Systems and Software Design Description (SSDD)
Communication Protocol Description (CPD)
Device Design Description (DDD)
And
Test Plan Documentation(TPD)

For

Autonomous COTS Bots

Version 1.0
09 May 2014

Prepared For
Bruce Bolden
bruceb@cs.uidaho.edu
JEB 232, University of Idaho
Moscow, ID 83844-1010

Prepared By
Team AutoCOTS
Chris Waltrip
walt2178@vandals.uidaho.edu
Robert Meine
mein1156@vandals.uidaho.edu
University of Idaho
Moscow, ID 83844-1010

Autonomous COTS Bots Record of Changes

Change Number	Change Date	Change Location	A M D	Brief Description	Author (Initials)
01	10-03-2014	All	A	Created template	CW
02	13-04-2014	All	M	Changed all tables to new format	CW
03	13-04-2014	Requirements Traceability	D	Removed Requirements Traceability for now.	CW
04	23-04-2014	Ch. 2–6	A	Initial Creation	RM
05	25-04-2014	Ch1	M	Populated subsections	CW
06	26-04-2014	Ch. 2–6	M	Corrections and updates	RM
07	26-04-2014	Appendix A–B	A	Initial creation	RM
08	29-04-2014	Ch. 7	A	Initial Creation	RM
09	29-04-2014	Ch. 8	A	Converted document from Microsoft Word format to LaTeX	RM
10	06-05-2014	Ch. 8	M	Finished conversion	RM
11	06-05-2014	All	M	Combined all documents into one master document	CW
12	07-05-2014	Preamble	M	Removed pagestyle	CW
13	13-05-2014	All	M	Proofread and rewrote entire document	CW
14	13-05-2014	Bibliography	A	Added references and citations	CW

Legend

A	Added
M	Modified
D	Deleted

Table of Contents

Part I

System and Software Design Description

Chapter 1

System and Software Design Description (SSDD)

1.1 Introduction

1.1.1 COTS Bots

1.1.2 Our Task

1.1.3 Identification

Title : System and Software Design Description (SSDD)
Revision: 0.3 Software: COTS Bots Communication Protocol Release : 0.3

This can be a standalone document and if so has no identification numbers other than the version number. Currently, this document is chapter 1 of the COTS Bots documentation written by Team AutoCOTS.

1.1.4 Document Purpose, Scope, and Intended Audience

Document Purpose

The COTS Bots communication protocol is being developed according to the set of requirements set forth by Dr. Terence Soule and Dr. Robert Heckendorn which are outlined in the Autonomous COTS Bots System and Software Requirements Specification (Rev. 0.00?). This document provides detailed information regarding the design implementation of these requirements.

Document Scope and/or Context

This document includes information regarding the design and components of the COTS Bots Communication Protocol. The structure and interactions between devices, along with the rationale behind the design decisions is included.

Intended Audience for Document

This document may be referenced by the following people or groups:

- Students and Faculty of the Computer Science Department at the University of Idaho;
- Students, Faculty and Others involved with the Interdisciplinary Capstone Design of the College of Engineering at the University of Idaho;
- Anyone affiliated with the COTS Bots group at the University of Idaho.

This document may also be referenced by anyone given express permission by a member of an above group.

1.1.5 System and Software Purpose, Scope and Intended Users

System and Software Purpose

The COTS Bots Communication Protocol is intended to allow an unspecified number of COTS Bots to coordinate with one another to complete various tasks.

System and Software Scope/or Context

The COTS Bots Communication Protocol will be designed to provide the ability for an unspecified number of COTS Bots to create an ad-hoc network together and to use this network to communicate with one another to receive instructions, assign tasks and report the status of events.

Intended Users for the System and Software

This protocol is intended to be used by anyone who is interested in using a number of COTS Bots for any given purpose. The initial releases of this protocol will be limited to the COTS Bots program at the University of Idaho and to people that are given access to the protocol by the COTS Bots program at the University of Idaho.

1.1.6 General Definitions, Acronyms, and Abbreviations

The following definitions are used throughout the entire document. Individual chapters will also have specialized definition sections as needed.

Term or Acronym	Definition
COTS	Commercially available Off-The-Shelf (or Commercial-Off-The-Shelf) products are any products that are sold in substantial quantities in the commercial marketplace and that isn't bulk cargo.[1]
LAN	Local Area Networks are computer networks that connect computers together within a limited area (e.g. a house or building).
WLAN	Wireless Local Area Networks are LANs where devices can use some wireless method (radio waves typically limited to a frequency range).
WAN	Wide area networks are networks that cover a broad area, such as an entire city or region.
Point-to-Point Topology	A network topology that establishes a connection between two nodes. Messages from one node to another have to each node in the network until they reach their destination.
Star Topology	A network topology where a coordinator node exists that each other node (called endpoints) connects to. The coordinator is responsible for delivering messages between two endpoint nodes.
Partially-Connected Mesh Topology	A network topology in which each node is connected multiple other nodes. Messages are routed to their destination by using intermediary nodes that know where the destination (or at least the next hop in the chain) is located.
SSDD	The System and Software Design Description document outlines the entire design for a project with references to the different requirements documents. In this document it serves as the introduction of the protocol that is then detailed in subsequent documents.[3]
CPD	The Communication Protocol Description document details exactly how the communication protocol that we are designing is defined, including protocol requirements, formal rules and notation and what testing should be implemented.
Device Design	The Device Design document details the requirements of devices that will use our protocol and how that protocol needs to be implemented.
TPD	The Test Plan Document lists out what exactly needs to be tested in order to verify that all of the project components work correctly. It also outlines requirements from the customer that must be fulfilled before the project is considered complete.

1.1.7 Document References

This document makes heavy use of the SSDD Template created by the Center for Secure and Dependable Systems (CSDS) at the University of Idaho.[3]

1.1.8 Document Overview

Chapter 1 This chapter gives a broad overview of the design decisions and constraints imposed by the operational environment and system requirements, and then identifies the system stakeholders and lists their concerns and mitigations to those concerns.

Chapter 2 This chapter's intent is to fully describe the structure of the protocol for the COTS Bots communications, including a discussion of both the current capabilities and structure, but also a discussion of the methodology, notes, and the methods to modify the protocol.

Chapter 3 This chapter discusses the encoding features of the COTS Bots communication protocol.

Chapter 4 This chapter describes all of the general requirements for devices that will support this protocol.

Chapter 5 This chapter contains a discussion and description of the testing methodology and tools available for the COTS Bots communication protocol.

Chapter 6 This chapter describes the design of the software for the Arduino Romeo v.2. This is the complete the requirements for documentation as outlined in chapter 4.

Chapter 7 This chapter provides a clear and defined test plan for the COTS Bots senior design project. It contains all the necessary information to unambiguously test all parts of the COTS Bots project, including software and hardware. It also provides a list of what should be tested, the requirements for any testing, how a requirement should be tested, and how the results should be presented and interpreted.

Appendix A This appendix contains complete descriptions of all of the Packet Data Units (PDUs) defined for the COTS Bots communication protocol.

Appendix B This appendix contains all of the capabilities available under the current version of the COTS Bots communication protocol in SMI format.

1.1.9 Document Restrictions

This document is for LIMITED RELEASE ONLY to UI CS personnel working on the project and College of Engineering Senior Capstone faculty.

1.2 Constraints and Stakeholder Concerns

1.2.1 Constraints

Environmental constraints

The COTS Bots Communication Protocol itself does not pose any environmental constraints. However, the electronics used will have environmental constraints, including required weather conditions and the indoor and outdoor ranges of the wireless devices.

System requirement constraints

The packets created by the COTS Bots Communication Protocol must be small enough that all intermediary devices in the network have the capability to perform their required tasks with the packet while not having a noticeable impact on the performance of other packets on the network. This includes, but is not limited to, devices that will forward the packet to another device, devices that will parse packets and devices that will write new packets.

User characteristic constraints

The typical user of the protocol should be able to understand that format of the packet and how to send data in the correct format.

1.2.2 Stakeholder Concerns

Hardware Concerns

Stakeholder Concern	Stakeholder List	Concern	Mitigation Mechanism	Appropriateness of the System
Low Cost	Users and Developers	The monetary cost of the parts should remain low.	Low Cost Additions	The hardware that is added is inexpensive as long as the COTS Bot is using an Arduino-based microcontroller. This isn't guaranteed otherwise.
Simple and Quick to Build	Users	The COTS Bots should not take very long to build	Minimal soldering	Most parts should not need to be soldered. The parts will use shields to stack parts on top of one another. This is slightly more expensive, but makes assembly much simpler.

Software Concerns

Stakeholder Concern	Stakeholder List	Concern	Mitigation Mechanism	Appropriateness of the System
Maintainability	Developers	The protocol should be robust enough to not require maintenance often.	Flexible and Extensible Protocol	The protocol is designed to be extensible from the very beginning. The protocol itself should only need to be modified if a radical feature such as encryption were to be added or if a vulnerability is found.
Simplicity	Users	The protocol should be able to handle a large myriad of unknown requests	Built-in Functionality	The protocol has a "data" payload. The protocol does not care what the payload is or how the payload is handled.

Part II

Communication Protocol Description

Chapter 2

Structure

2.1 Introduction

The intent of this document is to fully describe the structure of the protocol for the robot communications. This includes a discussion of both the current capabilities and structure, but also a discussion of the methodology behind the use of this protocol structure, notes, and the how to modify the protocol itself in the future.

This protocol is based on, but is not a copy of the SNMP protocol. In general, the broad intent and structure are similar, however, some of the details are different. For example, the OIDs used by SNMP must be unique world-wide, while the OIDs in the COTS Bots protocol do not need to be unique outside of the protocol.

This protocol emulates the various parts of the SNMP protocol, including, but not limited to, the use of SMI and OIDS and the underlying support of ASN.1 notation. Therefore, to completely understand the description of this protocol, an understanding of ASN.1 and SMI is required.

However, since these are not very complex, comprehensive study of these probably is not required.

The intent of this protocol is to be simple to use, but also flexible and expandable without any need to change the communication packets used by the protocol in order to expand or modify the capabilities of this protocol. In order to achieve this intent, this communications is the only allowed communication among devices (e.g. from the phone out into a network and from the network into the phone). In order to assess the capabilities of this protocol, the application uses an API that converts network communication (including those from the phone to the Arduino) into this protocol.

The purpose of this protocol is provide a systematic and consistent means of transferring information among nodes.

The COTS Bots protocol eliminates any need for backwards compatibility with older protocols used in the COTS Bot project previously. This protocol is designed to meet all of the requirements to make the previous statement true. This protocol, however, does not replace the packets for network layers below this application layer.

2.1.1 Communication Protocol Definitions, Acronyms, and Abbreviations

Term or Acronym	Definition
Capabilities	All of the OIDs available through this protocol.
Device	Any equipment that implements this protocol and any part of the capabilities of this protocol.
Node	Two endpoints in a communication.
PDU	Protocol Data Unit. An individual protocol message.
OID	Object Identifiers.
API	Application Programming Interface.
SMI	Serial Management Interface.
ASN.1	Abstract Syntax Notation One is a notation for defining the syntax for defining of information data. It defines a number of simple data types and specifies an notation for referencing these types and for specifying values of these types.[4]

2.1.2 What This Document Covers

This documents discusses the structure (i.e. only the logical representation) of the protocol. The only assumption that is made in this document is that the encoding will be ASN.1 compliant and therefore assumes that the requirements for ASN.1 compliance are met.

2.1.3 What This Document Does Not Cover

This document does not discuss specific encoding (i.e. the physical representation) specifications. This document does not discuss the capabilities available in this protocol, because this protocol does not require any specific protocols (and so the capabilities are not a necessary part of this protocol), although the (communication of the) capabilities are the reason this protocol is necessary.

2.1.4 Note for Users

Because the protocol is hidden behind an API that is below the application layer of the phone, the robot users should not need to interact at all with the protocol unless some problem develops.

2.1.5 Note for Developers

The protocol was designed to be accessed through an API and therefore requires no directly interaction. However, if the phone does not yet implement all the required features of the protocol or the protocol capabilities must be expanded to meet some requirement, then further reading of this document.

2.1.6 Overall Methodology of This Protocol

This protocol contains all of the necessary features to control all communication between applications that must communicate between devices over the network. This protocol is not designed for, but can be used to, control communication between applications on the phone. In order to allow for use of this protocol, an API must be developed for each device that uses this protocol and any of the capabilities of this protocol. It is

not required or possible that all capabilities of this protocol be implemented by any device. The API should be created to deal with any attempt to access or use any capability of this protocol. The API should also document for any user of the protocol, the procedure or result that occurs when any device that implements this protocol fails to implement a capability of this protocol.

The purpose of this protocol is to facilitate communication between devices in an ordered, device-independent manner. This protocol is designed not to be complete in capabilities, but to be complete in structure; that is, any capability can be added to this protocol without need to change the structure of this protocol.

This protocol is designed to independent of the specific medium of communication (e.g. wired, wireless, XBee or Bluetooth). The first use of this protocol will use XBee radio modules and Bluetooth.

2.2 The Structure of a PDU

The robot communication packets are assumed to sit inside the packets of lower level communication layers. Therefore duplicate information might be present in both the lower level communication packets and the robot communication packets. If space is an issue, then, as long as no undesirable problems arise, duplicate information can be stripped out of the packet, but only if another version number is used in the packet. This is explained later. A complete message is referred to as a PDU.

The first field of the protocol is the version(also known as type) number. This number (an unsigned 32 byte number) is common to all packets. From this field, the remaining fields of the packet are determined. If any additional packet descriptions are required, a new definition is required using a free number.

2.2.1 PDU Types

Packets fall into two categories, requests, responses, and notifications. For requests, there are two types, `GETREQUEST` (also known as `GET`) and `SETREQUEST` (also known as `SET`). `GET`s are used to retrieve information without changing any information on the remote machine while `SET` packets are used to change information on the remote machine. Responses to `GET` packets are called `GETRESPONSE` and responses for `SET` packets are called `SETRESPONSE`. A `NOTIFICATION` packet is sent to inform of a change in information.

A complete description of all PDUs defined for this protocol are found in Appendix A.2.

2.3 Requirements for Packets

2.3.1 Request and Response

If this protocol defines a PDU of type `REQUEST` (such as `GETREQUEST`), then a PDU of type `RESPONSE` (such as `GETRESPONSE`) must be defined. Although this responses do not need to be unique for each request, only one packet can be defined to be a response to each request.

2.4 Descriptions of Fields

2.4.1 versionid

10 versions are defined in this document.

Version 0

This number is unassigned and reserved.

Version 1

If the version field is set to 1, this packet is of type `GETREQUEST`. The corresponding response version is 2. Since this is a request for information, all of the varbind values should be `0`.

Version 2

If the version field is set to 2, the packet is of type `GETRESPONSE`. Besides the fields already defined in the version 1, two more fields appear: error-status and error-index. The varbind values should be equal to there values on the device.

Version 3

If the version field is set to 3, the packet is of type `SETREQUEST`. This packet has the same structure as version 1, except for the different version number. The sequence of OIDs are those that the sender wishes to set on the receiving node (along with their corresponding values).

Version 4

If the version field is set to 4, the packet is of type `SETRESPONSE`. This packet has the same structure as version 2 except that the variable-binding field is missing. This is because all values are assumed to be set to the values as sent in the `SETREQUEST` packet. If there are any errors, the error-status and error-index fields will be set and NO field will be set on the receiving device, i.e. each `SET` is assumed to be atomic.

Version 5

If the version field is set to 5, the packet is of type `NOTIFICATION`.

As the `NOTIFICATION` is not in response to a request, there is no request-id field nor error fields.

Versions 6-10

Each network is different and have different abilities. Since the initial network topology will be DigiMesh over XBee with API type packets, the source and destination address packets will already be present in the API packet (which contains the robot communication packet and is available for reading). Because the network interface ID is also already known on both sides of the communication (which does not have to be the same), these three fields are not present. Versions 6–10 map to versions 1–5 respectively.

2.4.2 network-id

The network id indications what interface this packet is for (in case multiple interfaces are available) or the path of the packet, or NULL if only one network is available or routing should be automatic or the path is unknown.

In this version of the protocol (1), this field should be set to NULL with the assumption that only one path exists for any one packet.

2.4.3 source-address and destination-address

The source and destination addresses are the network IDs for the sending and receiving nodes respectively on the interface. These values do not have to be the actual IDs of the next hop; rather, if a device is acting as an intermediary node, the intermediary node can talk to devices on that interface as a part of the device, so that explicit routing information is not required.

For example, for the COTS Bots project, the robot is comprised of several different devices, e.g. an Android phone, Romeo v2 Arduino MCU, and XBee RF module. Information coming into the XBee module can be first processed by the XBee module and returned, or can be forwarded to the Android phone. In both of these cases, the destination address of the packet will be the XBee module address on its network. For the phone, information being sent to the Arduino will be either processed by the Arduino, such as motion, or simply forwarded to the necessary node on the network via XBee. In both of these cases, the source and destination address are pure XBee addresses.

If the Android has more than one communication device available through the Arduino, the network-routing field can be used to choose the desired network. However, depending on the destination address for example, the Arduino might not need explicit routing information.

2.4.4 request-id

The request ID is a randomly generated 32 byte number. This number also appears in the response PDU. This helps to identify duplicate packets and also which packets are replies to previously sent packets.

2.4.5 variable-bindings

The variable bindings contain the sequence of OID(s) that the sender is requesting information for.

2.4.6 error-status and error-index

These are used to report any error in the `GETREQUEST` packet of type version 1 as defined above. The variable-bindings field will contain the sequence of the requested OID(s) along with their values.

2.4.7 check-sum

This is a 16-bit checksum as used in the XMODEM-CRC protocol. This checksum has a very high probability of finding errors (>99%).

2.5 Structure of the variable-bindings

The variable-bindings is a sequence of OIDS, i.e. the identification string of the capabilities. When an OID is being referenced, an additional .0 will be need to be appended for those capabilities that are single instances, otherwise a .X will be appended, where X is the number of the specific sequence.

2.6 Further Discussion

2.6.1 One Packet Versus Multiple Packets

Each PDU that is sent should be atomic, if possible. This might not always be possible because some commands may need to be executed in a specific order (such as turn, forward, turn) and the possibility of future steps is not known when the first packet is received. This should be noted in the capabilities that this command is not atomic.

Otherwise, the difference between using one packet with several OIDs or multiple packets with a single OID, is only in terms of network traffic generated.

Also, there is no field to specify a continuation of a message, such so specify the next packet in a sequence of packets. This instead, is specified using indexes on the appropriate OID.

If a continuous field is required, another version of this protocol can be created with the fields as required.

2.7 Further Discussion and Description

For communication between to XBee-equipped COTS Bots robots, the packets can be of type:

$$\text{Sending Phone} \rightarrow \text{v.1} \rightarrow \text{XBee} \rightarrow \text{v.6} \rightarrow \text{XBee} \rightarrow \text{v.1} \rightarrow \text{Receiving Phone} \tag{2.1}$$

$$\text{Sending Phone} \leftarrow \text{v.2} \leftarrow \text{XBee} \leftarrow \text{v.7} \leftarrow \text{XBee} \leftarrow \text{v.2} \leftarrow \text{Receiving Phone} \tag{2.2}$$

Of course, both nodes must understand the necessary packet types and both be in API mode. If this is not true, the receiving phone should send a response packet back with the necessary error value.

Chapter 3

Encoding

3.1 Introduction

This document describes the encoding features of this protocol.

3.2 Currently supported protocols

The only protocol encoding supported is the ASN.1 BER encoding. An in-depth description will be included in a later revision of this document.

Chapter 4

Requirements

4.1 Introduction

This document describes all the general requirements for devices supporting this protocol and general requirements for requirement documents to follow.

Refer to ASN.1 literature for a discussion of terms.

4.2 Minimum Versions

Any device supporting this protocol must support receiving packets of protocol version 1 and sending packets of protocol version 2.

Also, the device must support all protocol versions as defined by the `DEVICE.SUPPORTED-VERSIONS` capability of the device and all versions required by the device-specific protocol document.

If a device supports a version that is of type `REQUEST`, then the protocol must support the `RESPONSE` type of that version.

4.3 Minimum Capabilities

Any device supporting this protocol must support the `DEVICE.SUPPORTED-VERSIONS` capability of all protocol versions supported by the device.

Also, if the protocol-specific document of a device requires that a capability must be present, then that capability must be supported by the device.

4.4 Process When Receiving a PDU

4.4.1 Header

When a device receives a PDU of this protocol, the header information must be checked. The first field should have a type of `SEQUENCE`. If not, the header is considered to be invalid. If the length is incorrect, the

header information is also considered invalid. If the header is invalid the packet should be discarded or data should be discarded until a valid header is found, depending on the interface (packets or stream).

4.4.2 Version

The next field is the version. If the version is not supported or the encoding is invalid, the PDU is discarded and no reply is sent.

4.4.3 Other Non-Varbinding Fields

If any non-varbinding fields are invalid, then the error-status and error-index fields in the response message should be set to `invalidPDU` and the index of the field (with `version-id` labeled as 0), respectively. If the varbinding field is defined for the PDU, it should be set to NULL.

4.4.4 Varbinding

TBD

Chapter 5

Testing

5.1 Introduction

This document contains a discussion and description of the testing methodology and tools available for this protocol.

To test this protocol, a Python 3.2+ script is being developed that is capable of both emulating the communications from the phone and also emulating remote nodes.

5.2 Requirements

For each device that implements this protocol, a TCS and FT must be developed to test both the requirements as described in Part 4 of this document, but also implementation specific details as unique to the code (as partly described in the Test Plan).

Unless otherwise noted, all PDUs received or send must be valid and conform to the requirements under these documents.

5.3 Referencing Testing Cases from this documents

Each test case in this document should be referenced by using the tag `PROTOCOL.X.Y`, where `X` is the corresponding section name in upper case with no spaces and `Y` is the number of the test case in that section. Therefore, for example, the first test case shall be referenced by `PROTOCOL.SUPPORTEDVERSIONS.1`.

If the test case is no longer applicable, it should be marked as such. However, the section number is not to be reused.

5.4 TCS

5.4.1 Integration Tests

To be determined.

Part III

Device Design Description

Chapter 6

Romeo v2 Design

6.1 Introduction

This document describes the design of the software for the Arduino Romeo v2. This is the complete the requirements for documentation as outlined in chapter 4.

6.2 Arduino Definitions, Acronyms, and Abbreviations

Term or Acronym	Definition
Hardware Serial	Serial data connection that uses pins 0 and 1 (RX and TX) on the Arduino.
Software Serial	Serial data connection that uses any pins other than 0 and 1. This requires the NewSoftwareSerial library in order to be used.
USB Interface	Serial data connection that uses the USB port on the Arduino.
RX	Receive
TX	Transmit

For a list of definitions related to the protocol, please see chapter 2.

6.3 Methodology

Many of the steps should be sequentially executed so that, as many of the errors and error processing can be found in as few locations in code as possible. In this program, these steps should be in the `ParseProcotol()` function.

The Romeo v2 has an ATmega32U4 microcontroller running at 16 MHz with 32 Kbytes of flash memory[2], and as such is relatively slow at processing information. Code should therefore be optimized for speed and space.

6.4 Purpose

Our code for the Romeo v2 must handle the routing of communications from the various ports, run the motor controller, and handle protocol messages that are specific to the Romeo v2 and any additional hardware that is attached directly to or is controlled directly by the Romeo v2.

6.4.1 Routing

Information sent to the Romeo v2 can come from three sources: the USB port, the Hardware Serial pins (pins 0 and 1) or the Software Serial pins (pins other than 0 and 1). Not all information sent through these interfaces are intended for the Arduino.

In the current design, the Android phone interfaces with the Arduino through the Software Serial library. Some of these messages are designed to be passed directly to the XBee and some are indented for the Arduino to execute (e.g. turn on and off motor control) or reply to itself. The USB interface is currently only set for testing and debugging. Data from the XBee should be always routed directly to the phone because all of the application-level authentication happens on the phone's hardware.

If the source and destination addresses are equal to the address of the XBee unit, then the assumption is that the packet is intended for the current device, otherwise, the message is assumed to be intended for another device on the network.

6.4.2 Parsing

All traffic should be done using the Communications Protocol defined in part II. Although messages from the XBee unit should be routed to the phone for processing, messages from the phone to the Arduino might be intended just for local processing. Therefore, the Arduino should understand the protocol in order to carry out the commands or provide information as contained in the message.

6.4.3 Motor Control

Currently, the primary purpose of the Arduino is to drive the motor controller.

6.5 Setup and Loop

All Arduino programs start by executing the `setup()` function. This function is for initializing any serial ports and any other tasks or instructions that should be executed only once per power cycle.

The `loop()` function contains all the code that will run in a loop from after the `setup()` function is executed to the device being powered off.

6.6 Parsing Protocol Packets

6.6.1 loop

The `loop()` function just waits for one of the functioning serial interfaces to have available data. If any of of the serial interface have available data (using the available member function), the `ParseProtocol()`

function is called with the file descriptor (as explained in section 6.7.1).

All interfaces will use the `ParseProtocol()` function, even if the Arduino is receiving data from the XBee, (which will always be passed to the Bluetooth connection). This is because only one PDU should be processed at a time. Although the assumption that the XBee will receive only one packet, the XBee information will contain its own protocol information that will need to be stripped off and the version of the protocol possibly converted to a new version.

6.6.2 ParseProtocol()

This function manages the parsing of the packet into the individual fields by creating a Packet object and calling its `Initalize()` function.

6.6.3 Packet::Initialize()

This function retrieves the necessary bytes depending on the length parameter of the PDU. The Arduino code operates on the assumption that the entire PDU is available at once. If at any time, the rest of the data from the serial interface is not available (such as if the specific serial object's function is not available), the current data will be discarded and command will return to the loop function.

If all of the packet is read, then the `ParseProtocol()` function calls the `Packet::Parse()` function.

6.6.4 Packet::Parse()

At this point, since all of the packet has successfully been obtained, the individual fields of the PDU can be created. This is done by first grabbing the version number of the PDU. From this, the remaining fields can be parsed by creating a PDU and by then initializing all of the fields found in the particular version of that PDU.

There is only one type of PDU that contains member variables for each of all of the possible values found in any version supported by the Arduino. The remaining fields remain uninitialized. This is to simplify the code and since the type of fields do not vary widely, should not waste a material amount of memory.

If there is an error, the appropriate action should take place. If the PDU is a supported version, but contains some error in its structure or if it makes an invalid request (for example because the requesting device does not have the authorization, the information does not exist or the information cannot be set), or if everything is correct, a response packet should be created.

First a ResponsePacket object is instantiated. Then the `CreatePacket()` function is called by passing in the Packet object that contains the PDU.

6.6.5 ResponsePacket::CreatePacket()

The Packet object should contain all the necessary information at this point to create the appropriate response. However, the Packet class does not know how to encode the necessary fields. Instead, by using the member functions of the ResponsePacket class for each field in the response packet, the appropriate response PDU can be formed.

6.6.6 ResponsePacket::SendPacket()

Finally, the packet should be available to send over the appropriate interface.

6.6.7 ProcessOIDS()

Once the packet has been parsed successfully. The OIDs should be processed (by either retrieving information for a `GETREQUEST` or by setting information for a `SETREQUEST`. All of the information should now be ready for a response packet.

6.7 Further discussion

6.7.1 Using Different Serial Interfaces

Since the different serial objects for the Arduino, (e.g. USB interface, Hardware, and Software Serial) come from different classes, only the main `loop()` function deals with the Serial objects directly. Instead, a file descriptor is passed. If information needs to be read or written, the `getByte()` and `sendBytes()` functions are used instead of the specific object's `read()` and `write()` functions.

Currently, the value of 1 is for the USB interface, 2 for the hardware serial interface, and higher numbers for Software Serial interfaces. These are defined in `Utilities.h` and `Utilities.cpp`.

Part IV

Test Plan Document

Chapter 7

Test Plan Document (TPD)

7.1 References

The repository for this project is located at https://bitbucket.org/cawaltrip/autocots/.

7.2 Purpose

The purpose of this test plan is to provide a clear and defined test plan for the COTS Bots Team AutoCOTS Senior Design Project. This document should contain all the necessary information to unambiguously test all parts of the COTS Bots project, including software and hardware. This document therefore should provide a list of what should be tested, the requirements for any testing, how a requirement should be tested, and how the results should be presented and interpreted.

Although the language of this document might insinuate that all of the information is contained within in this physical document, the contents of the test plan are not actually confined to this specific document. To find other parts of the test plan, please consult the reference for other document names mentioned in the document, containing important pieces of the master test plan. This includes separate documents for each device/software's SRSs and TCSs.

7.3 Test Items

The hardware items under test for this project are:

- XBee Module
- Arduino Board
- Android Phone

The software items under test for this project are:

- Arduino Code

7.4 Software/Hardware Risk Issues

The software will be developed for both the Android and Arduino environment. The languages for each are the Android programming environment (based on Java) and the Arduino programming environment (based on C), respectively. Both have numerous standard libraries. These standard libraries however, will be considered a low levels of risk in our testing because we believe that they are thoroughly tested themselves before release. Third party libraries, however, may not be tested as thoroughly and have the potential be a much higher level of risk. Therefore, a standard library is preferred over a third party library in the COTS Bots Team AutoCOTS Project Code. However, a cost-benefit analysis should be done when deciding whether to use any given piece of supporting software. Also, not all third party software contains the same level of risk because some have undergone a rigorous testing and so can be considered to have a moderate to low level of risk depending on the library. Software developed under this project will always be considered to have a high level of risk and therefore must undergo testing as required under this document for software that is defined to be high risk.

Because the project involves robot movement, risk can arise from the movement and the unpredictability of movement. Although theft is also a risk to this project, theft is outside the scope of this testing plan.

The first semester deadline for this project is 16 May 2014. The final semester deadline (for Team AutoCOTS) is 19 December 2014. Both the hardware and software implementations must be in place at the end of the final semester deadline in order to fulfill the requirements for the final demonstration. All components of this project are critical to the success of the overall project.

7.4.1 Critical Risk Areas

+ TBD

7.4.2 Features that will be Tested

Features that will be tested include:

- Radio Protocol
- Radio Modules
- 3rd party libraries
- Our software

7.4.3 Features that may be Tested

Features that may or may not be tested include:

- Legacy code that was previously developed for the COTS Bots Project

7.4.4 Features that will not be Tested

The features that will not be tested include:

- Other hardware not listed above (e.g. motor controllers);

- Standard libraries used in the programming environments.

7.4.5 Approach

Because two separate languages will be used in this project, two types of testing tools will be used.

Also, because the codebase for the Arduino will be small, no specific tool is required. However, unit tests should be designed for all code nonetheless. This may include a manual run of the robot or a simulation in software of the inputs and outputs of the unit of code.

The majority of the code for this project will be developed under Android. Because Android is derived from Java, many tools are available for use. For now, we believe that we will be using JUnit for our unit tests, but this will not be finalized until we start on the Android development phase.

7.5 Metrics

+ TBD

7.6 Configurations

+ TBD

7.7 Software

- Android 2.4+?

7.8 Hardware

- Romeo v2 Arduino MCU
- XBee Series 1

7.9 Testing Methodology

7.9.1 Categories

Testing will fall under four categories:

1. Unit Testing
2. Integration Testing
3. System Testing

4. Acceptance Testing

7.9.2 Levels of Risk

There are three levels of risk defined for this project:

1. Low
2. Medium
3. High

7.9.3 Classification

There are three levels of classification defined for this project:

1. Level 1
2. Level 2
3. Level 3

This section lays out the minimum requirements for the classification designations. Any additional testing is encouraged, but is not required.

Level 1 Level 1 tests should be run anytime the software or device has been changed.

Level 2 Level 2 tests should be run anytime the software is ready to be released for integration into the code.

Level 3 Level 3 tests should be run before any major demonstrations and at least once during the lifetime of the software.

7.10 Testing Tools

This project uses multiple platforms and languages, therefore multiple tools are available not just for a language but one for at least all languages used in this project.

7.10.1 Arduino

GoogleTest has good support for mock objects to emulate the Serial objects.

7.10.2 Android

JUnit?

7.11 Testing Process

7.11.1 Pass/Failure Criteria

If a failure happens during the execution of any test, a failure report should be submitted to provide information of the failure. Failure should be reported in the issue page of the repository. The name of the issue should be the name of the code file and the name of the test if applicable.

What Constitutes a "Failure"?

Any deviation from a specification, e.g. SSDD or CPD.

What Does Not Constitute a "Failure"?

Any unit or action that does not have any requirements documentation, cannot cause a 'failure'. In this case, appropriate action should be carried out to address any perceived deficiencies.

What to do when a "Failure" is Discovered"

When a failure is discovered, an SCR should be produced to document each "failure" that needs to be corrected.

Incorrect Specification Document

A specification document being incorrect (e.g. outdated or misstated) also constitutes a "failure" and an SCR should be created.

What Sections to Include in an SCR

The following items should be included in an SCR:

- The failure that was identified;
- The expected outcome;
- The actual behavior;
- The steps required to reproduce the failure.

Each section should be concise, yet convey enough information for the developer to remedy the problem.

Test Development

Either the developer or another person not associated with the code may write tests. It is preferable that the developer not write the tests. This is to provide another perspective to the required. The coder should not code the tests besides unit tests developed white writing the code. Non unit tests tests should be written, coded, and ran by someone not involved in the code to reduce the change of bias.

7.11.2 Unit Testing

All software that fails under section 7.4.2 requires at least some unit tests. The coverage and number of these tests should be directly proportional to the risk level of the piece of code. Each piece of code should at least test the parameters of the function.

7.11.3 Integration Testing

Integration testing can start before the ending of unit testing. Integration testing also may also act as a substitute for unit testing if, during the integration testing, the necessary test coverage can be obtained that would have been obtained from the unit tests. If so, the integration test results should also list which unit tests were covered.

7.11.4 System Testing

TBD

7.11.5 Acceptance Testing

The acceptance test should be a battery of tests that are specified by the customer as proof that the concept and solution work correctly. Although similar to a system test, the acceptance test is based on the requirements of the customer, whereas the system testing should contain all tests that are necessary and applicable to cover the entire concept. Therefore, acceptance tests can be a subset of system tests or a combination of system tests depending on the scope of the acceptable tests.

7.11.6 Manual Testing

Some tests, generally those that include more than just software to be tested or those tests that are too complex to be automated, can be tested manually. Although automated tests are superior in that they can be ran multiple times with little work to repeat, some automation efforts do not pass a cost-benefit analysis and so should be manual. Manual tests have the same requirements as automated tests and the documentation generated from manual tests should reflect this.

7.12 Code Reviews

All code developed that will be used in the finished product (the code that will be used by the customer) must pass a code review. Code that is used in the development or testing phases of the project but that is not a part of the project codebase does not need to be code tested, although if possible, should be done.

7.13 Intra-Group Collaboration

Collaboration with members outside of the COTS Bots Teams is encouraged for tasks such as code reviews, test code audits and documentation proofreading to facilitate an unbiased review of the project and the project's status.

References

Chapter 8

Bibliography

[1] Federal Acquisition Regulations System.

[2] Atmel. ATmega32U4. Online.

[3] CSDS. Systems and Software Design Description Template.

[4] International Telecommunications Union. X.680: Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. Online.

Appendices

Appendix A

COTS PDU

A.1 Introduction

This document contains complete descriptions of all of the Protocol Data Units (PDUs) defined for this protocol as discussed in chapter 2.

A.2 COTS-PDU

```
COTS-PDU DEFINITIONS ::= BEGIN

IMPORTS
   Counter32   FROM SNMPv2-SMI

ObjectName ::= OBJECT IDENTIFIER

ObjectSyntax ::= simple SimpleSyntax

SimpleSyntax ::=
   CHOICE {
      integer-value   Counter32 (-2147483648..2147483647),
      string-value    OCTET STRING (SIZE (0..65535)),
      objectID-value  OBJECT IDENTIFIER
   }

NetworkAddress ::=
   CHOICE {
      xbee-my-address OCTET STRING (SIZE 4) ('0000'H..'FFFF'H),
      xbee-serial-number OCTET STRING (SIZE 8) ('00000000'H..'FFFFFFFF'H)
   }

NetworkRoute ::=
   CHOICE {
      unspecified NULL,
      network-id INTEGER,
      network-address NetworkAddress
      SEQUENCE OF NetworkRoute
   }


CyclicRedundancyCheck ::= INTEGER


ErrorStatus ::=
   INTEGER {
      noError(0),
      tooBig(1),
      noSuchName(2),
      badValue(3),
      readOnly(4),
      genErr(5),
      noAccess(6),
      wrongType(7),
      wrongLength(8),
      wrongEncoding(9),
      wrongValue(10),
      noCreation(11),
      inconsistentValue(12),
```

```
      resourceUnavailable(13),
      commitFailed(14),
      undoFailed(15),
      authorizationError(16),
      notWritable(17),
      inconsistentName(18),
      noDevice(19),
      invalidPDU(20)
   }

PDU ::=
   CHOICE {
      COTS-v1 cots-pdu-v1,
      COTS-v2 cots-pdu-v2,
      COTS-v3 cots-pdu-v3,
      COTS-v4 cots-pdu-v4,
      COTS-v5 cots-pdu-v5,
      COTS-v6 cots-pdu-v6,
      COTS-v7 cots-pdu-v7,
      COTS-v8 cots-pdu-v8,
      COTS-v8 cots-pdu-v9,
      COTS-v10 cots-pdu-v10,
   }

cots-pdu-v1::=
   SEQUENCE {
      version-id INTEGER (1),
      network-routing NetworkRoute,
      source-address NetworkAddress,
      destination-address NetworkAddress,
      request-id Counter32,
      variable-bindings VarBindList,
      check-sum CyclicRedundancyCheck
   }

cots-pdu-v2 ::=
   SEQUENCE {
      version-id INTEGER (2)
      network-routing NetworkRoute,
      source-address NetworkAddress,
      destination-address NetworkAddress,
      request-id Counter32,
      error-status ErrorStatus
      error-index INTEGER (0..max-bindings),
      variable-bindings VarBindList,
      check-sum CyclicRedundancyCheck
   }

cots-pdu-v3::=
   SEQUENCE {
      version-id INTEGER (3),
      network-routing NetworkRoute,
```

```
      source-address NetworkAddress,
      destination-address NetworkAddress,
      request-id Counter32,
      variable-bindings VarBindList,
      check-sum CyclicRedundancyCheck
   }

cots-pdu-v4 ::=
   SEQUENCE {
      version-id INTEGER (4),
      network-routing NetworkRoute,
      source-address NetworkAddress,
      destination-address NetworkAddress,
      request-id Counter32,
      error-status ErrorStatus,
      error-index INTEGER (0..max-bindings),
      check-sum CyclicRedundancyCheck
   }

cots-pdu-v5 ::=
   SEQUENCE {
      version-id INTEGER (5),
      network-routing NetworkRoute,
      source-address OCTET STRING ('0'H..'FFFF'H),
      destination-address OCTET STRING ('0'H..'FFFF'H),
      variable-bindings VarBindList,
      check-sum CyclicRedundancyCheck
   }

cots-pdu-v6::=
   SEQUENCE {
      version-id INTEGER (6),
      network-routing NetworkRoute,
      source-address NetworkAddress,
      destination-address NetworkAddress,
      request-id Counter32,
      variable-bindings VarBindList
   }

cots-pdu-v7 ::=
   SEQUENCE {
      version-id INTEGER (7)
      network-routing NetworkRoute,
      source-address NetworkAddress,
      destination-address NetworkAddress,
      request-id Counter32,
      error-status ErrorStatus,
      error-index INTEGER (0..max-bindings),
      variable-bindings VarBindList
   }

cots-pdu-v8::=
```

```
   SEQUENCE {
      version-id INTEGER (8),
      network-routing NetworkRoute,
      source-address NetworkAddress,
      destination-address NetworkAddress,
      request-id Counter32,
      variable-bindings VarBindList,
   }

cots-pdu-v9 ::=
   SEQUENCE {
      version-id INTEGER (9),
      network-routing NetworkRoute,
      source-address NetworkAddress,
      destination-address NetworkAddress,
      request-id Counter32,
      error-status ErrorStatus,
      error-index INTEGER (0..max-bindings)
   }

cots-pdu-v10 ::=
   SEQUENCE {
      version-id INTEGER (10),
      network-routing NetworkRoute,
      source-address NetworkAddress,
      destination-address NetworkAddress,
      variable-bindings VarBindList
   }

VarBindList ::= SEQUENCE (SIZE (0..max-bindings)) OF VarBind

VarBind ::=
   SEQUENCE {
      name ObjectName,
      CHOICE {
         value    ObjectSyntax,
         unSpecified NULL,    -- in REQUESTs
         noSuchObject   [0] IMPLICIT NULL,
         noSuchInstance [1] IMPLICIT NULL,
         endOfMibView   [2] IMPLICIT NULL
      }
   }

END
```

Appendix B

COTS MIB

B.1 Introduction

This document contains all of the capabilities available under the current version of this protocol in SMI format as mentioned in chapter 2.

B.2 COTS-MIB

```
COTS-MIB DEFINITIONS ::= BEGIN

IMPORTS
   MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE, Counter32
      FROM SNMPv2-SMI,
   DisplayString
      FROM SNMPv2-TC
   sysUpTime
      FROM snmpMIB -- RFC 3418


COTSMIB MODULE-IDENTITY
   LAST-UPDATED "201404170000Z"
   ORGANIZATION "COTS Bots Working Group"
   CONTACT-INFO "mein1156@vandals.uidaho.edu
   DESCRIPTION  "The COTS Bots Management Architecture MIB

   REVISION     "201404170000Z "
   DESCRIPTION  "Started document."

   ::= { head } -- head is implicitly the top most octet, but is not
      explicitly part of the OID

-- the device Group ********************************************

device      OBJECT IDENTIFIER ::= { head 1 }

supported-versions OBJECT-TYPE
  SYNTAX       SEQUENCE OF INTEGER,
  MAX-ACCESS   read-only,
  STATUS       current,
  DESCRIPTION  "A list of versions supported by this device."
  ::= { device 1 }

deviceType OBJECT-TYPE
  SYNTAX    INTEGER {
      Unknown (0),
      Romeov2 (1),
      BlueTooth (2),
      XBee-RF-Modules (3)
      }
  MAX-ACCESS   read-only,
  STATUS    current,
  DESCRIPTION
      "The type of the device."
   ::= { device 2 }

sysUpTime :== { device 3} -- In milliseconds
```

```
current-interface OBJECT-TYPE
   SYNTAX   INTEGER,
   MAX-ACCESS read-only,
   STATUS   current,
   DESCRIPTION "Interface number of the interface that the PDU was
      received on"
   :== { device 4 }

interface-number OBJECT -TYPE
   SYNTAX INTEGER
   MAX-ACCESS read-only
   DESCRIPTION "Number of interfaces available on the device"
   ::= { device 5 }


-- the interface Group ******************************************

interface OBJECT IDENTIFIER ::= { head 2 }

interface-table OBJECT-TYPE
   SYNTAX table
   MAX-ACCESS read-only
   DESCRIPTION "Table of interfaces available on the device"

interface-status OBJECT-TYPE
   SYNTAX BOOL {
      On (1),
      Off (0)
      }
   MAX-ACCESS write, read
   DESCRIPTION "Status of the interface"
   ::= { interface 1 }

interface-type OBJECT-TYPE
   SYNTAX INTEGER {
      though (1),
      parent (2),
      debug (3)
      }
   MAX-ACCESS write, read
   DESCRIPTION "Type of interface"
   ::= { interface 2 }

interface-device OBJECT-TYPE
   SYNTAX deviceType
   ::= { interface 3 }

network-discovery OBJECT-TYPE
   SYNTAX --TRAP RESPONSE--
   MAX-ACCESS read-only
   STATUS current
   DESCRIPTION "Network Discovery"
```

```
   ::= { interface 4 }

network-address OBJECT-TYPE
   SYNTAX CHOICE {
      XBee OCTETSTRING (size (4)) ('0000'H...'FFFF'H)
      }
   MAX-ACCESS read, write
   STATUS current
   DESCRIPTION "Network address"
   ::= { interface 5 }


-- the movement Group ********************************************

movement    OBJECT IDENTIFIER ::= { head 3 }

movement-table OBJECT-TYPE
   SYNTAX table
   MAX-ACCESS read-only
   DESCRIPTION "Table of interfaces available on the device"

movement-status OBJECT-TYPE
  SYNTAX    INTEGER {
                        moveable (0),
          not currently moveable (1),
          not moveable (2),
        }
  MAX-ACCESS   read-only
  DESCRIPTION
                       "Indicates whether the device is moveable and
                          whether the device
                       can currently move."
  ::= { movement 1 }

  movement-type OBJECT-TYPE
      SYNTAX INTEGER {
         servo (0),
         dctrack (1),
         underwater (2)
         }
      MAX-ACCESS write, read
      STATUS current
      DESCRIPTION "Type of movement"
  ::= { movement 2 }

movement-servo OBJECT-TYPE
      SYNTAX SEQUENCE {
         turning INTEGER (0...180),
         power INTEGER (0...180),
         time CHOICE OF {
            continuous INTEGER (0),
            milliseconds INTEGER (NOT 0)
```

```
            }
         }
      MAX-ACCESS write-only
      STATUS current
      DESCRIPTION "Servo values for turning and power respectively"
      ::= { movement 3 }

movement-dctrack OBJECT-TYPE
      SYNTAX SEQUENCE {
         left (0...180),
         right (0...180),
         time CHOICE OF {
            continuous INTEGER (0),
            milliseconds INTEGER (NOT 0)
            }
         }
      MAX-ACCESS write-only
      STATUS current
      DESCRIPTION "DC track values for left and right track speed,
         respectively. Negative value for reverse"
      ::= { movement 4 }

movement-underwater OBJECT-TYPE
      SYNTAX SEQUENCE {
         turning (0...180),
         power (0...180),
         time CHOICE OF {
            continuous INTEGER (0),
            milliseconds INTEGER (NOT 0)
            }
         }
      MAX-ACCESS write-only
      STATUS current
      DESCRIPTION "Underwater values, for left and right track values,
         respectively."
      ::= { movement 5 }

-- the command Group ******************************************

command     OBJECT IDENTIFIER ::= { head 4 }

command-message OBJECT-TYPE
   SYNTAX OCTETSTRING
   MAX-ACCESS write-only
   STATUS current
   DESCRIPTION "User defined message"
   ::= { command 1 }


-- the components Group ***************************************

components     OBJECT IDENTIFIER ::= { head 5 }
```

```
leds OBJECT IDENTIFIER
   SYNTAX table
   MAX-ACCESS read-only
   STATUS current
   DESCRIPTION "Table for leds"
   ::= { components 1 }

leds OBJECT IDENTIFIER
   SYNTAX INTEGER
   MAX-ACCESS read-only
   STATUS current
   DESCRIPTION "Number of leds available"
   ::= { leds 0 }

leds OBJECT IDENTIFIER
   SYNTAX INTEGER
   MAX-ACCESS write-only
   STATUS current
   DESCRIPTION "Number of milliseconds for the leds to stay on"
   ::= { leds 1 }
```