Proc. Of the 2nd International E-Conference on Advances in Engineering, Technology and Management - ICETM 2020
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-189-4 DOI: 10.15224/978-1-63248-189-4-10

A Simple Sequential Data Protocol for IoT Applications

Đorđe Herceg, Ivan Petković, Dejana Herceg

***Abstract*— We present a simple sequential data protocol (SSDP) - an implementation of a lightweight message encoding and exchange protocol for Arduino-based sensor devices. The protocol supports simple data types, strings and binary arrays with low overhead. It defines several types of messages used to discover device metadata, send and receive sensor reading values and runtime parameter values, as well as alarms. The SSDP Arduino library is simple and has a low memory footprint. In conjunction with mesh networking, SSDP can be used to quickly and efficiently build sensor networks based on inexpensive Arduino hardware.**

***Keywords*—sensor networks, Arduino, microcontroller, protocol, message exchange, IoT**

I. Introduction

IoT applications and sensor network applications are the foundation of todays connected world. There exist a multitude of different microcontroller platforms and application frameworks together with many communication devices, standards, and protocols. The Arduino [5] family of single-board devices, originally intended as a learning platform, has outgrown its purpose and thanks to its affordability, simplicity of designing and programming, is today used to build IoT devices and sensor devices. There is ample offering of sensors, actuators, displays, memory, and communications devices which can be connected to the Arduino. The Arduino software is simple to use and it can be extended with numerous third-party libraries, ranging from low-level device drivers to RTOS to complete data-logging and IoT solutions. Internet connectivity can be achieved using technologies such as WiFi, LoRa and GSM.

A prototype Arduino-based sensor device is shown in Figure 1. It is built around the Arduino Nano board, together with a real-time clock module and a 2.4GHz, 2Mbit/second digital transceiver module NRF24L01. Sensors are connected to the two terminal brackets on top of the PCB. The Arduino Nano board has the 8-bit Atmel ATmega328 CPU at 16MHz, with 32Kb program flash memory, 2Kb RAM, 1Kb EEPROM, 22 digital I/O pins and 8 analog input pins. Actuators such as relays, motor drivers, valves etc. can also be connected.

This paper presents Simple Sequential Data Protocol - SSDP - an implementation of a lightweight message encoding and manipulation protocol for Arduino-based sensor devices. The main goal behind SSDP was to provide an encoding scheme for transfer of binary data between devices, together with a software component which would maintain device state in a consistent manner. The SSDP software maintains a storage for sensor readings and configuration parameters on the device, as well as a set of commands which enable remote assignment and querying of the device's state. During the initialization phase of the firmware, the developer uses metadata to specify a set of variables which comprise the device's state (sensor readings and runtime parameters). Based upon the metadata, the library takes care of maintaining the values, interpreting incoming commands, generating response messages, and raising data change notifications. The SSDP library consumes about 5Kb of program space and less than 500 bytes of RAM when compiled.

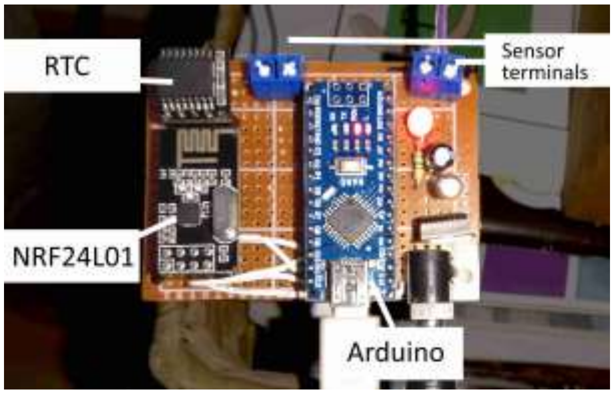

Figure 1. A prototype sensor device with a RF transciever

The motivation for this work stems from a specific need pertinent to deployment of a large number of sensor devices at one location. The devices need to be able to connect to the Internet in order to report sensor readings and alarms, and also perform certain actions when commanded by the server. It is common practice in such cases to set up a hub device surrounded by many sensor devices and connect them in a self-configuring wireless mesh network. There are many examples of successfully employed mesh networks, one example being honeybee farming. Honeybee hives are usually kept in remote locations, without access to electrical grid and possibly with weak Internet access. Hives are placed on electronic scales and equipped with thermometers, humidity meters and accelerometers to provide real-time monitoring and tampering detection. Simple Arduino devices with mesh networking capability, running on batteries, are used to monitor each individual hive. The hub device is equipped with a GSM modem, placed in a position which offers the best Internet access and access to the mesh network. Its task is to relay data between sensor devices and the backend located in the cloud, using industry-standard communications infrastructure, such as MQTT, HTTPS etc.

Proc. Of the 2nd International E-Conference on Advances in Engineering, Technology and Management - ICETM 2020
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-189-4 DOI: 10.15224/978-1-63248-189-4-10

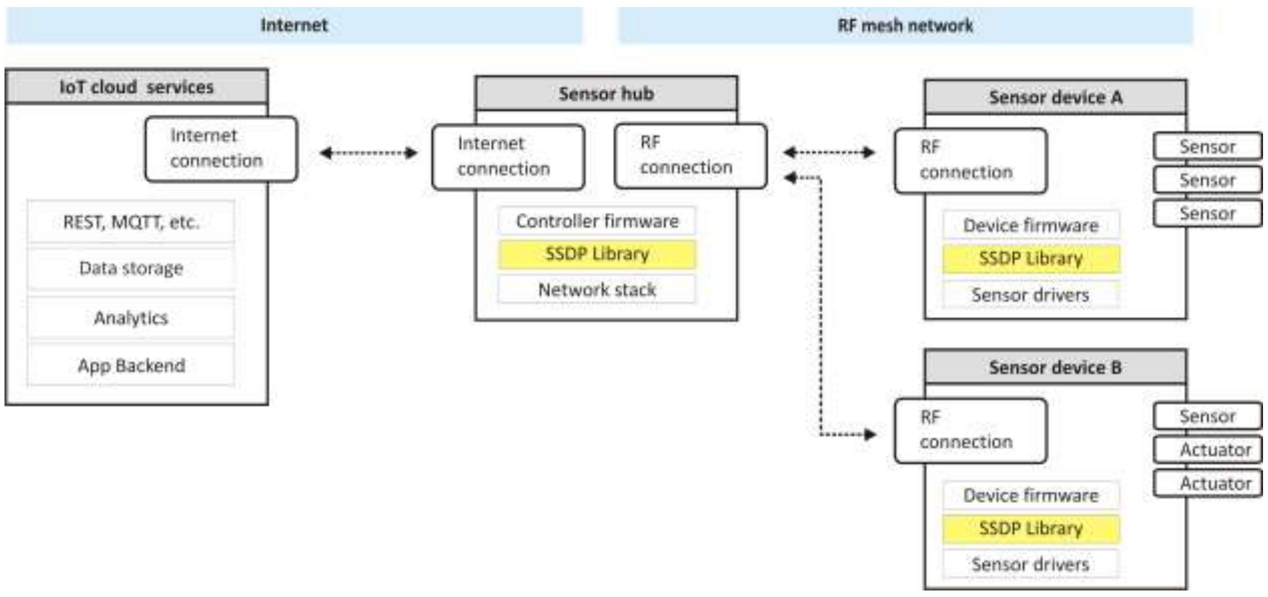

Figure 2. Network topology based on SSDP-enabled sensor devices

We based our solution on the inexpensive Arduino Nano hardware. Thanks to the low-power capabilities of the Atmel CPUs, a battery-powered Arduino can run for a week and up to several months. Mesh network was built upon the Nordic Semiconductor NRF24L01 digital transceiver, which enables digital communication in the 2.4GHz band via short unencrypted messages up to 32 bytes in length, at the speeds from 250Kbit/s up to 2Mbit/s. This module was chosen because it offers low power consumption, nearly instantaneous startup times and there exist a reliable mesh networking library for it [8].

Based on the chosen hardware we had to decide which communication protocol to use between the sensor devices. The limited amount of RAM on the Arduino (2KB) ruled out high level formats such as XML and JSON. Other protocols, such as MQTT and Protocol Buffers were more adequate, but due to specific requirements and hardware constraints, we decided to develop a custom solution which would satisfy the following conditions:

- Minimize overhead for message encoding,
- Require no additional RAM to encode/decode messages, besides the necessary input and output buffers,
- Consider the limitations of the hardware,
- Minimize the size of the program code.

The following terms are used throughout the paper:

Controller or **hub** – IoT-enabled device which acts as the Internet gateway and is tasked with controlling a group of sensor devices. It has constant or on-demand Internet access and a means of communicating with other sensor devices, thus functioning as a message relay between the Internet and sensor devices.

Sensor device – Arduino-based device, equipped with sensors for monitoring various parameters from the environment and optional actuators to interact with the environment. It connects to the controller via a mesh network, but has no Internet access. Each sensor device runs an instance of the SSDP software.

Mesh network – Ad-hoc network of sensor devices which provides dynamic configuration, message delivery and routing, as well as easy scalability, but without delivery guarantee.

Pin – numerical, textual, logical, binary, or other representation of a sensor reading or parameter on a sensor device. The name stems from hardware pins on microcontrollers. Each device has a set of variables in its firmware which hold the pins' values. This set is maintained by the SSDP software.

Command message – message sent from the controller to a sensor device, which triggers a certain action and possibly a response.

Status message – message containing pin values or other information from the sensor device.

II. Related work

Cisco PacketTracer [6] network simulation software has an extensive Arduino software library which encompasses the core methodology for developing IoT sensor networks based on Arduino-like single board computers. Tuxedo Message Buffers [9] is a message routing system produced by Oracle, with support for multiple message formats, one of which is a self-describing binary format. It enables transfer of atomic values, structs, lists, dested data etc. The Tuxedo software enables communication over domain boundaries, utilizing different communication technologies, as well as message queuing, and events. Google Protocol Buffers [7] is an flexible

Proc. Of the 2nd International E-Conference on Advances in Engineering, Technology and Management - ICETM 2020
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-189-4 DOI: 10.15224/978-1-63248-189-4-10

and extensible platform for serializing structured data. It is platform- and language-neutral and aims to provide a memory-efficient binary format. Message types are specified by writing metadata description files and compiling them with the protocol buffers compiler, thus producing a class that implements automatic encoding and parsing of the protocol buffer data.

In [3] the authors demonstrate the use of low-cost microcontroller-based sensor devices for data acquisition and logging, focusing on the real-world application of open-source hardware and software. The authors argue that the Arduino platform has great potential for implementation in scientific research applications, and can empower researchers with flexible, inexpensive tools for expanding their data-collection, automation and control capabilities. In [1], the authors demonstrate a multi-sensor monitoring system, consisting of open-source and inexpensive technology, which is used to collect high spatiotemporal resolution information on the presence of water and the occurence of flow in small temporary streams in mountainous headwater catchments. Their conclusions corroborate the need for wireless data transfer as a direction of future development. Further discussion on various sensor setups and their applications can be found in [2], [4].

III. Protocol description

SSDP is a simple binary protocol based on the exchange of command messages and status messages (Figure 3). Due to limited memory and computational resources of the hardware, messages are formed in such a way that they can be parsed in one pass, without the need for additional buffers in the RAM. A usual message has a header, followed by a sequence of values, tagged with pin IDs and data types, except when data types can be inferred from the message header itself.

Since in real-world scenarios, mesh networking may not be reliable, and devices may go offline at times, the "best effort" delivery strategy is a reasonable solution.

The communication strategy between the controller and a sensor device can be formulated as follows:

- The controller sends a command to the device.
- Device processes the received command. If an answer is requested, the device generates the appropriate status message and transmits it back to the controller.
- Status messages containing sensor readings can be periodically transmitted to the controller, out of band.
- Alarms trigger specific status messages to be transmitted to the controller, out of band. For example, too high a temperature in a beehive or sudden jolts and orientation changes on the accelerometer can trigger an alarm.

As there is no delivery guarantee, messages can be lost. The controller must take note of the devices which did not update their status for a long time and send the appropriate alert to the backend. Alarms raised on sensor devices must be reported repeatedly, for reasonably long periods of time, thus increasing the possibility of reception. Various handshake strategies can be used to signal reception of important messages between devices. However, they are not integral to the protocol.

A. *Data Types*

The following data types are natively recognized: Bool, SByte, Byte, Int16, UInt16, Int32, UInt32, ByteArray, String, Float, Time16, Date16.

The String and ByteArray data types follow the Pascal-style string format. The length of the value is kept in the leading byte, followed by an array of length characters (bytes). This approach enables strings to contain null characters and simplifies message construction and parsing. An example of a string 'Data' is shown in Figure X.

Offset	0	1	2	3	4
Data	4	'D'	'a'	't'	'a'

Figure 3. The word 'Data' stored as string

The Time16 data type packs time in two bytes, with a resolution of 2 seconds. As the SSDP is meant to be used in localized sensor networks, there is no provisioning for time zones. Such information, if needed, must be provided by the gateway or the backend server.

Bit	7	6	5	4	3	2	1	0
High byte	H	H	H	H	H	M	M	M
Low byte	M	M	M	M	S	S	S	S

Figure 4. The Time16 data type

The Date16 data type packs dates in two bytes, with epoch starting in the year 2000. Thus, dates from 1/1/2000 to 12/31/2127 are supported.

Bit	7	6	5	4	3	2	1	0
High byte	Y	Y	Y	Y	Y	Y	Y	M
Low byte	M	M	M	D	D	D	D	D

Figure 5. The Date16 data type

Additional data types can be dealt with in two ways. Fixed-length data types can be supported by extending the binary parser and writer components within the SSDP library, while variable-length data types can be stored within the ByteArray data and handled appropriately in the software.

B. *Message structure*

Messages start with a header byte (Table 1, Table 2), followed by data specific to the message type.

TABLE I. COMMAND MESSAGE HEADERS

Command	Name	Description
'S'	Set	Sets parameter values
'Q'	Query	Queries parameter values
'I'	Info	Queries metadata information
'V'	Version	Queries version information

Proc. Of the 2nd International E-Conference on Advances in Engineering, Technology and Management - ICETM 2020
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-189-4 DOI: 10.15224/978-1-63248-189-4-10

TABLE II. STATUS MESSAGE HEADERS

Status	Name	Description
'@'	Value	Parameter value
'#'	Metadata	Parameter metadata
'!'	Error	Error code

A pin value packet (PVP) is a special case of data, used to transfer one pin value (Table 3). A pin data packet consists of a pin ID, data type descriptor and actual data.

TABLE III. PIN VALUE PACKET STRUCTURE

Offset	Name	Length	Description
0	PinID	1 byte	Unique pin ID
1	Type	1 byte	Data type descriptor
2..N+2	Value	N bytes	Actual pin value

Multiple PVPs can be concatenated to enable transfer of a sequence of values within a message.

C. *Commands*

Commands are sent from the controller to sensor devices. The command packet is processed by the SSDP library, pin values are automatically updated, and the firmware is notified of the changed values. The firmware can decide to send a status message as a response to a command; out-of-band status messages can also be sent periodically or when a certain trigger at the device is activated.

1) Version

The Version command requests the identification number, software version and name from the device. The command is used to identify a device, but it can also be used to check whether the device is online. The DeviceID field usually contains a Date16 value followed by a Time16 value, designating the manufacture time and date for the device. Developers, however, can decide to use a different scheme for generating device IDs.

Command format: `'V'`

Response: Status message with the following payload:

Offset	Field	Type	Description
0	DeviceID	UInt32	Unique device ID
3	Version major	Byte	
4	Version minor	Byte	
5	Name	String	Device name

2) Info

The Info command requests pin metadata from the device. There are two variants. The first one requests the number of pins and the second one requests metadata for one or more pins, specified by the PinIndex values. In this way the controller can discover the information about the pins on each available device.

Command format: `'I'`

Response: Status message with the following payload:

Offset	Field	Type	Description
0	NumPins	Byte	Number of pins on the device

Command format: `'I' PinIndex(byte) {PinIndex(byte)}`

Response: Metadata status message, containing one or more of the following payload segments:

Offset	Field	Type	Description
0	PinID	Byte	Pin ID
1	PinType	Byte	Pin data type
2	PinName	String	Pin name

3) Query

The Query command requests actual pin values from the device. Pins are specified by one or more PinID values.

Command format: `'Q' PinID(byte) {PinID(byte)}`

Response: Value status message, containing one or more of the following PVPs:

Offset	Field	Type	Description
0	PinID	Byte	Pin ID
1	PinType	Byte	Pin data type
2	PinValue	*PinType*	Actual pin value

In the case of variable-length data types (String and ByteArray) the PVP has the following format:

Offset	Field	Type	Description
0	PinID	Byte	Pin ID
1	PinType	Byte	Pin data type
2	Length	Byte	Value length in bytes
3	Value	*Length*	Actual pin value

4) Set

The Set command sends one or more pin values to the device, as an array of PVPs. The device verifies the datatype of each passed value versus the corresponding pin's metadata, and, if successful, assigns the new value to the pin.

Command format:

```
'S' PinID(byte) PinType(byte) PinValue(PinType)
{PinID(byte) PinType(byte) PinValue(PinType)}
```

Response: This command produces no response.

IV. Application example

The example demonstrates a typical use case of SSDP in an Arduino project. The library is included in the project with the following directive:

```
#include <SSDP.h>
```

Runtime parameters and sensor readings on the device are stored in variables held in RAM. First, a structure is defined which holds these values. The fields can be accessed and modified by the device's firmware during program execution.

```
struct state_t {
  bool fan;
  float temp;
  float humi;
  float weight;
  uint8_t alarm;
} st;
```

To conserve RAM, field names are stored in program memory using the PROGMEM modifier.

```
const char strDeviceName[] PROGMEM = "IoT Device";
const char strTime[] PROGMEM = "Fan";
const char strTemp[] PROGMEM = "Temp";
const char strHumi[] PROGMEM = "Humi";
const char strWeight[] PROGMEM = "Weight";
const char strAlarm[] PROGMEM = "Alarm";
```

Next, the ConfigInfo object is created, which provides the core functionality of the SSDP. Metadata for the fields is registered in the SetupCfgFields() function. A global callback "gcb" is registered, as well as the pin-specific callback named "tcb" (Listing 3). Pin callbacks are invoked each time a value change is detected in the corresponding variable, and global callback is invoked on each value change.

```
ConfigInfo cfg;
void SetupCfgFields() {       // called from Setup()
  cfg.init(5);
  cfg.initField(101, strFan, &st.fan, pdtBool, pmOutput);
  cfg.initField(201, strTemp, &st.temp, pdtFloat, pmInput, tcb);
  cfg.initField(202, strHumi, &st.humi, pdtFloat, pmInput);
  cfg.initField(203, strWeight, &st.weight, pdtFloat, pmInput);
  cfg.initField(204, strAlarm, &st.alarm, pdtByte, pmInput);
  cfg.setEventHandler(gcb);
}
```

After initialization, the library is ready for use. The developer needs to implement the code to route incoming command packets to the config object, using the processCommand() method. The SSDP library takes care of updating the field values and raising appropriate notifications.

```
  uint8_t rez =
    config.processCommand(buffer, buflen, dest, destLen);
```

The processCommand() method invokes the four basic commands, Version, Info, Set and Query. It builds the appropriate response message and stores it into the destination buffer. The length of the response is returned as the result. It is the responsibility of the developer to transmit the response. The developer can choose to generate out-of-band status messages and transmit them to the controller.

In this way the developer can write his/her firmware around the SSDP library, with the single requirement that messages are regularly routed into and out of the library.

V. Conclusions

As the need for implementation of large sensor networks grows, economy and simplicity of development have become essential. Arduino devices are a suitable platform for the development of low-cost sensor devices, which support a simple and economical communications infrastructure. The SSDP protocol is a simple, low-footprint solution for command and data exchange between devices in a mesh network.

References

[1] R. S. Assendelft, H .J. I. van Meerveld, "A low-cost, multi-sensor system to monitor temporary stream dynamics in mountainous headwater catchments", Sensors, 19(21), pp. 46-45; 2019, https://doi.org/10.3390/s19214645

[2] D. Bri, H. Coll, M. Garcia, J. Lloret, "A multisensor proposal for wireless sensor networks," 2nd International Conference on Sensor Technologies and Applications, Cap Esterel, 25-31 August 2008, pp. 270-275.

[3] D. K. Fisher, P. J. Gould, "Open-source hardware is a low-cost alternative for scientific instrumentation and research", Modern Instrumentation, vol. 1, pp. 8-20, 2012.

[4] K. A. Noordin, C. C. Onn and M. F. Ismail, "A low-cost microcontroller-based weather monitoring system," CMU Journal, Vol. 5, No. 1, pp. 33-39, 2006.

[5] Arduino, "An Open-Source Electronics Prototyping Platform" http://www.arduino.cc (Accessed May 2020)

[6] Cisco PacketTracer – Network Simulation Tool, https://www.netacad.com/courses/packet-tracer (Accessed May 2020)

[7] Google Protocol Buffers, https://developers.google.com/protocol-buffers/ (Accessed May 2020)

[8] RF24Mesh, A user friendly mesh overlay for sensor networks using RF24Network and nRF24L01 radio modules, https://tmrh20.github.io/RF24Mesh/ (Accessed May 2020)

[9] Tuxedo Message Buffers, https://www.oracle.com/middleware/technologies/tuxedo.html (Accessed May 2020)

Đorđe Herceg
Faculty of Science, University of Novi Sad
Serbia

Ivan Petković
Faculty of Electronic Engineering, University of Niš
Serbia

Dejana Herceg
Faculty of Technical Sciences, University of Novi Sad
Serbia