Used to send plain text
Normalized	Used to send a numeric value that has been normalized to the range 0 to 100
Numeric	Used to send a numeric value along with the minimum, maximum, and the amount of change from the last message

Table 1: Current data types

- Develop an application to run on a host computer that implements the protocol and sends the encoded data
- Develop an Arduino library for receiving the encoded data, parsing it and passing control to the user's code running on the Arduino
- Design for the easy addition of other data types
- Support device identifiers to allow one Arduino to support multiple displays
- Support Ethernet and serial communication
- Minimize the amount of code a user needs to leverage the framework
- Build prototype ambient displays to evaluate the ADLib framework

Considerations were taken in the design of the protocol and library to ensure that the addition of new data types is a relatively simple process. To that end, the only fields that are fixed are the device ID and data type ID, with the fields following being data type specific. To add an additional data type, the next sequential data type ID becomes the new ID, and the specifics of the data type are implemented in the fields following. The addition also requires a change to the library to parse the new data type and call a handling function. Again, this process is rather simple with only minor difficulty being inherent in the complexity of the new data type.

The library includes support for multiple devices, and as such, it is not limited to one device per Arduino. For example, the multiple device support allowed the construction of a display consisting of two analog panel gauges using a single Arduino and host computer application. It is potentially very valuable to have

multiple device control for two reasons; the different types of ambient displays that one could build are limitless, and the Arduino has numerous inputs and outputs which can be expanded even further using additional electronic circuits.

As discussed above, using Arduino shields provides for incredibly varied options for expanding an Arduino; however, due to time constraints, only serial and Ethernet communications are included as part of this project. In the future, support for additional communication methods such as 802.11 wireless networking, could easily be added to the framework

Protocol details

ADLib defines a simple, unidirectional, message-based protocol for sending data from a host computer to an ambient display. No provisions are made for the Arduino to send data back to the host computer.

Initially, JSON was used as the encoding method, but the amount of memory needed to store and parse the JSON strings pushed the limits of the 2 kilobytes of random access memory of the CPU used on newer Arduinos. Because older Arduinos are too limited to store and parse JSON at all, its use would limit ADLib to the more expensive CPUs. As a consequence, ADLib uses a simple ASCII encoding scheme with the data fields separated by a vertical tab character (ASCII 11).

If and when an error occurs, such as the host computer ceasing to gather data, the error data type shown in Table 2 can be used. In this scenario, an error message can be sent to the display and used to change it in order to make the error visible to the user. There are many ways to make the error visual in the ambient display and known to the end user. Some possible examples are to display the error text on an LCD, make the device flash, or the text of the message could be further parsed by the user's code to allow for different actions to be taken based on the contents of the message.

Name	Description	Possible Values	Valid Values	Notes
deviceID	ID Number of the device	0 – 255	0 – 255	
dataTypeID	ID Number of the data type	0 – 255	0	Error message is ID 0
Value	String containing error message			255 Character maximum

Table 2: Error Data Type

The text data type shown in Table 3 is used when sending plain text messages from the host computer to the ambient display. Text messages could be used to send data for display on a traditional informational display, such as a scrolling LED sign, or a persistence of vision device. Similar to the error data type, the text of the message could be further parsed by the user's code to allow different actions to be taken based on the contents of the message. In effect, this allows the user to implement his or her own protocol while still using the ADLib framework for sending the data.

Name	Description	Possible Values	Valid Values	Notes
deviceID	ID Number of the device	0 – 255	0 – 255	
dataTypeID	ID Number of the data type	0 – 255	0	Text message is ID 1
Value	String containing text message			255 Character maximum

Table 3: Text Data type

The numeric data type shown in Table 4 is used when sending a numeric value, as well as the minimum and maximum acceptable values, and the amount of change from the last message sent. Using the Dow Jones Industrial Average (DJIA) example given in the introduction, the user's data gathering application could scrape a web page to obtain the minimum and maximum value of the average for the current day, the current value, and the delta from the last time the application performed the scrape. This information could then be sent along to the ambient display.

Name	Description	Possible Values	Valid Values	Notes
deviceID	ID Number of the device	0 – 255	0 – 255	
dataTypeID	ID Number of the data type	0 – 255	2	Numeric is ID 2
minValue	Minimum value possible			Implemented as a double
maxValue	Maximum value possible			Implemented as a double
currentValue	Current value			Implemented as a double
Delta	Amount of change from previous message			Implemented as a double

Table 4: Numeric data type

The normalized data type shown in Table 5 is used when sending a numeric value that has been normalized to fall between the range zero to one hundred. Again using the Dow Jones Industrial Average (DJIA) example, the user's data gathering application could scrape a web page to get the current percentage of change in the DJIA and send the number using the normalized data type to an ambient display.

Name	Description	Possible Values	Valid Values	Notes
deviceID	ID Number of the device	0 – 255	0 – 255	
dataTypeID	ID Number of the data type	0 – 255	3	Normalized message is ID 3
Value	Normalized / Percentage	0 – 100	0 - 100	Implemented as a double

Table 5: Normalized data type

Arduino library

The Arduino library portion of the ADLib framework consists of a C header and source code file. The header file contains the function and data type definitions, as well as constants used by the library. Of the constants, the two shown in Listing 1 are modifiable by the implementer if the default values are not suitable.

```
/* User modifiable variables */
#define ADLIB_TCP_PORT            2000    // User definded TCP port
#define ADLIB_DEFAULT_DELIMETER   "\v"    // Character delimiter used from incoming network message. MUST MATCH
SENDER
```

Listing 1: User modifiable constants

ADLIB_TCP_PORT defines the TCP port the library will listen on. ADLIB_DEFAULT_DELIMETER defines the field delimiter that will be separating the individual data fields. Of course, changing the delimiter requires a similar change to the command-line program (see below).

To use the library, it must be included in the Arduino program called a "sketch" just like any normal C include file.

For this project, an important goal was to make the use of the ADLib library on the Arduino as easy as possible. The result is that it requires only three steps to use. The first step is to set up a method of communication, either serial or Ethernet. This is typically done in the setup() function of an Arduino sketch. Currently there are two possible functions to call; ADLib_StartSerial() for establishing serial

communication, or ADLib_StartEthernet(mac_address, ip_address, subnet_mask, gateway) for Ethernet. Listing 2 shows an example of setting up Ethernet communication.

```
void setup() {
  // Start the network
  ADLib_StartNetwork(mac, ip, subnet, gateway);

  // Setup data type handlers
  ADLib_registerErrorHandler(&HandleError);
  ADLib_registerNormalizedHandler(&HandleNormalized);
}
```

Listing 2: ADLib setup steps

The second step is to register call back functions in order to handle the data types used by the display. The call back functions that can be registered directly correspond to the data types the library supports, and only the data types that will be used need to be registered. All data received from the host computer is passed to the handler functions as a C structure. Listing 2 shows how to register a handler function and Listing 3 shows example functions for handling a normalized and error data types.

```
// Update the ambient display
void HandleNormalized(ADLib_NormalizedDataType *p) {
  // Access the structure with p->value

  // Top of range where color is full green
  float greenRange = 30.0;

  if (p->value < greenRange) {
    setColor(ledPins, GREEN);
  } else {
    // Set color for current percent
    setPercentToColor(ledPins, (p->value - greenRange) * (100 / (100 -
greenRange)));
  }
}

// Update the ambient display for host error
void HandleError(ADLib_TextDataType *p) {
  // Access the structure with p->text
  …
}
```

Listing 3: Registering and using data type handlers

For the third step, the user calls the `ADLib_Parse(isBlocking)` function that reads data from the communications channel previously setup, parses the data and then passes control to the registered handler. The passed parameter is a Boolean value that tells the parse function to either wait until data is received and parsed before returning, or to immediately return if no data is present. Listing 4 shows an example call to the parse function which will read the incoming data, parse and set the values of the appropriate data type structures.

```
void loop() {
  ADLib_Parse(ADLIB_BLOCKING);
}
```

Listing 4: Calling the parse function

Command line Interface

To send data to the display, command line programs were written in C# and Python (See listing 5 for usage). One of these can be easily called from an implementer's data gathering application, allowing the implementer to use almost any platform or language

```
Usage: Network
       client --address=IPADDRESS --port=PORT --device= ID --type=0 --text=STRING
       client --address=IPADDRESS --port=PORT --device= ID --type=1 --text=STRING
       client --address=IPADDRESS --port=PORT --device= ID --type=2 --min=MIN --max=MAX --current=CURRENT --
delta=DELTA
       client --address=IPADDRESS --port=PORT --device= ID --type=3 --value=NORMALIZED

Usage: Serial
       client --serial=PORT --device=ID --type=0 --text=STRING
       client --serial=PORT --device=ID --type=1 --text=STRING
       client --serial=PORT --device=ID --type=2 --min=MIN --max=MAX --current=CURRENT --delta=DELTA
       client --serial=PORT --device=ID --type=3 --value=NORMALIZED

Options:
       None
```

Listing 5: Command line usage

Evaluation

While ambient displays are not the focus of this project, to evaluate the ADLib framework, we built three ambient displays that would test out the features and functionality of the library. The first is used to monitor the CPU utilization of a VMware cluster hosting approximately 400 virtual servers. The ambient display shown in Figure 2 consists of a translucent plastic flower pot with an Arduino, which receives its data via the Ethernet. It is connected to a prototype circuit which is built on perforated board mounted in the base. Three red, green and blue (RGB) LEDs on the breadboard are controlled by the Arduino to indicate the current utilization of the VMware cluster. The flowerpot glows green while the CPU utilization of the virtual machine farm is below 50% utilization, and then begins to fade from green to red between 50% and 100% CPU utilization. An evaluation of the effectiveness of this ambient display is outside the scope of this project (see Future Work below). The intent of this paper is to evaluate the ADLib framework and how it is able to simplify the development of ambient devices.

Figure 2: Flower pot ambient display

In order to gather the necessary raw data on all of the CPUs and to normalize into a percentage of overall CPU utilization of the VMware cluster, a data collection program was written in Windows PowerShell . The script then passes this value to the ADLib desktop application for encoding and transmitting to an Arduino. On the Arduino, a program structured like the one shown in Listing 1 was created to update the LEDs appropriately. The data collection program consists of seventeen 17 lines of PowerShell code. Of those, only one was necessary for communicating with the ADLib desktop application. This represents only five percent of the data collection program. Similarly, the Arduino program consists of two hundred lines of code, of which only five were necessary for using the ADLib library. This represents two percent of the Arduino program as shown in Figure 3.

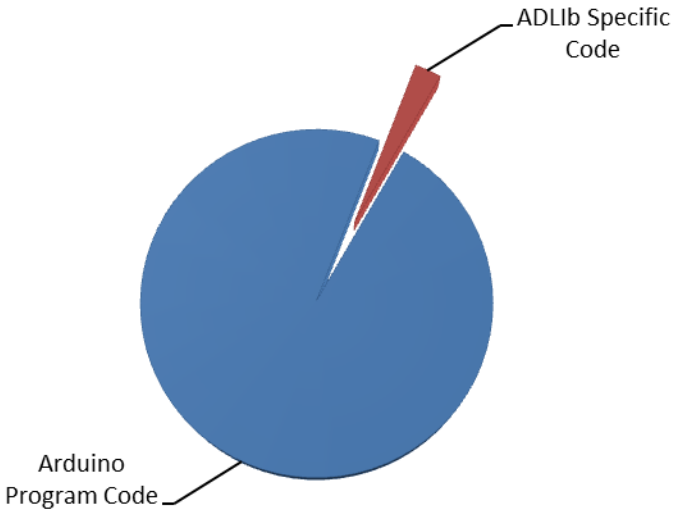

Figure 3: Lines of code to run display compared to ADLib code

The second ambient display pictured in Figure 4 is an alternative to the flower pot for monitoring VMware cluster CPU utilization. It consists of two analog panel gauges; one for overall CPU utilization, the other for overall memory utilization. This display is connected to an Arduino which receives its data from the host computer over a serial connection. The gauges swing from zero to one hundred percent with a resolution of 255 steps.

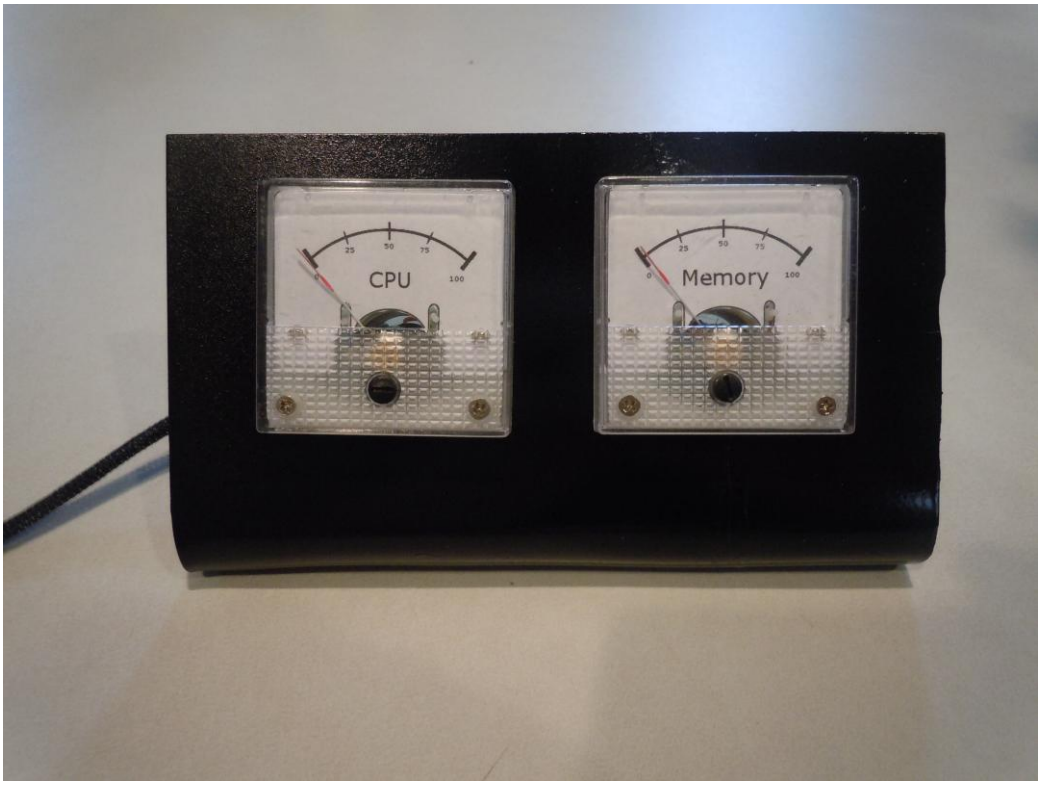

Figure 4: Analog gauge ambient display

The data collection program written for the flower pot display was reused to drive the analog gauges. Two additional lines of code were required, one to get the raw memory data and to normalize it from zero to one hundred percent, and the other to pass the data to the ADLib desktop application for encoding and transmitting to the Arduino. On the Arduino, a program structured like the one shown in Listing 1 was created to update the gauges appropriately. The data collection program consists of nineteen lines of PowerShell code. Of those, only one was necessary for communicating with the ADLib desktop application. This represents five percent of the data collection program as shown in Figure 5. Similarly, the Arduino program consists of one hundred lines of code, of which only five were necessary for using the ADLib library. This represents five percent of the Arduino program.

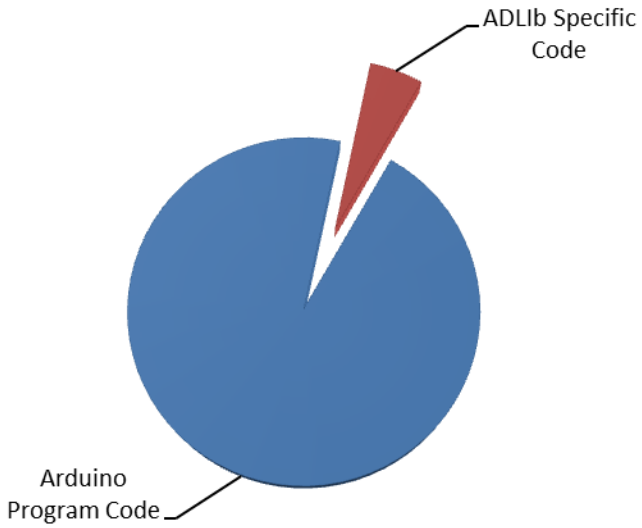

Figure 5: Lines of data gather code compared to send code

The third type of ambient display shows information regarding school closings. A data collection program reads school closing information from a web page and sends that information to the ambient display. An LED glows green if school is open, blue if there is a one hour delay, orange if there is a two hour delay or red if school is closed for the day.

A program was written in the Python programming language to gather school closing information from a local TV station's web page and pass that information to the ADLib desktop application as a normalized value. Of the fifty six lines of Python code written, only one line was necessary to execute the application on the host computer in order to send the encoded data to the display. This consists of less than two percent of the program. On the Arduino side, a program was written to accept incoming data and change the color of an LED accordingly. The Arduino program consists of almost two hundred lines of code, of which four (two percent) were required by ADLib.

Conclusions and Future Work

While the initial design goals detailed in the problem definition section have been met, there is still room to add features and flexibility into the ADLib library.

Additional features already under consideration are:

- Support for implementing timed events
- Moving the library from C to C++
- Revisiting JSON support
- Additional communication methods

Being an eight bit CPU with limited memory, these libraries can quickly exhaust available memory, which can cause many issues from instability to an outright hard lockup. The limits of the hardware will need to be considered in in future work as it may limit what can be implemented.

An important issue that needs to be considered is the issue of "trust" in ambient displays. Using the VMware flowerpot display, while everything may appear fine with it glowing green, how can we be sure that the data collection application running on the host is still functioning and sending data? What if there is a network issue between the two? One cannot know for sure if the status is truly accurate. [Antifakos et al. 2005] and others have discussed the issue of "trust" in an ambient display. One way trust may be realized is by providing the user some indication that the system is still functioning normally. One option for the flowerpot display considered was to place a ring of Electroluminescent Wire or LEDs around the inside, and have the Arduino control the color or brightness of the wire or LEDs to indicate the state of the environment. To implement an option such as this requires complex code to implement timers as well as logic to detect the state of the environment. It may be possible to extend the ADLib Arduino Library to support the ability to show the system is still "alive".

Moving to C++ would bring the features of object oriented programing like abstraction, data encapsulation, inheritance and polymorphism to the library potentially allowing an even simpler interface to the data and encapsulating the communication methods into a common class making the use of additional communication methods as similar as possible. Moving to C++ may make it possible to abstract the library away from the Arduino platform making it usable by other embedded systems such as AVR, PIC, PSOC, etc.

Revisiting JSON could further simplify the addition of new data types by providing a common parsing method that can be used by all data types. Combined with C++, data types could be completely external to the library in a separate class that implements common methods and properties.

Adding additional support for different communication methods such as Wi-Fi and Bluetooth would be advantageous. The way the library is structured, adding additional communication devices is not a difficult undertaking. To date, Wi-Fi support for the WiFly shield [Spark Fun 2011] and Async Labs Wi-Fi shields [Async Labs 2011] have been considered.

ADLib set out to simplify building ambient displays, and the result is a simple to use framework that handles both the details of encoding data and the heavy lifting of transmitting the data to a display very well. It only takes a few lines of code leverage, and if needed it can be extended with minor coding. All of this makes ADLib an excellent fit for building ambient displays using the Arduino prototyping platform.

Bibliography

1. Mark Weiser, John Seely Brown. 1995. Designing Calm Technology. http://www.ubiq.com/weiser/calmtech/calmtech.htm
2. William R. Hazlewood, Kay Connelly, Kevin Makice, and Youn-kyung Lim. 2008. Exploring evaluation methods for ambient information systems. In CHI '08 extended abstracts on Human factors in computing systems (CHI '08). ACM, New York, NY, USA, 2973-2978.
3. Mark Weiser. 1993. Some Computer Science Issues in Ubiquitous Computing. In Comm. ACM, vol. 36, no. 7, July 1993, pp. 75-84.
4. Jennifer Mankoff, Anind K. Dey, Gary Hsieh, Julie Kientz, Scott Lederer, and Morgan Ames. 2003. Heuristic evaluation of ambient displays. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '03). ACM, New York, NY, USA, 169-176.
5. Arduino 2009. Arduino prototyping platform. http://www.arduino.cc
6. Arduino 2009. Arduino Shields. http://www.arduino.cc/en/Main/ArduinoShields
7. Arduino, 2009. Arduino Serial Library. http://arduino.cc/en/Reference/Serial.
8. Arduino, 2009. Arduino Ethernet Library. http://arduino.cc/en/Reference/Ethernet.
9. Arduino, 2009. Arduino Messenger Library. http://arduino.cc/playground/Code/Messenger.
10. Ambient Devices 2010. Ambient Orb. http://www.ambientdevices.com/cat/orb/orborder.html
11. Stavros Antifakos, Nicky Kern, Bernt Schiele, and Adrian Schwaninger. 2005. Towards improving trust in context-aware systems by displaying system confidence. In Proceedings of the 7th international conference on Human computer interaction with mobile devices & services (MobileHCI '05). ACM, New York, NY, USA, 9-14.
12. Spark Fun 2011. WiFly Shield and library. http://www.sparkfun.com/products/9954
13. AsyncLabs 2011. WiShield 2..0 http://asynclabs.com/store?page=shop.product_details&product_id=26

Appendix A: Source code

The source code for ADLib and the supporting command line applications have been published to a public source code repository. This allows anyone to use the framework for building their own displays or to make changes or extend.

The framework is located at GitHub and can be accessed using a web browser and the URL below:

https://github.com/drobertadams/ADLib

The framework can also be accessed with the source code control application Git using:

git://github.com/drobertadams/ADLib.git

Details on Git can be found at:

http://git-scm.com/