

A JAUS Toolkit for LabVIEW, and a Series of Implementation Case Studies with Recommendations to the SAE AS-4 Standards Committee

Ruél R Faruque

Thesis submitted to the faculty of the Virginia Polytechnic Institute
and State University in partial fulfillment of the requirements for the
degree of

Master of Science
In
Mechanical Engineering

Dr. Charles F Reinholtz, Committee Co-Chairman
Dept. of Mechanical Engineering, Virginia Tech

Dr. Alfred L Wicks, Committee Co-Chairman
Dept. of Mechanical Engineering, Virginia Tech

Dr. Parag Batavia, Committee Member
Applied Perception Inc

December 8th, 2006
Blacksburg, Virginia

Keywords: JAUS, Architecture, SDK, LabVIEW, Standard,
Messaging, Unmanned Vehicles, AS-4

Abstract

A JAUS Toolkit for LabVIEW, and a Series of Implementation Case Studies with Recommendations to the SAE AS-4 Standards Committee

Ruél R Faruque

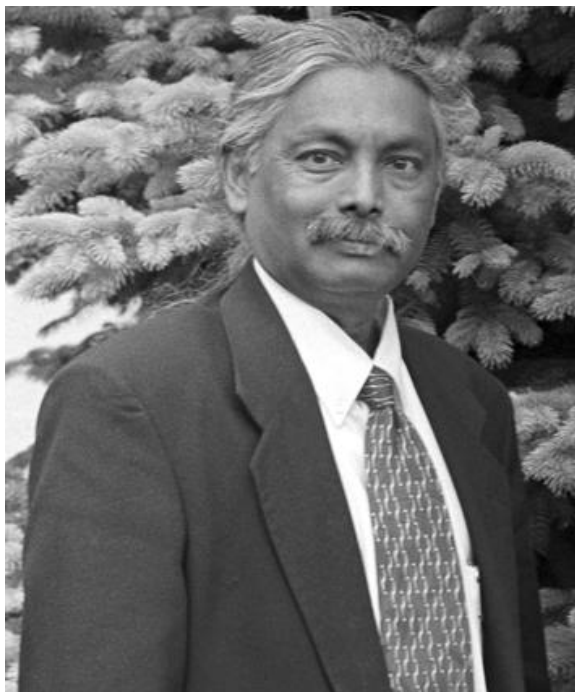
The Joint Architecture for Unmanned Systems (JAUS) is an emerging SAE standard for messaging between and within unmanned systems, enhancing interoperability, code reusability, and modularity. To date, most JAUS implementations have been produced by developers in the JAUS Working Group, who have an intimate understanding of the intent of the specification. As a result, the standards documents produced by the Working Group sometimes omit key implementation assumptions the developers had in mind.

The work described in this thesis is intended to provide objective feedback from the viewpoint of a developer who began implementing JAUS independent of the standards committee and is now embedded in the Experimentation Task Group.

This paper presents the lessons learned through this journey, from development of the first-ever JAUS software development toolkit for the LabVIEW programming language, through participation in Experimentation Task Group interoperability exercises 2.75 and 3.0, and to the establishment of the first internet-based testing environment for JAUS.

The results of this experience are presented as a collection of recommendations to the standards committee, organized in a series of diverse implementation case studies.

This paper is dedicated to my father, Azad Faruque, the Lead Data Architect for the United States Department of State. Before he died in August 2006, it was extremely important to him that I complete my Master's coursework.



Born in 1956 in Naryanganj Bangladesh, he emigrated to the United States, where he attended high school in Carrollton Ohio, achieved bachelor's degree in Mathematics and Physics at Warren Wilson College, met my mother Loren, and then pursued a master's degree in Computer Science at Ball State University. After completing his coursework and research in adaptive text-editors, but before completing his thesis, he took a job. At the time of my father's death, I was in this exact position of having my coursework and research complete, but not having started writing this thesis.

My father, and his wife Loren Faruque, are the most dedicated, selfless, and honorable people I have ever met. They did everything possible for my brothers, sister, and I, keeping our best interests in mind. I pray that God will help me aspire to their level of excellence.

Acknowledgements

A large and diverse group contributed to my success in this project. Truly, this work would not have been completed without their help.

Development Team

This project is unusual in that it is the work of a large group. Nearly everybody in JOUSTER had an impact on the toolkit, even if I just bounced an idea off them or copied a clever programming trick.

Grant Gothing and Steve Cacciola were instrumental in the programming of the toolkit, as the number 1 and 2 developers. They often spent hours putting my wacky ideas into code, and Grant endured many 10:00pm instant messages during my “Eureka!” moments. Without these two bright comrades, we would not have performed as well at the JAUS ETG experiments.

Cheryl Bauman, Jon Weekley, Matt Simone, Mike Fleming, Chris Terwelp, and Chris Sharkey all helped out with small yet vital parts of the project. Brett Leedy, one of the most focused unmanned systems developers I have ever met, was an inspiration for perfection. Andrew Bacha provided lots of programming tips, and a simulator which enabled us to be highly productive when testing with other implementers. Pat Currier and Jesse Hurdus were the first true beta testers of the toolkit. Bruce Billian helped out with networking issues and established the InterJECT VPN. Shane Barnett and Mike Avitabile helped with the outfitting of our primary test vehicle, Matilda.

Christian Hahn and Andy Watchorn from National Instruments, the makers of LabVIEW, have been very supportive in locating the answers to some of my difficult programming questions.

Advisors

Dr Charlie Reinholtz has been a mentor since I met him through ASME in 2002, and I am thankful for all the guidance he provided for all of the unmanned systems program at Virginia Tech. Dr R’s dedication to students and the autonomous vehicles we develop is the foundation of the success experienced by the autonomous vehicles teams at Virginia Tech.

I would probably not have graduated anytime soon had it not been for the constant reminders given me by Dr Al Wicks. This unconventional

professor has been extremely supportive of us as students, having our best interests in mind. But... I still don't like golf.

Dr Parag Batavia, chair of the Experimentation Task Group in the JAUS Working Group, has been extremely receptive to Virginia Tech as a newcomer in the ETG. I am thankful for the encouragement he provided to our group, for the time he spent evaluating this document, and for his answers to my many questions.

Finally, Mike Fleming and Chris Terwelp have done an excellent job as JOUSTER program managers, establishing clear initial goals, then allowing me to take the project beyond any of our expectations, all the while remaining excited about our work.

Working Group

My involvement in the JAUS Working Group has led me to meet an array of refreshingly eccentric and remarkably intelligent roboticists; I hope these relationships last long into the future.

Carl Evans from Applied Perception remained patient during my first few frustrating interoperability attempts. VPN testing with Joya Joseph from Defense Technologies helped forge the robustness of our vehicle software. Danny Kent from the University of Florida helped with the first-ever cross-organization demonstration of level-III interoperability. Sarah Gray and Lonnie Hansen from Autonomous Solutions, John Culbertson from RE2, See Yee from SPAWAR, Kathy Wienhold from iRobot, Jeff Wit and Arfath Pasha from AFRL, Parag Batavia from Applied Perception, and John-Mike Taylor and Jerry Lane from ARA have also given me many valuable lessons during the past two years.

The excitement of leaders such as Jeff Kotora, Woody English, Dan Gehring, Robert Wade, and Jim Albers is encouraging to me as a developer. Michael Clark from Harris has been extremely supportive of our adventures in JAUS, advocating for me at every opportunity.

Funding Support

This project was funded through the JOUSTER project, by the Army Research Development Engineering Command Simulation Technology Training Center (RDECOM-STTC) under contract N61339-04-C-0062.

Family

My mother, Loren, and brothers Imraan and Orion, and sister Anya, have continually reminded me to complete this work. Thank you for making me finish this. I blame you for me being able to wear that funny-looking robe on graduation day.

Contents

Title Page	1
Chapter 1: Introduction	1
1.1 Introduction	1
1.1.1 JAUS Toolkit for LabVIEW	1
1.1.2 Implementation Case Studies	2
1.1.3 Organization of Thesis	2
1.2 Motivation for a Standard	3
1.2.1 Need for an Interoperability Standard	3
1.2.2 The Birth of JAUS	4
1.2.3 JAUS Functionality	4
1.2.4 JAUS Working Group	7
1.3 Adoption of JAUS	8
1.3.1 Current Products	8
1.4 Motivation at Virginia Tech	8
1.4.1 JOUSTER	9
1.4.2 Evolution of JAUS Project at Virginia Tech	10
1.4.3 Why VT Uses JAUS	11
1.5 Introduction to JAUS	12
1.5.1 What is JAUS?	12
1.5.2 JAUS Purpose	14
1.5.3 Software Perspective	14
1.5.4 Terminology	14
1.5.5 System Layout	15
1.5.6 Components and Services	17
1.5.7 JAUS Messages	17
1.5.8 Compliance	21
1.5.9 Differences in VT Understanding of JAUS	22
1.6 Introduction to LabVIEW	23
1.6.1 Brief History	24
1.6.2 Benefits of LabVIEW	24
1.6.3 LabVIEW Environment	25
Chapter 2: VT JAUS Toolkit Overview	28
2.1 Toolkit Scope	28
2.1.1 Message Set Version	29
2.1.2 Interoperability	29

2.1.3	Assumptions	30
2.2	Toolkit Elements	30
2.2.1	Message Builders/Parsers	30
2.2.2	Component Model	32
2.2.3	Routing Components	35
2.2.4	Code Modules	36
2.2.5	Components	37
2.2.6	Message Replay Utility	38
2.3	Toolkit Deployment	38
2.3.1	Releases and Upgrades	39
2.4	Use of the Toolkit	39
2.4.1	New Primitive Driver	39
2.4.2	New Component	40
2.4.3	Experimental Message	40
Chapter 3: Implementation Case Studies		48
3.1	At a Glance	48
3.1.1	Vehicles	48
3.1.2	Payloads	49
3.1.3	Other Software	49
3.2	Design Process	49
3.2.1	Phase 1: Determine Interoperability Goals	49
3.2.2	Phase 2: Examine Existing Architecture	50
3.2.3	Phase 3: Define Component Boundaries	50
3.2.4	Phase 4: Construct Components	53
3.2.5	Phase 5: Define Node Boundaries	53
3.2.6	Phase 6: Integrate Components and Verify Performance	54
3.3	MATILDA	54
3.3.1	Functional Requirements	55
3.3.2	Interoperability	55
3.3.3	JAUS Layout	55
3.3.4	Operation	56
3.4	Simulator	57
3.4.1	Interoperability	57
3.4.2	JAUS Layout	58
3.5	Johnny-5	58
3.5.1	Interoperability	60
3.5.2	JAUS Layouts	60
3.5.3	Issues Encountered	61
3.6	Cadillac SRX	62
3.6.1	Why JAUS?	63
3.6.2	Interoperability	64
3.6.3	Software Approach	64
3.6.4	JAUS Layout	65
3.6.5	Issues Encountered	65
3.7	Gemini	66
3.7.1	Interoperability	67

3.7.2	J AUS Implementation	67
3.7.3	At Competition	68
3.7.4	Issue Encountered	69
3.8	Helium Red and the RMAX	69
3.8.1	Vehicle Overview	70
3.8.2	Interoperability	70
3.8.3	J AUS Layout	71
3.8.4	Issues Encountered	71
3.9	Rocky	73
3.9.1	J AUS Layout	74
3.9.2	Validation	74
3.10	Laser Range Finder Payload	76
3.10.1	J AUS Implementation	76
3.10.2	Obstacle Avoidance Payload	77
3.10.3	Issues Encountered	78
3.11	Micro Wireless Instrumented Payload	79
3.11.1	J AUS Implementation	79
3.11.2	Issues Encountered	80
3.12	Operator Control Unit	80
3.12.1	Service OCUs and the Full OCU	81
3.12.2	Aerial Display OCU	81
Chapter 4: Results and Recommendations		83
4.1	Message-related Issues	83
4.1.1	Interpretation of Wrench Efforts	84
4.1.2	Waypoint Management	85
4.1.3	Waypoint Tolerances	86
4.1.4	System Configuration	86
4.2	Protocol Issues	87
4.2.1	Message Ordering	87
4.2.2	Switching from Autonomous Mode to Teleoperation	88
4.2.3	Message Handling with respect to Status	90
4.2.4	Command Filtering with respect to Controller	90
4.2.5	(A)periodic Wrench Efforts	91
4.2.6	Watchdog Control Timeouts	92
4.2.7	System, Subsystem and Node Identification	93
4.3	Message Routing Issues	95
4.3.1	Incoming Message Handling	95
4.3.2	Service Connections	96
4.3.3	Multipart Message Sequence Number	96
4.3.4	Multipart Message Scope	97
4.3.5	Ack/Nak Rules	98
4.4	Other Issues	99
4.4.1	Desired versus Actual States	99
4.4.2	Command Completion	99
4.4.3	Function of the Subsystem Commander	100
4.5	Special Concerns	103

4.5.1	Deprecation of 0408h Set Local Vector	103
4.5.2	Command Service Connections	103
4.5.3	Analysis of Command Service Connection Performance	105
4.5.4	Level-III Interoperability	109
4.6	Innovations	112
4.6.1	Multiple Services per Component	112
4.6.2	Grouping Messages into Services	113
4.6.3	Header on Service Connection Messages	114
4.6.4	Latest-Data Connections for Commands	116
4.6.5	Experimental Messages	118
4.7	Summary of Recommendations	119
Chapter 5: Conclusion		123
5.1	Key Results and Recommendations	124
5.1.1	Benefits of JAUS	124
5.1.2	Key Issues	124
5.1.3	Future of JAUS	125
5.1.4	Accelerating Acceptance	126
5.2	JAUS in LabVIEW	127
5.2.1	Rapid Prototyping	128
5.2.2	Advanced Programming Concepts	128
5.2.3	Intuitive Debugging	128
5.3	Future Work on VT Toolkit	129
5.3.1	Additions to Base Functionality	129
5.3.2	Performance Modifications	130
5.3.3	Modifications for Ease of Use	133
5.4	Conclusion	133
References		135
Appendix A: Interoperability Examples		139
A.1	Rapid Integration: ARA at IVSS	139
A.2	Code Reuse: SRX and Rocky	140
A.3	Cross-Language, Cross-Organization, Level-III	141
A.4	Hypothetical Scenario	142
Appendix B: VT JAUS History		144
Appendix C: Other Recommendations		148
Appendix D: Component Model Screenshots		149

Acronyms

AOL IM	America OnLine Instant Messenger
ARA	Applied Research Associates
AUVSI	Association for Unmanned Vehicle Systems International
BDC	Blind Driver Challenge
Cm	Communicator
DARPA	Defense Advanced Research Projects Agency
EST	Eastern Standard Time
E-Stop	Emergency Stop
ETG	Experimentation Task Group
FCS	Future Combat Systems
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
GPS	Global Positioning System
GUI	Graphical User Interface
HeLiUm RED	Helicopter Lifted Unmanned Reconnaissance and Exploration Drone
ICD	Interface Control Document
IGVC	Intelligent Ground Vehicle Competition
IMU	Inertial Measurement Unit
INS	Inertial Navigation System
IRC	Inernet Relay Chat
IVSS	Intelligent Vehicle Systems Symposium
JAG Fleet	JAUS-enabled Autonomous Ground Fleet
JAUGS	Joint Architecture for Unmanned Ground Systems
JAUS	Joint Architecture for Unmanned Systems
JAUS ID	JAUS Identifier
JAUS WG	JAUS Working Group
JGRE	Joint Ground Robotics Enterprise (previously the JRP)
JOUSTER	JOint Unmanned Systems Test Experimentation and Research
JRP	Joint Robotics Program (now called the JGRE)
JSS	JAUS Services Specification
LabVIEW	Laboratory Virtual Instrumentation Engineering Workbench
LRF	Laser Range Finder
MIP	Mobile Instrumentation Platform
MoE2	Mobile Experimentation Environment
NFB	National Federation for the Blind
NI	National Instruments
NIC	Network Interface Card

NM	Node Manager
OCU	Operator Control Unit
OPC	OCU and Payloads Committee
OS(es)	Operating System(s)
PD	Primitive Driver
PLX WE	Propulsive Linear X Wrench Effort
PRZ WE	Propulsive Rotational Z Wrench Effort
RA	Reference Architecture
RA 3.2	Reference Architecture version 3.2
RDDF	Route Definition Data File
RDECOM	Research Development Engineering Command
RT	Real-Time
SAE	Society of Automotive Engineers
SC	Service Connection
SDK	Software Development Kit
STTC	Simulation Technology Training Center
SUGV	Small Unmanned Ground Vehicle
TCP	Transmission Control Protocol
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
UGV	Unmanned Ground Vehicle
USB	Universal Serial Bus
USPS	United States Postal Service
USV	Unmanned Surface Vehicle
UUV	Unmanned Underwater Vehicle
μ WIP	Micro Wireless Instrumented Payload
VaTAR	Virginia Tech Autonomous Rotorcraft
VI	Virtual Instrument
VT	Virginia Tech
WE	Wrench Effort
WIP	Wireless Instrumented Payload
WM	World Model

List of Figures

1.1	J AUS-interoperable Vehicle Control	5
1.2	Payload Expandability through JAUS	5
1.3	Level-I and Level-III JAUS Interoperability	6
1.4	JAUS Subsystem Configuration	16
1.5	JAUS Message Header (diagram from JAUS RA) citejausra	19
1.6	Repeated Query vs Periodic Event	20
1.7	LabVIEW Block Diagram	25
1.8	LabVIEW Front Panel	26
1.9	Loops in LabVIEW	27
2.1	Standard icons for message builders/parsers	31
2.2	Standard Packing/Unpacking VIs	32
2.3	Component Model Layered Structure	34
2.4	Component Model Front Panel, Settings Tab	41
2.5	Component Model Block Diagram	42
2.6	Front panel of the Node Manager	43
2.7	Real-time message display on the Communicator	44
2.8	Folder organization of the LabVIEW Toolkit	44
2.9	Primitive Driver for Johnny-5	45
2.10	Inside the Message Handler for a Reflexive Driver	46
2.11	Message Build for old version of E438h Report LRF Scan	47
2.12	Message Parse for old version of E438h Report LRF Scan	47
3.1	Virginia Tech’s JAUS-interoperable MATILDA	54
3.2	JAUS Layout for MATILDA	56
3.3	In-vehicle Visualization and 2D View for Simulator	57
3.4	JAUS Layout for Vehicle Simulator	58
3.5	Johnny-5 [17]	59
3.6	JAUS Layout for Johnny-5, at ETG Experiment 2.75	60
3.7	JAUS Layout for Johnny-5, at IGVC	61
3.8	Virginia Tech’s JAUS-interoperable Cadillac SRX	63
3.9	Message Flow for Cadillac SRX	64
3.10	JAUS Layout for Cadillac SRX	65
3.11	Preparing Gemini for an IGVC Run	67
3.12	JAUS Layout for Gemini at the IGVC	68
3.13	CAD Model of Helium Red SUGV [23]	70
3.14	JAUS Layout for Helium Red SUGV	71
3.15	Grand Challenge Entry, Rocky	73

3.16	J AUS Layout for Rocky	74
3.17	Level-III J AUS: Cadillac SRX Code on Rocky	75
3.18	Waypoints driven by Cadillac SRX code running on Rocky	76
3.19	Laser Range Finder Payload	77
3.20	J AUS Layout for Laser Range Finder Payload	77
3.21	Micro Wireless Instrumented Payload	79
3.22	J AUS Layout for Micro Wireless Instrumented Payload (μ WIP)	80
3.23	Primitive Driver Page for Full OCU	82
4.1	Message Flow for Recieving Node and Subsystem Name	94
4.2	Routing for Incoming Messages	96
4.3	System Configuration as Illustrated in J AUS RA [2]	102
4.4	System Configuration as Illustrated by Ruel Faruque	102
4.5	Waypoint Navigation without Command Service Connections [8]	106
4.6	Waypoint Navigation with Command Service Connections [8]	107
4.7	Global Pose / Velocity State Sensor: 2 Services in 1 Component	113
4.8	Message flow between Components on the same Node	115
4.9	Message flow between Components on different Nodes or Sub-systems	116
A.1	Subsystem table as displayed by University of Florida Node Manager	142
D.1	Front Page of Component Model 1.2.1	150
D.2	Settings Tab of Component Model 1.2.1	151
D.3	Events Tab of Component Model 1.2.1	152
D.4	Messages Tab of Component Model 1.2.1	153

List of Tables

1.1	Currently Available J AUS Software	9
3.1	Features of each level of J AUS interoperability	51

Chapter 1

Introduction

1.1 Introduction

JAUS (the Joint Architecture for Unmanned Systems) is an emerging standard for messaging onboard and between unmanned systems, aimed at enabling interoperability between unmanned systems developed by different organizations. The use of JAUS enhances code modularity and enables code reuse, leading to more rapid, less costly development and decreased deployment expense.

This thesis presents a series of case studies in implementing JAUS on a variety of unmanned systems using the first-ever JAUS software development toolkit for LabVIEW, and a collection of valuable implementation issues gleaned during the journey from isolated beginnings to embedded involvement in the JAUS Experimentation Task Group.

1.1.1 JAUS Toolkit for LabVIEW

In any implementation of JAUS, a substantial amount of software must be constructed to achieve even basic interoperability. Development of this core software requires considerable systems engineering and development time. While several groups have developed JAUS libraries for C++ and

Java, none has yet been proposed for LabVIEW, a graphical programming language produced by National Instruments.

The first portion of this thesis introduces the Virginia Tech JAUS toolkit for LabVIEW, describing its use and features.

1.1.2 Implementation Case Studies

To date, most JAUS implementations are produced by groups involved in the JAUS Working Group, which develops and documents the JAUS specification. As a result, the developers/implementers often know far more about the intent of the specification than what is included inside the actual documents. Naturally, the resulting standards document do not always mention assumptions the developers had in mind.

For this reason, it is valuable for an independent group to develop an implementation of JAUS, working from the only from information contained in the standards specification. In addition, these independent developers may take JAUS beyond the capabilities intended by the Working Group. The feedback from such work is invaluable to the JAUS Working Group.

The latter part of this thesis presents a series of implementation case studies and outlines a collection of implementation issues encountered when implementing JAUS on a variety of unmanned systems.

1.1.3 Organization of Thesis

A variety of background information is presented first, including a cursory survey of previous work in JAUS, and an introduction to JAUS, from the Virginia Tech point of view. Next the elements of the VT JAUS Toolkit are discussed, and a set of examples demonstrating its use are presented.

A series of implementation case studies follows, each with a problem definition, interoperability requirements, discussion of the JAUS layout, and implementation issues. Finally, a collection of issues encountered while im-

plementing JAUS is presented. In the conclusion, lessons learned, future work on the Toolkit, and the future of JAUS are discussed.

1.2 Motivation for a Standard

In 2000, the United States Senate mandated that 30 percent of all military vehicles should be unmanned by the year 2015 [1]. To accelerate the development of unmanned systems, several initiatives were taken, including:

- the Future Combat Systems program: a special military funding stream aimed at developing the deployable software and hardware to achieve the congressional mandate
- the DARPA Grand Challenge: initiated later, a series of race-like competitions intended to spark research and development in unmanned systems [15]

Development quickly ensued, and several teleoperated unmanned ground vehicles (UGVs) and many fully automated unmanned aerial vehicles (UAVs) have been successfully deployed in a variety of scenarios.

1.2.1 Need for an Interoperability Standard

Early on, retired Colonel Jeff Kotora noticed a key development issue with unmanned systems. The typical unmanned system (base vehicle platform, operator control unit, and any associated payloads), were all developed by one organization or company, and each company had implemented its own message set for communication between elements of its unmanned system.

As a result, each particular unmanned system was incompatible with other systems without substantial integration work. The problems were most evident in the deployment arena; if a military platoon deployed three different remotely operated UGVs, it needed three separate Operator Control Units (OCUs), one for each vehicle. Furthermore, the soldiers had to

be trained in the operation of three separate OCUs in order to fully deploy all three UGVs.

In addition, since no standard existed for communication between the components of an unmanned system, unmanned systems development was handled by large companies with a diverse skill set. Unless a time-consuming integration process was undertaken, it was impractical for smaller companies to focus on developing an expertise in niche areas.

1.2.2 The Birth of JAUS

With this set of problems in mind, Colonel Kotora proposed the idea for JAUGS – the Joint Architecture for Unmanned Ground Systems. The Joint Robotics Program, under the aegis of the Office of the Secretary of Defense, assembled a Working Group, made up of military research labs, industry partners, and academia, to develop the standard.

Later, the “G” was dropped, as JAUS (now the Joint Architecture for Unmanned Systems) sought to extend beyond the UGV arena into unmanned underwater vehicles (UUVs), unmanned water surface vehicles (USVs), and unmanned aerial vehicles (UAVs).

1.2.3 JAUS Functionality

The initial vision for JAUS is depicted in Fig 1.1. In this example of remote control, the Controller is translating joystick values to a standard JAUS mobility message. The Unmanned System receives this standard JAUS mobility message and translates it to vehicle-specific actuator commands.

Since the only interaction between the Controller and Unmanned System is a standard JAUS message, the two elements of the remote controlled vehicle could be developed independently by two separate organizations. Further, in the field, any JAUS-interoperable controller could theoretically be used to control any JAUS-interoperable vehicle. This is referred to in

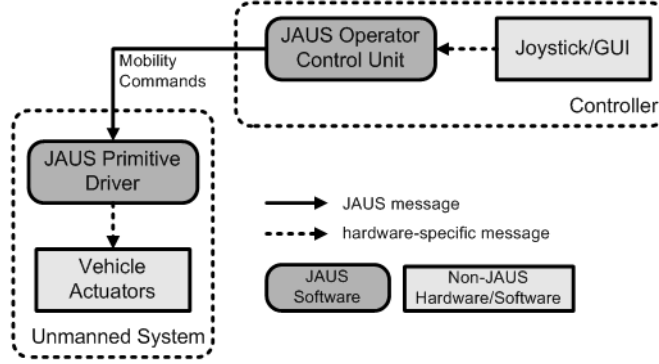


Figure 1.1: JAUS-interoperable Vehicle Control

JAUS as Level-I interoperability.

Payload Expandability

Taking interoperability one step further, we add a payload (another computer) to the vehicle, containing a GPS receiver and a Waypoint Driver, which receives Target GPS Waypoints from the Controller and generates the above standard JAUS mobility message. This message is then sent to the Unmanned System, which then translates the message into vehicle-specific hardware commands and executes them. Fig 1.2 illustrates this modification.

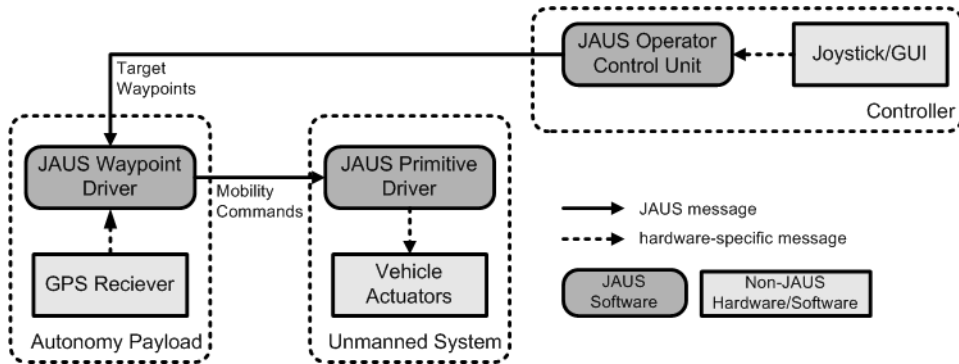


Figure 1.2: Payload Expandability through JAUS

In effect, we have added autonomous capability to the original Unmanned System *without modifying it at all*. We simply added an autonomy payload to simulate the mobility commands from the Controller.

In addition, since the payload interacts with the vehicle using only JAUS messages, it could be replaced with a Road-Following or Obstacle Avoidance payload, as long as the payload produced JAUS mobility messages for the vehicle. This type of payload expandability is referred to in JAUS as Level-II interoperability.

Code Re-use

If the original Unmanned System had supported a higher level of interoperability, we could have added the Waypoint Driver software directly to the Unmanned System, as shown in Fig 1.3. We could also separate out the hardware interface to the GPS receiver, so that the Waypoint Driver receives pose (location and orientation) from the GPS via a standard JAUS pose message. This type of payload expandability is referred to in JAUS as Level-II interoperability.

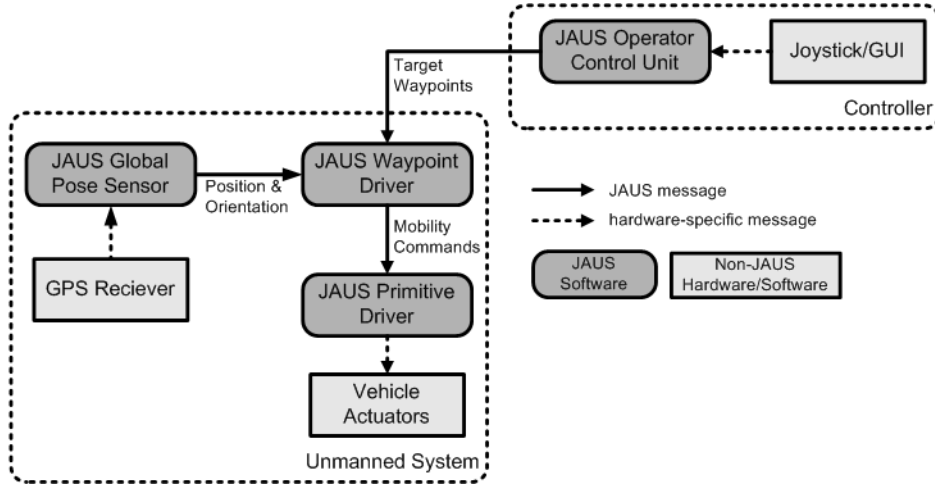


Figure 1.3: Level-I and Level-III JAUS Interoperability

Note that the Waypoint Driver now interacts with other software and hardware using only JAUS standard messages. This enables the Waypoint Driver to be replaced with another Waypoint Driver (employing a completely different algorithm), with no integration effort. This is referred to in JAUS as Level-III interoperability.

1.2.4 JAUS Working Group

The primary development of the JAUS standard is accomplished by the JAUS Working Group, a collection of military research labs, industry, academia, and other users. In meetings approximately four times per year, the group convenes to propose new ideas, discuss work since the last meeting, vote on documents, and approve changes to the standard.

In 2005, JAUS was adopted as an official standard by SAE (the Society of Automotive Engineers). Given the moniker AS-4, the Unmanned Systems Committee comprises the following subgroups:

AS-4A Architecture Framework

This committee is responsible for outlining the domain of JAUS, establishing compliance specifications, and promoting the standard.

AS-4B Network Environment

In this committee, rules are established for passing JAUS messages over existing mediums, such as TCP, UDP, serial, USB, Bluetooth, or CAN.

AS-4C Information Modeling

The messages and component behaviors making up the JAUS standard are the responsibility of this group. It often spawns task groups, which are given the job of developing new messages and behaviors for a particular application; example groups include the Mission Planning Task Group and the World Modeling Task Group.

Experimentation Task Group (ETG)

Previously known as the OPC (OCU and Payloads Committee) this group is responsible for conducting experiments to verify the validity and usability

of the JAUS standard. Recommendations from this group are brought to the other committees for review.

1.3 Adoption of JAUS

JAUS has currently been accepted by a variety of military research labs, small companies, and universities. With the recent requirement that elements of the Future Combat Systems must be JAUS-compliant, larger government contractors have taken more interest in the standard.

While most of the developers currently implementing JAUS are actively engaged in the JAUS Working Group, only a few organizations have implemented JAUS independent of the Working Group. As mentioned before, these independent, isolated developers can often provide valuable feedback on the completeness and clarity of the publications the JAUS Working Group has released.

1.3.1 Current Products

While most of the work in JAUS is in the research/development phase, several groups have generated production software and hardware, available for purchase. A few of the more prominent products available are mentioned in Table 1.1.

1.4 Motivation at Virginia Tech

In 2004, the Virginia Tech Unmanned Systems Group (headed by Mechanical Engineering professor Dr. Charles Reinholtz) was awarded a multi-million-dollar military contract by the Army Research Development Engineering Command Simulation Technology Training Center (RDECOM STTC) to develop instrumentation and infrastructure for a dedicated unmanned vehicle test site.

Table 1.1: Currently Available JAUS Software

Operator Control Units

- Autonomous Solutions, Mobius
- Defense Technologies Inc, MURC
- Applied Perception, API OCU
- Spawar San Diego, MOCU

Software Development Kits (SDKs)

- University of Florida, OpenJAUS
- Virginia Tech, JAUS Toolkit for LabVIEW
- Applied Perception, jLib and jNM
- Autonomous Solutions, jausNOW
- kairosAutonomi, JAUS SDK
- RE2, JAUS SDK

1.4.1 JOUSTER

The site was to be called JOUSTER (JOint Unmanned Systems Test Experimentation and Research). It would provide a unique Mobile Experimentation Environment (MoE2) to support the testing of unmanned vehicles, as well as an array of JAUS-interoperable vehicles available for customer use. Several key components of the JOUSTER site include:

- Mobile Instrumentation Platform (MIP): a trailer-mounted tower providing wireless 802.11 network coverage, a GPS differential base station, a weather station, and pan-tilt-zoom cameras.
- Wireless Instrumented Payload (WIP): a collection of off-the-shelf components assembled into an enclosure to be mounted on a vehicle. The WIP records position, orientation, vibration, temperature, and other data, independent of the vehicle systems. All data is logged locally and streamed wirelessly to the experiment base station.
- Micro Wireless Instrumented Payload (μ WIP): smaller version of the WIP, designed for use on-board smaller UGVs or UAVs.

- Multi-Level Wireless Emergency Stop System: a system of remotely controlled relays, which can be interfaced to a vehicle’s hardware to enable the vehicle to be issued three levels of Emergency Stop.
- Variable Weight Tow Sled: a tow-capacity test device for evaluating the pulling capacity of small UGVs.
- VaTAR: an autonomy-enabled Yamaha RMAX unmanned helicopter
- Helium Red: a small JAUS-enabled UGV deployable from the RMAX.
- MATILDA: a JAUS-enabled Mesa Robotics MATILDA.
- JAG Fleet: a fleet of JAUS-enabled autonomous ground robots.
- JAUS-interoperable Laser Range Finder Payload: a JAUS-interoperable payload capable of object detection and ranging, typically used to add obstacle avoidance or sensor capability to a vehicle.
- Command and Control Trailer: a mobile command and control trailer for experiment staging and control.
- Base Station Software: Command and Control software capable of tracking instrumentation and vehicle control.
- Experiment Coordination and Support: a team of underpaid graduate students available for experiment coordination and support.

1.4.2 Evolution of JAUS Project at Virginia Tech

Initially, JAUS was undertaken simply to provide the MATILDA with JAUS interoperability. As work progressed, the project evolved into a JAUS software development toolkit, implemented on nearly every element of the JOUSTER MoE2 environment.

Currently, almost all communication within MoE2 is accomplished via JAUS. With such a setup, customers of the site can easily integrate their own software/hardware in advance, saving valuable test time.

1.4.3 Why VT Uses JAUS

Unmanned systems developers at Virginia Tech have a slightly different set of reasons for implementing JAUS than the majority of developers.

Traditional: External Interoperability

Traditionally, most groups employ JAUS to ensure interoperability with unmanned systems developed by other organizations. The focus is on level-II or level-I interoperability.

Such an approach lends itself to an after-the-fact JAUS implementation. Since JAUS is only a deployment concern, the development team can create the vehicle software just as they would without JAUS, and then add a JAUS translator once the vehicle has been appropriately tested.

Large Team: Internal Interoperability

In unmanned vehicle design, software is often relegated to a small team of two to four individuals who interact closely during the project, eliminating any interoperability issues. In a small development team such as this, a standard may not be necessary.

However, with the growing complexity of problems that unmanned vehicle software must solve, a larger team is generally dedicated to software. On a large development team of software engineers, modular design and a standard for inter-component communication are imperative. While the team could establish its own messaging standard, it often makes more sense to implement JAUS, a standard already in existence.

In addition, the possibility for testing software on multiple vehicles necessitates a standard for messaging between the vehicle's intelligence software and its hardware interface software.

Virginia Tech Reasons for JAUS

Hence, the following summarize the motivation for implementing JAUS at Virginia Tech.

- JAUS facilitates the division of work in a large team, through level-III interoperability
- JAUS enables rapid redistribution of processes to balance computing resources and software complexity
- JAUS enables easy transport of code from vehicle to vehicle
- JAUS provides level-I and II interoperability with other MoE2 (Section 1.4.1) elements and with customer equipment

1.5 Introduction to JAUS

This section presents an introduction to JAUS as we understand it at Virginia Tech. The current JAUS specification is vague in some areas, and those areas are filled in here with information gleaned from long discussions with the Virginia Tech JAUS development team, and from participation in the Working Group and Experimentation Task Group. These ambiguities in the specification and the assumptions I made in implementing JAUS are the focus of this thesis, and are presented in Chapters 3 and 4.

1.5.1 What is JAUS?

JAUS is an emerging standard for messaging within and between unmanned systems, aimed at enabling interoperability between unmanned systems developed by different organizations. The use of JAUS enhances code modularity and enables code reuse, leading to more rapid, less costly development and decreased deployment expense. SAE is in the process of adopting JAUS as an aerospace standard, AS-4.

At its core, JAUS is simply a message set, specifying the bits and bytes. A standard header is defined, and body compression for the data associated with each particular message code.

JAUS also defines components – software modules that accept and produce JAUS messages. Each component is given a unique JAUS identifier which enumerates it as a Component and locates it in a Subsystem and on a Node. To support the components and messages, transport components and routing rules are defined, which actually take the messages from one component to another.

Scope

It is important to understand that JAUS is not a communication standard such as Ethernet. JAUS defines the data being passed, not the physical mechanism for passing the data, although it does provide rules for passing JAUS messages over various mediums, such as TCP, UDP, serial, USB, Bluetooth, or shared memory.

In other words, JAUS does not define a transport medium, but it specifies how to make use of existing mediums.

It may help to understand JAUS by thinking about it as a language, in which the messages are words. If I understand English, I can write it on paper, I can speak it, or I can print it in Braille. In each of these three mediums, there are different rules of use, but in all three cases I am still communicating in English. Likewise, JAUS defines messages (AS-4C, Information Modeling), and also defines rules (AS-4B, Transport) for passing JAUS messages over a particular medium, such as UDP, TCP, serial, USB, Bluetooth, or shared memory.

Extending the language analogy, the grammar in English corresponds to protocol in JAUS, answering the questions: In what order must messages be sent, and what behavior can the sender expect?

1.5.2 JAUS Purpose

For a graphical introduction to JAUS, illustrating the purpose behind and functionality provided by the standard, see Section 1.2.3.

1.5.3 Software Perspective

Software can be understood from two points of view. First, we can examine the logical flow, without regard to how data actually gets from one process to the next. This approach is much like a program flow diagram.

The second point of view details the physical interactions between the processes. It answers questions such as, are the processes all running on the same computing unit? Are they even on the same vehicle? Does the data pass in shared memory, FIFO queues, over a UDP port, serial port, or some other medium?

Because JAUS is a messaging standard, it should not restrict the logical flow of the software, only the physical manner in which the software modules interact. Thus, in applying JAUS to an unmanned system, we view software from the second perspective.

1.5.4 Terminology

Components are the basic building blocks for a JAUS-interoperable unmanned system. In order to support services, components interact by passing JAUS messages. The following terms are helpful in understanding JAUS.

- **Component:** A program that provides one or more services, and interacts with other components using JAUS messages.
- **Node:** A computer on which components run. For our purposes, something with a distinct IP address.
- **Subsystem:** A vehicle or OCU, for most purposes. Strictly, a logical grouping of components spread across one or more nodes [2].

- **Service:** A functional capability provided by a JAUS component, such as mobility or positioning.

1.5.5 System Layout

In defining an example JAUS-interoperable unmanned system, let's start with two components, ComponentA and ComponentB. Being software programs, the components reside on a computer, Node1. Now we introduce rule 1.

Rule 1: Every node must have one and only one Node Manager service, which is responsible for routing all message traffic on the node.

Node Manager as a Post Office

We can visualize the Node Manager (NM) as the post office in a small town. When a component sends a message, it puts it in a FIFO queue analogous to the big drop box at the post office. The NM dequeues messages, determines where they go, and either puts them in a FIFO queue for a local component (post-office box), sends them to another node (USPS truck to a different city's post office), or forwards them to a special component managing the connections to other subsystems (international air mail).

Service Connections

One subset of messages does not pass through the Node Manager – service connections. Designed for periodic updates where only the most recent data is desired, service connection messages bypass the Node Manager and are never queued. Instead, they flow directly from one component to another via shared memory, similar to a bicycle messenger in a large city.

Subsystems

We can also add a second node, Node2, with some other components, and group everything as a Subsystem. For communication between the nodes, most implementers use Ethernet, but other transport mediums are acceptable as well, such as serial.

Now that we have a subsystem, JAUS specifies two more services. First, a Communicator serves as a single point of entry to the Subsystem, managing the communications link. All traffic in and out of the subsystem passes through the Communicator. While this is specified in the current Reference Architecture, it can cause a bottleneck problem, and may change as experimental port forwarding (recently evaluated by the JAUS Experimentation Task Group) is implemented. Second, a subsystem commander coordinates the components within the subsystem. A subsystem contains only one Communicator and Subsystem Commander service, typically located on the primary node. Fig 1.4 depicts a typical JAUS subsystem configuration.

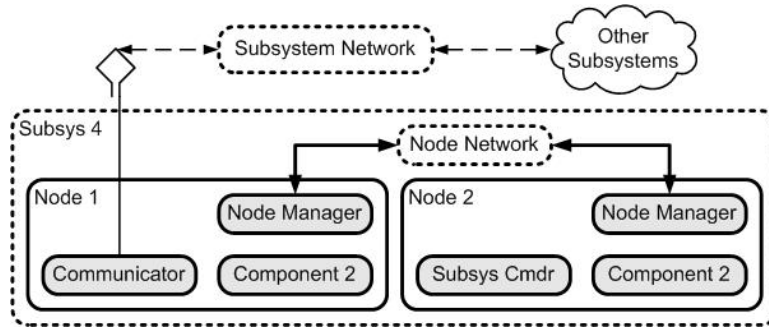


Figure 1.4: JAUS Subsystem Configuration

Rule 2: A subsystem can contain one and only one Communicator service, which serves as a single point of entry to the subsystem.

Identifiers/Addresses

There is one more rule. Each component has a JAUS identifier (ID) allowing it to be addressed. When a component sends a message, it needs to only

specify the destination JAUS address (JAUS ID), and the routing inherent in JAUS transports the message to the recipient. The JAUS ID is defined as three fields: Subsystem.Node.Component. Currently there is an additional field for Instance, but that will deprecate in the next major revision of JAUS.

Rule 3: Every JAUS component has an identifier, defined as Subsystem.Node.Component. Component ID 1 is reserved for the Node Manager.

1.5.6 Components and Services

While components are JAUS entities, services are functions that a component may provide; messages are passed between components to support a service. For example, if a component, supports the Global Pose Sensor service (38), it responds to and generates messages related to the Global Pose (location and orientation – latitude, longitude, elevation, roll, pitch, and yaw) of the platform. JAUS defines a list of services and their associated messages.

Another service discussed later is the Primitive Driver service (33), which translates the standard JAUS Wrench Efforts message into vehicle-specific actuator commands. The Wrench Efforts message contains the percent of available propulsive and resistive effort in 3 rotational and 3 translational degrees of freedom.

In the current revision of JAUS, a component can provide one and only one service, bound to the component field of the JAUS ID. However, with the introduction of the experimental Query/Report Services messages in the ETG, it is possible for one component to provide multiple services, such as obstacle avoidance and waypoint driving, if the designer so chooses.

1.5.7 JAUS Messages

There are two core parts to JAUS: routing and messages [35]. Messages provide the words for the JAUS standard, while Node Managers and Com-

municators handle the routing and transport of messages from source to destination.

Parts of a JAUS message

Depicted in Fig 1.5, the JAUS message header is 16 bytes long, is attached to every JAUS message, and contains the following information:

- Message Properties: Version, Ack/Nak, Service Connection Flag, Experimental Message Flag
- Command Code: Unique to the message, determines behavior and what data is included in the body
- Destination ID: JAUS address of the destination, Subsystem, Node, Component, Instance
- Source ID: JAUS address of the sender, Subsystem, Node, Component, Instance
- Sequence Number: Increments on successive messages, used for a variety of purposes
- Message Length and Multipart Reassembly fields: Used for packet transmissions

The body of a JAUS message ranges from 0-4080 bytes in length, and often contains a presence vector, where each bit indicates the presence of a particular field in the message body. [2]

Message Taxonomy

On a high level, messages are divided into three classes: Command, Query, and Report. Commands effect an action and may be countered with a confirmation or rejection. Query messages are requests for information, and result in Report messages containing the information. The messages can also be organized into functional groups.

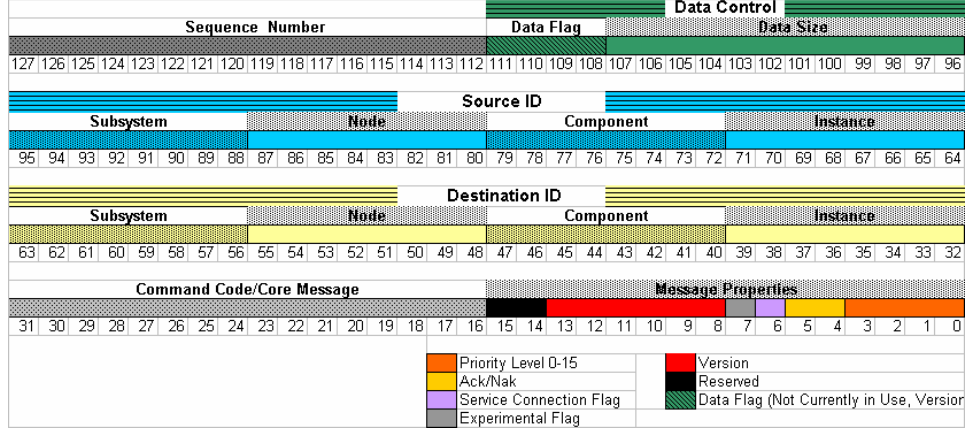


Figure 1.5: JAUS Message Header (diagram from JAUS RA) citejausra

Core

This set contains messages that every component must support, regardless of what other services it supports. At Virginia Tech we group them into the following subsets:

Status Control: Standby, Resume, Shutdown, Reset, Set Emergency, Clear Emergency.

Access Control: Request Uninterruptible Control, Release Uninterruptible Control.

Dynamic Configuration: Using dynamic configuration messages, a component can query another for a list of the services (in effect, messages) it supports, and thus determine what a component does. This group also contains messages for reporting the components on a node or subsystem, and a human-readable name for a component, node, or subsystem.

Event Setup: Suppose a GPS Waypoint Driver component desires the current location (Global Pose) of the vehicle at 5 Hz. Event setup messages allow the waypoint driver to set up a periodic event with the Global Pose Sensor, instead of querying at 5 Hz, as indicated in Fig 1.6.

Two types of periodic events are available: periodic without replacement, and service connections. Periodic without replacement causes the messages

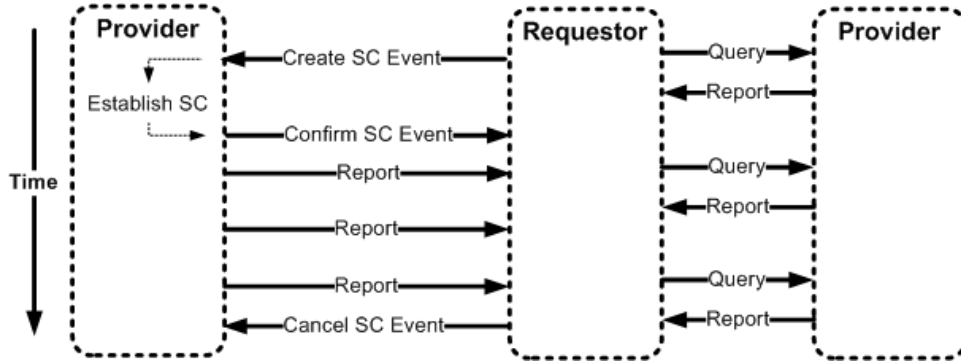


Figure 1.6: Repeated Query vs Periodic Event

to be sent via normal routing, while service connection messages travel on latest-data connection shortcuts, with new data replacing previous data, the messages never being queued [2].

Change-based events can also be requested, where the providing component is asked to send a message only when a particular field crosses a threshold. For example, an component could request an alert when the altitude dropped below a certain value.

Heartbeat

The heartbeat pulse is used to proclaim “I’m alive”. In most applications, each component sends a periodic heartbeat to the local Node Manager, indicating that it is operating. Each Node Manager usually broadcasts a periodic heartbeat message to other nodes within the subsystem, and one component within the subsystem broadcasts a periodic heartbeat to the other subsystems.

These messages are most often used for dynamic configuration and safety watchdog timeouts. For example, a teleoperated robot could enter a standby state if a heartbeat is not detected from its controlling OCU within a certain time period.

Platform

Messages related to the vehicle platform, such as Set GPS Waypoint, Query Transmission Gear, Set Travel Speed, Report Velocity State, etc.

Environment Sensor

Messages related to sensors (mostly exteroceptive), such as Query Global Pose, Set Camera Pose, etc.

Mission Spooling

In JAUS, missions are simply lists of messages, in the form of n-ary trees of sub-tasks. A Mission Planner can generate a mission (for example, to patrol an area for intruders), for a Mission Spooler to distribute subtasks to the patrol vehicles. Change-based events are typically used to monitor mission progress.

1.5.8 Compliance

Currently, there is no formal method for assessing JAUS compliance, so we cannot claim an unmanned system is JAUS-compliant. However, there are metrics to discuss the level of JAUS interoperability present within an unmanned system. Later in Section 3.2, Table 3.1 presents an overview of the features of each level of JAUS interoperability.

Note that successive levels do not include the previous level. It is possible to implement Level-III interoperability without implementing Level-II or Level-I.

Level-I Interoperability

Messages between subsystems are JAUS, but the implementation inside each subsystem is not constrained. This allows multiple robots to communicate with eachother, or one OCU to control several robots.

Level-I interoperability is illustrated in Fig 1.1.

Level-II Interoperability

Messages between Nodes are JAUS, but the node internals are unconstrained. This would allow a computer to be shared between vehicles, perhaps as a payload.

Level-I and Level-II interoperability are demonstrated in Fig 1.2.

Level-III Interoperability

Messages passed between Components are JAUS. Theoretically, this would allow components to be shared between unmanned systems. Certainly, the hardware issues of data sharing would require resolution first, but JAUS provides the language for the transactions. To date, level-III interoperability has not been formally defined.

Level-I and Level-III interoperability are shown in Fig 1.3.

Interoperability vs Compliance

Since there are currently no formal methods for assessing compliance, we do not refer to Virginia Tech systems as JAUS-compliant. Instead, we describe the JAUS-interoperability of a system. For example, the level-III test in Section A.3 demonstrates level-III *interoperability*, not level-III *compliance*.

1.5.9 Differences in VT Understanding of JAUS

My understanding of JAUS is slightly different than that of other developers in the JAUS Working Group, largely because I began implementing JAUS by reading the specification alone, without the extra knowledge the developers in the Working Group possess. The resulting assumptions and ambiguities are the topic of this thesis, but the key differences in my above explanation of JAUS are noted below.

- Rules: The above “rules” are my own creation. While this information is contained in the JAUS specification, it is not presented in this fashion.
- Multiple Services per Component: Judging by the reaction of the Working Group to this issue, it is my understanding that the prevailing assumption is one service per component.
- Payload Definition: Most implementers understand payloads to contain their own user interface, which is displayed on the OCU via the payload message set in a fashion similar to Remote Desktop for Windows XP [9]. At Virginia Tech, we understand a payload to be a computing device with a JAUS interface (a JAUS node) which can support any functionality that can be encapsulated in a JAUS component, such as obstacle avoidance, a sensor interface, or even an operator control unit component.
- Service Connections on Shortcuts: From interacting with many groups in the ETG, the idea of passing service connection messages on shortcuts is relatively unique to the VT implementation. Many other groups simply pass the messages through the normal FIFO queues for routing.

1.6 Introduction to LabVIEW

Since 2002, the Virginia Tech autonomous vehicle program has employed LabVIEW for most of its programming needs, primarily for its ease of use. This graphical programming language developed by National Instruments dramatically decreases development time, enabling research code to be prototyped extremely fast.

This section presents a cursory introduction to several LabVIEW concepts that will help in understanding the toolkit.

1.6.1 Brief History

First introduced in 1986, LabVIEW was initially intended to enable engineers with little programming experience to rapidly integrate data acquisition and testing equipment in the laboratory. National Instruments sought to revolutionize instrumentation by divorcing the processing software from the acquisition hardware in laboratory instruments. Harnessing the power of the personal computer, the idea of virtual instrumentation enabled engineers to conduct multiple tests with the same acquisition hardware.

Since then, LabVIEW has evolved into a fully functional programming language, and today, engineers use LabVIEW to automate manufacturing and inspection in factories, acquire test data for the Xbox video game controller, and control lasers to perform eye surgery [4]. Perhaps the most significant indication of the software's maturity is that software engineers at National Instruments currently develop new LabVIEW features using LabVIEW [4].

1.6.2 Benefits of LabVIEW

The plethora of available drivers, combined with the intuitive graphical environment, enable novice programmers to develop complex applications with a very short learning curve.

In addition, LabVIEW simplifies the use of advanced programming concepts such as parallel processing, dataflow programming, real-time execution, FIFO queues, and notifications, for example. New features allow the same source code to be compiled for Linux, Windows, Macintosh, real-time OSes, embedded devices, and even FPGAs (Field-Programmable Gate Arrays).

1.6.3 LabVIEW Environment

A LabVIEW program (or *virtual instrument*, VI) has two parts: the *block diagram* (Fig 1.7), which contains the code to perform operations, and the *front panel* (Fig 1.8), or the user interface. Each input (control) or output (indicator) on the front panel corresponds to a terminal on the block diagram, and data is passed via wires, in a parallel dataflow fashion, with function nodes executing when all inputs are available.

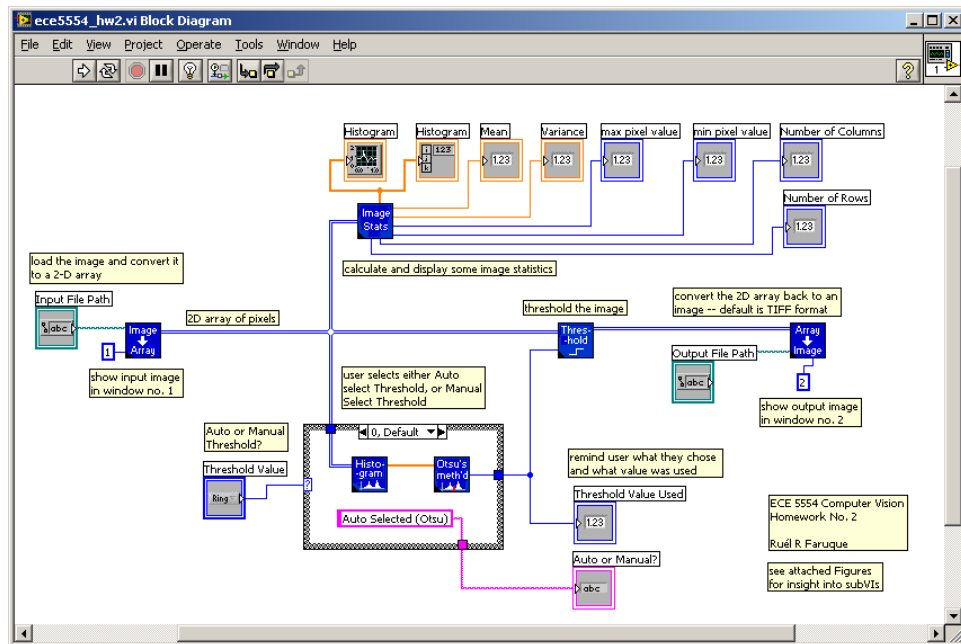


Figure 1.7: LabVIEW Block Diagram

Loops

Loops in LabVIEW are represented as boxes enclosing the code to be repeated. Fig 1.9 depicts a do-while loop in LabVIEW; the code inside the box repeats until the boolean wired to the stop condition becomes true. A for loop, also shown, operates in a similar fashion.

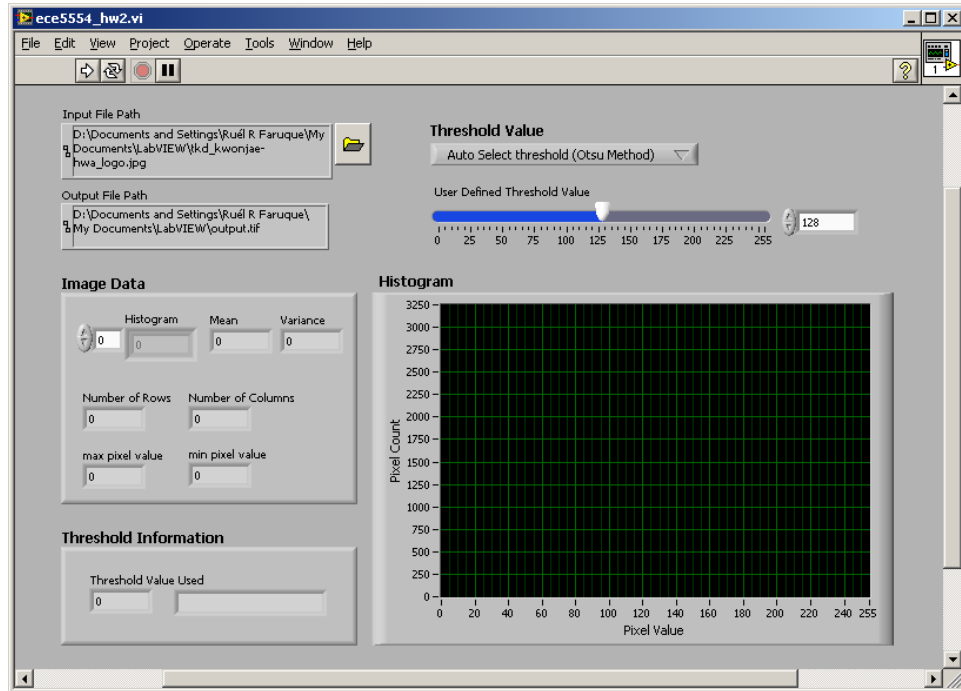


Figure 1.8: LabVIEW Front Panel

Clusters

A cluster in LabVIEW is simply a bundle of data, analogous to a struct in traditional text-based programming languages. It is good practice to make a type definition for clusters, so that any changes to the type definition are reflected in all VIs which use it as a control or indicator (input or output).

SubVIs

A subVI is the LabVIEW term for subfunction. By assigning the controls and indicators to terminals and creating an icon, a programmer can develop a subVI that can then be dropped onto the block diagram of other VIs. The blue VIs in Fig 1.7 are subVIs to perform basic computer vision operations.

Application Control

A set of application control VIs allows the programmer to dynamically modify and call LabVIEW objects, and in the near future, generate new Lab-

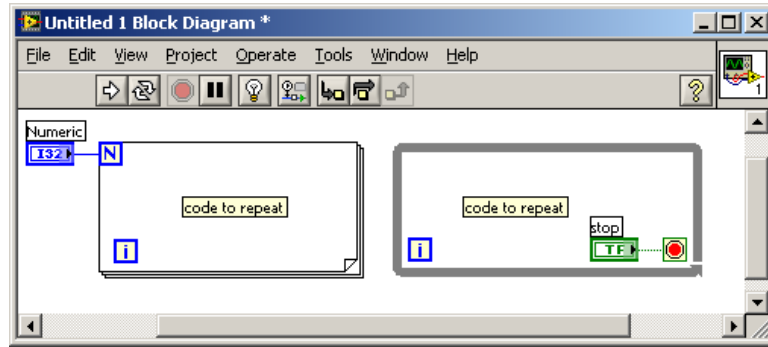


Figure 1.9: Loops in LabVIEW

VIEW code.

This collection of VIs is colloquially referred to as VI Server, and its typically used when one VI needs to programatically interact with another VI's user interface. Note that this is far different from placing the controlled VI as a subVIs on the block diagram, and instead describes an interaction between two top-level programs. The closest example outside of LabVIEW would be .NET.

Chapter 2

VT JAUS Toolkit Overview

In any implementation of JAUS, a substantial amount of software must be constructed before even basic interoperability can be achieved: data structures, message builders and parsers, component templates, routing logic, and routing components. Development of this “core software” requires considerable systems engineering and development time if the result is to be versatile, modular, and reusable.

Several groups have developed JAUS libraries for C++ and Java (Table 1.1), but none has yet been proposed for LabVIEW , a graphical programming language produced by National Instruments. This chapter presents an overview of the Virginia Tech JAUS Toolkit for LabVIEW. We begin with a discussion of the Toolkit scope, followed by a discussion of the Toolkit elements. The chapter concludes with a set of examples demonstrating use of the Toolkit.

2.1 Toolkit Scope

As discussed in the introduction, one of the primary goals for the JAUS Toolkit was to enable rapid implementation on a variety of equipment at the JOUSTER site, mostly vehicles and instrumentation. Hence, the focus was on the baseline communications architecture and the vehicle-related

services in JAUS.

2.1.1 Message Set Version

The JAUS Toolkit for LabVIEW is based on the messages used in Experiments 2.75 and 3.0 conducted by the JAUS Experimentation Task Group (ETG).

These messages are available in the JAUS Reference Architecture version 3.2 (RA-3.2) and in the Interface Control Document (ICD) for ETG Experiments 2.75 and 3.0 [2], [3].

Exclusions

Nearly all messages from the RA-3.2 and ETG ICD are included in the toolkit, except for the Manipulator and Payload Interface messages. The toolkit allows for rapid implementation of new or experimental messages, so implementation of these would be trivial.

2.1.2 Interoperability

Systems developed with the VT Toolkit are interoperable with systems developed according to JAUS Reference Architecture version 3.2, as well as subsystems that have also implemented messages contained in the Interface Control Documents for ETG experiments 2.75 and 3.0.

Though the Toolkit implements experimental messages contained within the Interface Control Documents for the ETG experiments, it is fully capable of interoperating with an RA-3.2 subsystem. While the Toolkit employs the messages from the ETG ICDs, it makes no requirement that other subsystems do. This is evidenced by the test with Applied Research Associates at the Intelligent Vehicle Systems Symposium (see Section A.1).

2.1.3 Assumptions

Given that the Toolkit was refined during its use in JAUS ETG Experiments 2.75 and 3.0, most of the assumptions about protocols and system layout are in line with those of other participants in the ETG. This was not always the case. At ETG 2.75, we corrected many assumptions gleaned from an independent understanding of the JAUS specification. These assumptions are the focus of this paper, and are presented in Chapters 3 and 4.

2.2 Toolkit Elements

The JAUS Toolkit for LabVIEW comprises the following parts:

1. Message Builders/Parsers: Subroutines to convert data to/from JAUS strings
2. Templates: Model message builders/parsers, Model message handlers, and the Component Model, an empty component VI supporting the core message set and containing “put your code here” directions
3. Routing Components: A Node Manager and Communicator, to transport messages from component to component
4. Code Modules: Pre-made message handlers and state machines, to support a service such as Primitive Driver, and pre-built messaging routines, such as “Establish Global Pose Service Connection”
5. Components: several pre-defined components, such as a Primitive Driver or Global Pose Sensor
6. Utilities: debugging tools such as a playback function for message logs.

2.2.1 Message Builders/Parsers

The VIs in this portion of the toolkit handle the unpacking and packing of the data in a JAUS message, except for the header. For example, a VI

called `jaus_0405-build-Set-Wrench-Effort.vi` is responsible for packaging the body of the Set Wrench Effort message, and a corresponding VI, `jaus_0405-parse-Set-Wrench-Effort.vi` would unpack the body into a set of Wrench Efforts.

Standard VI Structure

All messages provided in the Toolkit follow a standard structure and use a standard icon template (shown in Fig 2.1), making them readily identifiable. In addition, a standard naming scheme for VI filenames allows rapid integration of new experimental messages. A blank “Message Model” aligned with the standard structure is provided with the Toolkit, containing instructions for implementing a new message.



Figure 2.1: Standard icons for message builders/parsers

Standard packing VIs

On a message builder/parser block diagram, individual message fields are formed using standard packing VIs, ensuring that the packing details are correct. By removing the burden of packing details from a programmer implementing a new message, fewer syntactical errors are encountered in implementing new messages. Fig 2.2 shows these VIs on a block diagram, with the drop-down field-select menu expanded.

Experimental Messages

Using the Message-Model and the standard field packing/unpacking subVIs, it is simple for a novice to implement new experimental messages in the

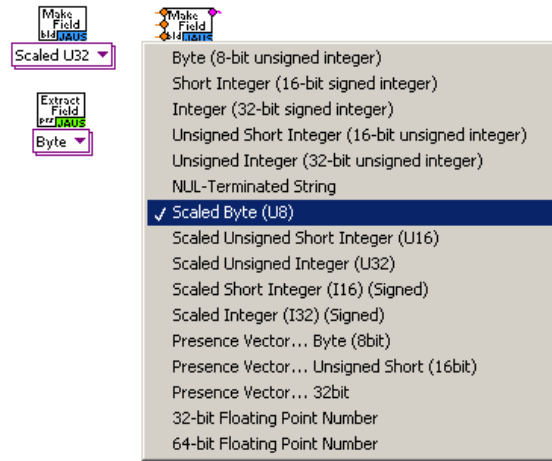


Figure 2.2: Standard Packing/Unpacking VIs

Toolkit. This is demonstrated in Section 2.4.

2.2.2 Component Model

The Virginia Tech implementation of JAUS centers on a Component Model framework. The Component Model is implemented as template VI serving as a baseline JAUS component, supporting the core message set [2][sect4.3].

The four main subVIs of the Component Model handle the following tasks:

- load config and allocate resources for component
- establish and maintain connection to Node Manager
- send heartbeat and perform dynamic configuration
- receive messages, parse header, handle ack/nak, reassemble packet transmissions, sort by priority
- handle core messages (request/release control, status control, event setup, dynamic configuration, heartbeats), maintaining a “core state”
- logging of all incoming message traffic

Typical Use

To develop components, a programmer augments the Component Model by adding extra functionality, in the form of services modules (see Section 2.2.4). If the services modules provided in the toolkit do not suffice, it is easy to develop a new service module. This is demonstrated in Section 2.4.

Usually, two main sections of code must be written. For example, for a Global Pose Sensor, the toolkit provides a message handler to handle incoming messages (queries for global pose). A separate vehicle-specific portion of code must be written to interact with the hardware GPS, INS, or Compass.

Layered Structure

The structure of the Component Model, depicted in Fig 2.3, indicates this distinct layering of functionality. A vehicle-specific Primitive Driver is only a slight augmentation of a Primitive Driver model, which itself is built upon the Component Model.

Other Features

A real-time message display can be filtered by ID and command-code on the fly, helping to answer the pervasive “did you get my message?” question that arises during testing.

A programmer-accessible console.out display is available to easily show internal errors, an idea taken from Autonomous Solutions Mobius OCU [24].

The message logging feature automatically timestamps messages, records optional notes with each message, and can be turned on or off at any time.

Front Panel Organization

The Component Model front panel organizes the GUI indicators and controls into four tabs:

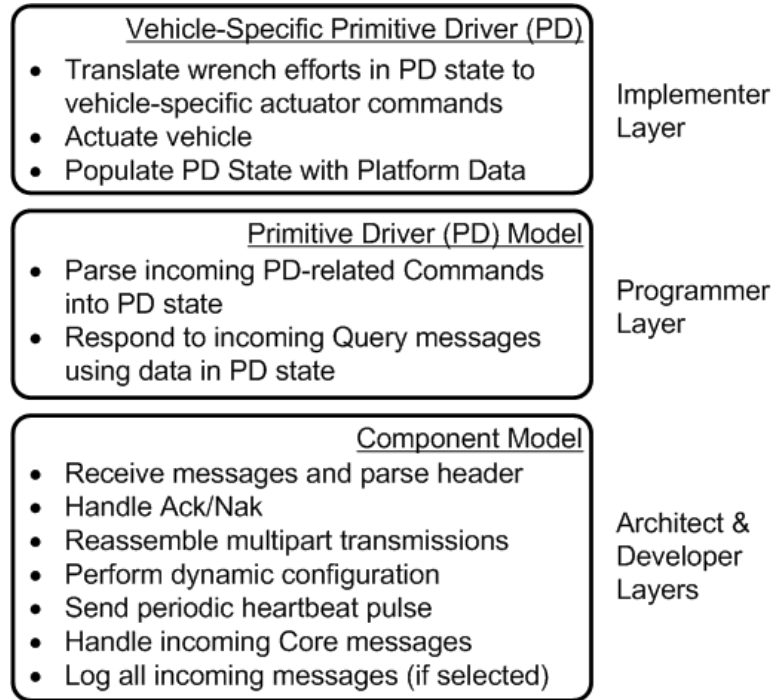


Figure 2.3: Component Model Layered Structure

1. Component: contains the main status indicator for the component...
this is where component-specific indicators would go
2. Settings: here, the user sets the component ID and component name...
also has settings for message logging and display
3. Events: shows the active Events and Service Connections
4. Messages: real-time message display, console.out display

A common significant design problem with many LabVIEW front panels is a Stop button that is difficult to find; often, stopping a VI requires scrolling to find the Stop button. The Component Model solves this problem by splitting the panel into two panes. A sticky pane at the window's lower edge contains the Stop and Reset buttons for the component, as well as a string indicator for status, and a progress bar. When the window is resized, the Stop and Reset buttons remain visible.

Block Diagram Structure

The block diagram of the component model contains two loops and only five subVIs, as indicated in Fig 2.5:

1. Initialize: start up JAUS, connect to Node Manager
2. Multithreads: several asynchronous, parallel threads for input message parsing, ack/nak and multipart message handling, message display, event watcher for the run-time menu, sending the periodic heartbeat, and message logging
3. Process: de-Queue incoming message, handle if core message
4. Shutdown: stop JAUS, disconnect from Node Manager
5. Handler Model: an empty message handler, to be replaced with a new or existing service module

The three wires stretching across the VI contain:

1. an incoming message (parsed header, unparsed body) to be processed by the message handlers
2. the core state (status, component ID, current controller, active events, component identification, etc)
3. a reference to the output queue and message logging queue

The rest of the diagram consists of comments and decorations serving as instructions for where to add code in designing a new component. As will be demonstrated in Section 2.4, this consists mostly of defining a service state and modifying the message handler to affect that state.

2.2.3 Routing Components

Perhaps the least-discussed, but most important parts of any JAUS implementation are the routing components, namely the Node Manager (NM) and Communicator (Cm). While the functions of the NM and Cm are

simple, their error-free operation is critical to the application of a level-III-interoperable JAUS system.

The VT Toolkit provides a well-tested Node Manager (Fig 2.6) and Communicator (Fig 2.7) supporting JAUS over UDP. An array of configuration options allows for slight modifications to enable interoperability with groups not making the assumptions involved in the JAUS ETG.

In addition, a dynamically configurable real-time message display assists with the everpresent question heard when testing interoperability: “Did you get my message?”. Both IP address and port number are shown for each message packet, which assists in debugging of networking problems. This is shown in Fig 2.7.

2.2.4 Code Modules

Three main categories of code modules are included in the Toolkit: services modules, message routines, and programmer tools.

Services Modules

A service module provides the messaging interface for a particular service, such as Primitive Driver. Consisting of a message handler and a state machine, the service module would be responsible for parsing incoming command messages into the Primitive Driver state, and responding to incoming query messages using the data in the Primitive Driver state.

Note that a service module is separate from the hardware interface, as indicated in Fig 2.3. For a Primitive Driver, the hardware interface would convert the Wrench Efforts in the Primitive Driver state to vehicle commands, and populate the Primitive Driver state with vehicle information to be used in responding to incoming queries.

While the Primitive Drivers for all JAUS vehicles at Virginia Tech use the same service module, their hardware interface is different.

Messaging Routines

These code modules perform commonly executed messaging routines, such as establishing a service connection with the Global Pose Sensor. Currently, there exist only a small set of routines, but this group will soon accomplish functions such as detecting the Global Pose Sensor via dynamic configuration messages, establishing a service connection, and monitoring the service connection for health.

Programmer Tools

This group of VIs affords the programmer extra options for publishing data, either for debugging or for user interface. For example, `jaus_console_send_message.vi` used for publishing internal errors or events to the console.out display on the Component Model front panel. Another useful VI, `jaus_speak.vi`, allows the programmer to queue up run-time-defined strings for a speech synthesis module in the Node Manager to announce.

2.2.5 Components

The following components are provided with the VT JAUS Toolkit for LabVIEW:

- Component Model (CM): discussed in Section 2.2.2
- Node Manager (NM): handles message routing, supports level-II interoperability over UDP, maintains table of subsystem configuration
- Communicator (Cm): handles message routing, supports level-I interoperability over UDP, maintains table of systems
- Subsystem Commander: instructs NM to forward external subsystem heartbeats to it, maintains table of all subsystem configurations in system
- Global Waypoint Driver with Mission Spooling support

- Primitive Driver
- Reflexive Driver
- Global Pose / Velocity State Sensor: multiple services in one component... see Section 4.6.1
- Global Pose Sensor: interfaces with NovAtel GPS
- Visual Sensor: supports Firewire, USB, and Axis Network Cameras
- Laser-Range-Finder Sensor: interfaces with Sick LMS-200 LRF
- Two Operator Control Units: discussed further in Section 3.12
- World Model Vector Knowledge Store
- Velocity Recorder: establishes service connection for Velocity State and saves data to file
- Global Pose Recorder: establishes service connection for Global Pose and saves data to file in Grand Challenge RDDF format [15]
- Text Messenger: an Instant Messenger client over JAUS, similar to AOL IM, Google Chat, IRC, ICQ, or Windows Live

2.2.6 Message Replay Utility

A message replay utility is provided for playback of the message logs. This has proven to be extremely valuable in testing and achieving interoperability with other groups. For example, in testing with Applied Research Associates at the Intelligent Vehicle Systems Symposium (see Section A.1), I used the message replay to capture user-defined messages from ARA's Operator Control Unit and then send them to ARA's vehicle, from the Virginia Tech OCU.

2.3 Toolkit Deployment

The JAUS Toolkit for LabVIEW is provided as a collection of VIs and DLLs, organized into folders, as shown in Fig 2.8. Folders labelled jaus-

something are required, containing VIs used by all components, while folders labelled *jaus_something* are optional, containing VIs required only by a specific component. VIs not contained within folders are components, such as the Component-Model or the Node-Manager.

2.3.1 Releases and Upgrades

Releases of the Toolkit contain two text files indicating the date of release. “revision-dates.txt” contains the date of the last release, and “changelog.txt” contains a list of all updates since the last release, and a list of affected VIs. For example, 2006-11-09-1630EST indicates a release on November 11th, 2006.

Incremental Upgrades

To avoid needlessly copying the 1000+ files making up the Toolkit when updating the release on a vehicle, incremental upgrades are provided. A user simply downloads the ZIP file for the upgrade from his/her current release to the latest release, and extracts it into his/her JAUS folder; only the changed files are replaced.

For example, if you have release 2006-10-31-2047EST and you wish to update to the latest release (2006-11-09-1630EST), you would apply incremental upgrade “from-2006-10-31-2047EST_to-2006-11-09-1630EST.zip” to your code.

2.4 Use of the Toolkit

This section presents a series of examples illustrating the use of the Toolkit.

2.4.1 New Primitive Driver

Suppose we wish to create a Primitive Driver for a particular vehicle. Since the Primitive Driver service is already defined, we can add its state cluster

and message handler to an empty Component Model, or start with the Primitive Driver Model.

As shown in Fig 2.9, one only needs to modify the upper portion of the block diagram, which interacts with the vehicle hardware based on the information in the Primitive Driver (PD) state.

2.4.2 New Component

In this example, we implement a new component, the Reflexive Driver. This component receives Wrench Efforts from a Navigation Driver, and modifies them if they are unsafe.

In developing the Reflexive Driver service module, there are four steps: (1) List messages, (2) Develop system state, (3) Build message handler, and (4) Test. Upon initialization, the Reflexive Driver will establish a Service Connection for Laser Range Finder (LRF) Data (E438h) and request control of the Primitive Driver. In normal operation it receives Wrench Efforts (0405h) and LRF Data (E438h), and produce Wrench Efforts for the Primitive Driver.

The Handler Model is used, and the resulting Reflexive Driver message handler (service) is shown in Fig 2.10. While this section of code provides the JAUS interface, the algorithm outside the handler on the Component Model diagram calculates the Outgoing Wrench Efforts.

2.4.3 Experimental Message

Implementing a new JAUS message with the VT JAUS Toolkit is simple using the standard packing VIs (Fig 2.2) and the Message Model. Fig 2.11 and fig 2.12 depict the build and parse subVIs for an early version of E438h Report LRF Scan.

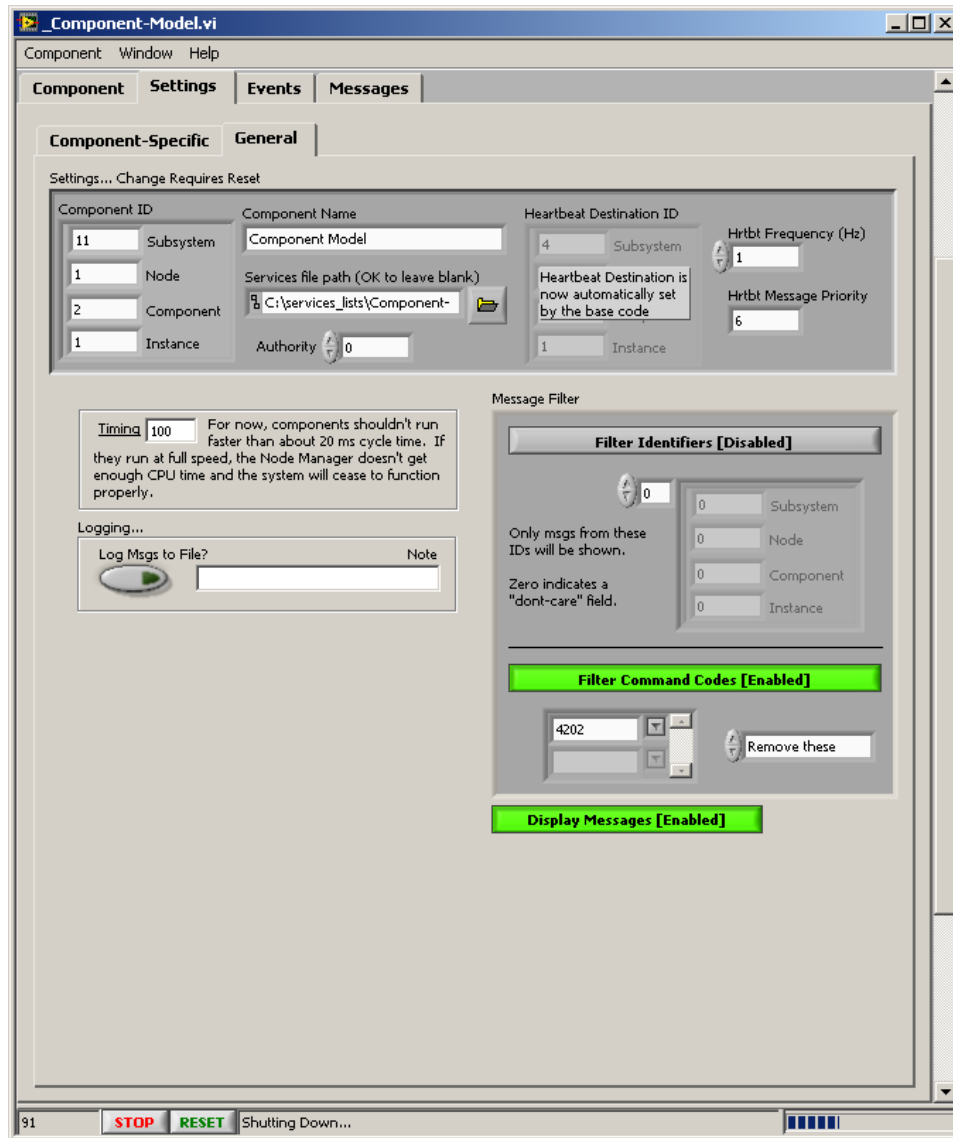


Figure 2.4: Component Model Front Panel, Settings Tab

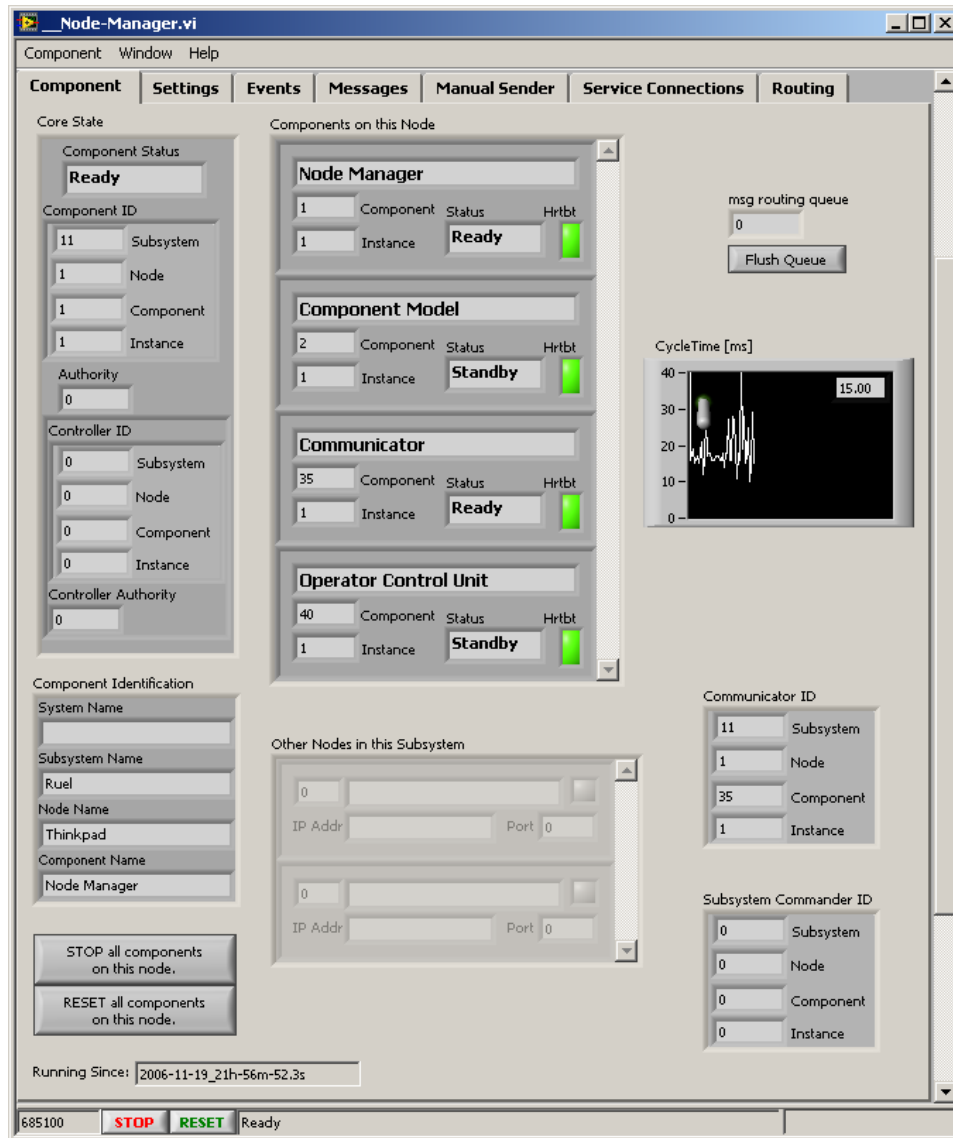


Figure 2.6: Front panel of the Node Manager

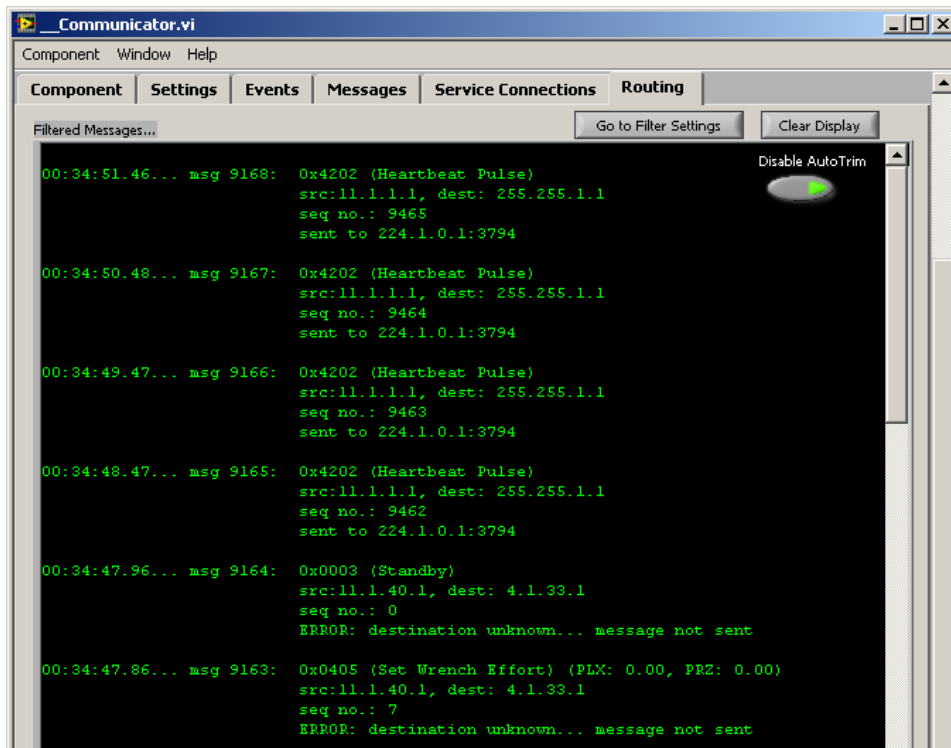


Figure 2.7: Real-time message display on the Communicator

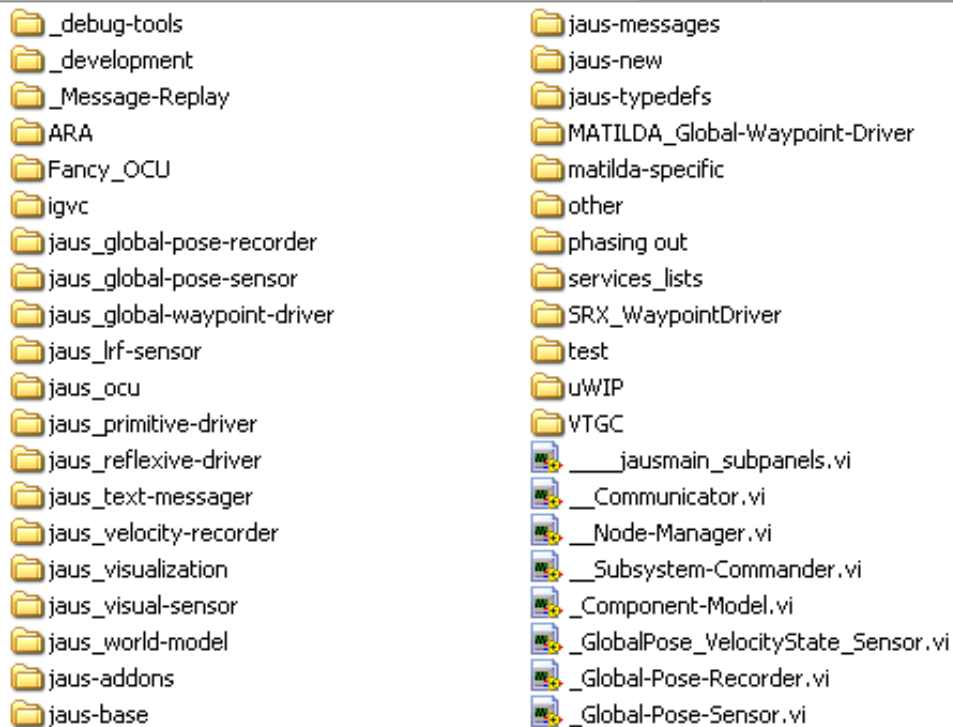


Figure 2.8: Folder organization of the LabVIEW Toolkit

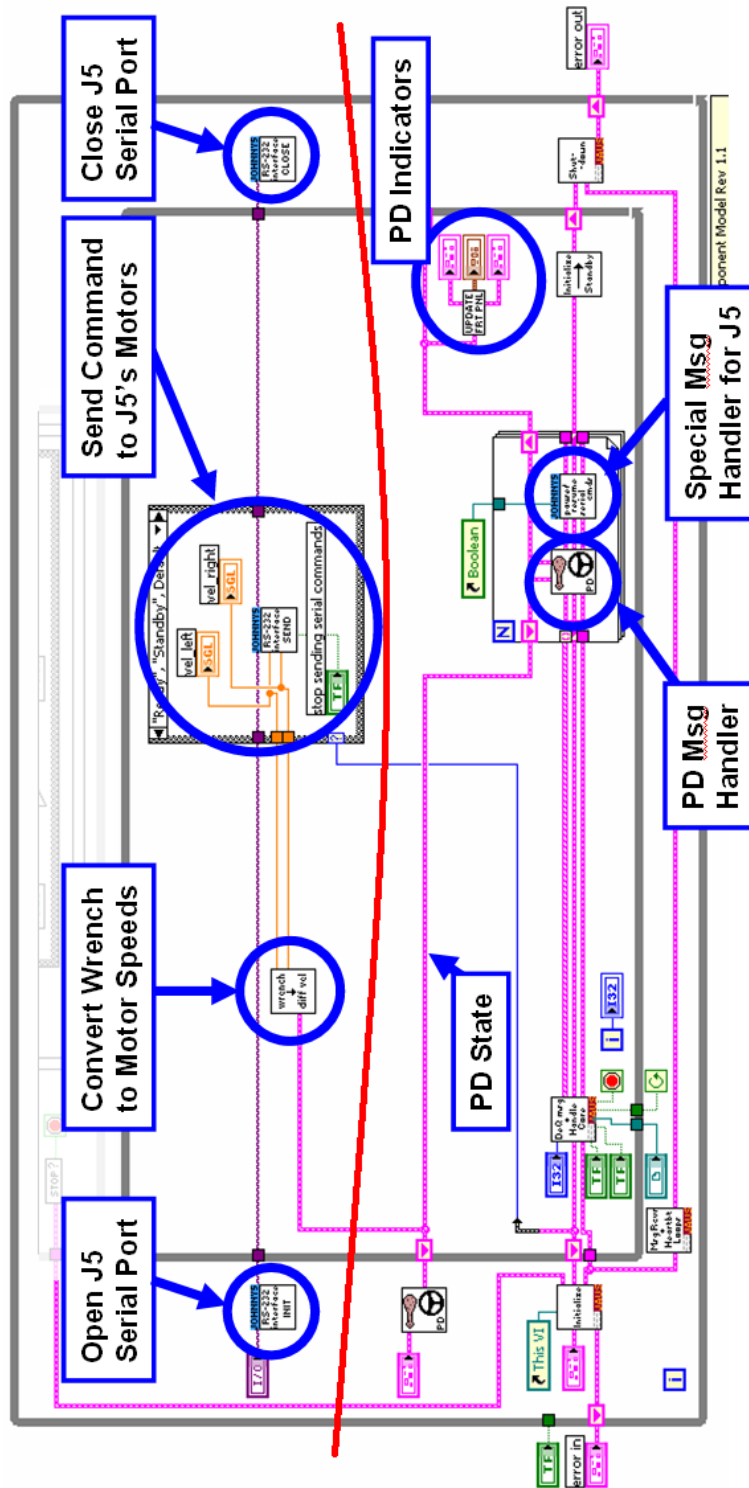


Figure 2.9: Primitive Driver for Johnny-5

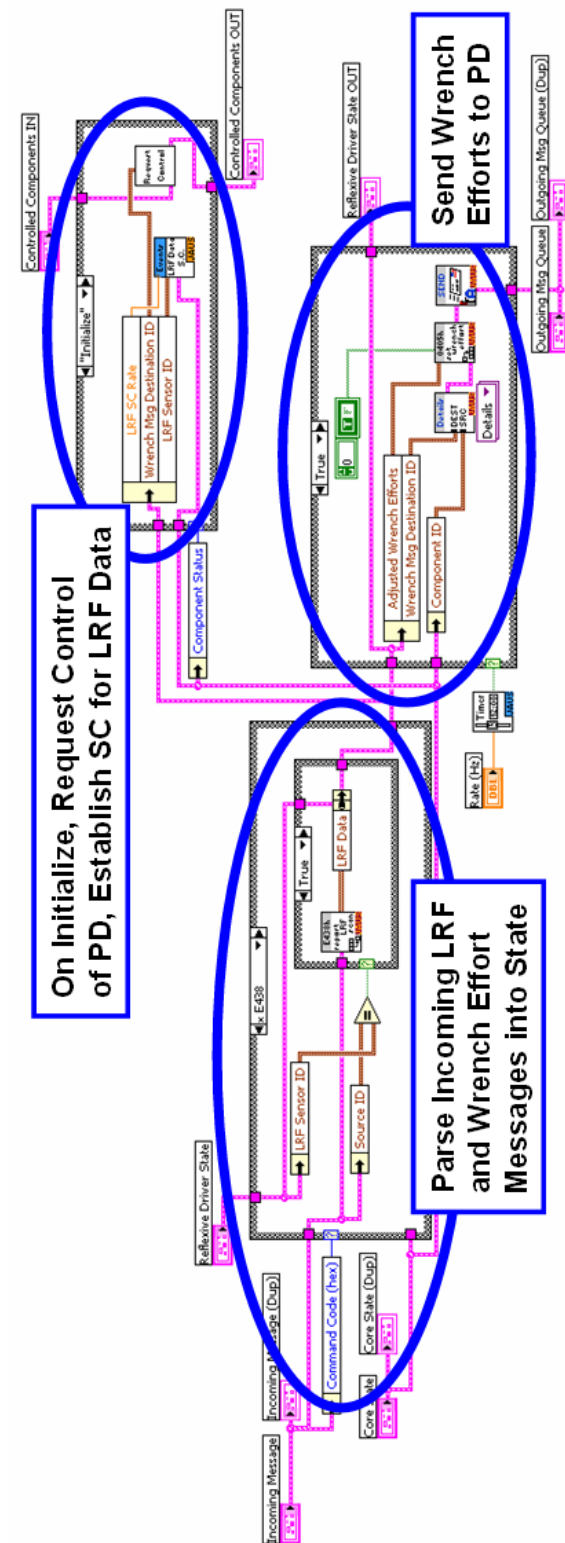


Figure 2.10: Inside the Message Handler for a Reflexive Driver

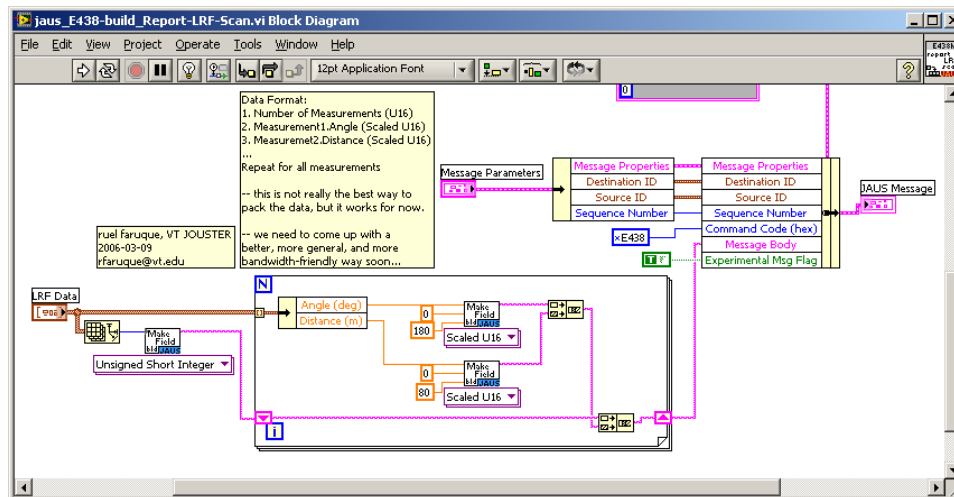


Figure 2.11: Message Build for old version of E438h Report LRF Scan

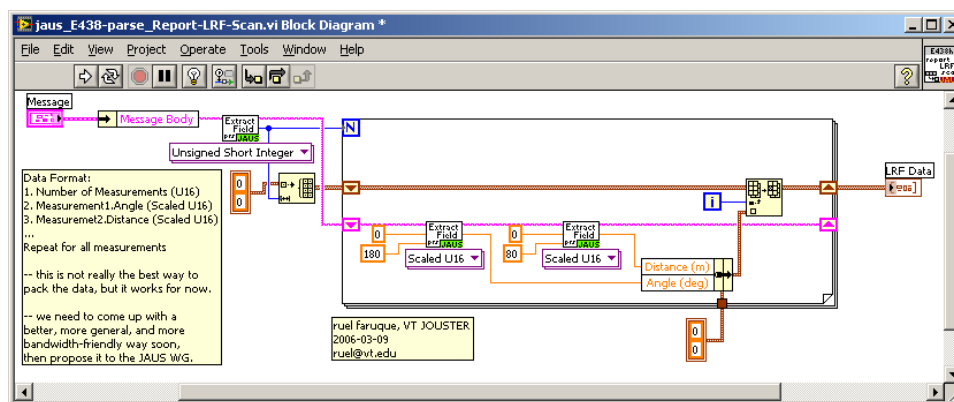


Figure 2.12: Message Parse for old version of E438h Report LRF Scan

Chapter 3

Implementation Case Studies

In this chapter, we present an overview of the JAUS implementations at Virginia Tech. Through use of the Toolkit, and by following a standard design process for implementation, JAUS has been rapidly applied to a wide array of vehicles, payloads, instrumentation, and software.

3.1 At a Glance

First, we present the design process used in implementing JAUS, followed by the collection of implementation case studies. For each particular case study, an introduction to the hardware is presented, along with a discussion of the JAUS implementation, and the key issues related to that implementation. General implementation issues arising from multiple case studies are discussed in Chapter 4.

3.1.1 Vehicles

To date, JAUS has been successfully implemented on seven unmanned vehicles at Virginia Tech. Ranging from 12 lbs to 3000 lbs, 2mph to 30mph, and including Ackerman-steer, differential-drive, wheeled, tracked, and even articulated vehicles, this diverse group helps to validate the versatility of the Toolkit, and JAUS in general.

3.1.2 Payloads

Two distinct types of payloads are presented here. The Laser Range Finder (LRF) Payload (Section 3.10) is a true JAUS payload that adds functionality to an unmanned vehicle, while the μ WIP (Section 3.11) is an instrumentation device used in the testing of unmanned vehicles.

3.1.3 Other Software

Finally, a purely software implementation is presented: an Operator Control Unit.

3.2 Design Process

At Virginia Tech, a strict design process is followed for implementing JAUS on an unmanned system, consisting of the following phases:

1. Determine Interoperability Goals
2. Examine Existing Architecture
3. Define Component Boundaries
4. Construct Components
5. Define Node Boundaries
6. Integrate Components and Verify Performance

The process is presented here in the context of converting an existing vehicle to JAUS-interoperability. The process differs only slightly for a new vehicle incorporating JAUS from the beginning, since the application of JAUS usually occurs after the software architecture has been determined.

3.2.1 Phase 1: Determine Interoperability Goals

First, the design objectives are established, with the primary goal of deciding whether to implement level-I, level-II, or level-III JAUS-interoperability. During this phase, the following questions are addressed:

- With what other JAUS-interoperable objects should the unmanned system be compatible?
- Will developers be adding more code modules often?
- Is future code reuse a priority?
- Will the vehicle be accepting payloads?
- How much time is available for the conversion?

Several characteristics of each interoperability level, which are helpful in this decision, are given in Table 3.1.

3.2.2 Phase 2: Examine Existing Architecture

Despite the appearance of the word “architecture” in the acronym JAUS, the standard is not an architecture. As previously mentioned, JAUS should not restrict the logical interaction of the software modules, only their physical interaction. However, the JAUS conversion process will be influenced by the vehicle’s software architecture. Is it hierarchical, reactive or deliberative? How are the behaviors combined – subsumption, potential fields, or some other method? The architecture will heavily impact the next phase.

3.2.3 Phase 3: Define Component Boundaries

Here, the developer decides where to divide the vehicle software into JAUS components, and which services are to be contained within each component. For a level-II interoperable system, this step is unnecessary however, as everything is wrapped in a single monolithic component. Two primary strategies exist for establishing the components in level-II or level-III system: based on functional divisions or based on available messages.

Functional Decomposition

Nearly all autonomous vehicle software contains three distinct processes: (1) Sensing, (2) Interpreting the sensed data and making decisions, and (3)

Table 3.1: Features of each level of JAUS interoperability

Level I Interoperability

- Interoperability only with other complete JAUS subsystems
- Very little future modification possible
- Highly efficient operation
- No internal messaging overhead
- Very short conversion time
- Limited code reusability

Level II Interoperability

- Interoperability with other complete JAUS subsystems (other vehicles or an OCU)
- Interoperability with other JAUS nodes (distinct computing elements which can be added to the unmanned system, typically payloads)
- Most future modification is accomplished by adding JAUS nodes.
- Secure. The existing computing resources are dedicated to the base capabilities.
- No internal messaging overhead in node.
- Short conversion time
- Limited code reusability

Level III Interoperability

- Full interoperability with other JAUS subsystems, nodes, and components.
- Allows JAUS components to be added to the base software, facilitating future work.
- Encourages modularity
- Introduces messaging overhead
- Short to medium conversion time, depending on level of component granularity
- Highest level of code reusability

Acting upon the decisions [5]. With these Sense, Plan, and Act primitives in mind, I propose two further levels of Level-III granularity:

Level-III-A: Define a component for each sensor interface, a component to accomplish motion control, and one component to handle all decision-making abilities. Such a division effectively decouples the vehicle-specific software (sensors and motors) from the intelligence software, allowing intelligence software to be easily ported from vehicle to vehicle. This approach is best suited for small projects where the core decision-making software is written by a small group of developers who interact often.

Level-III-B: Define the components just as in Level-III-A, only decompose the decision-making module into smaller, more distinct components, such as path planning and obstacle avoidance. A large development team would benefit more from such an approach. New unmanned systems are better suited to this higher level of granularity, since applying Level-III-B to an existing autonomous vehicle requires that the software already be modular before attempting a JAUS conversion.

Messaging Decomposition

The second method of decomposition relates to the available messages in the standard. In following this approach, the developer examines the current message set and defines components based on the available JAUS messages. At this early stage in the development of the JAUS standard, the messaging decomposition method is highly restrictive, as the standard does not yet provide messages for all applications of unmanned vehicles.

This approach would be best suited for a developer who seeks only to implement the existing JAUS messages, and not to help extend the standard by defining experimental messages and proposing them to the JAUS Working Group.

3.2.4 Phase 4: Construct Components

In this phase, the components are actually developed. Virginia Tech's JAUS Toolkit provides two main avenues. First, one can copy the vehicle code into the Component Model framework; this provides the user with a nice graphical user interface for the new JAUS component, with user controls and indicators already placed on the LabVIEW front panel. Alternatively, the developer can place the same JAUS subroutines onto his/her existing program's block diagram, without copying his/her existing code.

In both approaches, we typically construct the components via a four-step process, outlined below. Note that steps 2 and 3 are often simplified by the presence of services modules contained in the JAUS Toolkit.

1. List the JAUS messages coming in and out of the component, and their associated data.
2. Based on the data associated with the messages, establish a data state cluster for each service in the component.
3. Assemble the message builders/parsers into message handlers to interact with the service state, responding to incoming messages.
4. Test each service independently, validating the component.

3.2.5 Phase 5: Define Node Boundaries

Once the components have been established, the node boundaries are defined. Just as in any non-JAUS-interoperable system, this decision process involves balancing the considerations of computing power and complexity.

Due to the routing inherent in JAUS, the layout can be easily changed in the future simply by moving a component to another node and changing minimal settings.

3.2.6 Phase 6: Integrate Components and Verify Performance

All messaging between components is tested and verified, moving upwards from the lowest level. Once this is complete, the JAUS-enabled unmanned system should be tested against its pre-JAUS configuration. Most often, a system lag will have been introduced by the messaging overhead (Section 5.3.2). As discussed in Section 4.5.2, this delay can usually be reduced through the use of latest-data connections.

3.3 MATILDA

As one of its contract deliverables, researchers in JOUSTER outfitted a stock Mesa Robotics Block II MATILDA with a JAUS-interoperable autonomous control package [25]. The first JAUS-interoperable vehicle at Virginia Tech, the modified MATILDA is shown Fig 3.1.



Figure 3.1: Virginia Tech’s JAUS-interoperable MATILDA

Primarily, the MATILDA is used as a platform for evaluation, development, and demonstration of the JAUS standard at JAUS ETG Experiments. In addition, the vehicle is available as a research platform for customers of

the JOUSTER site. In summary, typical uses include:

- carrying prototype payloads under test
- participation in vehicle teaming exercises
- serving as a test vehicle for new navigation algorithms
- participation in educational and promotional demos

3.3.1 Functional Requirements

With the typical use pattern in mind, the following interoperability requirements were established for the JAUS-enabled MATILDA:

1. provide teleoperation via an OCU
2. autonomously drive GPS waypoints commanded via an OCU
3. interact with other JAUS subsystems (vehicles and OCUs)
4. accept a third-party JAUS payload
5. allow easy addition to and modification of intelligence software
6. facilitate demonstration, evaluation, and testing of the JAUS standard

3.3.2 Interoperability

Given that the MATILDA is intended for the evaluation and development of JAUS, it was decided to pursue the most rigorous JAUS implementation possible. The desire to test component-to-component interactions mandated level-III interoperability, while payload expandability and interaction with other robots/OCUs required level-II and level-I interoperability, respectively.

In addition, to fully evaluate and demonstrate the standard, the highest level of component granularity was chosen, level-III-B interoperability.

3.3.3 JAUS Layout

As indicated in Fig 3.2, the MATILDA employs a Primitive Driver, Global Pose Sensor, and Global Waypoint Driver, in addition to the Node Manager

and Communicator.

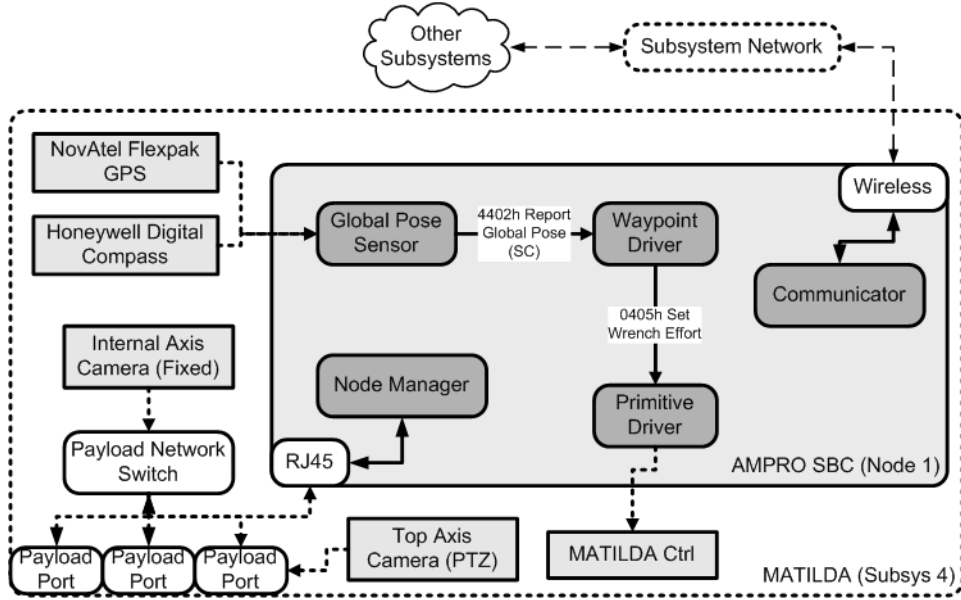


Figure 3.2: JAUS Layout for MATILDA

An internal node network isolated from the subsystem network provides three payload ports. In this way, a JAUS payload aboard the MATILDA is registered as a secondary node in the subsystem, and communicates to other subsystems via the MATILDA’s Communicator component.

3.3.4 Operation

Upon startup, the MATILDA’s Waypoint Driver requests control of the Primitive Driver, establishes a service connection to the Global Pose Sensor, and begins commanding Wrench Efforts to the Primitive Driver.

To command the vehicle waypoints, an OCU must send, to the Waypoint Driver, the “Request Control”, “Set Global Waypoint”, “Set Travel Speed”, and “Resume” messages in any order. The Waypoint Driver also supports missions sent via the Mission Spooling messaging set.

To teleoperate MATILDA, a higher-authority OCU can simply request control of the Primitive Driver, causing the Primitive Driver to reject control by the Waypoint Driver. To resume Waypoint Driving then requires sending

a “Reset” message to the Waypoint Driver.

3.4 Simulator

Virginia Tech’s recent successes in the AUVSI Intelligent Ground Vehicle Competition [16] and the DARPA Grand Challenge [15] are largely due to the use of a custom-developed software simulator. Developed by Virginia Tech graduate student Andrew Bacha, the simulator interfaces with the prototype software by accepting motor commands, calculating dynamic response and motion, and providing simulated sensor data from a virtual, idealized world, depicted in Fig 3.3.

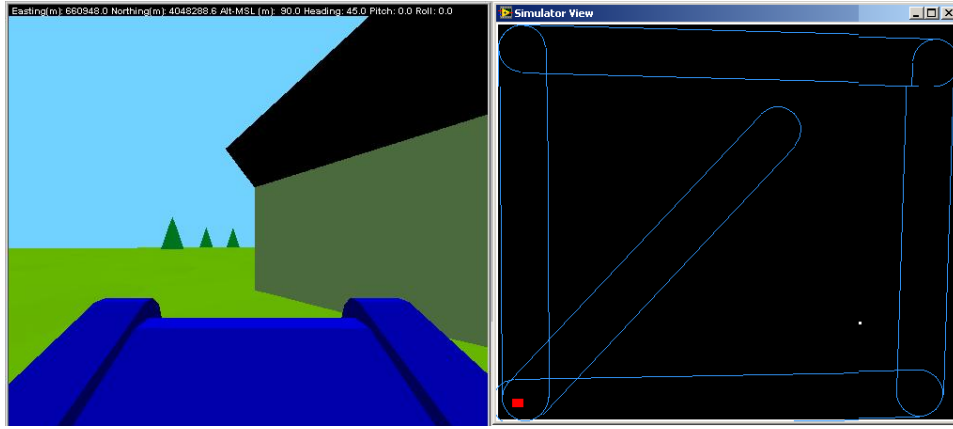


Figure 3.3: In-vehicle Visualization and 2D View for Simulator

The simulator is ideal for interoperability testing, where the performance of the robot is not under test, only its messaging interface. When testing interoperability at the ETG experiments, Virginia Tech used the simulator extensively, and code was only run on the vehicle as a final validation, after the JAUS interface had been proven with the Simulator.

3.4.1 Interoperability

Since the simulator is used for nearly all JAUS testing Virginia Tech does with other groups in the JAUS Working Group, the goals for the JAUS-

enabled simulator were very similar to those of the MATILDA (Section 3.3.2). The most rigorous JAUS implementation possible was desired: level-I, level-II, and level-III-B.

3.4.2 JAUS Layout

Fig 3.4 presents the typical JAUS layout of the simulator. Since the system is configured especially for the interoperability test at hand, it is very flexible.

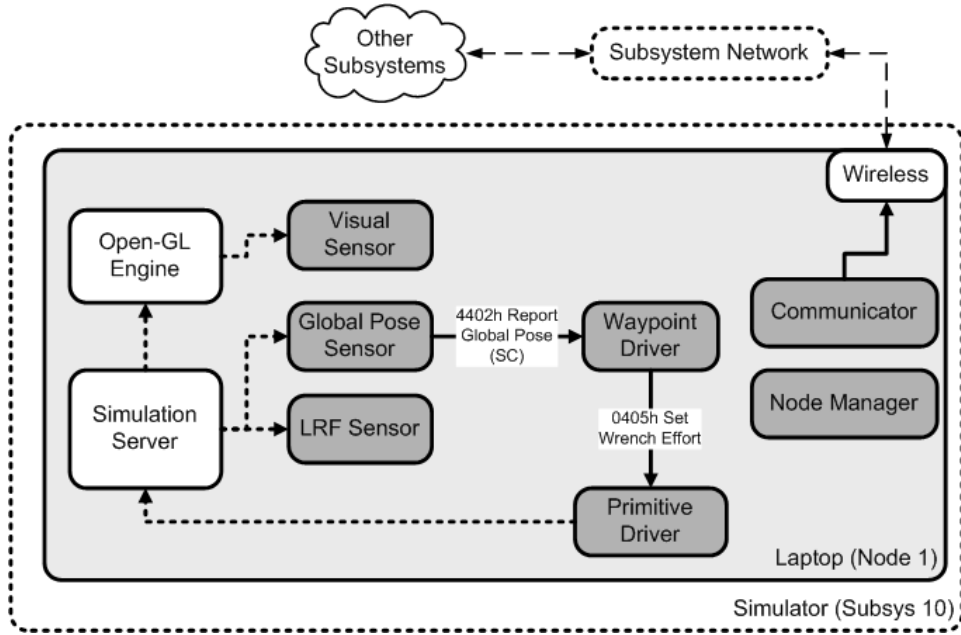


Figure 3.4: JAUS Layout for Vehicle Simulator

A virtual Visual Sensor was also integrated into the simulator (with the aid of Michael Fleming and Chris Terwelp), to provide OpenGL simulated video feedback (Fig 3.3), for teleoperation. However, the JAUS interface was never successfully tested outside Virginia Tech and still needs verification.

3.5 Johnny-5

Johnny-5 (pictured in Fig 3.5) was developed for the 2004 AUVSI Intelligent Ground Vehicle Competition, as a capstone senior design project [17]. Due to the vehicle's robust mechanical and electrical foundation, Johnny-5 has

served as a research testbed for several master's theses [19], [20], [21]. In addition, Johnny-5 has returned to the IGVC each year, and continues to perform well [16].



Figure 3.5: Johnny-5 [17]

When the Virginia Tech Unmanned Systems Lab suffered a fire in November 2005, the MATILDA slated for participation in ETG Experiment 2.75 was badly damaged. With less than two weeks to the experiment, we chose Johnny-5 as a replacement vehicle for participation in the experiment. Using the VT JAUS Toolkit, the conversion to JAUS-interoperability took approximately 2 hours.

Johnny-5 participated successfully in the ETG Experiment, with GPS waypoint navigation capability. The following spring, Johnny-5's original

IGVC software was converted to full JAUS-interoperability for entry in the 2006 IGVC [17]. Both configurations are presented here.

3.5.1 Interoperability

Given that Johnny-5 was readied to substitute for the damaged MATILDA at ETG Experiment 2.75, level-I and level-III interoperability were implemented. For the IGVC, level-III interoperability was maintained, and changes were only to the JAUS layout.

3.5.2 JAUS Layouts

Fig 3.6 shows the JAUS layout for Johnny-5 used at ETG Experiment 2.75. It is nearly identical to that of the MATILDA, and operates in the same manner.

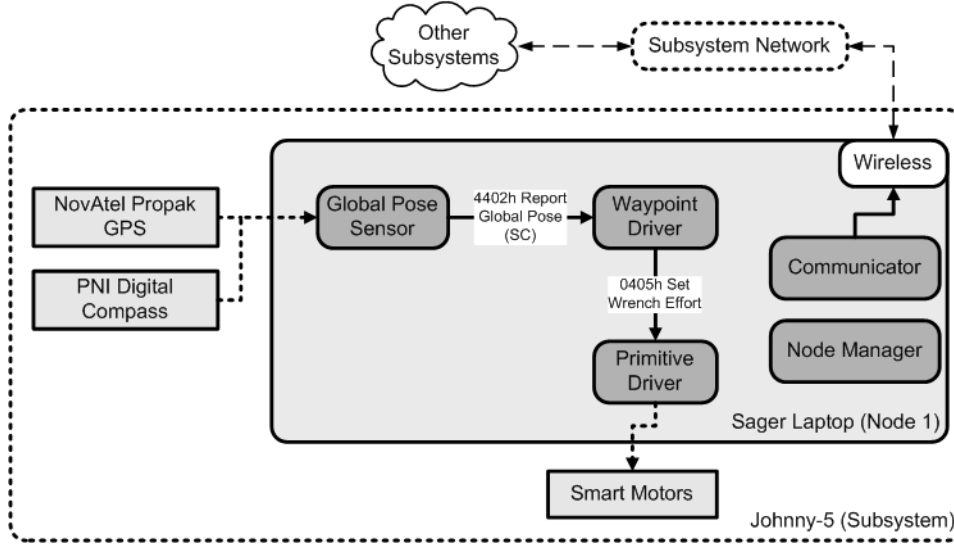


Figure 3.6: JAUS Layout for Johnny-5, at ETG Experiment 2.75

For the IGVC, we replaced the Waypoint Driver with a Navigation Driver to handle lane maintenance and obstacle avoidance, shown in Fig 3.7. This Navigation Driver acquired obstacle data through a service connection to the LRF Sensor component and produced a set of Wrench Efforts for the Primitive Driver component.

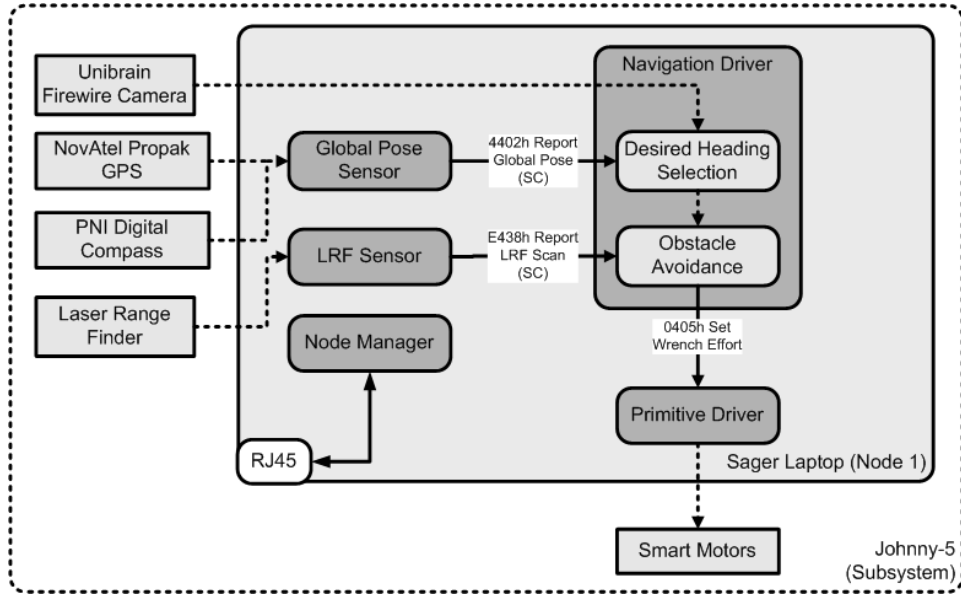


Figure 3.7: JAUS Layout for Johnny-5, at IGVC

3.5.3 Issues Encountered

The following issues were encountered in implementing JAUS on Johnny-5 for the Autonomous Challenge of the IGVC.

Camera Interface

JAUS messages were used for all vehicle-specific interfaces to the navigation code, except for the camera interface. The decision was made to retain the image acquisition process inside the Navigation Driver to avoid the overhead incurred by packing, packetizing, sending, reassembling, and unpacking the JPEG image messages. As further discussed in Section 5.3.2, this “overhead” is more a cause of the implementation than the standard.

The associated delay would have been especially detrimental in the IGVC Autonomous Challenge, where the higher-speed driving necessary to remain competitive demands lower cycle times.

Wheel Speeds vs Wrench Efforts

Pre-JAUS, the Autonomous Challenge code produced a set of two wheel speeds, which were then commanded directly to the vehicle. When the Primitive Driver was introduced, it became necessary to convert these wheel speeds to Wrench Efforts.

Since Wrench Efforts are dimensionless and there is no specified way to interpret them, interoperability is lost when the Navigation Driver does not know how the Primitive Driver will unpack the Wrench Efforts. To resolve this dilemma, simple pre-JAUS and post-JAUS converters were employed, but such a practice requires the same conversion is used on any robot on which the Navigation Driver runs.

3.6 Cadillac SRX

The Virginia Tech Cadillac SRX (shown in Fig 3.8) is the first JAUS-interoperable luxury passenger vehicle in the world ¹. The project began as an entry in the Blind Driver Challenge, sponsored by the National Federation for the Blind, with the goal of developing a car that a blind person could drive [26].

In the Blind Driver Challenge, the focus is less on demonstrating full autonomy and more on developing a car that safely allows a blind person to remain in control [27]. Two distinct problems arise: (1) conveying road information to the driver and (2) integrating variable levels of autonomy, where the car can perform difficult tasks such as parallel parking or override commands when the blind person makes a mistake.

¹This assertion is made after consulting with prominent members of the JAUS Working Group



Figure 3.8: Virginia Tech’s JAUS-interoperable Cadillac SRX

3.6.1 Why JAUS?

Modularity was the primary reason for implementing JAUS on the SRX. With the potential for human-interface projects that would extend into the multi-year Blind Driver Challenge, it was crucial that a documented standard for commanding the vehicle’s drive-by-wire software was implemented.

In addition, I saw the opportunity to validate JAUS on a full-size, high-speed autonomous vehicle in urban settings. Previously, JAUS had been primarily deployed on slower-moving vehicles intended for unstructured environments.

Finally, the SRX implements a Potential Fields method for navigation, a strategy that has often been relegated to experiments in simulation. Two of the project members, Jesse Hurdus and Grant Gothing, sought to demonstrate an implementation of potential fields on a full-size, JAUS-interoperable vehicle [8].

3.6.2 Interoperability

To enable the highest level of modular design, level-III interoperability was chosen. In addition, level-II interoperability would allow component software to run on multiple laptops during testing. Level-I interoperability was not a concern, since the car would not be under control by an OCU or Mission Spooler.

3.6.3 Software Approach

As previously mentioned, the SRX implements a Potential Fields method for navigation. In this paradigm, an array of behaviors produce vectors, which are then intelligently combined (usually via a weighted summing) to produce a single vector that the vehicle drives.

This strategy is implemented in JAUS on the SRX via a series of Driver components, each producing a 0408h Set Local Vector message. Each driver sends its Vector output to the Vector Summer, which produces a single vector for the Vector Driver. As shown in Fig 3.9, the Vector Summer service and the Vector Driver service are encapsulated in a single component, the Vector Navigator.

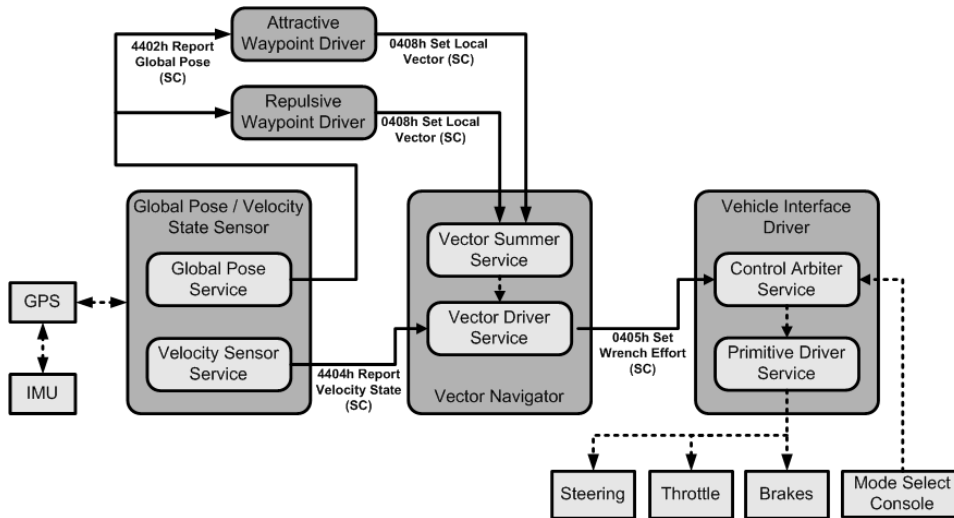


Figure 3.9: Message Flow for Cadillac SRX

In this particular example, an Attractive Waypoint Driver guides the vehicle toward waypoints, while a Repulsive Waypoint Driver repels the vehicle from an obstacle expressed as a GPS point. In the future, other Drivers could be added, such as a Lane Maintenance Driver interfacing with a camera or an Obstacle Avoidance Driver interfaced with a laser range finder [8].

3.6.4 JAUS Layout

Fig 3.10 shows the JAUS layout for the Cadillac SRX. To enable easy testing, a multi-node setup is typically employed, where the each navigation driver component runs on the laptop of the programmer who developed it.

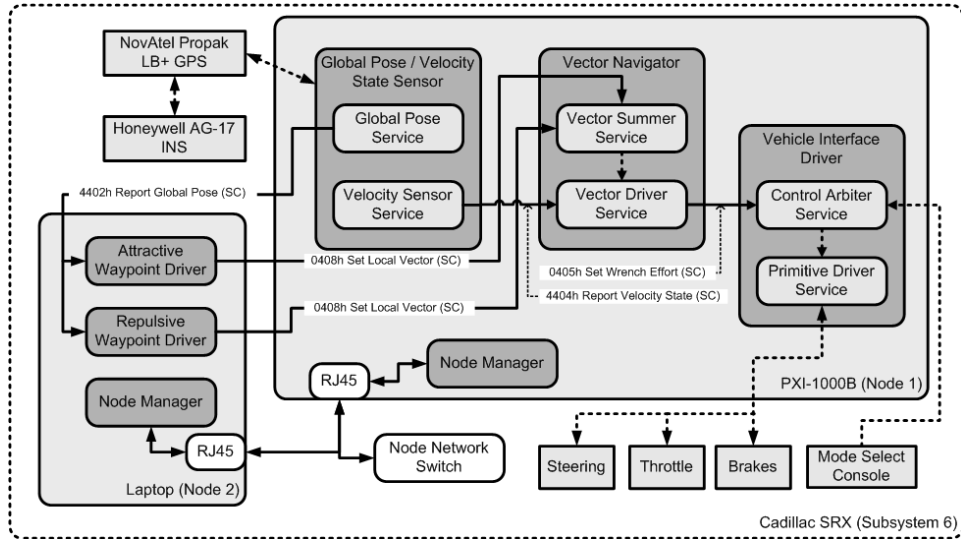


Figure 3.10: JAUS Layout for Cadillac SRX

3.6.5 Issues Encountered

Several key implementation issues were encountered in the SRX JAUS implementation, the most notable of which are presented below.

Deprecation of 0408h Set Local Vector

In version 3.2 of the JAUS Reference Architecture, the message 0408h Set Local Vector is set to deprecate at the next revision. As demonstrated in the SRX, this message is required for implementation of a Potential Fields navigation strategy.

Surely, one could use the Set Global Vector message, but this requires each Driver to have knowledge of the vehicle's Yaw angle. While this seems at first like a reasonable requirement, it creates unnecessary message traffic, it introduces susceptibility to noise in the yaw measurement, and not all vehicles have Yaw sensors.

Vector Summing

The nature of Potential Fields requires the Vector Summer service in the Vector Navigator to receive command messages from multiple components. Currently, the exclusive control messages in the JAUS specification only allow one controller per component.

To work around this issue, Gothing and Hurdus implemented a special protocol for checking in Drivers, and disabled the exclusive control filter in the Set Local Vector message handler for the Vector Summer [8].

Command Service Connections

Service connections were implemented for all periodic command messages. The performance increase was drastically significant in the case of the SRX, and is discussed in Section 4.5.2.

3.7 Gemini

Gemini is a four-wheeled, articulated robot developed alongside Johnny-5 for the 2004 AUVSI Intelligent Ground Vehicle Competition [18]. It has

returned to competition annually since 2004, and has also consistently performed well, most recently receiving the Grand Award and the JAUS Award at the 2006 IGVC [16]. Pictured in Fig 3.11, Gemini's key features are its refined design, long (5-hour) battery life, innovative mobility, and JAUS-enabled payload expandability.



Figure 3.11: Preparing Gemini for an IGVC Run

3.7.1 Interoperability

JAUS was implemented on Gemini primarily for the JAUS Challenge and the Design Competition of the IGVC. To minimize the performance impact to Gemini's winning software, Gemini's implementation was Level-II. Level-I interoperability was also required for the IGVC JAUS Challenge, where the command messages are sent from an external subsystem [28].

3.7.2 JAUS Implementation

The JAUS layout for Gemini is shown in Fig 3.12. All software functions are contained within a single Navigation Driver, which acquires camera and

laser range finder data, then actuates the motors.

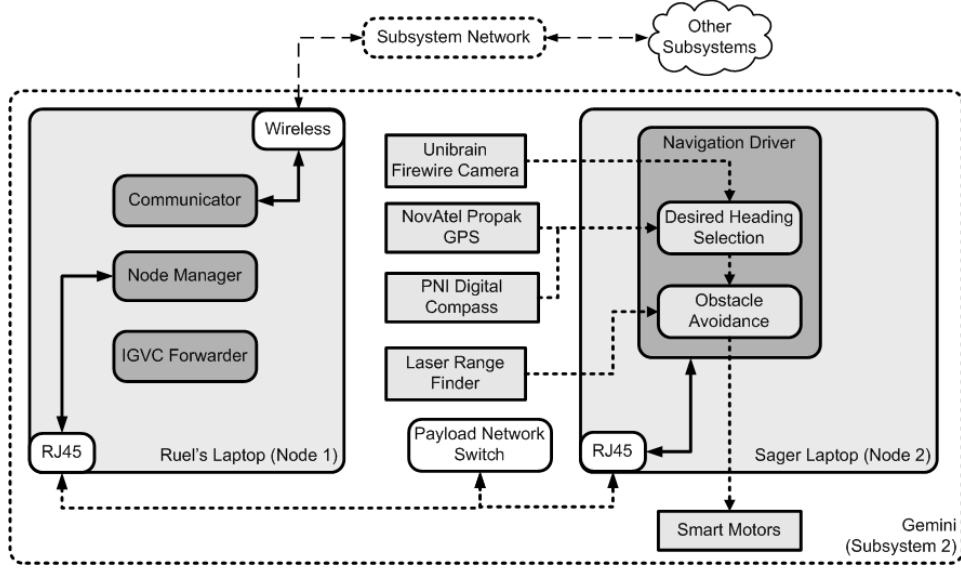


Figure 3.12: JAUS Layout for Gemini at the IGVC

Gemini only responds to the messages specified for the JAUS challenge: 0004h Resume, 0003h Standby, and 0406h Set Discrete Devices [28].

3.7.3 At Competition

Since Gemini's computer (a Sager laptop) had no wireless card, and the VT team did not have an external card at IGVC, level-II interoperability was leveraged to enable Gemini to receive JAUS messages for the IGVC JAUS Challenge.

Fig 3.12 depicts the final layout, in which a separate laptop with two network cards (NICs) ran a component that mimicked Gemini's Navigation Driver. Messages from the IGVC OCU were received through the Communicator, and re-sent by the IGVC Forwarder to the Navigation Driver, where they were executed.

3.7.4 Issue Encountered

One might ask, looking at Fig 3.12, why is the Forwarder component required, and why was Gemini’s computer reconfigured for a JAUS address of Node 2 for the JAUS Challenge?

There are two reasons. First, the IGVC JAUS Challenge specified that all OCU messages would be sent to JAUS address Subsys.1.33.1 [28]. Second, due to configuration constraints implied by the new port-forwarding specification, the Communicator component must be on the primary node, Node 1 [3]. The VT Communicator contains an error-check, where it will not run on a secondary node; hence the extra laptop needed node address 1.

3.8 Helium Red and the RMAX

In a typical UAV-UGV collaboration, problems can arise from the speed and range differential between the vehicles. Suppose a UAV detects a suspicious site on the ground, and a UGV is dispatched to investigate; if the UAV had flown far from the base, it could take hours for the UGV to reach the site. [23]

HeLiUm RED (HElicopter LIfted UnManned Reconnaissance and Exploration Drone) redefines the traditional idea of UAV-UGV collaboration. This small unmanned vehicle is light enough to be carried by Virginia Tech’s Yamaha RMAX UAV, which has a much longer operating time and range than the typical UGV. [23]

With Helium Red, it is no longer necessary for the UAV to wait hours for a diagnosis on the suspicious location in the previous example. Instead, the UAV can simply deploy and retrieve Helium Red directly at the site, obtaining information more rapidly. [23]

3.8.1 Vehicle Overview

To enable the largest variety of missions, Helium Red is outfitted with an array of sonar sensors, a digital compass, a GPS receiver, and a camera. Helium Red has two modes of operation: (1) GPS waypoint navigation with obstacle avoidance, and (2) teleoperation.

The entirely custom-designed robot is shown in Fig 3.13. Helium Red weighs about 12 lbs and is about 18 inches long. All vehicle software is implemented in LabVIEW Real-Time, on a PC-104 computer running the Pharlap OS.

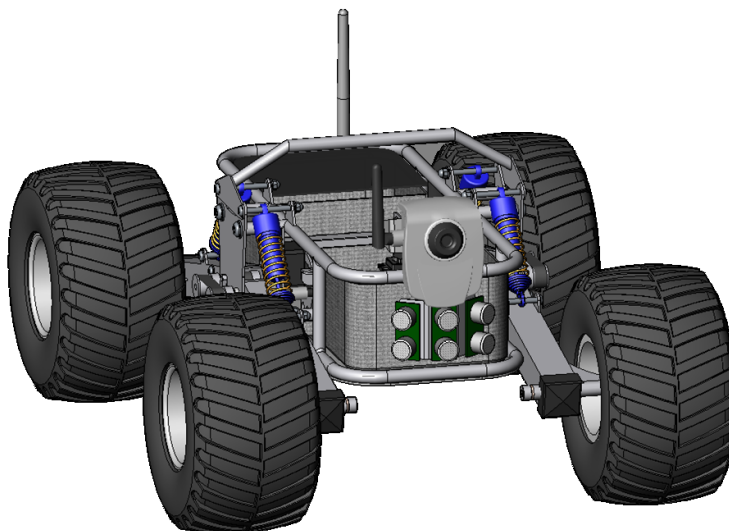


Figure 3.13: CAD Model of Helium Red SUGV [23]

3.8.2 Interoperability

For Helium Red, JAUS is implemented primarily to facilitate communication to/from the vehicle. Further, to conserve computing resources on the low-power PC-104 module, and to reduce the complexity of the software, level-II or I were possible.

Since the VT Toolkit does not yet have a LabVIEW RT implementation for Level-I, only level-II, and because Helium Red operates as part of the

RMAX, developer Patrick Currier proceeded with Level-II.

3.8.3 JAUS Layout

Currently, Helium Red is configured as shown in Fig 3.14. The vehicle operates as a single node, and Currier’s laptop interacts with it over a wireless node network. This configuration is further discussed below in Section 3.8.4.

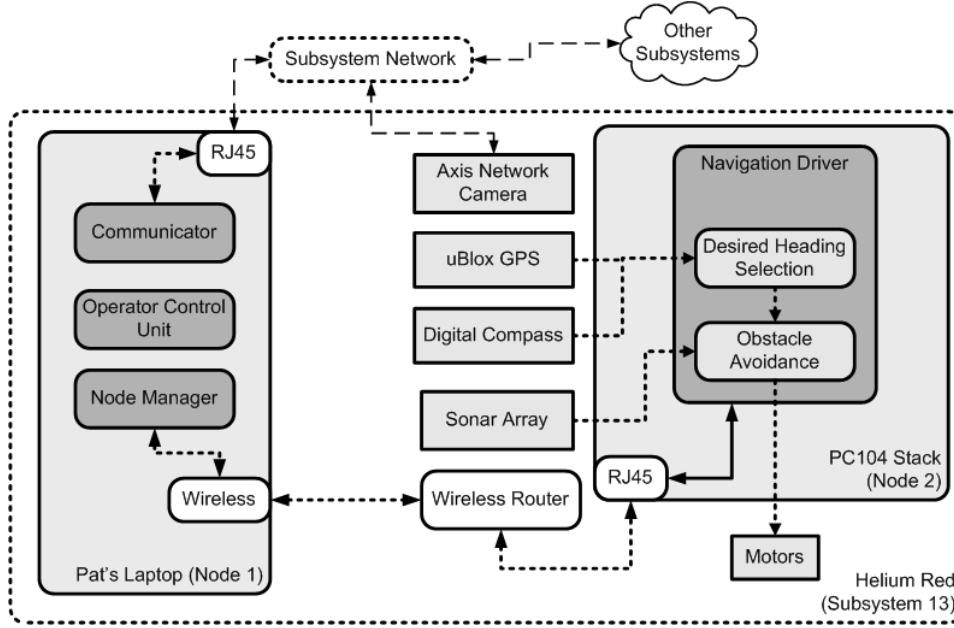


Figure 3.14: JAUS Layout for Helium Red SUGV

Similar to Gemini, Helium Red employs a single Navigation Driver performing all vehicle tasks. This single component supports teleoperation, obstacle avoidance, and waypoint driving capabilities, in addition to the routing functions. In other words, it is a single component supporting the the Primitive Driver, Reflexive Driver, Waypoint Driver, Global Pose Sensor, and Node Manager services.

3.8.4 Issues Encountered

The following issues were encountered in implementing JAUS on Helium Red.

Autonomous and Teleoperation Control in one Component

As mentioned above, Helium Red has only one component, supporting multiple services. While this helps to minimize the overhead of JAUS (see Section 5.3.2), it introduces a problem.

How does the Operator Control Unit switch between teleoperation and waypoint navigation? In a Level-III implementation, this can be accomplished by putting components in Standby via messaging, as discussed in Section 3.3.4. Unfortunately, such a method does not work for a single component such as Helium Red’s Navigation Driver.

For Helium Red, Currier implemented the following simple solution. When the component has no controller, it is in autonomous mode; if it has a controller, it is in teleoperation mode. To command Target Waypoints to Helium Red, the OCU requests control, issues the Set Waypoints messages, then releases control. This issue is discussed further in Section 4.2.2.

UAV-UGV Collaboration: Subsystem or Node?

Traditionally, and according to the JAUS Reference Architecture, vehicles are regarded in JAUS as subsystems, their computers and payloads as nodes, and their software modules as components [2]. Operator Control Units typically expect to register individual vehicles as subsystems, expecting to communicate directly to the vehicle’s JAUS address.

A curious issue arises with Helium Red. Is it *part of* the RMAX UAV aerial reconnaissance system? If so, what happens when there are multiple Helium Reds deployed by a single RMAX?

The issue of vehicles that cooperate with and are transported by others is unclear in JAUS. Should Helium Red be a node or a subsystem? Should it change dynamically? Should an OCU even have access to directly control Helium Red, or should the RMAX receive a Mission from the OCU, which it then decomposes into smaller tasks, one or more of which are delegated

to Helium Red? Or should the RMAX serve as a mesh-network proxy for Helium Red, making it visible as a subsystem to the OCU?

To date, these questions remain unanswered.

3.9 Rocky

Rocky is the second Grand Challenge vehicle from Virginia Tech, entered in the 2005 Grand Challenge, placing 9th after its generator failed [15]. Pictured in Fig 3.15, Rocky is based on a diesel-powered Ingersoll-Rand Club Car.



Figure 3.15: Grand Challenge Entry, Rocky

Rocky's JAUS implementation began after the DARPA Grand Challenge, at the announcement of the DARPA Urban Challenge in spring 2006. Though the utility cart form factor is not eligible for entry in the Urban Challenge, Virginia Tech sought to have multiple vehicle platforms available for software testing. A JAUS interface would allow for software to be easily ported from one vehicle to another.

3.9.1 JAUS Layout

With the above goal in mind, the JAUS implementation on Rocky mimics that of the Cadillac SRX, the vehicle slated for entry in the Urban Challenge. To simplify the conversion process, only the hardware-interface components were implemented. Fig 3.16 shows the JAUS layout for Rocky.

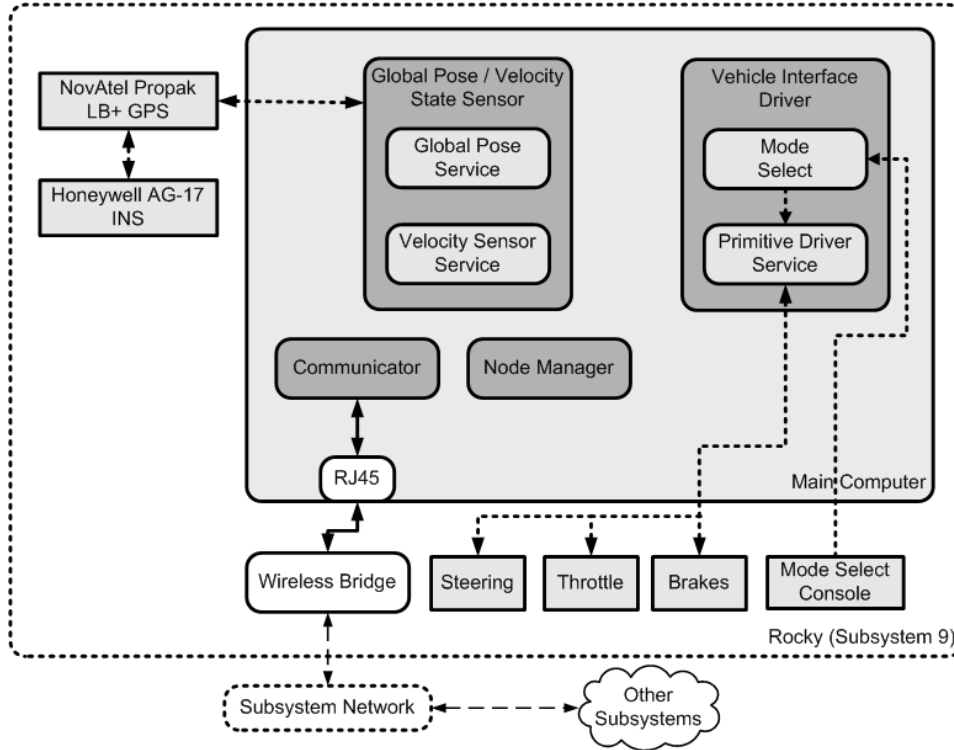


Figure 3.16: JAUS Layout for Rocky

3.9.2 Validation

Rocky's JAUS implementation was verified in two stages:

Teleoperation

First, the vehicle's Primitive Driver service was teleoperated using the Operator Control Unit component, while the Global Pose / Velocity State Sensor reported pose and velocity to the OCU via a service connection.

Directly Porting Cadillac SRX Code to Rocky

As a final validation, the Waypoint Driver and Vector Navigator components from the Cadillac SRX were ported directly to Rocky, resulting in the configuration of Fig 3.17. Without any software changes at all, Rocky successfully navigated GPS waypoints autonomously, as shown in Fig 3.18. The algorithm settings could be adjusted for better performance, but the vehicle navigated satisfactorily.

After converting Rocky's hardware-interfaces to level-III JAUS interoperability, we simply had a JAUS-interoperable remote control car. By leveraging previously developed software from a different Ackerman-steer vehicle (the Cadillac), autonomy was effectively added to Rocky in 5 minutes, without any extra integration work at all. This highly successful demonstration of code reuse should be encouraging to the developers and potential users of JAUS.

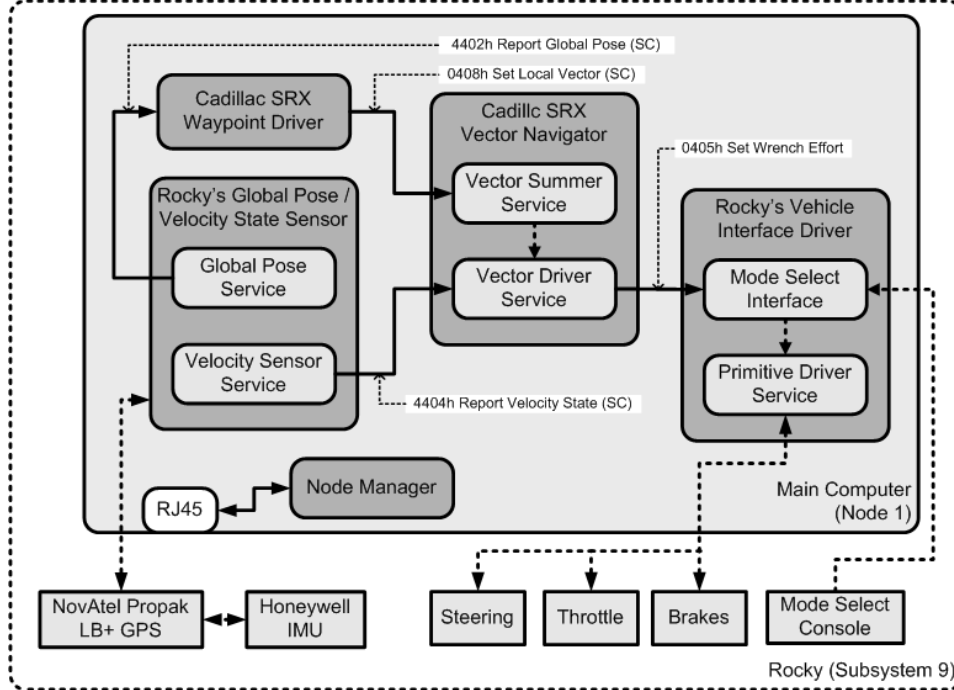


Figure 3.17: Level-III JAUS: Cadillac SRX Code on Rocky

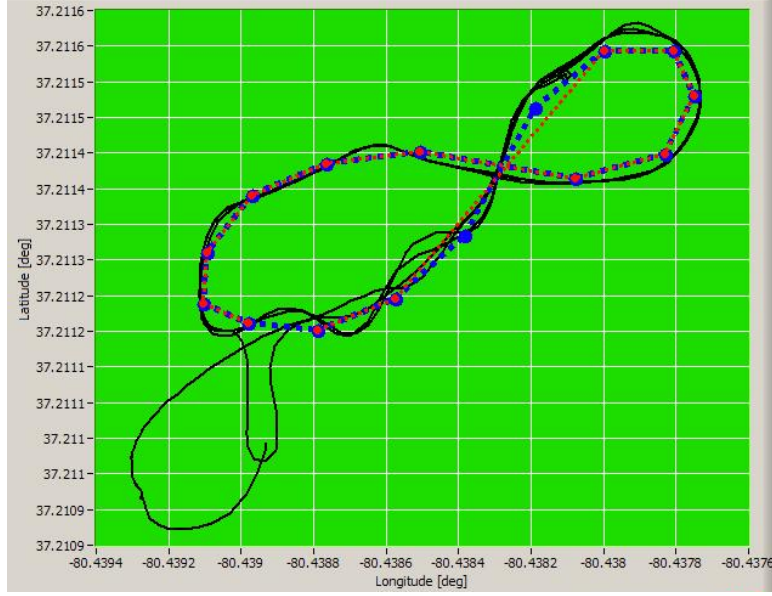


Figure 3.18: Waypoints driven by Cadillac SRX code running on Rocky

3.10 Laser Range Finder Payload

The Laser Range Finder Payload was developed for use at ETG Experiment 3.0, as an Intruder Detector Payload. Depicted in Fig 3.19, it comprises a National Instruments compactFieldPoint Real-Time controller, a Sick Laser Range Finder, and a DeWalt power tool battery.

The Intruder Detector Payload interfaces with a vehicle's node network through level-II interoperability. It continually monitors the Sick LRF scan for objects within a certain threshold distance, and once an object is detected, it calculates the position (latitude/longitude) of the object, and notifies the World Model.

3.10.1 JAUS Implementation

Intended for installation on a vehicle, the LRF Payload implements only level-II interoperability, as shown in Fig 3.20. Upon startup, the payload sends a broadcast query for services, in order to detect the subsystem's Global Pose Sensor and World Model. Once an object is detected and



Figure 3.19: Laser Range Finder Payload

localized, it is sent to the World Model via the Create Vector Knowledge Store Object message.

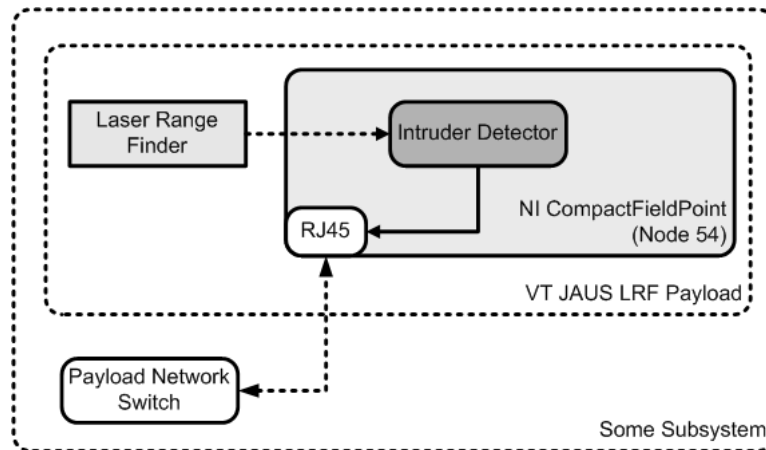


Figure 3.20: JAUS Layout for Laser Range Finder Payload

3.10.2 Obstacle Avoidance Payload

The Payload has also been configured as an Reflexive Driver, enabling the addition of obstacle avoidance to a vehicle through level-II interoperability.

In this arrangement, all Wrench Efforts (whether from an OCU, a Vector Driver, or a Waypoint Driver) are checked and modified if a collision is imminent.

3.10.3 Issues Encountered

Two issues were encountered in the LRF Payload JAUS implementation.

Experimental LRF Scan Message

While it is possible to report obstacle data using the World Model Messages, many simple low-latency reactive obstacle avoidance strategies are designed based on the raw LRF scan from the Sick LRF.

A pair of custom messages (E238h Query LRF Scan and E438h Report LRF Scan) were implemented to allow an Intruder Detector Console component to query the payload for its current scan. In addition, navigation algorithms off-board the payload can establish service connections for the LRF map data.

Configuration Messages

When the Obstacle Avoidance Payload is attached to a robot, the Navigation Driver must send Wrench Efforts to its Reflexive Driver, instead of the vehicle's Primitive Driver. With the current message set, the destination JAUS address for the Wrench Efforts must be manually set in the Navigation Driver.

Though we have not implemented this yet at Virginia Tech, a series of configuration messages, such as "Set Output JAUS Address" and "Request Control of JAUS Address", could be useful in this situation. Most likely, the Subsystem Commander would command these messages to the Navigation Driver after it detected a configuration change and determined that a Reflexive Driver had been added.

3.11 Micro Wireless Instrumented Payload

In unmanned vehicle testing, it is often helpful to track the vehicles independent of the vehicle's localization system. Researchers in the Unmanned Systems Group developed the Micro Wireless Instrumented Payload (μ WIP) to accomplish this exact purpose. In addition to position information, the μ WIP records gyroscope, magnetometer, accelerometer, and pressure sensor values. Pictured in Fig 3.21, the μ WIP is small, approximately 3"x5"x1", weighs about 0.8 lbs, and has a battery life of over 5 hours. It can stream data wirelessly to a ground station at 200Hz and record the data onboard a secure digital card for later download. [22]



Figure 3.21: Micro Wireless Instrumented Payload

In a typical application, a μ WIP is affixed to each vehicle in the test or demonstration, and the data recorded a ground station separate from the Experiment command and control. To enable non-Virginia Tech software to track the μ WIPs, a JAUS interface was implemented.

3.11.1 JAUS Implementation

Since limitations of the μ WIP's wireless transceiver allow only one open TCP connection at a time, JAUS was implemented via a simple translator com-

ponent running on a laptop computer at the control station. Level-I interoperability is used for the μ WIPs, since most OCUs expect to register distinct subsystems.

Fig 3.22 depicts the simple JAUS layout for the μ WIP, in which a Global Pose Sensor component is configured to interface with the μ WIP via its custom protocol.

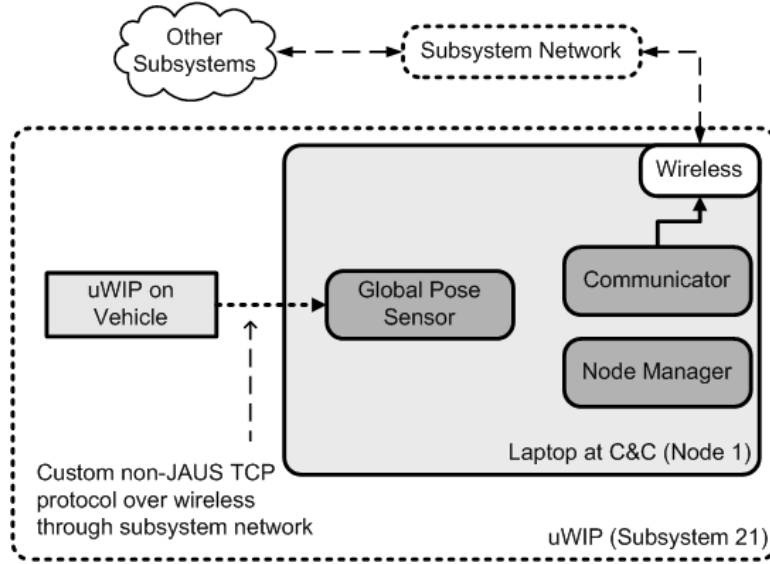


Figure 3.22: JAUS Layout for Micro Wireless Instrumented Payload (μ WIP)

3.11.2 Issues Encountered

The μ WIP records and streams a long list of data for which there is no JAUS message. Currently, this data is simply datalogged on the translator computer for later retrieval. Future work with the μ WIP's JAUS implementation will involve defining a set of experimental JAUS messages for instrumentation, to record some of this data.

3.12 Operator Control Unit

Primarily, Virginia Tech's early involvement in JAUS was focused toward vehicle components. Because of this, the Virginia Tech OCU is designed as

a testing tool rather than a field-able product.

Also, in contrast to typical OCU development where the OCU is an entire subsystem, the Virginia Tech Operator Control Unit is designed as a component. This configuration is more suited to testing, as the OCU can be used to test out a system where level-I interoperability is not implemented.

3.12.1 Service OCUs and the Full OCU

For each service implemented, a complementary OCU component was developed to test out the service’s messaging interface. Each OCU, for example the Primitive Driver OCU or the Waypoint Driver OCU was implemented as a service, with a manual message sender, a message handler, and a state. The user would select the message, enter the destination JAUS address and the relevant data, and click “Send”. Almost nothing was automated; in this way, the tester had full control of which messages are sent during the test.

Later, the services were consolidated into a “Full OCU” component. Still however, the OCU is simply a collection of message senders and message handlers. A screenshot of the Primitive Driver page for this OCU is shown in Fig 3.23.

3.12.2 Aerial Display OCU

In the summer of 2006, Jesse Hurdus and Grant Gothing developed a more user-friendly Operator Control Unit for vehicles with Waypoint Drivers. This component implements more automatic functionality, such as dynamic configuration. For example, instead of loading a text file of target waypoints and manually sending each message, the user simply selects a vehicle from the list and draws the waypoints on a aerial photo display, causing the appropriate messages to automatically be transcieved.

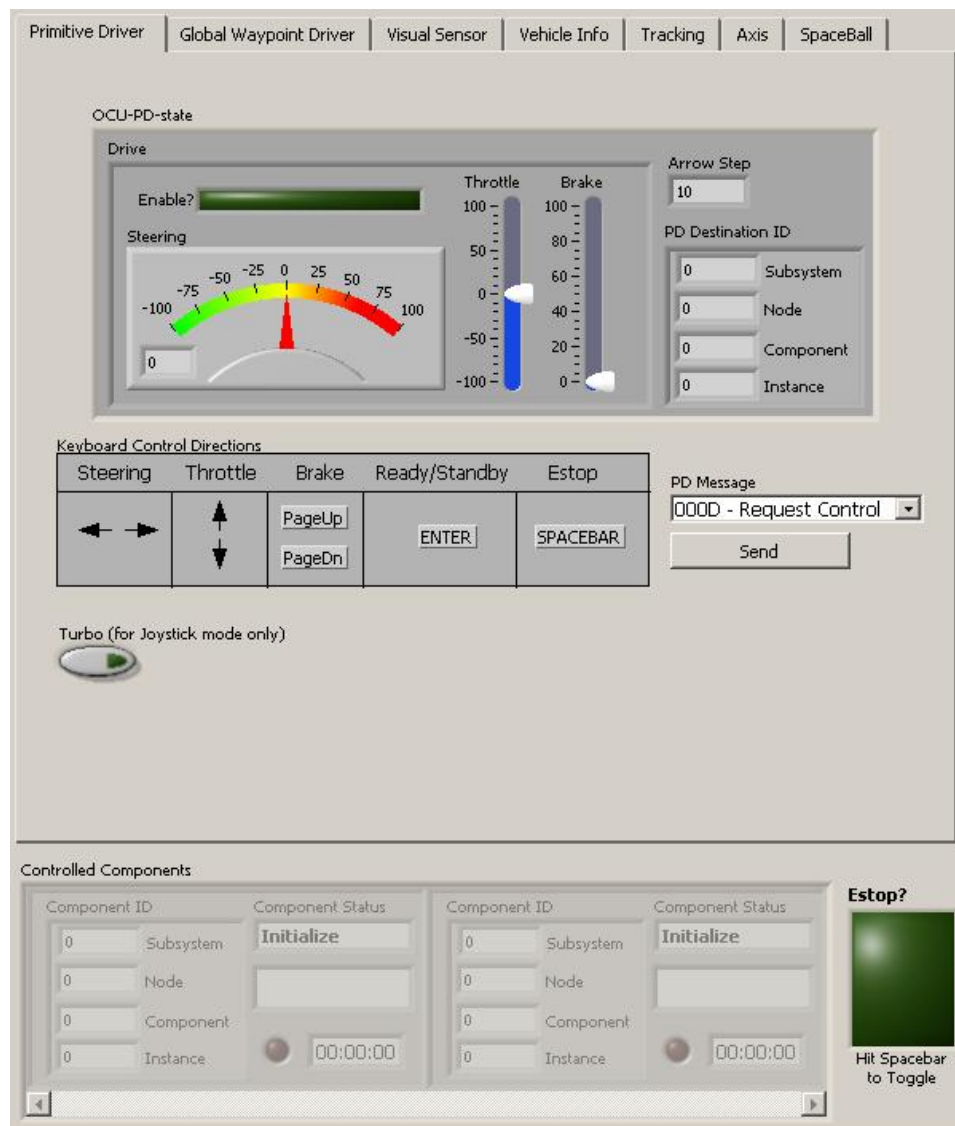


Figure 3.23: Primitive Driver Page for Full OCU

Chapter 4

Results and Recommendations

This chapter presents the issues encountered in implementing JAUS, and a set of recommendations to the standard. In each case, the problem is outlined, the Virginia Tech solution explained, and a recommendation to the standards committee suggested.

Several questions arising from ambiguous points in the standard are presented first, followed by collection of imminent changes to the standard that developers at Virginia Tech are concerned about. Finally, a set of innovations, scenarios which the VT toolkit approaches differently than other groups, are detailed.

4.1 Message-related Issues

In the JAUS specification, the interpretation of several JAUS messages is unclear. This section presents the ambiguous points about individual JAUS messages and the behaviors they are expected to effect.

4.1.1 Interpretation of Wrench Efforts

The Set Wrench Effort message contains six propulsive and six resistive elements; each of these sets is mapped to the specific mobility controls of a vehicle, in the vehicle's coordinate frame [2]. Each value is expressed as a percent, from -100 to 100, and is traditionally interpreted as the “percent of available effort”. Since Wrench Efforts were originally intended for teleoperation, it is likely they are intended to convey joystick position.

Two interpretation issues arise:

Propulsive Linear X Wrench Effort Interpretation

Typically, the propulsive linear X (PLX) wrench effort is mapped to the forward direction, while the resistive linear X effort is usually mapped to the brakes.

However, it is unclear whether a 25 percent PLX wrench effort means the vehicle should move forward at 1/4 its maximum speed, 1/4 its maximum acceleration, or it should hold the throttle at 1/4.

Propulsive Rotational Z Wrench Effort Interpretation

The propulsive rotational Z wrench effort (PRZ) is traditionally mapped to steering, and is ambiguous in two ways.

For an Ackerman-steered vehicle, does the PRZ wrench effort correspond to steering wheel angle, rate of rotation of the platform about the Z-axis (the traditional interpretation for differential drive), or steering wheel rotation rate?

In a differential-drive vehicle the issues are less significant. With regard to rate, the logical conclusion is that a 25 percent PRZ wrench effort means the vehicle should rotate at 1/4 its maximum rate, but this is not explicitly stated in the spec. There is also no standard way to combine PLX and PRZ in converting to differential wheel speeds.

VT Solution

At Virginia Tech, the following mappings are used for wrench efforts in an Ackerman-steered vehicle:

- Propulsive Linear X: Throttle Position
- Propulsive Rotational Z: Steering Wheel Position
- Resistive Linear X: Brake Pressure
- the remaining wrench elements are unused

For a differential vehicle, we use the following conversion equations to obtain two differential wheel speeds:

```
vel_left = (0.01*wrench_p_lx)*(maxSpeed_lx)
          + (0.01*wrench_p_rz)*(maxSpeed_rz)*(track/2);
vel_right= (0.01*wrench_p_lx)*(maxSpeed_lx)
          - (0.01*wrench_p_rz)*(maxSpeed_rz)*(track/2);
```

Recommendation

The specification should contain more detailed information on wrench effort interpretation. Clarity is becoming more and more of a requirement as wrench efforts are produced by autonomous drivers instead of teleoperation joysticks.

4.1.2 Waypoint Management

In the current specification, there is no specified way to delete target waypoints in a Waypoint Driver. Several groups (such as Applied Perception) have elected to set the waypoint number field in the Set Waypoint message to zero, indicating that the old waypoints should be deleted and replaced with the new set.

VT Solution

The Waypoint Driver service checks the waypoint number field in an incoming Set Waypoint message. If it is zero, the current waypoint list is cleared, and the new waypoint inserted as the first point. If nonzero, the waypoint is appended to the end of the list. Replacement of waypoints is not supported.

This arrangement simplifies the implementation in the Operator Control Unit, since to append waypoints it need not first query the Waypoint Driver for the waypoint count. To replace waypoints, the OCU starts numbering the Set Waypoint messages at 0, to append, it starts at 1.

Recommendation

The Set Waypoint message should contain options for “Clear All Waypoints”, “Append Waypoint”, and “Replace Waypoint”.

4.1.3 Waypoint Tolerances

Currently there is no way to specify the tolerance to determine whether or not a waypoint has been achieved. The Set Waypoint message should contain a waypoint tolerance.

4.1.4 System Configuration

Often, it is useful for components within the subsystem to have a listing of the other subsystems within the system, for applications such as service connection cleanup or safety timeouts (see Section 4.2.6). In most implementations, this list is maintained by the Communicator or the primary Node Manager, since all external heartbeats pass through these two components.

These components discover the system configuration by detecting external heartbeats and querying each subsystem for its configuration. Hence, it is impossible for any components not receiving external heartbeats to obtain knowledge of the system configuration.

Recommendation

Currently, the Query/Report Configuration pair of messages supports a Node configuration and Subsystem configuration [3]. A system configuration query type should be added to this message, enabling any component within the subsystem to query the primary Node Manager for the system configuration.

Since it is not always necessary to transmit the entire system configuration containing a listing of every component on every node in every subsystem, this message would provide an option to send only a list of subsystem numbers.

4.2 Protocol Issues

If we understand JAUS as a language, messages are the words, and protocols are the grammar. It was discovered in the ETG Experiments that, while the messages are present to accomplish the desired functions, the rules for using them need more definition [6].

This section presents an array of protocol questions which are not clearly answered in the specification.

4.2.1 Message Ordering

At ETG Experiments 2.75 and 3.0, many Waypoint Drivers had problems interoperating with Autonomous Solutions Mobius OCU, since they required messages in a particular order. For example, several groups would ignore any Set Travel Speed messages received before a Set Waypoint message. Such a problem is detrimental to on-the-fly interoperability in the field.

VT Solution

Virginia Tech’s message handlers do not usually care about message ordering. An Operator Control Unit may send a Set Travel Speed, a Resume, and a Set Waypoint message in any order, but the vehicle will not begin moving until all three have been recieved.

In situations where the message ordering does matter, clear documentation ensures the designer of an interfacing component has enough knowledge to correctly design for interoperability.

Recommendation

The specification should specify behavior for message ordering, so that an Operator Control Unit or other vehicle knows what behavior to expect when trying to interoperate. This issue is being addressed with the JAUS Service Specification, with the documentation of service interfaces [29].

4.2.2 Switching from Autonomous Mode to Teleoperation

There is no specified way to switch a vehicle between teleoperation and autonomous mode. While many methods exist, a standard method or set of methods should be mandated by the specification, to facilitate interoperability.

VT Solution

In a Level-III interoperable system, assume we have a Waypoint Driver and Primitive Driver, like the MATILDA (Section 3.3). Upon startup, the Waypoint Driver obtains control of the Primitive Driver and begins sending Wrench Efforts; at this point the vehicle is in autonomous mode. For an OCU to teleoperate the vehicle, it simply requests control of the Primitive Driver, causing the Primitive Driver to reject control by the Waypoint Driver. To resume autonomous operation, the OCU releases control of the

Primitive Driver and sends a “Reset” message to the Waypoint Driver, so that the Waypoint Driver reattains control of the Primitive Driver.

An alternative solution is for the Waypoint Driver to only request control of the Primitive Driver when commanded into the Ready State.

Unfortunately, both schemes only work in a system where autonomy and teleoperation are handled by two separate components. In a Level-II system, such as Helium Red, where teleoperation and waypoint driving are handled by the same component, messaging is no longer sufficient. Section 3.8.4 presents the solution implemented for Helium Red.

Another Solution

The above solution was chosen to maintain interoperability through only JAUS messages. A more straightforward and versatile solution is the implementation of a new “Set Command Mode” message, in which the body contains a field specifying the operation mode of the command link. The field could include the following values:

- Wrench Effort Control
- Vector Control
- Waypoint Control: includes Set Waypoint and Set Travel Speed

Note that in the case where a vehicle would receive waypoints from an OCU, then travel for miles, out of the OCU’s range, and potentially be controlled by a different OCU at its destination, a different operation mode would prevail.

Recommendation

The standard should specify a protocol for switching between different command modes. This could be achieved with the introduction of a new “Set Command Mode” message, which informs the controlled component which command messages it should expect.

4.2.3 Message Handling with respect to Status

In the specification, there are messages that are specified to be only effective when the receiving component is in a particular state [2]. If the receiving component is not in that particular state, should a Nak be sent as a reply, or should the message simply be dropped?

For example, if a component is in the Emergency state, does it require a “Clear Emergency” message to transition out of the Emergency state, or will a “Ready” or “Standby” message be sufficient?

There is a state machine slide in the JAUS Tutorial presentation published by the JAUS Working Group indicating the possible transitions from state to state [7]. Something similar to this should be included in the specification.

VT Solution

Any status control message will effect the component into the commanded state, unless some other service is restricting the state. For example, a Waypoint Driver may remain in Initialize until it has successfully established a Global Pose service connection and gained control of the Primitive Driver.

Recommendation

State transitions should be clearly specified in the JAUS Services Specification. Further, state machines should contain more states than the five specified JAUS states, and the “secondary status” field of the 4002h Report Status message should be used to report this.

4.2.4 Command Filtering with respect to Controller

In the JAUS ETG, it is typically required that a component be under control by the sender of a command before executing the command; all queries however, are handled regardless of the sender [3]. This is not stipulated

anywhere in the specification, and should be included in the JAUS Service Specifications.

VT Solution

In most Virginia Tech message handlers, commands messages are not executed if they do not originate from the current controller. The most significant exception is status control messages, in order to enable a broadcast Emergency stop or Shutdown message for testing.

Another exception is the message handler for the Vector Summer service in the Cadillac SRX (Section 3.6.5). Since the vehicle employs Potential Fields, this component must accept commands from multiple behavior components, even though it can be under control by only one component at a time.

4.2.5 (A)periodic Wrench Efforts

At Virginia Tech, the initial Operator Control Unit sent a Set Wrench Effort message to a Primitive Driver only when the Wrench Efforts had changed (the joystick had moved).

At ETG Experiment 2.75, we learned that most other groups send Wrench Efforts periodically, at a very high rate. Initially, we viewed this as unnecessary message traffic, but later learned that the Wrench Efforts were being used for a watchdog timer in the Primitive Driver (see Section 4.2.6).

In addition, these messages were sent at a very high rate through normal message routing, opening up the possibility for queueing. While one solution is to simply decrease the receiving Primitive Driver's cycle time, a better solution would have been a command service connection. Command service connections are discussed further in Section 4.5.2.

Recommendation

Though the specification should not govern implementations to this level of detail, a set of guidelines may be helpful. These guidelines would help minimize unnecessary message traffic, an issue that will arise when JAUS is implemented on smaller vehicles with less computing power and systems are deployed on networks with lower bandwidth.

4.2.6 Watchdog Control Timeouts

In the Interface Control Document for ETG Experiment 2.75, it is stipulated that a vehicle under control by an Operator Control Unit should release control by its current controller and transition to the Standby state if it has not heard from the Operator Control Unit in a specified amount of time [3].

VT Solution

To accomodate this requirement, a heartbeat forwarding mechanism was implemented, making use of the periodic heartbeats already being broadcast by each subsystem. When the Node Manager detected a “Confirm Control” message originating from a local component, it began forwarding all heartbeats from the controller’s subsystem to the controllee. In this way, the controlled component could watch for a break in the incoming heartbeats and enter the safe state if necessary.

Other Solutions

Most other groups at the ETG used the periodic Wrench Effort messages for the watchdog timer. While this works for teleoperation (where Wrench Efforts are often sent periodically), it breaks down when the OCU tries to command a Waypoint Driver, where the OCU may only send a command message every minute or two.

One further solution employs the Configuration and Event messages. A controlled component would establish a change-based event for configuration, to receive an update whenever the subsystem configuration had changed. This method introduces the least extra traffic, since the Node Managers monitoring node, subsystem, and system configuration are already checking for heartbeat timeouts. Unfortunately, in order to implement this, a modification to the Query/Report Configuration message pair is required (see Section 4.1.4).

Recommendation

Control timeouts should not be based on command messages, as the messages may arrive aperiodically. Three alternatives are proposed:

1. Implement heartbeat forwarding as described above in Section 4.2.6
2. The controlled component should set up a change-based Event with the Node Manager for the System, Subsystem, and Node Configuration. When the configuration changes, causing the component to receive a notification, it checks if its controller has disappeared from the list. This would require a modification to the Query/Report Configuration message pair (see Section 4.1.4)
3. If the commands are sent periodically, use a command service connection and implement a watchdog pursuant to the update rate of the service connection

4.2.7 System, Subsystem and Node Identification

In the dynamic configuration messages for the JAUS ETG, there is a set of Query/Report messages (D2A4h, D4A5h) to transmit a human readable name for a component, node, subsystem, and system [3]. This is useful in determining what organization is responsible for a certain subsystem number during a test. A minor set of issues arise:

1. If each Node Manager stores the node name and the primary Node Manager stores the subsystem name, how are these transmitted to components?
2. Where is the system name stored?
3. When a subsystem joins a system (signs on to a network or is powered on), how does it receive the system name?
4. The messages are called Query Identification and Report Identification. Since this can be confused with a component ID (subsystem, node, component, and instance), an alternative description would be Query Name and Report Name.

VT Solution

The VT solution has two parts, in order to support level-III interoperability. First, when a new component comes online, it queries the Node Manager for the node name and subsystem name. In addition, the Node Manager automatically reports the subsystem name and node name to any component that has just come online, just in case the implementer of the component (suppose it is not a VT component) did not decide to query for the name.

These are both illustrated in Fig 4.1.

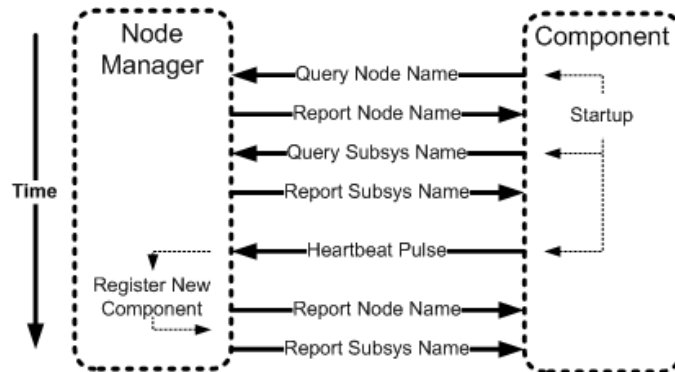


Figure 4.1: Message Flow for Receiving Node and Subsystem Name

Subsystem and node names are only applied if the Report Identification message originated from the local Node Manager.

Recommendation

The specification should either specify this protocol for sharing the subsystem name, or stipulate that only the primary Node Manager or Subsystem Commander can be queried for the subsystem name, and only a Node Manager can be queried for the node name. The message spec should also clarify use of the system name.

4.3 Message Routing Issues

In this section, several issues related to basic message routing between components are discussed.

4.3.1 Incoming Message Handling

Under normal operation, a JAUS component can receive two types of incoming messages: normal messages routed through the Node Manager, or service connection messages. Since both messages contain a priority level in the JAUS header, and service connection messages are not to be queued, a slight ambiguity arises regarding the order of message processing.

VT Solution

Fig 4.2 depicts the input message processing structure for the VT Toolkit, in which each component has a single input FIFO queue. When the Node Manager routing process enqueues a message in this queue, it is immediately dequeued by the component, parsed, and either stored for reassembly if it is part of a multipart message, or enqueued in one of sixteen different priority queues.

On each cycle of the main component thread, the oldest highest priority message from these sixteen queues is dequeued for processing. In parallel, any of the active service connections which have triggered are read,

yielding an array of messages. This array of SC messages and one normal message are then processed in the message handlers.

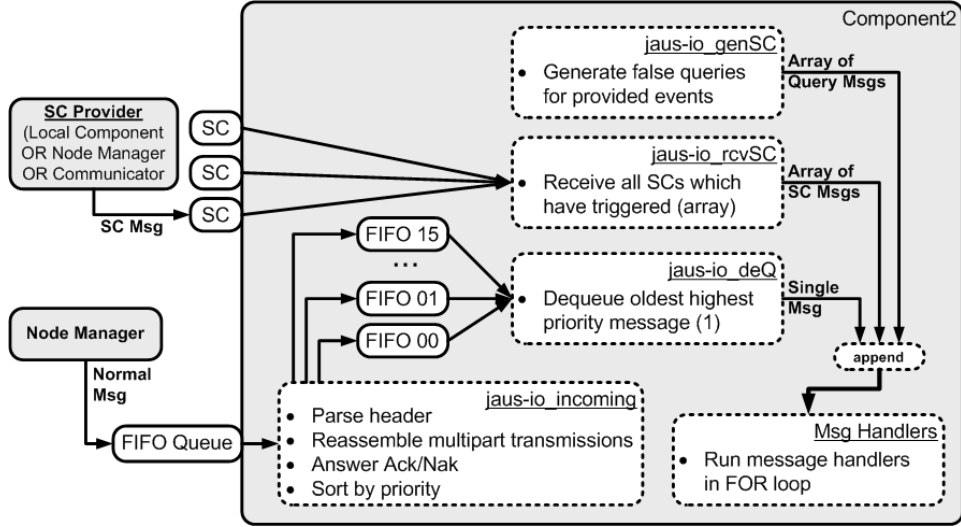


Figure 4.2: Routing for Incoming Messages

4.3.2 Service Connections

The handling of service connections is discussed in Section 4.6.3.

4.3.3 Multipart Message Sequence Number

To handle large data sets, the current JAUS message header contains a set of data control fields used for the handling of multipart (packetized) transmissions [2]:

- **Multipart:** four bits with the following interpretations: Normal message (single packet transmission), First packet in stream, Middle packet in stream, Retransmitted packet, Last packet in stream.
- **Sequence Number:** increments for each packet in a large data set transmission.

A Potential Problem

There is no way to indicate what percentage of the transmission has been recieved so far. Suppose we are on a lossy link and we decide to reassemble the Camera Image message after we recieve 80 percent of its packets. Currently, this 80 percent cannot be calculated, since the recieving component never knows how close it is to the end until the last packet is recieved. What happens if we never recieve that last packet?

Surely, the component could simply reassemble the current transmission when it recieves another “first packet” message (presumably from the next multipart transmission). But, suppose this packet is lost as well? One further issue is that the transport medium does not guarantee order of delivery – and the standard should be prepared to handle this. Packet 7 should not be discarded even if it arrives between packets 11 and 13.

Recommendation

The Sequence Number field should be instead interpreted as the number of packets remaining in the transmission; for a transmission of N packets, the sequence number would decrement down from N-1 on the first packet. With this arrangement, upon recieving a packet, the recieving component knows exactly how many more packets are on the way.

4.3.4 Multipart Message Scope

One further issue related to multipart messaging poses the question: Where are message packetized – when they leave the component, or when they enter a subsystem or node network?

It is evident from reading the JAUS specification that multipart messaging should be used between any JAUS entities, including components. However, several groups have decided to break a message into packets only when it is transmitted over a network (from node-to-node or subsystem-to-

subsystem). While this decision enables efficient intranodal processing, it inhibits true level-III interoperability.

This issue is further discussed in Section 4.5.4.

4.3.5 Ack/Nak Rules

Two main ambiguities exist regarding the use of the ack/nak field in the JAUS header: (1) What are the criteria for a Nak message? and (2) What constitutes an Ack?

The JAUS RA version 3.2 provides the following rules governing the response when an acknowledgement is requested [2].

- Set AckNak to “Msg Not Rcvd” to signal that the command is not supported
- Set AckNak to “Msg Not Rcvd” to signal that the destination is unknown to the Node Manager
- Set AckNak to “Msg Rcvd OK” to signal that the command was received OK

The following questions arise:

- If the message size in the header does not match the message body, should a Nak be sent?
- If the command is supported but the message is malformed, ie the length of the body does not match the presence vector, should a Nak be sent?
- When sending an Ack, should it be sent when the command was received or executed?

VT Solution

In the VT Toolkit, Ack/Nak is handled entirely by the level-III transport layer. When a message header is parsed, but before the message is enqueued

in to the priority queues (Fig 4.2), Ack/Nak is handled. If an acknowledgment is requested, an Ack will be sent, unless:

- the Message Size field is outside the allowable range 0-4080
- OR Message Size is incorrect
- OR the Data Flag bits compose an illegal value
- OR the Service Sonnection bit it set AND the Ack/Nak field is not set to Response Not Required

in which case a Nak is sent.

4.4 Other Issues

This section presents a variety of implementation issues that do not fit in the categories above.

4.4.1 Desired versus Actual States

Suppose an OCU is ready to teleoperate a stationary vehicle with a Primitive Driver. The OCU sends a Set Wrench Effort command to a Primitive Driver, with Propulsive Linear X set to 100 percent. 30 milliseconds later, the OCU sends a Query Wrench Effort message. Should the Primitive Driver respond with the current value being commanded to the motor, which will be lower since the vehicle will only have started to move, or the set point of the Wrench Effort?

This issue comes up for other messages as well; the specification should clarify how queries should be handled.

4.4.2 Command Completion

Suppose an OCU spools a mission to a vehicle, containing a Set Waypoint message and a Set Travel Speed message, and establishes a change-based event for mission status. Should the vehicle report that it completed the

Travel Speed command when it is established as the set point, or when the speed is attained?

A similar question exists for the Waypoint message. Should the vehicle report the message as completed when the new waypoint is added to the list of target points, or when the waypoint is actually achieved? It appears that in mission spooling, the Set Waypoint command is used more often as an “Achieve Waypoint” command.

Current Solution and Recommendation

Current implementations in the ETG mix behaviors. Vehicles report a Set Travel Speed message as completed immediately, since a vehicle with no Target Waypoints should not simply begin moving in a random direction. Likewise, vehicles report a Set Waypoint message as complete only when the waypoint has been achieved.

The specification should attempt to standardize behavior for commands or explicitly clarify the appropriate behavior for each command.

4.4.3 Function of the Subsystem Commander

According to the Reference Architecture version 3.2, the Subsystem Commander “coordinates all activity within a given subsystem”. It further specifies that these functions comprise “mission planning, issuing commands, querying status for operation, and may be performed by a human being or by a computer”. [2]

In this section, I present (perhaps encouraging laughter), my initial perception of the Subsystem Commander function. Only when I read the Interface Control Document for ETG Experiment 2.75, did I discover the understanding intended by the JAUS Working Group.

Initial Perception of System Layout

From the beginning, I thought all components inside a subsystem were private, their existence unpublished to external subsystems. Components existed primarily to facilitate the division of modularity in software.

I thought the Subsystem Commander was the only visible component in the subsystem, and all messages from external subsystems were addressed to it. For example, a Set Wrench Effort would arrive into the Subsystem Commander, and it was responsible for forwarding the Wrench Effort to the internal Primitive Driver. In the case of queries, the query was forwarded internally by the Subsystem Commander, and the response generated by the component was reformed for the external component, the source JAUS address replaced with that of the Subsystem Commander.

Realization this was Incorrect

When I read the Interface Control Document for ETG Experiment 2.75, I was shocked to see that external components communicated directly with components internal to a subsystem. It seemed odd to me that an external subsystem would be privy to such detailed configuration information about an unmanned system intended for military use.

Possible Reasons for Theory

In retrospect, several reasons led me to my initial theory of operation for the Subsystem Commander. First, is the diagram typically used to illustrate the hierarchy of JAUS entities, reproduced in Fig 4.3. I concluded from this diagram that Subsystem Commanders and Components were not peers, but rather the Subsystem Commander in effect “encompassed” the Components.

Coupled with the fact that the Subsystem Commander coordinates all activity within the subsystem, and the idea that JAUS was being developed

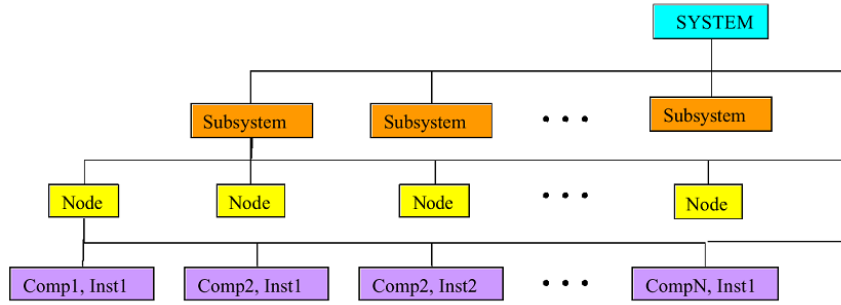


Figure 4.3: System Configuration as Illustrated in JAUS RA [2]

for military applications, I arrived at this erroneous understanding of the system architecture.

Recommendation

Two further issues arise from Fig 4.3. First, components are contained within a node, and are not subordinates to a node. Second, the Node Manager is not shown in the diagram.

I propose that a second figure, similar to Fig 4.4, be used to illustrate subsystem layout. This figure clearly depicts the fact that a subsystem is “a logical grouping of component [software running] on one or more [computing] nodes” [2]. It also communicates that each node contains a Node Manager, and that all traffic to and from the subsystem passes through the Communicator.

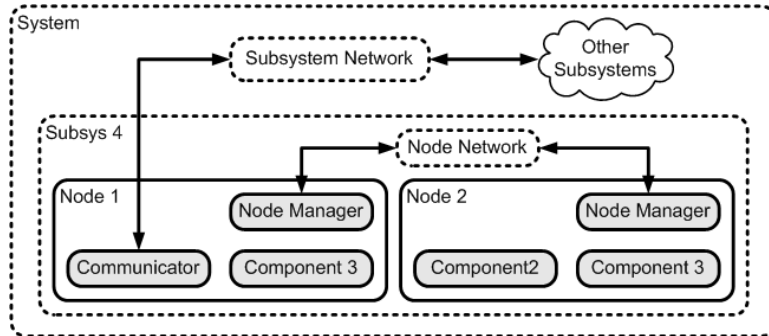


Figure 4.4: System Configuration as Illustrated by Ruel Faruque

4.5 Special Concerns

This section discusses several imminent changes to the JAUS specification which developers at Virginia Tech are concerned about.

Many of these “imminent changes” were overheard in discussions at the JAUS Working Group, or inferred from observing the implementations of key developers in the Experimentation Task Group. As a result, some of them may have already been addressed.

4.5.1 Deprecation of 0408h Set Local Vector

While the Reference Architecture version 3.2 mentions that the Set Local Vector message is set to deprecate at the next revision, this particular message is necessary to implement a Potential Field strategy for navigation. This issue is discussed in Section 3.6.5.

It is recommended that the 0408h Set Local Vector message not be deprecated in the next revision of JAUS.

4.5.2 Command Service Connections

Latest-data connections are critical in an autonomous vehicle application, especially at high speeds. Queueing of commands can cause grave issues with performance when a vehicle begins executing old path planner commands. Even more catastrophic is queueing of sensory data, when a Navigation Driver generates commands based on out-of-date information.

In JAUS, latest-data connections are implemented via service connections. As discussed earlier, messages flowing on a service connection are treated specially – they are never queued. New data replaces old, ensuring that only the most recent data is available.

Removal

In the ETG Experiments, the service connection management messages in RA 3.2 are replaced with a new set of experimental Event messages [3]. While the RA 3.2 messages contain provisions for command service connections, these new otherwise-superior Event messages do not. Coupled with the fact that nobody in the ETG employs command service connections¹, it is apparent that they are being deprecated.

Alternative

At an ETG Experiment, When our vehicle's Primitive Driver running at 10Hz experienced substantial latency when controlled by an OCU sending command messages at 10Hz and query messages at 1Hz (the sum exceeding the component's frequency), we were advised to decrease the cycle time of our Primitive Driver. While this can (and did) solve the immediate problem, such a system still remains vulnerable to network and computing hiccups.

The problem at hand is perfectly suited for a command service connection, since only the current Wrench Efforts are important to the Primitive Driver.

VT Solution

Instead of implementing the command service connections detailed in Reference Architecture version 3.2, we elected to develop a simpler service connection method. At my suggestion, Virginia Tech developer Grant Gothing implemented two new experimental messages to generate a command service connection.

There were several reasons for deciding not to implement RA-3.2 command service connections, and for introducing new command codes.

¹This assertion is made without a formal query, but based on observation.

- The complex multi-controller scenarios supported by RA-3.2 service connections are rarely, if ever, used in implementation.
- Introduction of new Create and Confirm messages is required, since there is no Query Message (a field in the Create Event message for a traditional Query/Report service connection) for a Command service connection

In addition, a slightly different, more refined solution, optimized to reduce message traffic, is presented in Section 4.6.4.

Verification

Gothing and Hurdus verified the command service connections on the Cadillac SRX, and clearly demonstrated the improvement provided by latest-data connections [8]. With the subsystem configured as shown in Fig 3.9, the same waypoint navigation course was driven multiple times, first with command service connections disabled, and then with command service connections enabled.

Fig 4.6 and Fig 4.5 show the results obtained by Gothing and Hurdus. An analysis of the mediocre performance without service connections is presented in Section 4.5.3.

4.5.3 Analysis of Command Service Connection Performance

Why did the Cadillac SRX experience such horrible performance with regular message routing? This is especially astounding given that many other JAUS-interoperable vehicles have performed satisfactorily without command service connections. Most notably, NaviGator, the University of Florida vehicle entered in the 2005 DARPA Grand Challenge, traversed 14 miles without issue [15] [14]. What then, is the justification for reinstating command service connections based on the performance of the Cadillac SRX?

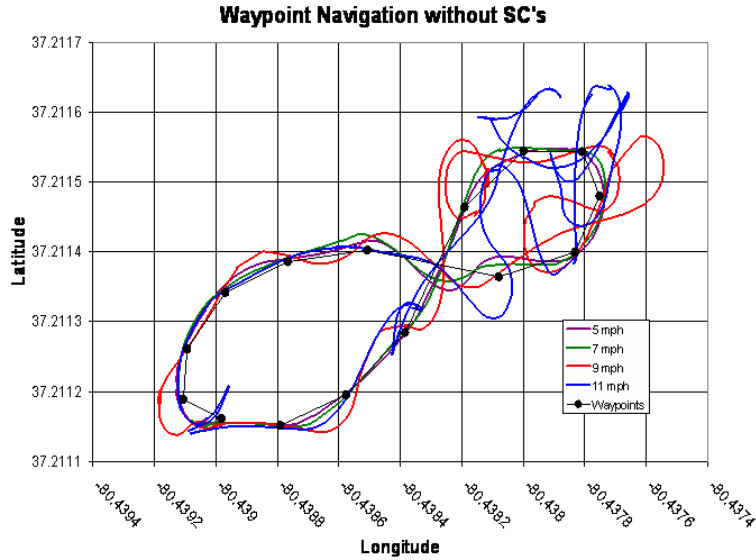


Figure 4.5: Waypoint Navigation without Command Service Connections [8]

Possible Reasons for Bad Performance

As the lead developer of the core software that handles all message routing in the VT Toolkit, I would attribute the mediocre performance to the following potential causes:

- The VT implementation is not Real-Time. Hence, the allocation of computing resources is non-deterministic.
- Currently, the VT implementation is not completely event-driven. While many features *are* event-based, the Node Manager routing loop and the message handling loop in a component still run at a fixed cycle time.
- Sending all messages via normal command routing means they must pass through the Node Manager. If the Node Manager thread is not given a high priority, a bottleneck can occur.
- The components may not have been running fast enough to receive incoming messages. A component running at 10Hz can only receive messages at rates below 10Hz, and any delays will cause messages to

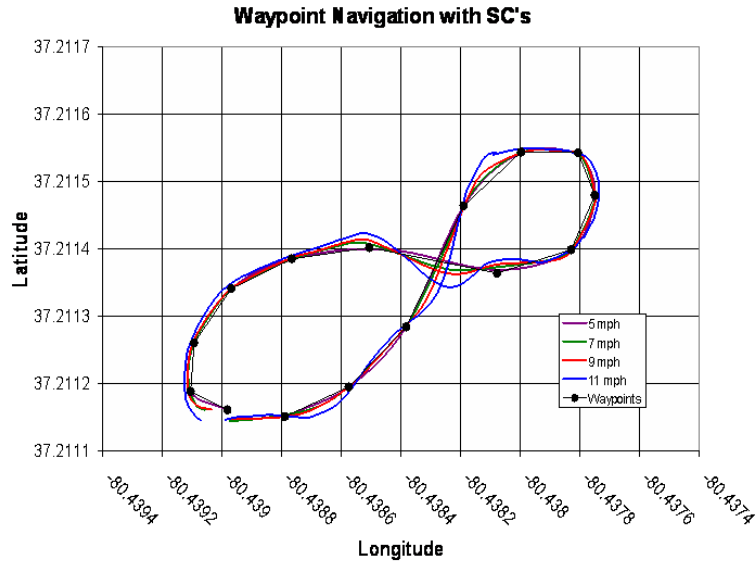


Figure 4.6: Waypoint Navigation with Command Service Connections [8]

be queued.

- Incorrect implementer setup, where a component's cycle time is set to 10Hz, and it is expected to receive messages at 10Hz or faster. In this situation, even one unexpected message causes the message handler to fall behind. Gothing and Hurdus were careful not to make this mistake.
- The test was executed with open-source VIs running in LabVIEW rather than compiled code; in addition, other Windows programs were running.

Two Cascading Causes

Since Gothing and Hurdus set the component cycle times appropriately for the rate of incoming messages, the root cause is not incorrect setup. The problem is then attributed to two cascading causes.

First, sending all messages via normal message routing requires they pass through the Node Manager, causing a potential bottleneck. This bottleneck however, only occurs if the system has not been designed to give the Node Manager a higher priority. Hence, the second cause is the non-real-time

implementation.

When service connections are employed, messages between local components do not pass through the Node Manager, and instead flow directly from component-to-component. Inter-nodal service connection messages follow a similar shortcut, being recieved by the component directly after being read from the network, as shown in Fig 4.9.

Note that the mediocre behavior does not worsen throughout the test. Rather, drastic overshoot is experienced, yet the system remains stable; it builds error, then recovers. Computationally, this behavior is attributed to the thread manager. Corroborating this conclusion is the observation that the number of elements in the Node Manager's input queue would peak rapidly, then recover, periodically.

Root Cause

I conclude the root cause is the bottleneck of messages being routed through the Node Manager. This was exacerbated by the non-deterministic thread management.

Both of these problems were present at all speeds, but were evidenced more at higher travel speeds since the vehicle travelled further during the recovery time.

Justification to Reinstate Command Service Connections

With this analysis in mind, the reinstatement of command service connections is still recommended, for several reasons.

First, the very nature of reactive navigation relies on having access to the most recent data. The JAUS standard should not remove the capability of latest-data connections for commands.

In addition, by introducing magnified latency issues, the clear impact of command service connections was highlighted and evidenced. It is in-

disputable that implementing command service connections had a positive effect on performance.

4.5.4 Level-III Interoperability

To date, there has only been one successful demonstration of cross-organization level-III interoperability – that is, two components from different organizations running on the same computing unit (node). This first successful test, accomplished by Virginia Tech and the University of Florida in December 2006, is detailed in Section A.3.

In this section, we present a collection of issues related to level-III interoperability.

Transport Specification Should Extend Into Level-III

In the newest draft of the Transport specification, the data control flags and the multipart message sequence number (used in packet transmissions) have been moved from the traditional RA-3.2 message header (Fig 1.5) used by JAUS applications (message handlers), to the lower-level JAUS transport layer. As a result, the details of packet reassembly are handled by the JAUS transport layer, and a condensed JAUS message header passed to the application layer (the message handlers). [37]

In this new transport specification, the packaging of the multipart data control flags and multipart sequence number are specified for each medium, such as UDP or serial. In effect, the traditional JAUS message from RA-3.2 (less the multipart fields) is carried inside a transport-medium-specific packet defining the multipart message flags. [37]

As a result, multipart messages can only be passed between nodes (level-II) or subsystems (level-III). Components on the same node wishing to send large messages to each other must implement a non-JAUS protocol to pass messages, since the data control field is not in the new application header.

To address this problem, JAUS should govern the byte sequence passed between components. Since there are many physical mediums for passing data between components, the actual medium-specific packet need not be specified, only the byte sequence (application header + transport-level multipart fields) to be passed in the FIFO queue, POSIX shared memory, over the NIC loopback device, or whatever other medium is used for Component-to-NodeManager communication.

In the future, the Transport specification could define the packet for each level-III medium, or simply leave that to the implementer. In either case however, the sequence of bytes (modified application header + multipart reassembly fields from transport header) must be established up front to enable true level-III interoperability.

University of Florida / OpenJAUS NMI Standard

To accomplish the level-III interoperability test of Section A.3, the Node Manager Interface (NMI) used by OpenJAUS was implemented in the Virginia Tech components [32]. The NMI specifies a UDP port for each component, and a port for the Node Manager, JAUS messages being passed via the NIC loopback device. Each UDP packet contains a single JAUS message, using the RA-3.2 header.

In addition, OpenJAUS defines a small set of non-JAUS messages to enable Check-In and Check-Out of components, and to assign available JAUS identifiers to components checking in. There are also messages to verify the existence of a particular JAUS address in the subsystem, and to query/report the JAUS identifiers of local components supporting a particular service.

The OpenJAUS specification was very simple to implement, allowing a Virginia Tech component to interact with the University of Florida Node Manager after less than an hour of work.

With a few clarifications and enhancements, the OpenJAUS NMI is a suitable model for level-III interoperability. The most significant enhancement would be to the handling of service connections. In the OpenJAUS Node Manager, all messages, including service connections, are routed through a single FIFO queue. It is recommended that service connections flow instead on shortcuts directly between components, as shown in Figs 4.8 and 4.9. To accomplish, service connection messages would flow directly from component to component via the loopback device, bypassing the Node Manager.

Recommendations

The following recommendations are suggested to facilitate level-III JAUS interoperability:

- JAUS should define the sequence of bytes (application header + transport-level multipart fields) to be passed between components and the Node Manager. The details of how to pass these bytes can be left to implementers, or later defined by the standard for different mediums.
- The OpenJAUS NMI (Node Manager Interface) provides one workable solution for standardizing intraprocessor communication where a network loopback device is available.
- The special routing of service connection messages (bypassing the Node Manager, flowing on shortcuts) should be clarified.
- To facilitate the processing of messages, the JAUS message header should be retained on messages flowing along service connections.
- A system configuration message should be implemented, to enable components to verify the presence of another JAUS address (e.g. service connection provider should know if the requestor has disappeared). An alternative would be a special change-based event in which the

Node Manager (already watching for timeouts) alerts subscribers when a particular JAUS address has signed off.

4.6 Innovations

In this section, we present several issues Virginia Tech has approached differently than other groups.

4.6.1 Multiple Services per Component

In the Reference Architecture version 3.2, it is specified that a component may perform one and only one service, bound to the component ID [2]. More recently though, with the introduction of the experimental dynamic configuration messages in the ETG, a component can support more than one service, but must support at least one [3].

However, from my experience in the JAUS Working Group, only a few groups have actually implemented multiple services per component. Below, we present an example Virginia Tech component which supports multiple services through composition of services.

Global Pose / Velocity State Sensor

When implementing JAUS on the Cadillac SRX (Section 3.6), it became necessary to interface with the NovAtel GPS/INS system – a GPS receiver coupled with an Inertial Measurement Unit (IMU). Since both pose information and velocity information are obtained from the same hardware interface, we added a Velocity State Sensor service module to our Global Pose Sensor.

Fig 4.7 shows the block diagram for the Global Pose / Velocity State Sensor. Each service has an message handler (circled), and a state wire (labelled). The latest INS data is copied into the state wires, and the message handlers act upon the current data in the state wire.

4.6.3 Header on Service Connection Messages

According to Dan Gehring of AMRDEC, service connections (SCs) were initially intended never to be transmitted via messaging. Instead, the data that the requester desired would simply be written to shared memory by the provider at the requested rate, and the requester would read it as desired. The JAUS message header is only used for service connection transfers when the data must be sent across a network. [34]

While this interpretation is highly efficient, it does not facilitate level-III interoperability, since setting up the service connection requires knowledge of the sharing mechanism. In addition, the way in which a Provider component publishes the service connection data differs depending on where the Requestor component is located in the system.

Most significantly, a receiving component cannot automatically receive all triggered service connections via standard Component Model functionality and simply run the messages through its normal message handling chain. Rather, it must be configured to read the shared memory as needed and parse it into the component state.

VT Approach

At Virginia Tech, service connections are understood instead as “periodic latest data connections”, where data is sent via JAUS messaging, only via shortcuts enabling messages to bypass any queues. The JAUS header is retained on SC messages for the following key reasons:

1. The service connection bit is used to signal the routing software that the message is flowing on a service connection shortcut.
2. The destination JAUS address (for the requestor) is used by the Node Manager or Communicator (mimicking a local requestor) to route the message over a network according to the node or subsystem table. In this way, the Provider component does not need to publish the data

differently depending on the Requestor component’s location in the subsystem, as detailed below.

3. The JAUS header allows the reciever to parse the message through its normal message handlers. Such an approach allows for rapid integration of new experimental messages, by only writing a message handler. This is discussed further in Section 4.3.1, where the handling of incoming service connections is presented in Fig 4.2.

VT Message Flow

Fig 4.8 illustrates the message flow for communication between two components on the same node, in the VT Toolkit. When a Provider component puts a message into an outgoing service connection (“outSC”), the Requestor component recieves notification and can read the message immediately from (“inSC”).

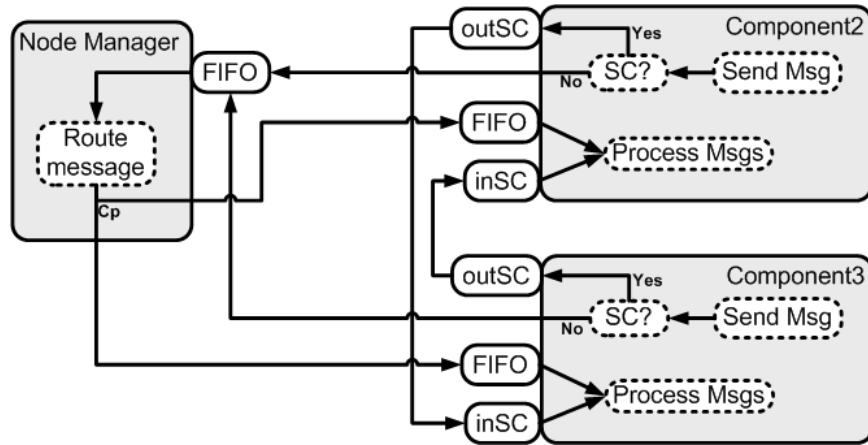


Figure 4.8: Message flow between Components on the same Node

The paths traveled by a messages traveling on service connections between components on different nodes or subsystems are depicted in Fig 4.9. If a local component is a Provider on a service connection to another node or subsystem, the Node Manager or Communicator mimics a local requestor. Conversely, for a local Requestor, the Node Manager or Communicator mim-

ics the Provider.

In both cases, the local component treats the service connection in an identical manner.

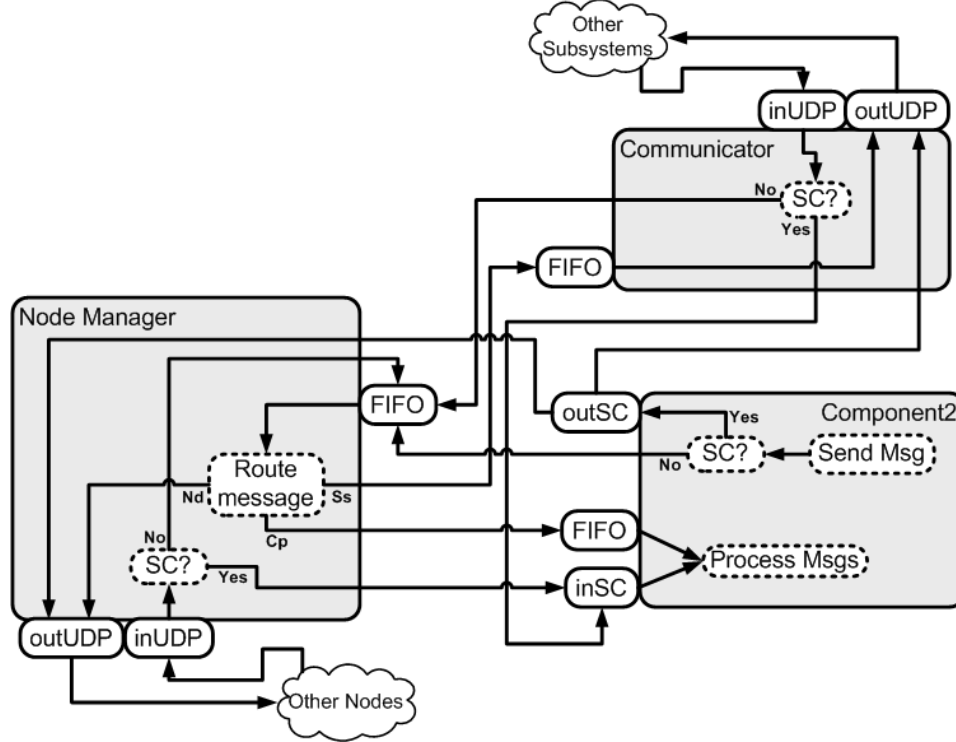


Figure 4.9: Message flow between Components on different Nodes or Subsystems

4.6.4 Latest-Data Connections for Commands

As illustrated with the Cadillac SRX in Section 4.5.2, latest-data connections have a profound impact on performance. Developers at Virginia Tech have recommended the reinstatement of command service connections, however in this section I present a more refined solution, optimized for the application.

In JAUS, latest-data connections are typically referred to as service connections, however the reasons for implementing a latest-data connection are different depending on the data being sent.

Two Reasons for Latest-Data Connections

Put concisely, the primary reason for employing a service connection is to ensure the latest data is available to the Requestor. Old data should replace new. With this key point in mind, there is a slight difference between a standard Query/Report service connection, and a Command service connection:

- In a Query/Report latest-data connection, it is the Requestor that determines the message rate.
- In a Command latest-data connection, it is the Provider that determines the message rate.

Realizing that the more intelligent software (the Navigation Driver, as opposed to the Global Pose Sensor or Primitive Driver), determines the rate for the latest-data connection, we can establish the following:

- In a Query/Report latest-data connection, the Provider is instructed (during setup) by the Requestor when to update the data in the latest-data connection. Since the connection is set up in advance, there is no possibility for varying the update time intelligently (such as sending a message only when the data changes).
- In a Command latest-data connection, the Provider is calculating the commands to be passed in the latest-data connection, and as such only needs to update the data when necessary (or as frequently as it likes).

Proposed Solution

I propose that latest-data command connections be implemented in a slightly different manner than traditional command service connections. The message path would be established in the same fashion as for a report service connection, shortcutting and eliminating queueing. However, the Provider (the producer of the commands to be sent on the connection), would not be re-

quired to send data at the predefined rate, rather could send it only when required.

One further practical use of service connections for watchdog timers. When a Requestor determines that it has not recieved a message within a multiple of the service connection update rate, it may take some action, such as reporting an error, trying to reestablish the service connection, or entering a safe mode. To enable support for this in latest-data command connections, a timeout field could be added to the message used to create the connection.

4.6.5 Experimental Messages

The following experimental messages have been implemented by Virginia Tech. Many of them have been discussed in detail in the Implementations chapter, but are still presented here.

Laser Range Finder

Two messages were implemented to report the raw scan data for a Sick Optics planar Laser Range Finder. As discussed in Section 3.10, it possible to send obstacle data via world model messages, but many higher-speed reactive obstacle avoidance strategies are designed for the raw scan data, expressed as polar plot.

- E238h Query LRF Scan: This message causes the recieving component to respond with msg code E438h Report LRF Scan.
- E438h Report LRF Scan: This message is used to pass the raw scan data for planar laser range finder. The body supports variable resolutions.

Command Service Connections

Two messages were implemented by Grant Gothing, to enable the setup and management of Command Service Connections. The justification for implementing command service connections is presented in Section 4.5.2.

- FFA1h Create Command Service Connection
- FF09h Confirm Command Service Connection

Note that a slightly different, more refined solution for command service connections (termed “latest-data connections for commands”), optimized to reduce message traffic, is presented in Section 4.6.4.

E000h Message Not Supported

This message is used to notify the sender that the recieved message is not supported. So that the original sender may know which message was not supported, the body of E000h contains the original message header.

4.7 Summary of Recommendations

The recommendations to the JAUS standard are summarized below:

- The interpretation of Wrench Efforts is ambiguous and should be clarified. Section 4.1.1, 3.5.3.
- The set Target Waypoint message needs options to clear all waypoints, append a waypoint, and replace waypoint. Section 4.1.2.
- A method to command a waypoint tolerance should be added. Section 4.1.3.
- The dynamic configuration message needs a field to report system configuration. Section 4.1.4, 4.5.4.

- Where the ordering of messages is important, the specification should explicitly state so, or specify that order should not matter. Section 4.2.1.
- A protocol should be established for switching from teleoperation to waypoint navigation; the solution would be different for systems where Global Pose Sensor and Primitive Driver services are provided by the same component. Section 4.2.2.
- The handling of and behavior effected by status control messages should be specified. Section 4.2.3.
- Where access control is required before executing command messages, it should be specified. Section 4.2.4.
- For certain applications, commands must be accepted from multiple sources; this should be supported in some way. Section 4.2.4.
- A set of guidelines should be put into place to minimize unnecessary traffic, such as periodic wrench efforts. Section 4.2.5.
- Watchdog control timeouts should not be based on periodic commands, since command messages are not always periodic. 4.2.6.
- A protocol for sharing the subsystem and node names with components should be specified. Section 4.2.7.
- The use of the system name should be clarified. Section 4.2.7.
- The order for handling/processing of incoming messages (service connections, priority messages) should be outlined. Section 4.3.1.
- The sequence number for multipart messages should indicate the number of remaining packets, not the number of packets transmitted so far. Section 4.3.3.
- The transport specification should clarify the use of multipart messages between components on the same node. Sections 4.3.4, 4.5.4.
- The rules for Ack/Nak should be clarified. Section 4.3.5.

- A distinction between desired and actual states should be established, for messages such as Query Wrench Effort. Section 4.4.1.
- The definition of command completion should be specified, to enable consistent implementation of messages for mission spooling. Section 4.4.2.
- A different diagram should be used in the RA to illustrate subsystem hierarchy and layout. Section 4.4.3.
- The 0408h Set Local Vector message should not be deprecated, as it is required for potential fields navigation strategies. Section 4.5.1, 3.6.5.
- Command service connections should not be deprecated; an alternative is proposed below. Section 4.5.2.
- A new latest-data command connection is suggested, where messages flow on shortcuts similar to a service connection, and new commands replace old (using the SC bit), but there is no requirement to send the command periodically. Section 4.6.4.
- The header should be retained on all service connection messages, to simplify the construction of message handlers and generic component templates, and to enable true latest-data connections (critical in reactive navigation). Section 4.6.3.
- The special routing of service connection messages should be clarified, explicitly stating that they flow on shortcuts, eliminating any queuing. Section 4.6.3, 4.5.4.
- To facilitate level-III interoperability, the use of transport-level header fields should be specified for passing messages between components on the same node. Section 4.5.4.
- The OpenJAUS NMI (Node Manager Interface) is a good starting point for level-III interoperability, and should be further developed. Section 4.5.4.

- Services should be grouped into separate sets, ie accepting control *by* something else, gaining control *of* something else, and not supporting control *by* something else. Section 4.6.2.
- An experimental message is suggested to pass the raw polar plot from a planar Sick laser range-finder. Section 3.10, 4.6.5.
- A message indicating “Message not supported” should be included in the core message set. Section 4.6.5.
- The question of whether a marsupial vehicle is a node or subsystem should be addressed. Section 3.8.
- A series of configuration messages for use by the Subsystem Commander, such as “Set Output JAUS Address”, “Request Control of JAUS Address”, etc, should be implemented to enable smart reconfiguration. Section 3.10.
- A few additional comments are included in Appendix C.

Chapter 5

Conclusion

This thesis has presented the results of the Virginia Tech experience with JAUS, from development of the first-ever JAUS software development toolkit for the LabVIEW programming language, through successful participation in Experimentation Task Group interoperability exercises 2.75 and 3.0, to the establishment of the first internet-based testing environment for JAUS [33].

During this journey from isolated beginnings to embedded involvement in the Experimentation Task Group, the VT JAUS Toolkit has evolved as its developers learned more of the traditional assumptions about JAUS and conducted interoperability tests at every possible opportunity. The lessons learned during this evolution were presented in a series of diverse implementation case studies and a set of recommendations to the standards development committee.

Since the summer of 2005, the goal of developing a JAUS implementation versatile enough to be installed on any unmanned system in the Virginia Tech Unmanned Systems Group has been achieved. The LabVIEW Toolkit for JAUS has been successfully deployed on over 10 unmanned systems at Virginia Tech, with more implementations in the near future.

The Toolkit's JAUS implementation has proven to be highly versatile

in achieving interoperability, evidenced by the fact that Virginia Tech has never failed to achieve interoperability with another group. When initial testing is unsuccessful, Toolkit features such as message logging and replay, and LabVIEW’s ideal prototyping environment usually enable the problem to be resolved quickly.

5.1 Key Results and Recommendations

5.1.1 Benefits of JAUS

At Virginia Tech, we found the modularity and portability achieved through JAUS to be its most valuable asset. While the message set is lacking in fullness, the routing infrastructure in JAUS for passing messages and basic communications is more mature. The ability to develop applications that can interact across networks just as easily as on the same computing node is extremely useful in orchestrating tests and demonstrations of unmanned vehicles.

From a development point of view, the convenience of modularity and portability provided by JAUS far outweighs any messaging overhead. As demonstrated in running Cadillac SRX code on Rocky (Section A.2), the potential cost-savings in industry due to code reuse prove to be significant. Integration time is also drastically decreased, enabling more research to be conducted in the same amount of time.

5.1.2 Key Issues

In implementing JAUS, no serious problems were encountered; the majority of issues were related to ambiguities in the specification, and were easily resolved by consulting developers in the group. These recommendations are summarized in Section 4.7, however two concerns of especially high importance to developers at Virginia Tech are presented below:

Latest-data Connections

Latest-data connections are critical in the operation of an unmanned vehicle, especially for sensor data and commands in the reactive paradigm. While JAUS provides service connections for query/report message pairs, command service connections are effectively no longer in use. It is strongly recommended that latest-data command connections be reinstated (with slight modifications in Section 4.6.4), as their positive impact was clearly demonstrated with the Cadillac SRX.

Level-III Interoperability

Many groups in the JAUS community discount the need for level-III interoperability, feeling that it would be imprudent to allow a implementer or third party to install software which could starve the important vehicle-control code for processor time.

While this is a valid concern, level-III interoperability is not usually implemented to support integration with other groups, as level-I and level-II are. Rather, level-III interoperability in an unmanned system is intended to facilitate code reuse and modularity within a company's software development projects.

By designing JAUS to enable level-III interoperability, the JAUS Working Group may see a surge in the development of unmanned systems software, as software becomes more modular and code re-use becomes easier. The OpenJAUS standard for intra-nodal communication provides a good starting point for defining level-III interoperability, although its management of service connections could be optimized.

5.1.3 Future of JAUS

In a survey of unmanned systems developers conducted for his master's thesis, Sean Baity recieved the following response from one developer in the

final comments section:

“...there needs to be a major paradigm shift toward a focus on software. how often have we gone ‘to see a robot’... what is there to see? its just a drive-by-wire vehicle. the software is what makes these machines unique. countless times ive heard ‘our platform...’, ‘we have a platform’, etc. software is often seen as more of an afterthought. this i believe is the underlying reason that current UGV development progress is slow.” [30]

The developer above is addressing a key issue related to unmanned vehicle development – an issue JAUS can help to address. If level-III JAUS- interoperability is enabled, the modularity and standard messaging interfaces will facilitate the coordination of large software development teams. In addition, small groups or individuals can develop niche expertise in small software elements of unmanned systems.

While this type of development of infrastructure is possible now, it requires lengthy meetings to establish a standard for the project at hand. JAUS effectively skips this step, allowing individual groups to specialize in developing Reflexive Drivers, or Primitive Drivers. This is the beginning of plug-and-play.

Finally, by facilitating the dispersion of software work in unmanned systems among small companies, the collective status of unmanned systems technology will see a marked increase in productivity.

5.1.4 Accelerating Acceptance

Two recent developments have the promise to accelerate JAUS development and positively impact the acceptance of JAUS as a standard.

InterJECT VPN System

The InterJECT VPN system was established by Virginia Tech researchers to facilitate and accelerate development and testing of new JAUS messages

and protocols [33]. By allowing developers to quickly conduct tests across the country without the time and expense of travelling to a central site, interoperability testing occurs more frequently, thus accelerating development of the specification.

In the last month alone, InterJECT has been used extensively in the validation of the mission spooling interface between Defense Technologies Inc and Virginia Tech. It is very unlikely that such progress would have been made without a testing arena such as InterJECT.

OpenJAUS

OpenJAUS is the first open-source distribution of JAUS. Initially developed by researchers at the University of Florida, OpenJAUS is now managed by Nomadio Inc [32]. OpenJAUS provides a reference implementation of the standard, developed by implementers embedded in the JAUS Working Group. With an open-source distribution available, the upfront cost for a company to become involved in JAUS is much lower.

5.2 JAUS in LabVIEW

Early in the development of the VT Toolkit, at a JAUS Working Group meeting, a developer asked me “What are you doing in LabVIEW?”. When I explained that we were developing a JAUS SDK for LabVIEW, the developer laughed, “you’re joking right?”. Like most others, he believed LabVIEW was only intended for data acquisition and measurement in the laboratory.

To the contrary, LabVIEW has evolved into a fully-functional programming language, with many features that make it ideal for a research implementation of JAUS.

5.2.1 Rapid Prototyping

With its fast learning curve and intuitive environment, even non-software engineers can develop complex applications very quickly. LabVIEW handles most memory management automatically and the details of complex structures such as arrays, freeing the user to focus on the algorithm being prototyped. In addition, The front panel / block diagram structure eliminates the need for developing a separate user-interface software module, required in most text-based languages.

5.2.2 Advanced Programming Concepts

LabVIEW simplifies the use of advanced programming concepts such as multi-threading, queueing, data-sharing, and user-interface control, nearly to the point of being trivial. Other concepts used in nearly every LabVIEW program include dataflow programming, parallel processing, and event capture routines to interact with the user. These concepts are immediately available to the novice programmer, where in a text-based language, only experts understand enough to implement multithreading, parallel processing, and dataflow programming.

It is not uncommon for a LabVIEW programmer to think nothing of dropping parallel loops onto a block diagram, all sharing data through a system of notifiers and queues, while manipulating and interacting with objects on the front panel.

5.2.3 Intuitive Debugging

Next to the advanced programming concepts, this is probably the most useful set of features about LabVIEW.

RunTime Probes

While a VI is running in LabVIEW, a user can click on any wire and see the data currently flowing inside it, where in a text-based language, the programmer would have to either print the data to a console or employ variable watching features in the compiler.

Programs Resemble Flowcharts

By the very nature of graphical programming, LabVIEW VIs tend to resemble the program flowchart describing the subroutine function. This greatly simplifies the design and debugging of code, facilitating the communication of issues to other programmers. As demonstrated by Fig 2.9, an instant grasp of program functionality can be obtained, whereas in a text-based programming language, extensive reading is required before even the basic program structure is evident.

5.3 Future Work on VT Toolkit

While the VT JAUS Toolkit for LabVIEW has evolved substantially since its inception, including a thorough restructuring of the code in December 2005, several modifications are planned. The items fit into three groups.

5.3.1 Additions to Base Functionality

Several upgrades are planned for the base functionality, extending the scope of JAUS implemented by the VT Toolkit;

Ack/Nak Retry

JAUS specifies that, if an acknowledgement is requested on a message and none is recieved, the sender may try up to three times to re-send the message. Since nobody else in the ETG implemented this side of Ack/Nak,

VT did not. To ensure the most reliable message passing, the full features of Ack/Nak could be implemented in the future, however the ambiguities about when to send an Ack or Nak would require resolution first.

OpenJAUS Protocol Optimization

While the OpenJAUS UDP protocol was implemented for the Level-III test, all messages now flow through the Node Manager, whether they are service connections or not. To implement true service connections in which messages are never queued, a different system will be implemented to handle service connection messages.

Error/Status Reporting

From the beginning, very little error-handling was implemented, since developers were primarily concerned with developing code that worked for ETG. However, with the deployment of the Toolkit on other vehicles, a more robust system of logging and recovering from errors is necessary.

Automated Service Connection Providing

Currently, if a developer implements a new message, a separate piece of code must be written to support a service connection for that message. With automated service connections, it will be necessary only to create the message VI and a message handler.

This enhancement also enables the components to become more event-driven, one of the performance modifications.

5.3.2 Performance Modifications

As mentioned earlier, the VT implementation is not optimized for performance. Due to an incomplete understanding of event-driven programming in LabVIEW early on, the Toolkit has a fair amount of processor overhead.

However, the developers have learned more about LabVIEW's features, and are now sufficiently prepared to correct this problem.

By eliminating polling and implementing an event-driven structure instead, messages are handled faster, and overall processor usage is reduced. In addition, the elimination of cycle times for polling loops will decrease message transit times substantially, with message routers waiting for messages instead of alternately checking and entering a delay state.

“Overhead” of JAUS

The concept of overhead has been mentioned several times in earlier chapters. A common concern in the implementation of any standard is the overhead incurred by passing messages in standard formats. These concerned developers often prefer a highly optimized method of data-sharing, customized for the application, often stating that the extra overhead associated with *any* standard is very detrimental.

It is important however, to understand that two factors can cause the symptoms of latency: the standard itself, and the implementation. Even in a standard with quite low overhead, an improper implementation can introduce errors. In the JAUS standard, the two potential causes of overhead are the routing of messages through the NodeManager (a potential bottleneck), and the presence of a 16-byte header on all messages. The first issue, bottleneck routing, can be addressed via service connections, as demonstrated in the Cadillac SRX. Removing the message header however, removes the plug-and-play interoperability and code-reuse benefits afforded by JAUS, and should not be done.

In the VT implementation, the overhead is usually manifested as a latency. While several potential causes are possible, the most probable is the non-event-driven programming. Since the components and Node Manager run at a fixed cycle time, message transmission requires a finite amount of

time. In the worst-case scenario, the following message delays are possible for a Node Manager and components running at 20 and 50 ms cycle times, respectively.

- Immediately after the polling Node Manager checked its queue and began waiting for 20 ms, a component sends a message, which is immediately enqueued
- 20 ms later, the message is dequeued by polling Node Manager. After a second 20 ms wait, the message is routed (enqueued for a local component)
- The component's event-driven incoming message parser immediately receives the message, parses the header, and enqueues it in the 16 priority queues. Suppose this happens immediately after the component's message handler checked the priority queues, and is now in its 50 ms wait
- 50 ms later, the component's message handler dequeues the message from the priority queues
- The component then waits another 50 ms before releasing the message into the component-specific message handler and executing the command, a cumulative 140 ms after the message was sent, assuming the routing and parsing processes take no time
- If the command is only parsed into the service state and executed on the next loop iteration (as is the case with most current VT components), an additional 50 ms delay is introduced, increasing the total delay to 190 ms

The elimination of fixed cycle times and the adoption of more event-based programming in LabVIEW will reduce the cumulative 190 ms delay above. With this in mind, the cause of "overhead" latencies in the VT implementation is less a cause of LabVIEW as a programming language, and more due to the system design. The only strong exception is the JPEG image compression, which LabVIEW currently implements using a very inefficient file-write system.

Preliminary exploration into event-driven programming in LabVIEW suggests that these latencies can be nearly eliminated, leaving only the processing time for routing, and parsing messages, which was assumed zero in the above example.

5.3.3 Modifications for Ease of Use

The last group of upgrades concerns ease of use for the implementer.

Automatic Services List Generation

Currently, the list of services used in response to D2A7h Query Services is loaded from a configuration file. While this was simple to implement, it can cause problems for interoperability when the wrong configuration file is loaded, or a message is added to the message handler.

Soon, components will automatically determine the services supported, by detecting the composition of the message handlers that make up the component.

Modules

Finally, the collection of modules in the VT Toolkit will be extended to include more automated functionalities. Common messaging tasks requiring three separate VIs will soon require only one.

5.4 Conclusion

In summary, a JAUS software development kit was developed in LabVIEW, and validated in a series of implementation case studies. The specification was found to be highly conducive to enabling modularity in design and facilitating code re-use, however contains several ambiguous points, mostly regarding message interpretation and protocol.

In addition, two key recommendations are brought forth: (1) the reinstatement of command service connections (however, slightly modified as latest-data command connections), and (2) the introduction of guidelines to enable level-III interoperability.

J AUS was found to have high potential for message routing and infrastructure, but the message set and protocol specification must be filled out before the standard will be widely implemented.

References

- [1] United States Senate, *Fiscal Year 2001 Defense Authorization Bill*, Sec 217.
- [2] JAUS Working Group. *JAUS Reference Architecture version 3.2*. Available at www.jauswg.org
- [3] JAUS Experimentation Task Group. *Interface Control Document for JAUS ETG Experiments 2.75 and 3.0*. Available at www.appliedperception.com
- [4] NI Week 2006 Program Booklet. Available: www.ni.com/niweek
- [5] Murphy, Robin M. Introduction to AI Robotics. MIT Press, Cambridge MA. 2000.
- [6] Minutes from OPC Meeting at JAUS Working Group Meeting in Atlanta, August 2006.
- [7] JAUS Tutorial presentation published by the JAUS Working Group. Available at www.jauswg.org
- [8] Gothing G, Hurdus J. *Implementation of JAUS on a Cadillac SRX Using a Potential Fields Architecture*. AUVSI 2006.
- [9] Introduction to Remote Desktop. Available at www.microsoft.com
- [10] University of Florida Grand Challenge Paper. Available at www.darpa.mil/grandchallenge
- [11] Evans, Carl. *Development of a Geospatial Data-Sharing Method for Unmanned Vehicles based on the Joint Architecture for Unmanned Systems (JAUS)*. Master's thesis, Dept of Mechanical and Aerospace Engineering, University of Florida, 2005.
- [12] Crane III, Carl; Armstrong II, David; Ahmed, Maryum; Solanki, Sanjay; MacArthur, Donald; Zadowny, Erica; Gray, Sarah; Petroff, Thomas; Grifis, Mike; Evans, Carl. *Development of an Integrated Sensor System for Obstacle Detection and Terrain Evaluation for Application to Unmanned Ground Vehicles*.
- [13] Gray, Sarah; Harrison, Joseph. Smith, Brian. *JAUS Operator Control Unit (OCU) Interoperability Experiment: Preparation and Results*.

- [14] Crane III, Carl, Armstrong II, David; Touchton, Robert; Galluzzo, Tom; Solanki, Sanjay; Lee, Jaesang; Kent, Daniel; Ahmed, Maryun; Montane, Roberto; Ridgeway, Shannon; Velat, Steve; Garcia, Greg; Griffis, Michael; Gray, Sarah; Washburn, Sarah; Routson, Gregory. *Team CIMAR's NaviGATOR: An Unmanned Ground Vehicle for Application to the 2005 DARPA Grand Challenge*.
- [15] DARPA Grand Challenge website.

`www.darpa.mil/grandchallenge`
- [16] Intelligent Ground Vehicle Competition Website. `www.igvc.org`
- [17] Virginia Tech Autonomous Vehicle Team. Johnny-5 Design Report for 2006 IGVC. Available at `www.igvc.org`
- [18] Virginia Tech Autonomous Vehicle Team. Gemini Design Report for 2006 IGVC. Available at `www.igvc.org`
- [19] Simone, Matthew. *A Novel Method to Locate Targets Using Active Vision and Robotic Inertial Navigation Data*. Masters thesis, Dept of Electrical and Computer Engineering, Virginia Tech, 2006.
- [20] Bacha, Andrew. *Line Detection and Lane Following for an Autonomous Mobile Robot*. Masters thesis, Dept of Mechanical Engineering, Virginia Tech, 2005.
- [21] Naik, Ankur. *Arc Path Collision Avoidance Algorithm for Autonomous Ground Vehicles*. Masters thesis, Dept of Mechanical Engineering, Virginia Tech, 2005.
- [22] Hastings, Ben. *Design of a Micro Wireless Instrumented Payload for Unmanned Vehicle Testing*. Masters thesis, Dept of Electrical and Computer Engineering, Virginia Tech, 2006.
- [23] Currier, Patrick. *Helium Red: a Prototype JAUS UVG Deployable from a UAV*. Presentation at Unmanned Systems Summit during NI Week 2006.
- [24] Mobius Product Page at Autonomous Solutions Website

`www.autonomoussolutions.com/products/products-main/mobius.html`
- [25] JOUSTER Project Website for JAUS-interoperable MATILDA:

`http://www.jouster.me.vt.edu/projects.php?project_id=12`
- [26] JOUSTER Project Website for High Speed Autonomous Vehicle:

`http://www.jouster.me.vt.edu/projects.php?project_id=8`

- [27] NFB Blind Driver Challenge 2006 Rules. www.bdc.nfb.org
- [28] Rules for JAUS Challenge at 2006 IGVC. Available at www.igvc.org
- [29] Draft JAUS Services Specification. Available at www.sae.org
- [30] Baity, Sean. *Development of a Next-generation Experimentation Robotic Vehicle (NERV) that Supports Intelligent and Autonomous Systems Research*. Masters thesis, Dept of Mechanical Engineering, Virginia Tech, 2005.
- [31] Huang, Hui-Min; Messina, Elena; Albus, James. *Autonomy Level Specification for Intelligent Autonomous Vehicles: Interim Progress Report*. Performance Metrics for Intelligent Systems (PerMIS) Workshop Proceedings, 2003.
- [32] OpenJAUS Website. www.openjaus.org
- [33] InterJECT project page on Joustter website

http://www.joustter.me.vt.edu/projects.php?project_id=25
- [34] Personal Communication with Dan Gehring of AMRDEC
- [35] Pedersen, Jorgen. *A Practical View and Future Look at JAUS*. RE2 white paper available at www.resquared.com.
- [36] Buzz, Woody. *JAUS Rant*. Blog available at

www.gatago.com/comp/robotics/misc/14392392.html.
- [37] Wienhold, Kathy. Transport Committee Session Presentation at JAUS Working Group Meeting, San-Diego CA, October 2005.
- [38] B. Gombard, C. Terwelp, M. Fleming, A. Wicks, C. Reinholtz, "JOUSTER: Joint Unmanned Systems Test, Experimentation, and Research Facility," *ITEA Journal*, Dec 2004/Jan 2005, pp. 9-13.
- [39] Clark, M.N. *JAUS implementation: Robots Gather for Successful Interoperability Experiment*. IEEE Robotics and Automation Magazine, March 2005. p75.
- [40] Spencer, Lloyd. *Analysis - Security: Robotics and the Need for Standards*. Robotics Trends online magazine.

www.robotictrends.com/displayarticle669.html
- [41] Faruque, Ruel; Bacha, Andrew; Reinholtz, Charles. *Implementing a JAUS Design Process for Level-II and Level-III Interoperability on Two Autonomous Robots for the Intelligent Ground Vehicle Competition*. SAE/NDIA Intelligent Vehicle Systems Symposium, June 2006.

- [42] Clark, M.N. Presentation at AUVSI 2006, Baltimore Maryland.
- [43] JAUS Overview on Drexel University Website

http://www.pages.drexel.edu/~jw73/jaus_overview.htm
- [44] Torrie, M.W.; Cripps, D.L.; Swensen, J.P. *Joint Architecture for Unmanned Ground Systems (JAUGS) Applied to Autonomous Agricultural Vehicles*. Automaton Technology for Off-Road Equipment Proceedings, 2002.

Appendix A

Interoperability Examples

This Appendix presents a collection interoperability examples illustrating the capabilities of JAUS. It is implementation successes like these that keep me excited about developing and implementing JAUS.

A.1 Rapid Integration: ARA at IVSS

At the 2006 Intelligent Vehicle Systems Symposium (IVSS) in Traverse City Michigan, Applied Research Associates (ARA) had a teleoperated front-end-loader on display. The night before I was to deliver two presentations, I wandered through the exhibit area looking for some free food, and noticed ARA was letting attendees try their hand at teleoperating the front-end-loader using the ARA Operator Control Unit.

JAUS-interoperable?

Curious, I approached one of the engineers, and asked “Does it run JAUS?”. The engineer, John-Mike, replied, “Yes, it’s JAUS-compliant”. Intrigued, I offered to get my laptop, which had a JAUS-interoperable Operator Control Unit (OCU) on it, and try to interoperate with the front-end-loader. All of this occurred slightly after 7:30pm.

Integration

I returned from my hotel room at about 8:00pm. As the engineer, the manager, and I chatted about JAUS, we got my laptop properly connected to the ARA network. Once I was able to successfully ping the front-end-loader, I began capturing packets. It took a few minutes for me to realize that ARA was not using the standard UDP preamble employed by other implementers in the JAUS Working Group.

Front-End-Loader Moves

By 9:00pm I had successfully commanded a Wrench Effort to the Front-End-Loader from my Virginia Tech OCU, and the Front-End-Loader moved.

However, since ARA had implemented several non-standard JAUS messages, I was unable to disable the Emergency brake and enable the drive-by-wire systems from the Virginia Tech OCU.

User-Defined Messages

Over the next few hours, John-Mike and I chatted about all manner of topics, and continued testing. Since John-Mike did not have the source code or the documentation for the user-defined JAUS messages his OCU was sending, I used the message logging feature in the VT JAUS Toolkit to capture the messages. To do this, I started up a simulated vehicle on my laptop, and John-Mike directed the ARA OCU toward my simulated vehicle.

Once I had captured the user-defined messages, I set up the Virginia Tech OCU to send them to the Front-End-Loader. As we did not know what each message accomplished, we proceeded by making educated guesses based on the captured messages combined with empirical testing.

Final Test

At 10:45pm, John-Mike “turned off” the ARA OCU – he actually kept it on so we could send an Emergency Stop just in case, but no other messages were sent from the ARA OCU to the Front-End-Loader. Then, only by sending messages from Virginia Tech OCU, I:

- turned the engine on
- enabled the drive-by-wire systems
- removed the parking brake
- drove the vehicle around
- E-stopped the vehicle
- turned the vehicle off

Time to JAUS interoperability was 1 hour. Full interoperability with ARA’s user-defined messages took 2.75 hours.

A.2 Code Reuse: SRX and Rocky

In early 2006, I guided an undergraduate team of mechanical engineering seniors in implementing JAUS on a Cadillac SRX. We implemented a vehicle-specific mobility interface, a vehicle-specific sensor interface, a speed and heading control algorithm, and an algorithm enabling the SRX to autonomously navigate a course of GPS waypoints. All software modules had JAUS messaging interfaces.

Later that summer, I lead a team that converted Virginia Tech’s previous Grand Challenge vehicle, Rocky, to JAUS-interoperability. We implemented a vehicle-specific mobility interface, a vehicle-specific sensor interface, both with JAUS messaging interfaces. Rocky was now JAUS-compliant, but only

as a remote-control car, with none of his autonomous features from the Grand Challenge.

Directly Porting Autonomous Code

To test the portability of the previously developed code, I copied the speed and heading control algorithm and the waypoint navigation algorithm from the Cadillac SRX, and installed them on Rocky. I then changed one setting – the JAUS subsystem number for the vehicle – in six places.

We then loaded a set of GPS waypoints into the waypoint navigation algorithm, and Rocky successfully navigated the GPS waypoint course, as shown in Fig 3.18. Total time to interoperability was 5 minutes.

A.3 Cross-Language, Cross-Organization, Level-III

Early in 2006, I proposed to Danny Kent (a lead JAUS developer from the University of Florida), that we attempt a level-III interoperability test. My idea involved running a Virginia Tech JAUS component on a University of Florida (UF) JAUS Node, with a UF Node Manager and Components.

It was not until the release of OpenJAUS that I investigated the UF method for NodeManager-Component interaction. Making use of the loop-back device in a network card, each Component is assigned a UDP port to pass JAUS messages on. An additional set of non-JAUS messages, on a different set of ports, is used for Check-In and verification of the JAUS address when a Component signs on.

Step 1: UF Node Manager in Windows

Since we had never compiled the Virginia Tech (VT) code for Linux, we decided the first test would run a UF Node Manager and a VT Component on a Windows computer. Danny Kent was able to compile the UF Node Manager, written in Java, for Windows very quickly.

Step 2: Implement OpenJAUS Node Manager Interface protocol in VT Components

Implementing and integrating the OpenJAUS Node Manager Interface (NMI) with an existing Virginia Tech component took less than 1 hour.

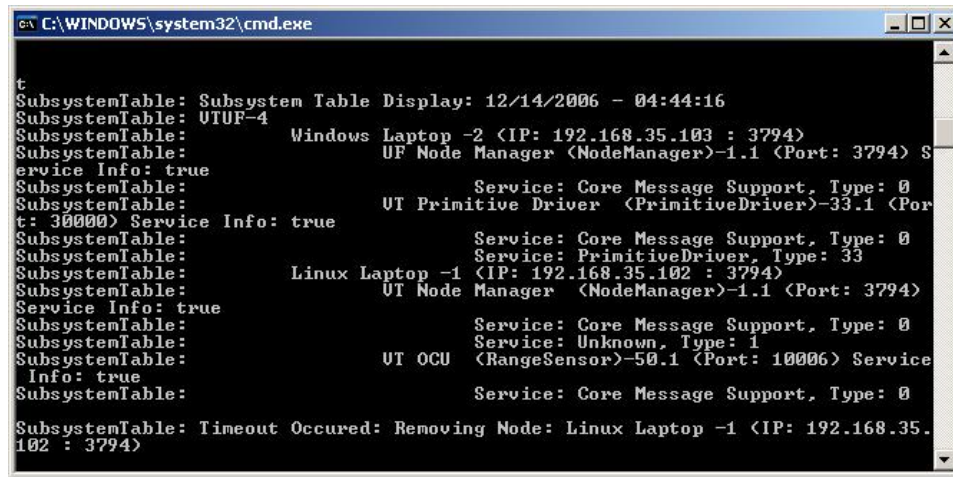
Next, I integrated the OpenJAUS NMI into the Virginia Tech Component Model and compiled an Operator Control Unit (OCU) component. Ninety minutes after starting integration, I used the compiled OCU component to teleoperate a Primitive Driver running in LabVIEW, interfaced with the vehicle kinematics simulator. All message traffic for this test passed through the University of Florida Node Manager.

Step 3: VT Components in Linux

Working towards running a Virginia Tech component or Node Manager on a University of Florida Linux node, I compiled the Virginia Tech components for Linux. This worked on the first try, with only one minor change to the Windows source code – commenting out a speech synthesis module that relies on the Microsoft text-to-speech module.

Validation and Future Work

Fig A.1 depicts the subsystem configuration as displayed by the University of Florida Node Manager during one of the tests.



```
C:\WINDOWS\system32\cmd.exe
t
SubsystemTable: Subsystem Table Display: 12/14/2006 - 04:44:16
SubsystemTable: UTUF-4
SubsystemTable: Windows Laptop -2 <IP: 192.168.35.103 : 3794>
SubsystemTable: UF Node Manager <NodeManager>-1.1 <Port: 3794> S
ervice Info: true
SubsystemTable: Service: Core Message Support, Type: 0
SubsystemTable: UT Primitive Driver <PrimitiveDriver>-33.1 <Por
t: 30000> Service Info: true
SubsystemTable: Service: Core Message Support, Type: 0
SubsystemTable: Service: PrimitiveDriver, Type: 33
SubsystemTable: Linux Laptop -1 <IP: 192.168.35.102 : 3794>
SubsystemTable: UT Node Manager <NodeManager>-1.1 <Port: 3794>
Service Info: true
SubsystemTable: Service: Core Message Support, Type: 0
SubsystemTable: Service: Unknown, Type: 1
SubsystemTable: UT OCU <RangeSensor>-50.1 <Port: 10006> Service
Info: true
SubsystemTable: Service: Core Message Support, Type: 0
SubsystemTable: Timeout Occured: Removing Node: Linux Laptop -1 <IP: 192.168.35.
102 : 3794>
```

Figure A.1: Subsystem table as displayed by University of Florida Node Manager

In conjunction with the submission of this thesis, I am working with Danny Kent to run VT components and UF node manager on a Linux computer, and hopefully will have achieved level-III interoperability on Linux by the end of 2006.

Conclusion

With this experiment, we proved conclusively that Level-III interoperability is possible through JAUS. Virginia Tech components written in LabVIEW communicated with University of Florida components written in Java. The OpenJAUS NMI provided the ground rules for communication between Node Manager and components, greatly accelerating the process.

A.4 Hypothetical Scenario

All of the previous examples illustrate the current capabilities of JAUS. If the the appropriate steps are taken in the continued development of the standard, the following scenario may take place in 10 years on a battlefield.

2016, on a Battlefield

A convoy of soldiers and unmanned vehicles is travelling across the desert. From prison, a terrorist leader somehow communicates to his loyal insurgents that this convoy of enemies must be destroyed. The terrorist army successfully bombs the convoy, destroying nearly all the unmanned vehicles and killing all but 3 soldiers, who are severely wounded, and unable to walk. One of the unmanned vehicles is still in operation, but the bomb has destroyed its primary sensor (a Laser Range Finder payload). In its current state, the unmanned vehicle is incapable of obstacle avoidance.

One of the soldiers drags his body across the sand, recovering a Vision Processing payload that was part of another unmanned vehicle. He manages to mount it to the unmanned vehicle with the missing Laser Range Finder. All three soldiers climb aboard the unmanned vehicle, and one of them commands the vehicle to take them home, using his sleeve-mounted computer.

The three soldiers are transported safely back to base, riding aboard an unmanned vehicle that is only able to operate safely because it successfully integrated with and managed to make use of a payload scavenged from another damaged vehicle.

Appendix B

VT JAUS History

Following is a summary of key milestones in the history of JAUS development at Virginia Tech.

Fall 2004:

- JAUS OPC 2.0 Experiment takes place at JOUSTER
- Brett Leedy writes header parser and basic OCU

January 2005

- Ruel returns to grad school after one year at AREVA
- begins work on MATILDA

February 2005

- Ruel assigned to JAUS project
- responsible for multipart message reassembly and ack/nak

June 2005

- Ruel assumes leadership over JAUS Project
- development of multipart messaging and new header parser code
- continue learning about JAUS by reading RA

July 2005

- attend JAUS WG meeting in Chicago
- begin design for component structure
- begin thinking about design for Primitive Driver

August 2005

- Joustler leadership mandates participation in OPC Experiment 2.75
- begin Communicator
- begin Node Manager

Sept 8, 2005

- MATILDA drives using original Leedy OCU and new Primitive Driver
- also used barebones Node Mgr and Communicator
- no dynamic configuration yet

Sept 2005

- begin integrating Primitive Driver with simulator
- 1st Component Model release

October 2005

- attend JAUS WG meeting in San Diego
- receive/parse first non-VT heartbeat (from Applied Perception)
- small mistakes in header parser
- no further interoperability possible since didn't have dynamic config msgs in OPC 2.5 ICD
- Grant Gothing joins project, Chris Terwelp joins project
- begin preparing for JAUS OPC Experiment 2.75
- begin implementing:
 - dynamic configuration messages
 - event notification messages [Grant Gothing]
 - mission spooling messages [Steve Cacciola]
 - world model messages [Cheryl Bauman, Chris Sharkey]

November 2005

- lab suffers fire, damages MATILDA 2 weeks before OPC 2.75
- implement JAUS on Johnny-5 robot instead
- participate in OPC Experiment 2.75 with Johnny-5
- OCU teleoperated the NSWC Talon
- waypoint driving via Harris OCU
- semi-compliant with Mobius
- dynamic config with nearly all groups
- timing issues affected interoperability
- routing software rewritten almost entirely while at OPC

December 2005

- begin major revision of codebase
- develop compact Component Model
- implement standard VI connector blocks + colors
- rewrite Communicator, Node Manager
- rewrite core message support

January 2006

- transition to LabVIEW 8.0
- rework integration with simulator
- integrate visual sensor with OpenGL world
- result: virtual visual sensor
- attend JAUS WG Atlanta
- develop Text Messenger over JAUS
- begin preparation for OPC Experiment 2.875 (3.0)

February 2006

- begin implementing JAUS on Cadillac SRX
- develop LRF sensor component

March 2006

- develop JAUS LRF Payload (runs intruder-detector software) [Chris Terwelp]
- develop World Model VKS for use at ETG Experiment 2.875 [Grant Gothing]

April 2006

- participate in JAUS Experiment 3.0
- compliance with Mobius
- dynamic configuration interoperability with everybody
- intruder-detector payload achieve level-II interoperability with UFL
- create objects in UFL world model
- participate in backup scenario at end of week

May 2006

- implement JAUS on Gemini for the IGVC
- begin implementing JAUS on Rocky and Cliff

June 2006

- win the IGVC JAUS Award
- attend intelligent Vehicle Systems Symposium
- present JAUS on two JAUS-enabled IGVC Robots
- present JAUS on Cadillac SRX
- at IVSS, teleoperate ARA's CAT Front-end-loader with VT OCU
- time to vehicle move: 1 hour
- after 2 more hours implmenting some custom non-RA messages
- turn ARA OCU Off
- with VT OCU, start engine, drive around, stop, emergency stop, etc
- through Internet, teleoperate MATILDA in Blacksburg Virginia from Traverse City Michigan
- implement JAUS-interoperability on Helium Red UGV (deployable from RMAX UAV)

July 2006

- InterJECT VPN system goes online
- teleoperation interoperability with Applied Perception OCU over InterJECT VPN
- Autonomous Solutions asks: "Can you leave your virtual vehicle online for us to test against?"
- interoperability with Autonomous Solutions Mobius
- recieve phone message "Hey this is Sarah at ASI, we're teleoperating your simulator right now".

- i was not even in Blacksburg when this happened
- JOUSTER field day with 5 JAUS subsystems tracked simultaneously by VT Fancy OCU

August 2006

- NI Week 2006: present JAUS toolkit in Controls Summit
- JAUS-interoperable MATILDA driven around for entire NI Week conference
- coordinated JAUS session at NI Week 2006
- AUVSI, presented JAUS-interoperable Cadillac SRX using Potential Fields [Grant Gothing, Jesse Hurdus]
- at AUVSI, achieved dynamic configuration, control with Harris' iRobot Packbot, but no movement
- OCU achieves partial interoperability with UF virtual vehicle

Sept 2006

- monitor experiment/demo at VIR for JOUSTER program review
- JAUS-enabled uWIPs on each vehicle
- JAUS-enabled MATILDA commanded waypoints
- Aerial Display autodetects and tracks uWIPs and MATILDA via JAUS
- ported vectorDriver and waypointDriver code directly from SRX to Rocky
 - Rocky successfully drives waypoints
 - time to add autonomy: 5 minutes

October 2006

- waypoint driving interoperability with DefenseTechnologies MURC OCU
- two permanent simulated robots installed on InterJECT VPN
- attend JAUS WG meeting in Jacksonville
- present Virginia Tech JAUS experience to JAUS WG
- interoperability with JCTS (JAUS Compliance Tool Suite)
- interoperability with kairosAutonomi
- teleoperation interoperability with SPAWAR San-Diego MOCU

November 2006

- begin implementing JAUS on GPS Differential Base Station [Mike Webster]
- mission spooling interoperability with Defense Technologies MURC OCU [Steve Cacciola]
- JAUS-interoperable MATILDA used in UGV characterization tests

December 2006

- Level-III interoperability achieved with University of Florida Node Manager running in Windows
- VT components compiled to run on Linux (Fedora core 6)

Appendix C

Other Recommendations

In this appendix, I list a few interoperability issues which I forgot to include in the main body of the thesis document. As such, they were not reviewed by the committee.

- **Position RMS:** In JAUS, there is only one uncertainty field currently reported for position accuracy: the position RMS. This error is a representation of the cumulative uncertainty in latitude, longitude, and altitude. In most ground-based unmanned systems, the accuracy of the latitude and longitude are important, but a less-accurate estimate of altitude is required. It may be useful to report the standard deviations of latitude and longitude separately, decoupled from the altitude error, which is normally much greater than for latitude and longitude.
- **Priority Level:** JAUS currently has sixteen levels of priority, in my opinion an excessive amount. When I asked developers embedded in the Working Group how they handled priority, I learned that most implementers ignore priority. I would recommend that a single bit be used for priority, representing “normal” or “emergency” priority. Using only two levels of priority simplifies implementations. With such a solution, an incoming header-parser could simply enqueue normal messages at the back of the message handling queue to be processed in the order recieved, and emergency-level messages at the front of the queue for immediate processing. I recommend that the Priority field be reduced from 4 bytes to 1, and that the extra three bytes be declared Reserved, for future use.

Appendix D

Component Model Screenshots

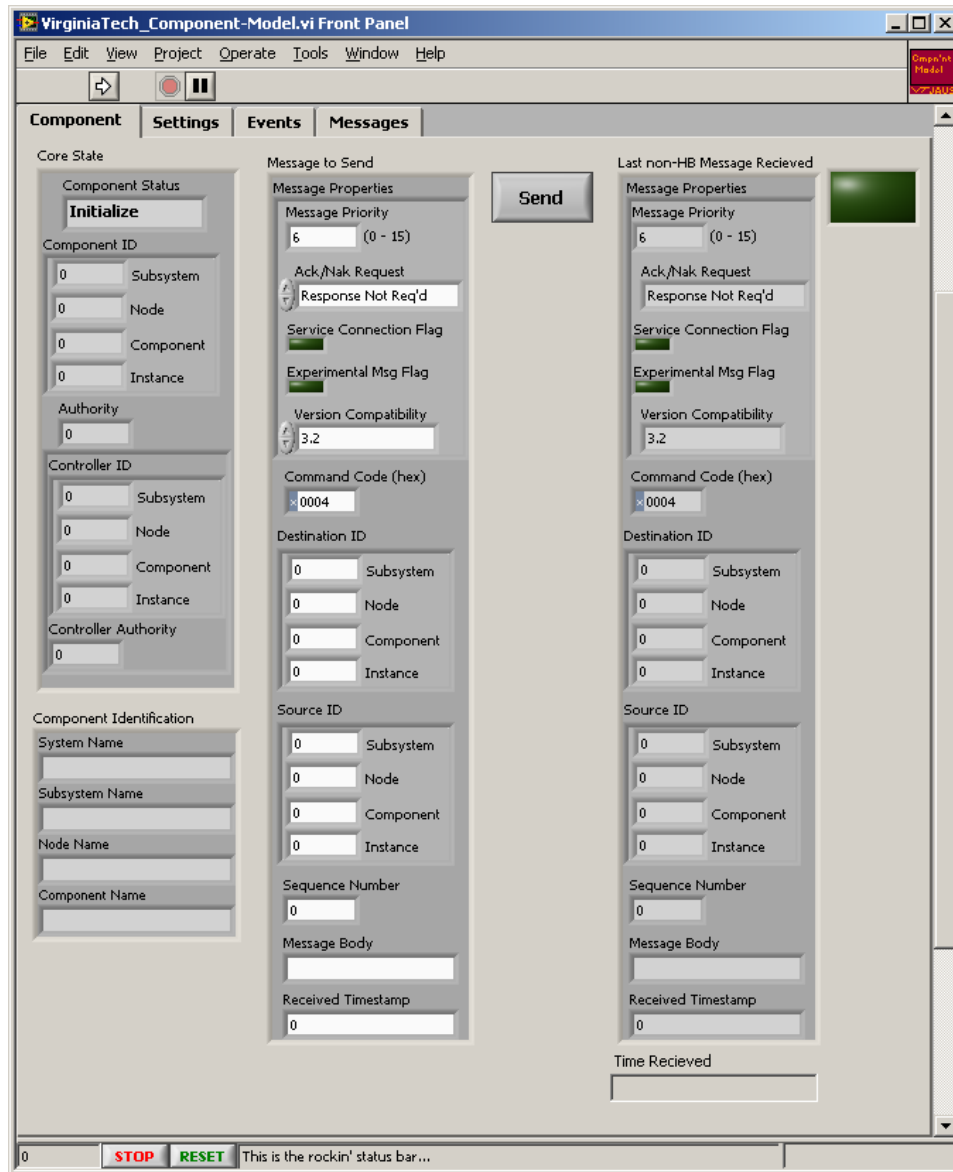


Figure D.1: Front Page of Component Model 1.2.1

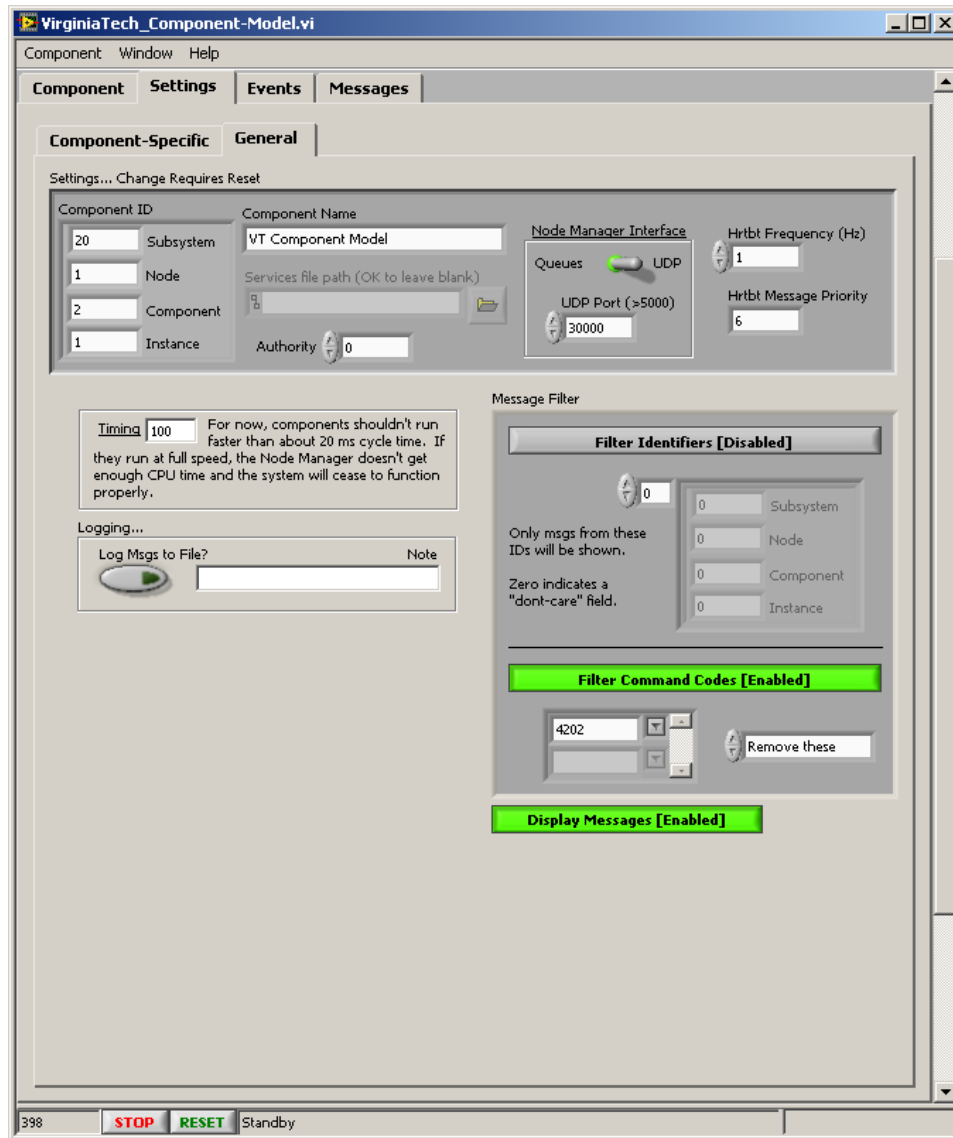


Figure D.2: Settings Tab of Component Model 1.2.1

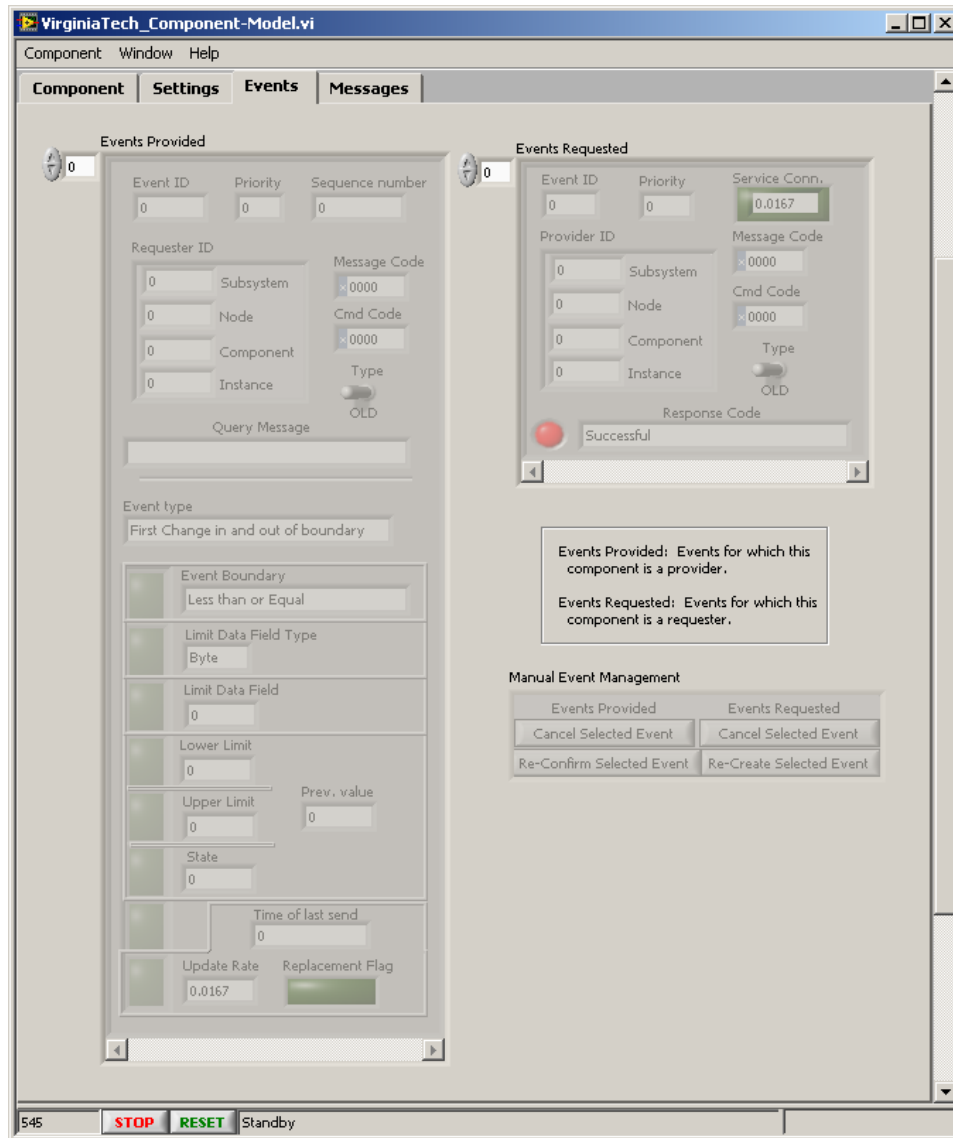


Figure D.3: Events Tab of Component Model 1.2.1

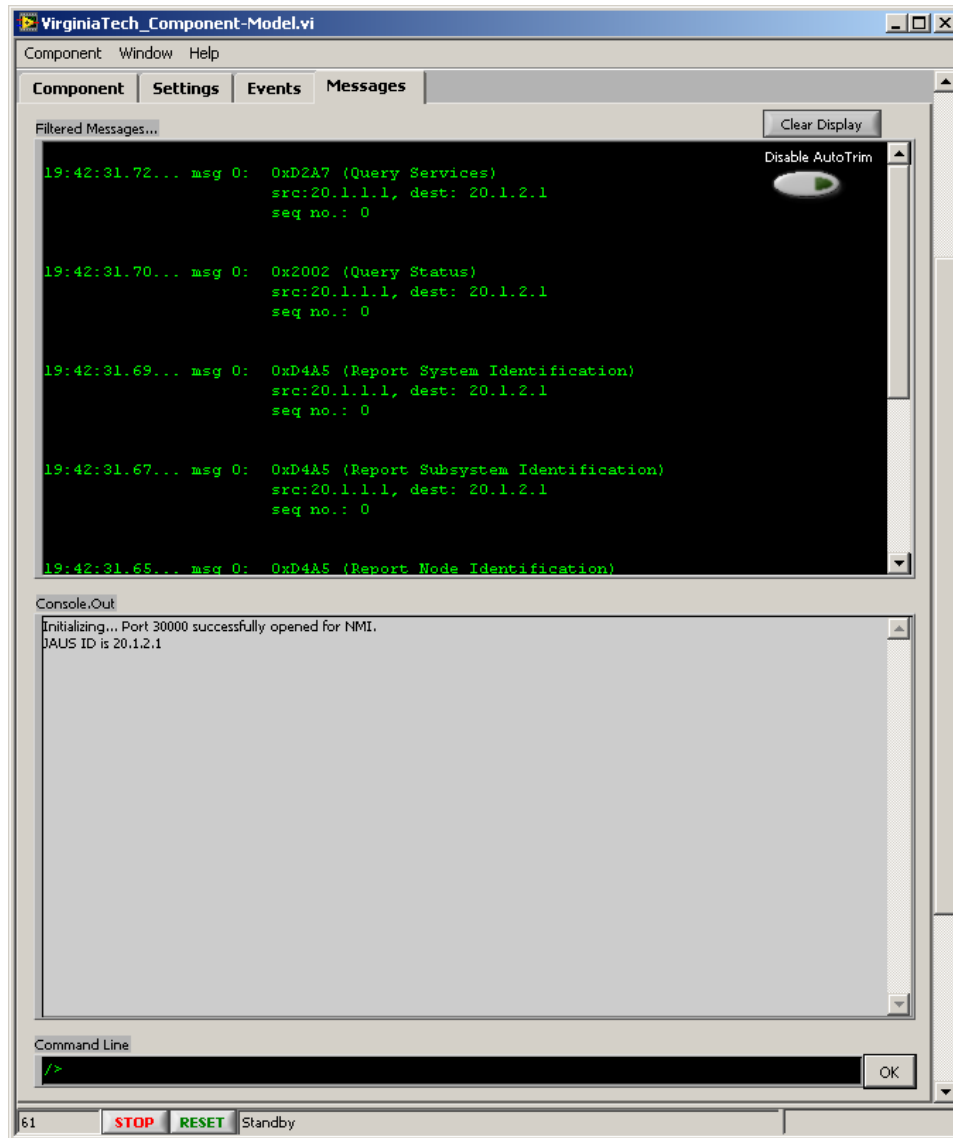


Figure D.4: Messages Tab of Component Model 1.2.1

Vita

Ruél Faruque is the son of Loren and Azad Faruque. He switched between homeschooling and Montessori school, then graduated Valedictorian in three years from United Christian Academy. In 2003, Ruél completed his B.S. in Mechanical Engineering at Virginia Tech, then took employment at AREVA Framatome ANP, developing robotic tooling to perform safety inspections inside nuclear reactors. In 2005 he returned to graduate school at Virginia Tech, and completed his masters in December 2006.

Ruél now works at TORC Technologies LLC (www.torcotech.com) in Blacksburg, Virginia. He serves on the AS-4C Information Modeling Subcommittee, the AS-4B Network Environment Subcommittee, and the Experimentation Task Group of the SAE AS-4 Unmanned Systems Committee, which is responsible for the development of JAUS.

Ruél did not obtain his driver's license until the age of 24. It is interesting to note that he wrote autonomous vehicle control software before driving a car himself.

