Development of a Universal Robotics API for Increased Classroom Collaboration within Robotics Education

A thesis

submitted by

Robert S. Linsalata

In partial fulfillment of the requirements
for the degree of

Master of Science
in
Mechanical Engineering

TUFTS UNIVERSITY

May 2012

Principal Advisor: Chris Rogers, Ph.D.

Committee Member: Ethan Danahy, Ph.D.

External Committee Member: Tully Foote

Abstract

An increasing number of low-cost, simple robotics systems are being targeted to the educational market as learning tools for a wide range of subjects, especially within the Science, Technology, Engineering, and Mathematics (STEM) disciplines. The diversity and inherent coupling of software and hardware in robotics leads to integrated development environments where programs and software are restricted to the original robot hardware upon which they are developed. The inability to collaborate and share robotics-based curriculum because of hardware-dependent code, is holding back the full potential of the technology as an educational resource. This thesis presents a Universal Robotics API (URAPI), a set of processor-independent robotics instructions, which enables teachers and students to write robotics code capable of running on any hardware. This enables educators to share and access curriculum and content, no matter the resources available in their classroom. The design and concepts of the URAPI system will be presented, along with initial explorations via a Universal Data API for data logging and experimentation. An implementation of URAPI in LabVIEW will then be demonstrated with a collection of commonly used educational robots. The URAPI system enables teachers to write universal robotics programs that allow educational robotics to break free of current constraints through cross-platform collaboration and sharing of curriculum and knowledge.

Acknowledgements

There are so many people who helped me, or pushed me, or kicked me, through these last few months, it would take as many pages as there are in this thesis to express my gratitude for their patience, kindness and motivation. For now, Thank You: My Mom & Dad; my brother, Dave; Jess; my advisor, Chris, and my Committee, Ethan and Tully; Caitlin; Magee; Jess; Matt; Eric; Mike; Mary; Rafi; Greg; Brian; and all the CEEO Grad Students and Jackson; and all the CEEO. Also, I would like to add a special note to all those who ever worked in the 'back room' and helped build the knowledge necessary to complete this thesis.

Table of Contents

1. Introduction

1.A. Motivation

Robots are increasingly finding a place in classrooms as effective aids and contexts for teaching a number of disciplines and topics, all the way from early elementary through college [1]. The explicit link between the software content and hardware application provides an effective platform for teaching and learning concepts in a variety of disciplines in an interactive and tangible manner. The "hands-on approach" and ability to engage students while giving the curriculum context and relevancy, makes robotics a uniquely optimal tool for teaching [2]. The wide range of application areas and the variety of robotic systems available mean that beneficial scenarios span the entire spectrum of educational subjects. Robotics applications have grown from the traditional core disciplines of Electrical Engineering, Mechanical Engineering, and Computer Science, to more innovative areas such as Biology [3] and human-social and human-robot interaction studies [4]. Robots have even been shown to be effective in aiding children with social disabilities and early cognitive development [5].

Within the educational market, there are many different robotic platforms currently being sold for classrooms. These robots all have various strengths that make them better suited for some learning situations than others. Some of the most popular robotic systems try to provide a balance in capability of hardware and flexibility of building and programming. Other systems target narrower use cases and either provide cheaper solutions in single, inflexible configurations, such as the iRobot Create [6], or offer modular kits, such as the Bioloid [7], that come with powerful specifications and large price tags [2]. Robots such as the Finch [8] provide students just starting out in robotics with a ready-to-go robot so they can concentrate on learning basic programming; graduating to platforms like the LEGO MINDSTORMS robotics toolset [9] allows the students to explore building and innovative solutions with their creativity leading to many different and unique robots [10].

The power of robotics in education is often driven by the explicit interconnection between the logical programming world and the 'real-world,' interactive physical instantiation; however, this tight connection, coupled with the diversity of applications and hardware, also leads to problems because the actual robot tools available are primarily provided as custom, integrated packages. The benefit of the packaged tools is that the link between the program and physical hardware components is explicit, and optimal control and functionality can be provided through customized features and functions. The primary disadvantage with this approach is that all the robotics software and content developed using the system is now tied to the specific robot hardware and package with which it is developed. Knowledge and content cannot be freely transferred without requiring rewrites of the programming or forcing collaborators to own the exact same system.

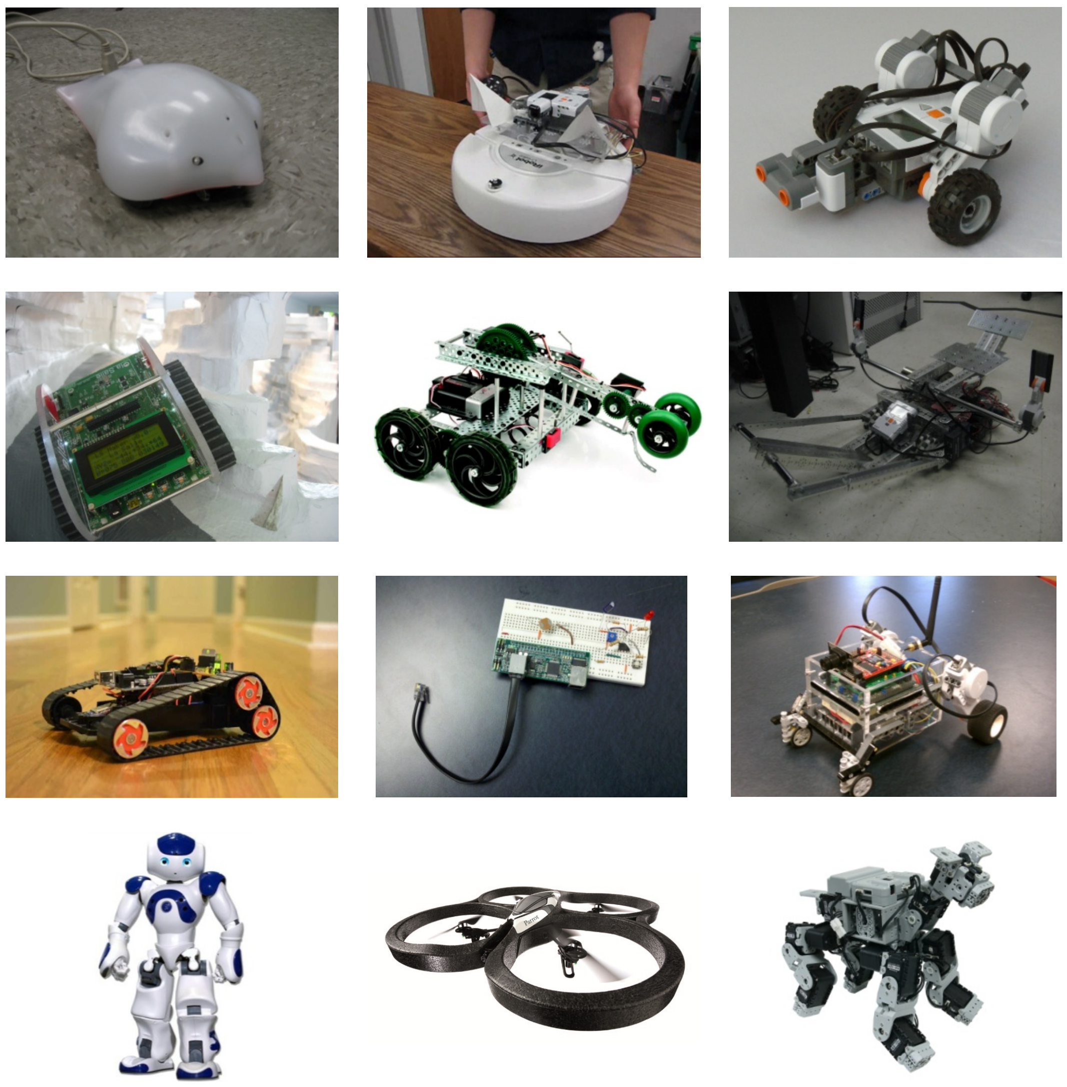


Figure 1.1: A sample of the diverse educational robot market. Top row, left to right: Finch, iRobot Create, LEGO NXT. Second row: LaSalle LSMaker, VEX Protobot, TETRIX robot. Third row: Arduino based Bluetooth-bot, HiTechnic SuperPro, Surveyor SRV-1 based robot. Fourth row: Aldebaran Nao, Parrot AR Drone, Robotis Bioloid.

The incompatibility between different robotic systems is the downside of the diversity that makes robots so useful. If a teacher wants to use a certain robot in a lesson plan, he must use the software environment that came with the robot, learn the programming language, and either find suitable curriculum developed for this specific robotic system or understand the language and robot well enough to convert the activities themselves to adapt for the specific hardware. This situation forces teachers into choosing "packages" of robot solutions that may use hardware appropriate for their students or lesson plans, but lack the software or curriculum base capable of making it an effective learning tool.

If there were a tool that allowed easy sharing of code across platforms, there would be more support for teachers developing curriculum using robots, and allow greater numbers of students to benefit from exposure to robotics. Freeing the choice of robot and hardware capabilities from the provided software will allow teachers to mix and match software and different robot models for a more diverse range of applications better suited to the needs of an individual learning environment. These needs can range from budgetary constraints to the training and skill knowledge of the teachers and students. Teachers wishing to integrate robotics into their lesson plan will be able to search for curriculum and activities based on the concepts they wish to teach – rather than the technology hardware they are able to afford. Being able to interchange robot hardware while keeping the basic skills learned with earlier robots means that the different strengths of robotic systems can be leveraged without the added cost of having to learn new syntax for coding the same programming concepts they already know.

A system for sharing code across robotic platforms would also enable smooth continuums for students as they grow in experience and skill. Beginning students will be able to master the foundational concepts of programming and robotics on cheaper, rudimentary platforms; then they will be able to 'graduate' to more complex, open-ended robots that allow them to explore advanced concepts and creativity in building and programming their own inventions. In this transition, not only will they be able to keep the basic robotics skills learned on the simple robot, they will even be able to keep their previous code and work and apply it to the new robot, to enable smoother and quicker transitions and training time.

Similarly, teachers will be able to design courses where special units or extensions can take advantage of additional niche robotic systems, while basing the core of the course on top of a more versatile platform. For instance, college robotics teachers may wish to teach introductory mechatronic concepts of building and programming interactive physical systems on a flexible platform such as the NXT, and then spend a couple weeks later in the semester teaching Image Processing on the SRV-1 [11], with its integrated camera and imaging capabilities. Being able to use the same software and programming language for both robots enables this style of course by minimizing the transitioning time and knowledge needed to be learned about the differences.

The loss of optimization that is inevitable in a system designed to universally include all robots may be a problem for some advanced classrooms; however, in the educational context, this generalization is much more likely to benefit the vast majority of educators. Ultimately, the teachers in these scenarios are less concerned about the optimal performance of the robot or even teaching the specific language of that robot. The teachers care how well the robotic system contributes to their teaching goals, and in this way, the fundamental actions and behaviors that will be common to all robots are the most important to the "power" of the robot as an educational tool in their classroom. Despite their crucial standing at the center of educational activities, these 'universal robotic behaviors' are also the most incompatible in current robotics software implementations.

1.B. Existing Technology

There are a number of solutions that attempt to solve the problem of hardware-dependent code and enabling the sharing of software content across multiple platforms. The most common approaches involve transferring code between different architectures and providing development environments, or common interfaces, for multiple robotic platforms. Many of the systems are capable of solving some of the issues facing educational and non-professional robotics users; however, most of the solutions still focus on the robotics specialist or are targeted at use cases that are too specific or outside the realm of the K-12 educational range.

1.B.i. Universal Sensor Data

In addition to robotic programming technologies, universal systems exist that focus on smart sensors and universal data communication. Cross-platform robot programming could be based on the ability to categorize the inputs and outputs of robots through a universal data system. The Institute of Electrical and Electronics Engineers (IEEE) defines the "IEEE Standard for a Smart Transducer Interface for Sensors and Actuators Wireless Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats" (IEEE 1451) [12]. The IEEE 1451 standard is composed of multiple layers that address a significant vertical range of sensor applications and aspects. These layers range from wireless network standards to the format of specification data stored on a sensor. The standard was designed to capture the full range of uses and variability within the professional sensor industry and application market. The details of the specification and the protocol are comprehensive, but also extremely complex [13]. For the educational market, the IEEE 1451 standard is too extensive to be implemented by the costrestrained educational product companies and too complicated for use in simple classroom scenarios. However, the format and architecture of the IEEE 1451 Transducer Electronic Data Sheets formed the basis for the universal data descriptions explored later in this thesis.

1.B.ii. Source Code Porting and Transfer

Current capabilities to transfer code between platforms fall mostly within two realms. One approach is to use a single shared language and function set for the entire scope of the code. The other approach provides standard libraries of functions for high-level functionality. Some software development environments within the second approach also provide consistent development environments and the same base language, but different libraries and 'perspectives' (customized user interfaces) for programming different platforms.

1.B.iii. GCC & Non-Robotic Computer Operations

The first approach to transferring code uses a shared programming language and rewriting the processor dependent implementation code. While common embedded languages such as C and multi-target compilers like GCC [14] aid this process, unique methods of controlling hardware peripherals, which are inherent to robotics code, still require a large portion of the user's code to be rewritten. The task of rewriting implementation code for each peripheral can easily venture outside the focus and ability of K-12 classrooms.

This approach breaks down all the functions in the source code into low-level, hardware-independent representations. These low-level operations can be subsequently translated into corresponding machine code for any fully described target interface. This approach is good at porting the computational code for many different architectures, as seen by the widespread use of single compilers like GCC. The idea of a universal middle layer that is inherently software and hardware independent is shown to be effective in projects such as LLVM, which can be found in many mainstream and commercial products [15], [16], [17]. These middle-ware 'substrates' have even driven the creation of effective virtual machines [18].

The major problem with these solutions is that while they have been proven to provide an effective means of creating portable processor code across architectures, the low-level breakdown causes significant loss of the meaning and purpose of higher-level functions. In robotics programming, abstraction needs to be done at the level of the robotic actions and physical hardware component control. Since every processor can implement the same control in different processor functions, breaking down these robotic action functions into processor commands does not guarantee "correct" representation of the code. Thus, the thing that is lacking is abstraction of the robotic actions, because robots do the same things differently.

There is a strong parallel between the attempts to create hardware-independent robotics code and the history of programming languages designed to compile on any computer. The distinguishing factor that further complicates the task for robots is the perception and interaction with an indeterminate physical world.

1.B.iv. Arduino & Wiring

Integrated environments, such as Wiring [19] and Arduino [20], maintain the meaning of higher-level robotic commands through libraries for different physical inputs and outputs, and functions that provide abstraction of basic actions rather than just processor operations. The large community backing, spanning disciplines from traditional engineering prototypes to art and design courses, drives ongoing development of easy-to-use examples and libraries for an extensive range of compatible peripherals [21]. The key to the wide-spread applications and collaboration in these environments are the standard microprocessor boards and layouts that are always used with the environment.

Early editions of the development systems used a single board and pin layout, but quickly expanded to include variations on these main boards [22]. The compatibility of the user's code and the libraries is mostly achieved through preprocessor definitions for standard pin naming conventions. For instance, there are a set of numbered Digital Input and Output Pins on every Arduino board variant that are labeled with the same names next to the physical pins as the variables used in software. A large board like the Arduino Mega has 54 Digital Inputs/Outputs, while the Arduino Uno only has 14. As Figure 1.2 illustrates, Users are still able to compile and run code with functions such as digitalRead(pin 12), because the header files for the targeted board define the board specific address corresponding to the physical pin 12.

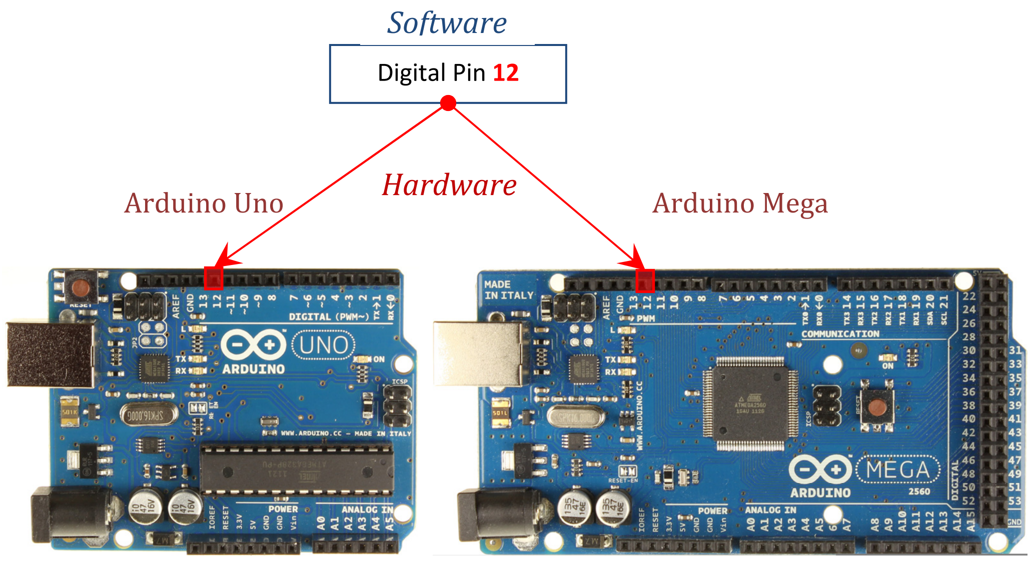


Figure 1.2: The same software address "Digital Pin 12" can map to different physical ports on the Arduino Uno and the Arduino Mega because of the common labeling.

While the tremendous growth of the Arduino market in recent years speaks to the success of the approach, there are still limitations in regards to a universal educational solution. The most substantial limitation is that all the boards are essentially the same. The Arduino programming environment is merely a wrapper of preprocessor definitions and procedures that transform the simple programming environment and language of the user's code into C/C++ and then pass normal C/C++ source code to a compiler. Because all the Arduino boards use preprocessor definitions for C/C++, they still rely on the C/C++ compiler to generate the executable code. This means that all the targets must be relatively similar in how they implement various actions since the compiler translates the low-level processor operations to the individual boards, not the original high-level behaviors.

The many differences in chip architectures mean solutions such as the Arduino/Wiring systems are necessarily targeted at similar microprocessors within the same chip family. The low-level incompatibility problems range from the manner in which the architectures define various functions, to simple differences in how the chips behave during certain actions. The Wiring developers are starting to examine the possibility of integrating new architectures and chip brands; however, the task of doing so effectively will not be straightforward [23]. When trying to overcome these architecture incompatibilities, increasingly complex pre-compiling passes and special function casing will need to take place in order to be able to use the same code with different chips. The alternative will be recreating the entire system for the new architectures, which does not lend itself as a simple approach for manufacturers wishing to integrate their new systems.

1.B.v. Firmata

Firmata is a generic protocol for microcontroller communication, designed to allow any computer host to communicate with the Arduino from software [24]. The Firmata protocol is designed to be used with a provided Arduino program, or sketch, that is compiled and run on the board and allows the MIDI-styled Firmata commands sent to the board to control various input and output actions. The protocol is designed to "co-exist" with the MIDI protocol, not by using the MIDI commands directly, but rather by interpreting the same commands as Arduino-specific actions. By adding this layer of abstraction on top of the Arduino, Firmata is separating the hardware dependent machine code from the desired software actions with a standard interface that allows any software to be used.

The level of abstraction in Firmata provides the hardware independence needed to freely transfer programs and content, but the current form of the protocol limits its application. While great for the Arduino, and even related boards such as the Sanguino [25], the protocol and firmware currently only allow any software to use the Arduino boards, not the other way around (allowing any hardware to be used with a given software program). The design of the protocol is Arduino-focused, making it viable for microcontrollers similar to the core Atmel chip and the Arduino board layout, but the commands are too specialized to abstract higher-level robotic behaviors or actions that use different functions. Even the current implementation of the Servo functions – the highest-level of abstraction as of now – groups the commands under the auxiliary MIDI System Exclusive message. Further extension into meaningful robotic functions would make the protocol unwieldy and over-generalized.

1.B.vi. RobotC & MRDS

A number of robotic programming environments are promoted as multi-platform software tools. These environments tend to be based around a single Integrated Development Environment (IDE) that uses a common language and methods to program any given robot. The programs themselves are written for specific target hardware and will present a customized library of robot-specific functions to access the Input and Output commands.

One of the more popular educational robotics systems is the RobotC development environment [26]. This software uses a simplified C-based language to program a number of robots and is supported by a large backing of tutorials and activities developed primarily by the Carnegie-Mellon Robot Academy. While the same base language is used for the LEGO NXT, the VEX Cortex, the VEX PIC [27], and soon the Arduino, different modified environments are required for the LEGO platforms and the VEX platforms. The code itself is primarily the same, but different robots require different hardware definitions and sometimes different functions to access their inputs and outputs. Recent developments of a Natural Language that offers higher-level behaviors are promising for allowing generic abstraction of robot functions; though, the current implementations are still platform specific and define different natural behaviors only for NXT, TETRIX, Cortex, and PIC.

Similar to the RobotC environment, Microsoft Robotics Developer Studio (MRDS) [28] supports a number of robots through generic services and 3rd party support implementations. These robots generally span a larger range of complexity and are more often used by researchers or university level projects. Robotic programs written using the Microsoft Robotics Developer Studio still use target-specific functions and commands to access the hardware and physical interaction. Additionally, the software only runs on Windows operating systems and provides little possibility of ever being able to compile programs on the small-processor systems found in education.

1.B.vii. Frameworks and Universal Interfaces (Urbi, Player, ROS)

A common approach to multi-platform programming provides high-level robotic abstraction interfaces. This is the system used in most of the current multiple robotic platform development frameworks such as the Player Project [29], Gostai's Urbi [30], and also in some services of the previously discussed Microsoft Robotics Developer Studio. Willow Garage's Robot-Operating-System (ROS) [31] follows a somewhat similar tactic, though the system is primarily focused on connecting various robots in more complicated systems, rather than the operational functions on a single platform. Behaviors and actions such as moving the robot to a position or grasping an object are offered as standard interfaces in systems like ROS, enabling the development of high-level algorithms. While this architecture allows the fluid communication and integration of various different robotic modules, the user is left to rewrite all the low-level implementation code for each specific processor to interface with these behaviors, tasks, and data. Even at the lowest level of abstraction found in robotic frameworks similar to Urbi, the libraries of functions for each robot are manually created by the developers to fit a static set of pre-specified functions [32].

1.C. Conclusion

The use of robots in the classroom reaches a broad and diverse base of applications. These applications vary from the traditional robotics endeavors as educators are more concerned with the generic behavior of robots. The traditional optimized functions that are customized and tightly linked to the robotic hardware are not as important as the ability to describe the general behavior of a physical robotic system in ways that can easily adapt to their lesson plans and curriculum.

The current robotics solutions all provide capable functionality and some even make a point of designing the environments to be accessible to teachers, who are focused on using the robots as tools. These solutions limit the effectiveness and impact of educational robotics use because they constrain the teachers to integrated packages of software, curriculum, and support exclusive to particular platforms. In order to take advantage of the principle that the robot solution should fit the needs of the mission, there needs to be a way for educators to utilize the general benefits and curriculum developed in the desired topics, no matter the current resources available to their classrooms.

A universal robotics language will allow software and content to easily transfer between robot platforms as long as it is able to abstract the generic behaviors and actions of robots in the physical world, rather than the microprocessor operations. From the perspective of the teacher, using a command called "Move Forward" should make the robot move forward, no matter what manufacturer's hardware is inside. Contrary to the goals of most robotics-specialists and researchers, a system that attempts to be universally inclusive to all robots, will inherently induce generalization and be unable to take full advantage of the features and power of particular robotic systems. However, in education this can be an advantage. In [5], Marina Bers discusses the importance of not focusing exclusively on teaching "technological literacy," such as knowing how to use a particular software or robotics package. A universal robotic system that generalizes by robotics concepts provides the benefit of putting the ideas and relationships first enabling lessons to concentrate on teaching the concepts, rather than only the application skills.

2. URAPI Design

2.A. Introduction Statement

Robotics software is composed of the algorithms, intelligence, and code (written by end-users of robots for their application), which calls functions that directly control or read the physical robot hardware. Presented in the following thesis is a system that inserts a universal interface layer between the two sides of software and hardware; by going through this standard middle interface, different robots can be easily substituted underneath while the program and intelligence on top remains the same, as illustrated in Figure 1. The interface layer does not remove either side from the other, but rather defines attachment points at the meaningful levels of physical actions and robotic behaviors. With this system, the process of substituting either the hardware or software while retaining the important elements is a smooth and painless procedure. Users will be able to create one line-following program and then run it on an NXT, a Create, or an Arduino based robot, without changing any of the original code.

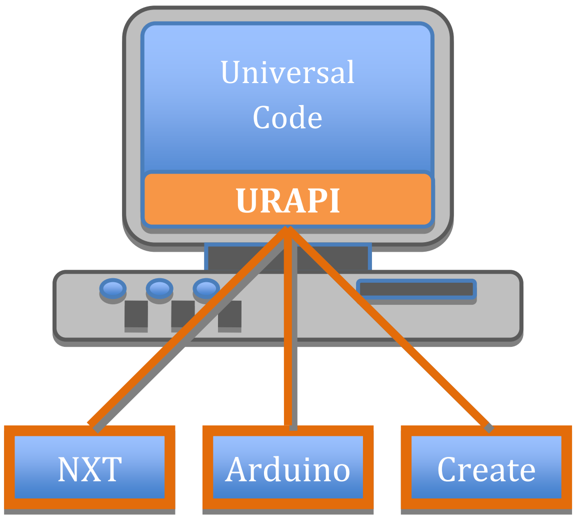


Figure 2.1: Universal Interface Layer; showing the user's universal code (with URAPI base) and the multiple individual robots as possible targets.[1]

2.B. Specification

The Universal Robotics API (URAPI) system allows a user to write universal robot programs capable of being executed on any robot platform. These robotic programs can be viewed as compositions of various behaviors and actions the robot should perform in order to accomplish some larger, overall task or goal. In the prior line-following example, the overall goal is to follow the edge of a line; the smaller behaviors and tasks are using a light sensor input to control the motors of the robot so the light intensity of the floor is halfway between the dark floor and bright line. The purpose of the URAPI universal program is to allow the user to define these behaviors in a manner independent of any particular robot platform. The URAPI system can then execute the universal programs on any chosen robot by directly translating the universal commands into commands in the native language of the robot. Thus, URAPI will translate the generic commands to control the motors of the

[1] Illustrated by Jessica Noble.

line-follower, to a built-in 'drive' command on the Create and to Pulse-Width-Modulation (PWM) output commands on the Arduino robot.

As depicted in Figure 2.2, the overall URAPI system is designed to keep the user's programs and content separate from the actual hardware. This means that content developers and curriculum designers can create dynamic libraries accessible by any user – no matter their hardware situation.

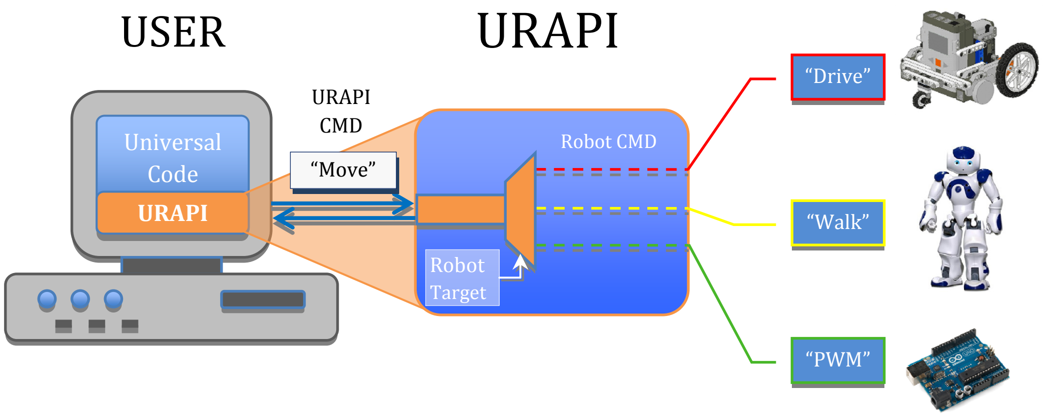


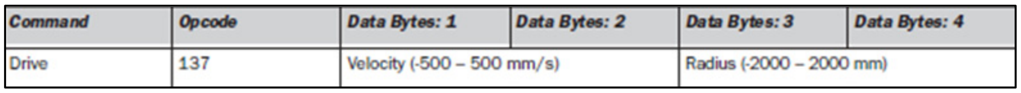
Figure 2.2: High-Level URAPI System Layout[2]

The URAPI system is able to unbind the software from the hardware through the use of an intermediate universal language. In this way, the hardware manufacturers only have to worry about adapting to the one set of standard universal commands. Similarly, now the software writers are able to write their programs and applications using the functions and components entirely within the universal language – knowing that all URAPI robots will be able to understand the code.

The universal language provides generalized robotic actions and universal descriptions of the hardware components, which describe the archetypical input or

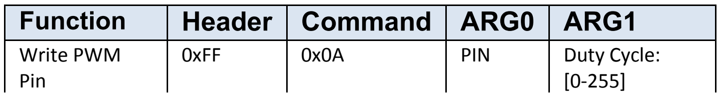
[2] Illustrated by Jessica Noble.

output processes and the essential properties of the devices in relation to their role in the robot's program. To illustrate, consider a simple robot program that moves forward until the robot sees a light. The universal language provides a generalized action for Moving in a given Direction at a given Power or Speed. Since there are many ways robots can "Move Forward," the generalized URAPI action itself does not stipulate whether this is accomplished using legs, wheels, tank treads, or helicopter blades. In fact, since some robots may have one high-level command explicitly for "driving" and others may require programmers to use a sequence of lower-level analog "Voltage Output" commands, the user does not stipulate which target-specific functions the robot will use, only the action they wish the robot to achieve (e.g. moving itself forward) Figure 2.3. The point of this level of definition is to define the actions in only as much detail as is necessary to describe the intended behavior, without going into the details unique to the particular robot's implementation of the task. The description of the robot's actions is intended to reflect only the level of detail the teacher or student cares about knowing or specifying how the robot actually carries out the desired task.



Command	Opcode	Data Bytes: 1	Data Bytes: 2	Data Bytes: 3	Data Bytes: 4
Drive	137	Velocity (-500 – 500 mm/s)		Radius (-2000 – 2000 mm)	

(a)



Function	Header	Command	ARG0	ARG1
Write PWM Pin	0xFF	0x0A	PIN	Duty Cycle: [0-255]

(b)

Figure 2.3: Two examples of Move Forward a) the iRobot Create Drive() Command, and b) the Arduino using analogOut() commands.

There are different commands and devices that robots can use to accomplish the same generic robot task or perform the same behavior. Continuing with the Moving Forward behavior from the previous example, there is a wide range of approaches found just within the education-targeted robot demographic. Straightforward differences appear in variations of mechanisms used by the robots: Humanoid robots like Nao (Figure 2.4.a) [33] use legs to move around the room by applying time-variant 'gaits' in order to move in a certain direction; while simple wheeled robots such as the Finch (Figure 2.4.b) just rotate basic DC motors in one direction or the other. However, sometimes the physical differences do not change the fundamental methods used to complete the robotic task. Treaded robots like the LaSalle Robot (Figure 2.4.c) [34] and wheeled robots similar to the iRobot Create (Figure 2.4.d) [35] both provide a differential drive as the main command to control their movement. Despite all the differences in physical and programmatic implementation, the key factor all the robots share is that they have some method to control a set of actuators, whose function is to provide a means of locomotion. The idea behind running a universal program on a variety of robots is that each robot will use its own special capabilities to achieve the same results – they will all perform the same behaviors, each in their own special way.

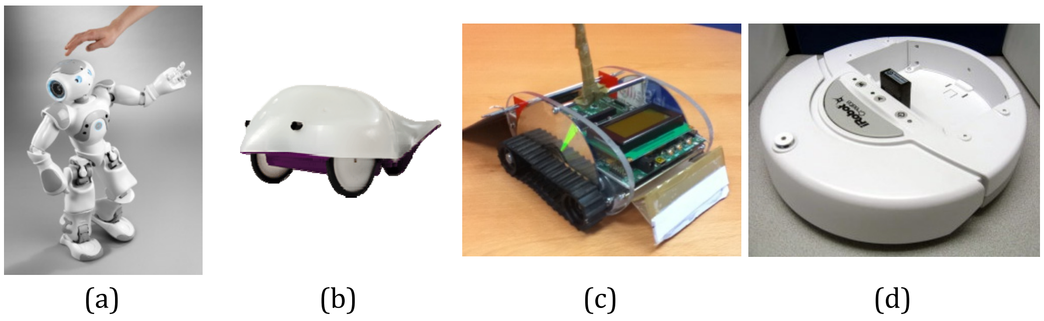


Figure 2.4: Popular Educational Robots: a) Nao Humanoid; b) the Finch Robot; c) LSMaker Robot from La Salle; and d) the iRobot Create.

Attempting to group all the special abilities and variations of such a diverse set of robots is bound to lead to some loss of functionality and potential on a subset of platforms. The advanced tracking capabilities of the AR Drone [36] would be difficult to abstract in a cross-platform manner and high-speed precision control loops may be limited by not being able to get to the “bare-bones” optimized functions. However, in the target educational classroom scenarios, optimized control of individual low-level actuators is rarely the main goal of the lesson. Many of the activities and curriculum currently developed around these robots are more concerned with the fundamental behaviors of robots; activities center around sensing the world, planning the motions and steps necessary to complete a maze, or designing a robotic solution to accomplish a simple engineering task. Being able to get started quickly on new hardware or utilize polished and tested curriculum developed by other teachers gives an important benefit to educators in teaching the actual concepts they want the students to learn, rather than bothering with the details of a particular robot’s operation or implementation.

Thus, the URAPI purpose-based Universal Commands and Universal Devices make it possible to create a standard interface layer on top of all robots that encapsulates the particulars of the robot’s implementation. The standardization of these URAPI structures means that the interface layers for all the robots have the same attachment points, thereby enabling one robot to be swapped out for any other URAPI robot. This standardization also allows for a common means of accessing and controlling the behavior of any robot – a quality that protects URAPI-based software from becoming obsolete. If a new version of a robot uses a different function to control motors, the underlying implementation code can be completely revamped while still providing the same universal interface to the software. Thus, the user's old program is automatically compatible with the new robot without requiring rewrites or "broken code."

2.C. Universal Data API (UDAPI)

2.C.i. Description

The development of a Universal Data API for Smart Sensors served as an initial trial ground for implementing some of the concepts central to the URAPI design. In robotic systems, the interaction with the physical world is conducted primarily by commands that send output values to specific actuators or read inputs from a certain sensor. Exploring a method of generalizing these inputs and outputs as simply reading and writing appropriate values to a certain location gives significant insight and a development base for a full universal robotics system. Data API focuses on the basic activities of data sampling and logging as they commonly take place in pre-college curriculum.

This Universal Data API, also known as UDAPI, is based on the simple idea that a user should be able to plug any sensor into their system, and the sensor tells the system all the information about what the sensor is and how to use it. This means no new drivers for every sensor while at the same time enabling data logging and analysis software to be not only compatible, but also knowledgeable about any sensor attached. The UDAPI specifications automatically give sensor data context and description to evolve datasets from 'just numbers' to descriptive measurements of phenomena in the real world.

The overall architecture of a UDAPI system is seen in Figure 2.5. The UDAPI specification prototypes a standard method of describing input and output devices, which is a critical aspect of the universal robotic system and programs. This means describing the capabilities of the hardware devices themselves and defining the structure and pieces used as building blocks to describe the data values and transducer control in a universally understood manner. The universal commands, used as the protocol to communicate between UDAPI devices, are defined according to the general actions and tasks common to any data logging activity, rather than to using a specific type of sensor or categorizing the types of actions based on the processor operations. Finally, one of the fundamental URAPI principles is the abstracting interface layer that provides a buffer between the goals that the user defines in software and the messy details of how the transducer implements those goals.

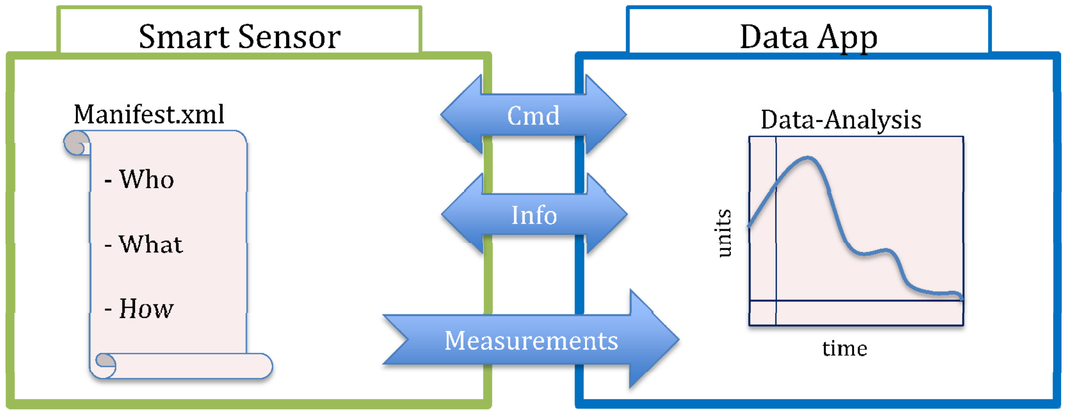


Figure 2.5: UDAPI concept image

In order to achieve the goal of a universal interface for sensors – a common layer through which all UDAPI systems could communicate and understand one another – there first must be a standard protocol, or 'way of talking,' between UDAPI systems. This provides the base level on top of which the data information can be transmitted and understood. The understanding of the data comes from being able to ask the UDAPI sensors (using the common protocol) for the necessary information that describes how to interpret the numeric values coming across the channel as meaningful, quantitative measures of the 'real' world. This 'necessary information,' known as the sensor's 'Manifest,' constitutes the third core element of

UDAPI – the standard format (syntax) and vocabulary (keywords and definitions) for characterizing the sensor, describing the functionality, and explaining the measurements.

2.C.ii. Elements and Principles

Previous work on defining a universal protocol for smart sensors was published by the Institute of Electrical and Electronics Engineers (IEEE) in the "IEEE Standard for a Smart Transducer Interface for Sensors and Actuators Wireless Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats" (IEEE 1451) [12]. The multiple layers of the IEEE 1451 standards are designed to completely cover every characteristic of any sensor to the level of detail required by an industrial sensor engineer. The intricacies and extensive definition required to implement the IEEE 1451 makes it unsuitable for the educational arena – a market heavily influenced by school district budgets. Researchers working with the sensor standard have suggested the implementation is too complicated to be completed by anyone other than industrial sensor engineers [13]. However, the design of the UDAPI protocol and specifications is heavily influenced by IEEE 1451 features and in many ways serves as a simplified version of the industrial standard.

The UDAPI system specifies the following main components: (1) the UDAPI Sensor, (2) the Embedded Manifest, (3) the UDAPI Commands, and (4) the software UDAPI Data Application. In the following sections, these subsystems and specifications will be discussed along with the role each part plays in a full UDAPI system.

1) The UDAPI Sensor

UDAPI sensors have three specifications: The sensors must understand and appropriately carry out the UDAPI commands; the sensors must send data and communicate in the format of the UDAPI protocol; and the sensors must carry with them the Manifest.xml file that defines their UDAPI characteristics. The UDAPI specification requires that the sensor be able to send and receive individual packets of data that contain information in the UDAPI format. The actual hardware or communication standard used to communicate the UDAPI packets is not specified. This allows the UDAPI protocol to apply to any sensor system capable of making a physical communication connection, no matter if it is across USB, Bluetooth, or even TCP/IP. Extensions into data experiments using multiple physical UDAPI devices simultaneously take advantage of "bus" communication protocols like USB and i^2c where multiple devices are able to communicate on a single line. The standard UDAPI interface layer allows new devices to be integrated into a data logging experiment just by plugging the new UDAPI sensor on to the bus. The onboard Manifest.xml file can be dynamically retrieved to determine the individual characteristics and operating information of the sensor. As described in Figure 2.6, this information falls under the general categories of "who" the sensor is, "what" data the sensor measures, and "how" to communicate with and use the sensor.

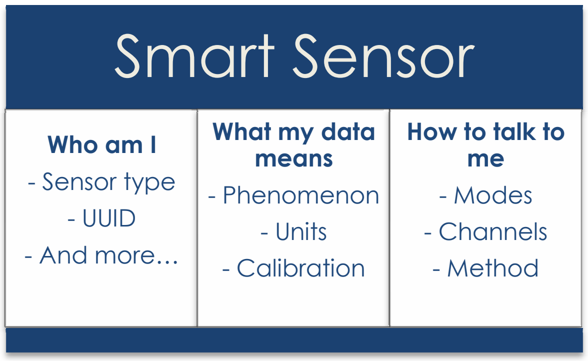


Figure 2.6: Smart Sensor Information

2) Embedded Manifest

The Manifest is the file found on all UDAPI sensors that contains all the information about the sensor and how to use it. The categories include 'ID', the information about the sensor product and other identification; 'Transducers', the information about each method and option available to sense data using the device; and 'Data Logging', the data logging capabilities. The general structure of the file is depicted in Figure 2.7.

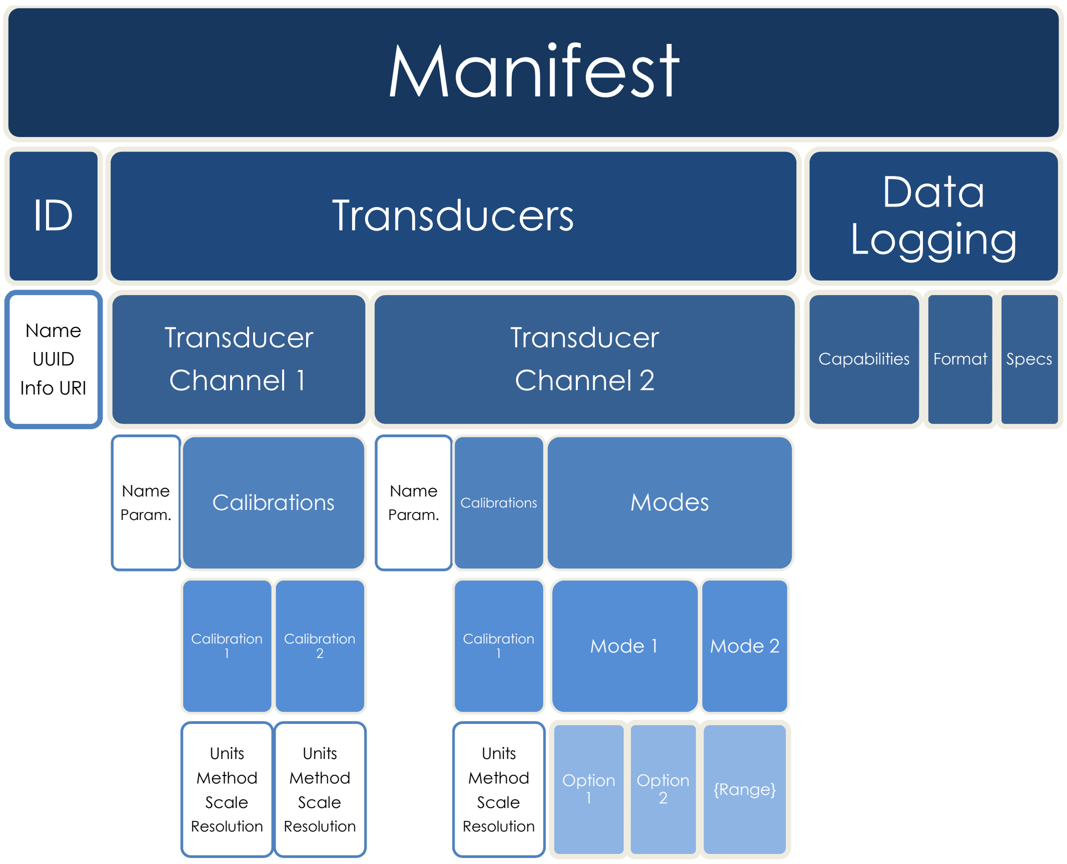


Figure 2.7: UDAPI Manifest Depiction

In cases of multiple sensors, or Transducer Channels (TC), the Manifest can also indicate default settings. Some categories are required to be present by the UDAPI standard, such as the Universal Unique Identifier (UUID) and at least one Transducer Channel (TC). Other fields may be optional or provide default values so as to simplify implementation. The standard format is written in XML (Extensible Markup Language) using standard UDAPI elements. An example for a simple NXT light sensor is shown in Figure 2.8. Part of the Manifest.xml has been collapsed for the sake of clarity.

```
<sensor lang='English'>
    <identity>
        <name>NXT Light Sensor</name>
        <uuid>LEGO.NXT.Light.9844</uuid>
        <version>9844</version>
        <info_uri>http://mindstorms.lego.com/en-
            us/products/default.aspx#9844</info_uri>
        <hardware/>+
        <manufacturer/>+
    </identity>
    <transducers>
        <transducer index='1'>
            <chan_num>1</chan_num>
            <chan_type>sensor</chan_type>
            <desc/>+
            <conversions>
                <conversion index='1'/>
                <conversion index='2'/>
            </conversions>
            <modes>
                <mode index='1'>
            </modes>
            <capabilities/>+
        </transducer>
    </transducers>
    <datalogging/>+
</sensor>
```

Figure 2.8: UDAPI Manifest.xml Code Snippet Example ('+' marks collapsed sections).

One key aspect about the UDAPI Manifest.xml is that the "file" is embedded on the sensor itself. The commands allow the Manifest to be requested and sent over the regular communication line, but the possibility to read the file directly off the sensor like a USB Flash Drive is a future option. This would ideally be coupled with the specification of a USB HID Device Type so that the UDAPI devices could be identified by any system. The goal of all the onboard information and self-identification is to avoid the difficulties and troubleshooting that inevitably occurs in attempting to install the proper drivers and software and performing configuration for individual hardware sensors.

Since the firmware details are physically stored on the device, this also creates a concrete link between software upgrades and device versions. The Manifest includes a Universal Resource Identifier (URI) field where more information and updates can be found for the sensor. It is possible that future implementations could use this URI to automatically update the firmware on the device. Incompatibilities from upgrades are avoided because any updates to the onboard firmware can check hardware compatibility directly before being installed. In addition, compatibility of new software features can always be guaranteed since the firmware on the device describes the capabilities of the specific sensor hardware.

3) UDAPI Commands

As discussed above, there is a simple set of API commands defined for reading data, getting information about the sensor, and controlling options. Examples of the UDAPI Simple Commands are seen in Table 2.2. The possible parameters and options are all informed by the Manifest file. Each 'mode' element will have a name and children 'option' elements that list and describe the type of arguments.

Command	**Description**
<R>	- Read default sensor value
<G, M, 2>	- Get the current setting of Mode 2 option
<S, M, 2, 33>	- Set the Mode 2 option to a value of 33

Table 2.1: Example of using UDAPI Simple Commands

For example, looking at the expansion of the 'modes' node in the prior light sensor Manifest, there is one 'state'-type option named "Floodlight" Figure 2.9. The 'state' type specifies an option with multiple discrete states, rather than a continuous number within a range of values. The Manifest is simply providing information on the available options. Using a 'Set' command to choose the

Floodlight mode will inform the control logic running on the UDAPI sensor, but not necessarily change any values in the XML file.

```
<sensor lang='English'>
    <identity/>+
    <transducers>
        <transducer index='1'>
            ...
            <modes>
                <mode index='1'>
                    <type>state</type>
                    <name>Floodlight</name>
                    <states>
                        <option index='1'>On</option>
                        <option index='2'>Off</option>
                    </states>
                </mode>
            </modes>
            ...
        </transducer>
    </transducers>
    <datalogging/>+
</sensor>
```

Figure 2.9: UDAPI Modes Manifest.xml Snippet

The full set of Simple Commands definitions can be seen in Table 2.2. The most important commands are Read, Get, and Set, and constitute the majority of the basic UDAPI functionality. The simplicity of these Commands allows them to be easily handled by control logic run on the UDAPI Sensor.

Command	Bytecode	Parameters	Returns	Description
Get XML	X		X, (r1), {Manifest.xml [text]} >> r1 = length X[-1] >> [-1] is a byte value	Request the Manifest.xml file from the Sensor. Sends the content in chunks.
Clear	+++			Resets buffer and clears temp data
Read	R		R, Sensor value	Reads the sensor value using the current configurations and calibration
Set	S	M,(v1) C,(v1)		Sets the possible options (M - for mode, C - for conversion) to the given value/selection (v1). (v1) selection should correspond to the index in the Manifest.xml for the desired configuration
Get	G	M C	G,M,(r1) >> r1=Current Mode G,C,(r1) >> r1=Current Conversion	Returns the current selections for the given option
Write	W			Opposite of Read, but for Actuators
(Error)			!!!,(bad command received)	

Table 2.2: UDAPI Simple Commands Protocol

In addition to these Simple Commands there are more complex UDAPI Data Logging Commands that enable the user or applications to setup data logging experiments using multiple universal transducers and various experiment configurations. These commands set up 'experiments' for multiple Transducer Channels, based on a Start and End condition and a 'Trigger' to signal when the UDAPI sensor(s) should take new data points. The sampling Trigger is usually based on time intervals, but can also come from other UDAPI sensors, such as a button on the UDAPI device used to sample on demand. The full set of UDAPI Data Logging Commands can be viewed in Appendix B.2-UDAPI Data Logging Commands and online at http://sites.google.com/site/ceeourapi/durapi/command-protocol.

4) Universal Data Logging / Analysis SW Applications

The final piece of the UDAPI system is the UDAPI-compatible applications that actually interact and use the UDAPI sensors. The benefit of UDAPI-based data logging applications is that they may be designed in a general way, and dynamically read the information from the sensors needed to populate the graphs, label axes, run experiments, and do any other number of data logging activities, just by writing the code to do it once with the generic UDAPI commands.

The point of the UDAPI system is that the user has a piece of hardware that enables them to uniquely interact with the physical world; how to interface with that piece of hardware need not be anywhere but contained within the device itself. The Manifest and universal data protocol facilitate this embedded intelligence, enabling more accessible sensors for students and meaningful explorations of the world.

2.D. URAPI

2.D.i. System Design Introduction

The goal of the URAPI system is to provide a single development environment from which any robotic platform can be easily programmed. Furthermore, the system aims to provide a means of writing universal programs that are capable of running on any robot of choice, with the same resulting behavior.

The following sections describe the goal of the URAPI system and the similarities in the larger robotics system to the UDAPI protocol that investigates the subsection of inputs and outputs. The rest of the chapter will then review the major aspects of the robotics API design, including the core universal language and translation architecture that connects the universal parts to particular robots.

1) Focus of System Design

The overarching URAPI model covers a large area of robotics uses including the two most typical operating modes: Compiled mode, where pre-downloaded programs are executed on the robot by itself; and Tethered mode, where the computer executes the program and sends commands to the robot on demand. The full URAPI model is depicted in Figure 2.10.

As discussed in the Introduction, the focus of the URAPI system is on the lower-end, small-processor systems commonly used in educational settings and by non-professional robotics programmers. In Compiled mode, programs are compiled and completely saved onto the robot hardware directly, where then the robot executes all of the actions and performs all computations as defined in the code. Earlier work of applying the larger URAPI concepts in this compiled mode shows a successful implementation of cross-platform compatible programming on small processors [37].

In Tethered mode the robot is linked, either physically or wirelessly, to a computer that is executing the program and sending across only the actual robotic input and output commands that involve the physical hardware. Tethered mode is the primary focus of the URAPI system design presented below.

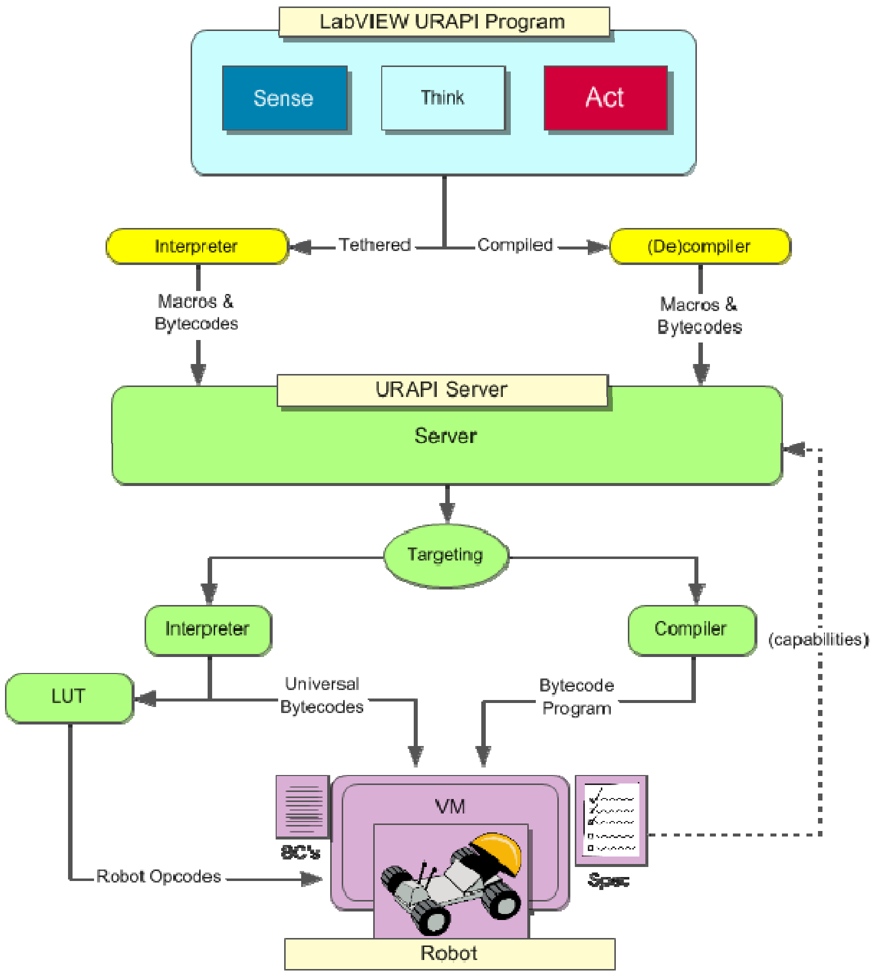


Figure 2.10: Full URAPI System Diagram

As Tethered programs execute most of the computational code on the computer, rather than on the actual robot hardware, the core of the system is concerned with the universal design in relation to robotic input and output commands. The "Sense, Think, Act" model provides a framing for the subsets of robotic operations used in both the larger URAPI model and the applied design of this thesis. The three classifications are meant to categorize the type of every robotic command at the most general level. Commands related to reading inputs or sensor values are considered 'Sense' tasks, any operations that change outputs or cause the robot to act on the world are considered 'Act' commands, and the majority of the remaining operations, which tend to deal with computations, logic, or program flow are classified as 'Think' behaviors.

The Sense and Act commands are the primary focus throughout the design of the following system due to the important role inputs and outputs play in robotics. The variety in sensors and actuators used in Sense and Act tasks is one of the major roadblocks to standardizing lower-level robotics programming [38]. An effective solution for this challenge will enable future integration with universal control logic explored in [37] to create an effective, comprehensive Sense-Think-Act URAPI system.

2) UDAPI Similarities

While the objective and structure of the URAPI system does not directly inherit from the UDAPI project, many of the central principles demonstrated in UDAPI are shared by URAPI. Most notably, the characterization of Generic Input and Output data provides insight for the development of Universal Devices capable of describing any type of hardware peripheral. A core premise of the URAPI system is that hardware control and interaction can be represented at the most fundamental level as simply reading or writing a value at a certain location in a specified format.

Another major shared concept is the design of the functionality-based abstracting interface layer between the hardware and software. In the URAPI system, the scale of the interface grows tremendously as the universal actions are no longer limited to the subcategories of data logging and device configuration. Because of this, maintaining a consistent functional definition across all the universal commands is an even greater challenge, which drove the design of new components and features in the universal system.

2.D.ii. Universal Language

Robotic programs are different from traditional computer programs in the respect that a robot's purpose is to achieve certain goals using various behaviors or performing tasks, all while interacting with the physical world [39]. This is in contrast to typical computer programs, which execute a specific deterministic 'plan' with predetermined computations and processes. Robotic programs use Sensing to ask questions about the environment, Thinking to understand the answers, and then make decisions on what Actions the robot should perform in order to achieve the current objectives. These questions, decisions, and actions form simple 'tasks' for the robot to complete. By linking together various simple tasks, the robot is able to complete more complex behaviors, which themselves can be used to achieve the goals and overall mission of the robotic program. Thus, by using these generalized tasks as the basis for the URAPI universal language, users are able to write complex robotic programs capable of running on any robot.

In the URAPI language, all Sense and Act tasks are defined by the combination of a Universal Device, which specifies the physical Input or Output component to use, and a Universal Command, which defines the action the robot should perform on the Device. This division allows a more flexible and meaningful definition of the task and its purpose. For example, a user may write a differential drive program that uses the difference between left and right ultrasonic sensor Universal Devices, read using two ReadAnalog() Universal Commands, to follow an object in front of his NXT robot. When he shares the program with his friend who has a Finch robot, she is able to run the differential drive smoothly, even though her Finch uses digital IR detectors as the Universal Devices. The robotic program is able to retain its purpose and meaning across the two platforms through the combination of the Universal Devices for the sensors and motors, the Universal Commands that read and write analog values, and the computational logic connecting the inputs and outputs.

1) Universal Commands

The Universal Commands define the actions a robot performs in a generalized task. These Commands accommodate a broad spectrum of robotic actions which vary in complexity from simple control of voltage on a line to high-level behaviors to drive mobile robots to a location. Corresponding directly to the classification of the Sense and Act behaviors, the majority of Universal Commands are divided into the two primary categories of Input and Output. All the Commands within the Input category define various ways in which to read values from sensors or obtain information about the robot and environment. Likewise, the Commands within the Output category define various ways the robot can act upon the world or control attached actuators.

Within each of these two categories, there is a hierarchical collection of Universal Commands. The most basic commands, such as ReadAnalog() and

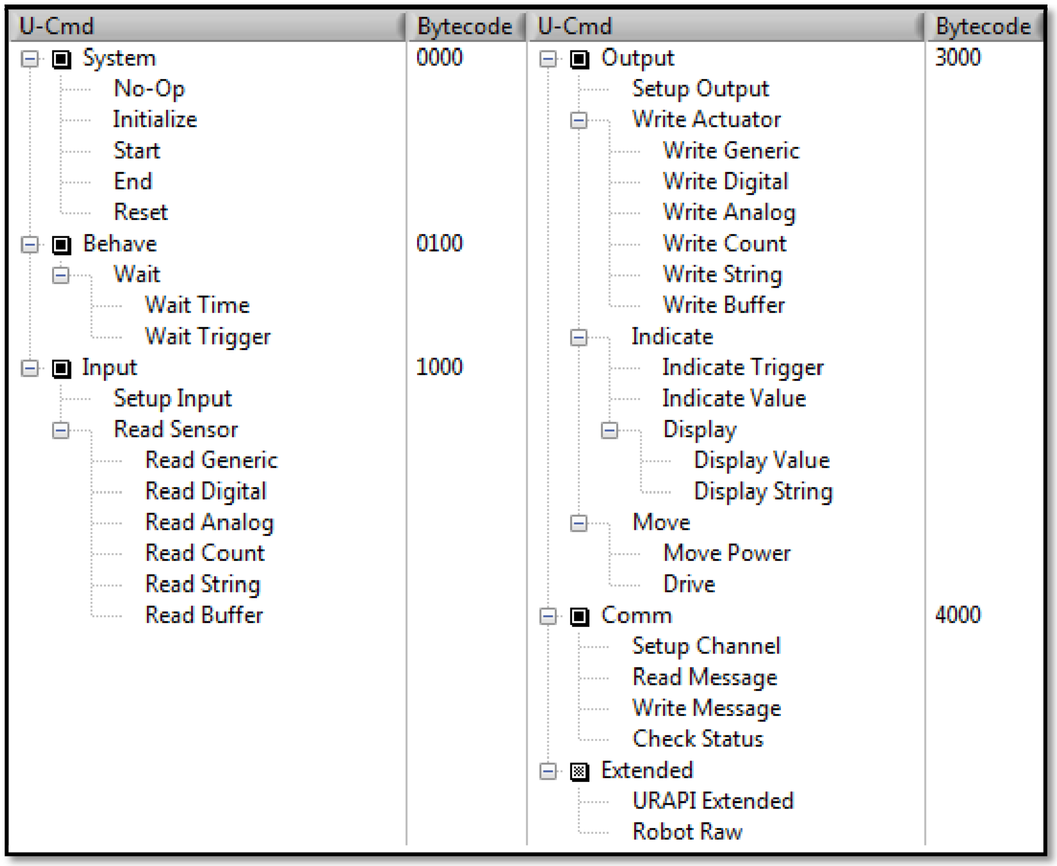
OutputDigital(), are defined by the data types of the values they return or use as control signals. The Digital-based Commands use simple On-Off Boolean states, while Analog Commands allow for a range of decimal point values between 0.0 and 100.0 (or [-100.0, 100.0] for signed numbers). More complex Commands describe the actions performed in a greater behavioral detail. For instance, the Move Command specifies the physical movement of the robot in space and the Spectrum sub-hierarchy of Universal Commands allows users to define frequency or time variant outputs that may be valuable for sound or light based robotic programs. For every Universal Command, no matter what Universal Device is used, the Command will always Input or Output the same data types. In this way, the Universal Commands ensure a consistency to the expected format of input arguments and output values.

Each Universal Command is linked to a unique numeric identifier, or bytecode. The bytecode allows the URAPI system to do two things: first, the URAPI system uses the bytecode to look up the format of the Command including the arguments expected and what the function should return; second, the URAPI system uses the bytecode to find the corresponding implementation of the Universal Command for the current robot. Each Universal Command has a set format of input arguments and output return values that are consistent across robotic platforms. The input and output variables are mostly defined within standard data types, with the exception of Universal Devices. The bytecode identifier and standard arguments also allow the URAPI Universal Commands to be recognized and parsed on the small microcontrollers found in the educational robots. This specification means that robots can be directly controlled and communicate in the URAPI Universal language by implementing a Virtual Machine firmware to interpret the bytecode Commands.

The hierarchical structure of the Input and Output Commands defines the higher-level behavior-based robot actions inside families of related, more generic Commands. By generically classifying the higher-level Commands according to function, a clear lineage is set forth that allows the more basic URAPI robots to define 'Macros,' or sequences of simpler commands, in place of direct function translations. More qualitative or behavioral tasks such as driving forward will often be implemented with Macros on small processors that use simpler commands to achieve as close a behavior as feasible on the robot. By including Universal Commands for both the high-level robotic actions, such as Drive(), and the low-level Commands, like PWM Output(), URAPI is able to overcome the common problem of discrepancies in the complexity of available commands on various education robots.

Table of Commands

The list of Universal Commands is shown in Figure 2.11. The fundamental, data-type based Input and Output Commands are seen under Read Sensor and Write Actuator, respectively. Each fundamental Command corresponds to a certain type of variable that is used with the given Command. For instance, the Read Digital Command outputs a Boolean `TRUE/FALSE` value, while Read Analog outputs a floating point number. When a robot's native functions are defined in the URAPI Translations, each function falls under the Universal Command that corresponds to the values it provides or receives. Universal Devices are similarly categorized by the data-type of the Base Command used to communicate with the Device.



U-Cmd	Bytecode	U-Cmd	Bytecode
System	0000	Output	3000
No-Op		Setup Output	
Initialize		Write Actuator	
Start		Write Generic	
End		Write Digital	
Reset		Write Analog	
Behave	0100	Write Count	
Wait		Write String	
Wait Time		Write Buffer	
Wait Trigger		Indicate	
Input	1000	Indicate Trigger	
Setup Input		Indicate Value	
Read Sensor		Display	
Read Generic		Display Value	
Read Digital		Display String	
Read Analog		Move	
Read Count		Move Power	
Read String		Drive	
Read Buffer		Comm	4000
		Setup Channel	
		Read Message	
		Write Message	
		Check Status	
		Extended	
		URAPI Extended	
		Robot Raw	

Figure 2.11: URAPI Universal Commands, listed by Command category

Within the URAPI Universal Command set, there are two special functions, GenericInput() and GenericOutput(), that serve as all-encompassing extensions of the abstracted Sensing and Acting actions. Essentially, the generic functions serve as interfaces for all the Commands so that robots need not specify how to handle every single function in the URAPI instruction set, to still achieve sufficient compatibility. In general, the goal of the generic function is not to provide precise adaptation, but rather to "do well enough" in situations that are trying to convey high-level concepts or are outside the reasonable jump to be made (in terms of transference of a program onto a significantly different platform than it was intended). An example of an impractical scenario would be a 3D face tracking program written for a 6-Degree-of-Freedom flying robot, being targeted to a two-wheel robot car with a light sensor and an 8-bit processor.

The URAPI Extended and Robot Raw Commands provide means to extend and customize users' code in order to capture unique features or advanced capabilities that are custom to particular robots. The URAPI Extended Command allows for future patches and modules to implement advanced functions that do not fit the standard URAPI protocol format. An example of such a function is the ability to send pictures or stream live video from the robot. The Robot Raw Command allows programmers and manufacturers to send raw byte buffers to the robot in order to use custom functions for the particular robot or implement any additional operations directly through highly optimized code. The purpose of these two Commands is not to be the primary means of implementing robot features, but rather to make URAPI extensible for manufacturers and users through an integrated option for handling special cases.

2) Universal Devices

With respect to the generalized tasks the robot performs, the distinction of the Universal Devices as an argument of the Universal Commands allows users and robots to easily substitute physically different hardware components that can be used to accomplish the same task. There are four main properties connected to a Universal Device: the Class, or purpose, of the Device; the Device Type, which describes how to interact with the hardware; the Identity, or ID, which simply uniquely identifies Devices of the same Class or Type; and the Base Command, which specifies the Universal Command used to talk to the Device.

The Class is the most important property with regards to the universal definition of the Device. The Class describes the purpose of the Universal Device with respect to how the component is used in a given task or the characteristics of the information the Device reads or writes. This can vary from base data-type descriptions of the inputs or outputs, such as a Boolean input or a generic analog output, to a more functional 'job' the device does, like the left wheel of a robot car. The logic behind the Class is to retain a consistent definition of what role the hardware plays in the program. By describing the function rather than the particular specification of the hardware, different robots can pick the most appropriate hardware available for the task. Users are able to describe this functionality by defining a simple list of characterizations starting at the top with the most important traits (ex: Locomotion or Light Sensor) and then getting more general with each subsequent level down the list (ex: `Analog_Input`, `Digital_Output`), until the required base descriptions at the end (Input or Output). The Class mechanism allows the user to indicate what properties are most important to try to match when targeting new hardware.

The Class list is critical to URAPI's ability to adapt across robot capabilities while retaining the meaning of the program. A Universal Device with a Class list starting with a high-level specification like `Left Wheel`, as on the iRobot Create, will include intermediate Classes after the first entry such as `LocoMotor`,[3] `Motor`, `Analog_Output`, and finally `Output`. This means that when a user chooses to use a simple robot, like an Arduino where `Left Motor`, or even `LocoMotor`, have no applicable meaning, the URAPI system can gracefully fall back on the intermediate Class entries. In this case, that means URAPI can automatically assign the same

[3] A `LocoMotor` Class indicates a motor that can affect the robot's physical location.

“Left Wheel” Universal Device to a PWM Output on the Arduino that includes Motor and Analog_Output in its own Class list. Similarly, the same “fall back” mechanism allows URAPI to smoothly substitute different high-level Devices that perform the same function. For example, a Passive Infrared sensor, an Ultrasonic sensor, and a Bump sensor, may all include an intermediate Class of Obstacle Sensor that would allow any of these Devices to be interchanged, even if the top-level Classes are all different.

In contrast to the user program-focused Class field, the type of the actual matched hardware component on the robot is found in the Device Type property. For instance, the Device Type will indicate whether the Universal Device is a DC Motor or a PWM Servo. More specifically, the URAPI system identifies the proper robot function to use and the port address on the robot to read or write based on the Device Type. Every Universal Command with a Universal Device argument specifies a corresponding default value for the Device Type. The fundamental, data-type based Commands simply use the data-type that is read or written, such as Analog_Input (Double-precision) for Read Analog, or Count_Output (Integer) for Write Count. Higher-level Commands specify appropriate higher-level categories, such as Move that uses the LocoMotor Output Type and Display that uses the Display Output Type. When manufacturers create robot hardware Manifests (device descriptions and definitions similar in function to the UDAPI Manifest.xml), they describe each possible variety of sensor as corresponding to a specific Type, starting with the defaults for the Universal Commands. There should be a Type for every sensor that uses a different robot function (instruction and opcode) to talk to it; this means that robots with multiple sensors that return, say, Analog values, but use different functions on-board, will have multiple Types even though the sensors use the same Universal Command. Depending on the transducer matched by the Device Class on a given platform, it is possible for the Device Type to vary across robots for the same Universal Device in a URAPI user program.

For any given transducer on a robot, no matter how the device is used nor how you transform the return values, the final output to the robot boils down to a single operation, the Base Command, which will always have the same command code, port, and format of arguments. All together, this means that even when the Class of a Universal Device specifies a Digital Button or trigger, the robot can use a Light Sensor with an Analog Device Type, and URAPI can automatically interpret the Analog values returned from the sensors Base Command as Digital Boolean values.

2.D.iii. Target Robot Translations and Definitions

As described previously, URAPI universal programs focus on recording the generic tasks and behaviors of the robot; this enables individual robots to define their own implementations of these tasks based on the available robot-specific functions best suited for achieving the task. For each URAPI-compatible robot, there is a set of definitions that specify the functions the Universal Commands map to on the robot, and the available hardware and capabilities identified as Universal Devices. The URAPI system is able to load the proper definitions dynamically when the user targets a robot. The Universal Devices and any superficial customization layers created by the manufacturer can enrich the programming experience of the user; however, URAPI benefits most from being able to translate the Universal Commands directly to native robot functions using the translations provided in the Robot Look-Up Table (Robot LUT).

For each Universal Command, the robot manufacturer defines the corresponding native function the robot uses to implement the action. Since the Universal Commands necessarily generalize and categorize robotic actions, such as the Read Digital Input() Command, the Robot LUT allows the manufacturers to define multiple robot functions for a given command, based on the particular Universal Device being targeted. This is a necessary step as the tight integration of hardware in robot software means the mapping of robot functions to Universal Commands is rarely 1:1. Robots such as the Finch have multiple input sensors that return Analog values – and therefore are all accessed with the ReadAnalog() Universal Command; however, the Finch's simplified interface uses unique functions for each type of device: the 'T' bytecode reads the temperature sensor, the 'L' command returns the values of the two light sensors, and the 'A' command returns the accelerometer readings. On more flexible platforms such as the NXT, various types of devices can be attached to the same hardware connection ports, and values are returned using the same GetInVals() command. These two scenarios demonstrate the majority of cases that drive the robot function definitions based on the Universal Command and the Universal Device.

URAPI offers two approaches to handle possible differences in robot complexity: the first is generalizing, which uses the Class of Universal Devices and category of Universal Commands; second is the ability to support higher-level Universal Commands through a sequence of simpler commands called 'Macros'. Macros are most often used on bare-bones platforms where more complex Commands such as Drive may have no meaning. The basic idea is seen in movement and motor commands. The iRobot Create offers high-level Drive functions that control the velocities of the wheels directly. A robot built on top of an Arduino board may move its motors using Analog Output commands, or possibly even Servo library functions that let programmers specify duty cycles. The problem is really that the Create *only* provides the high-level commands to control its motors. This means that when a user tries to write a universal program to move both robots, they must use the single Drive command on the Create, but use a collection of Servo or PWM commands on the Arduino to provide the same behavior. In order for the same universal program to run on both robots without the user having to rewrite the code, there needs to be a way of matching the multiple Arduino functions and the single high-level Create function. Thus, the URAPI Robot LUT provides Macros as a means for manufacturers of "simpler" robotic platforms to define multi-function implementations for higher-level Universal Commands.

2.E. Conclusion

If all robots are considered to speak different languages with different vocabulary, grammar, and punctuation, the efficacy of a universal robotic system can be judged on how accurately the meanings of the sentences, or programs, translate between robot languages. The design of the URAPI system focuses on identifying the "main ideas" and "plot" of a robot program and accurately preserving them when translating to a given robot. The Universal Commands and the Universal Devices provide the building blocks for users to write meaningful universal robotic programs, while the design and flexibility of the system lets manufacturers define capable definitions for individual robots.

3. URAPI Demonstration

3.A. Example & System Walkthrough

3.A.i. Scenario Presentation

The following sequence of sample educational robotics scenarios illustrate the operation of the URAPI system and explore the various features of the implementation. The first scenario will walk through the creation and operation of a basic one Input, one Output robot program, while also providing a detailed examination of the function and implementation of the URAPI components. Consider a teacher designing an introductory Predator-Prey activity where students build a simple robot animal that "runs" from the light. The teacher wishes to create a starter program for the LEGO NXT that reads a light sensor and drives the motors forward at a given power.

3.A.ii. User's Corresponding URAPI Code

1) Diagram Components

Presented below, Figure 3.1 is the URAPI universal program that will complete this task. This code is written by the teacher for her NXT robot using the URAPI system in LabVIEW; however, the program is made of simple generic components that could run on any platform. The three main elements used in the URAPI Code are the (1) Universal Devices, the (2) Universal Command VIs (Virtual Instruments), and the native (3) LabVIEW code.

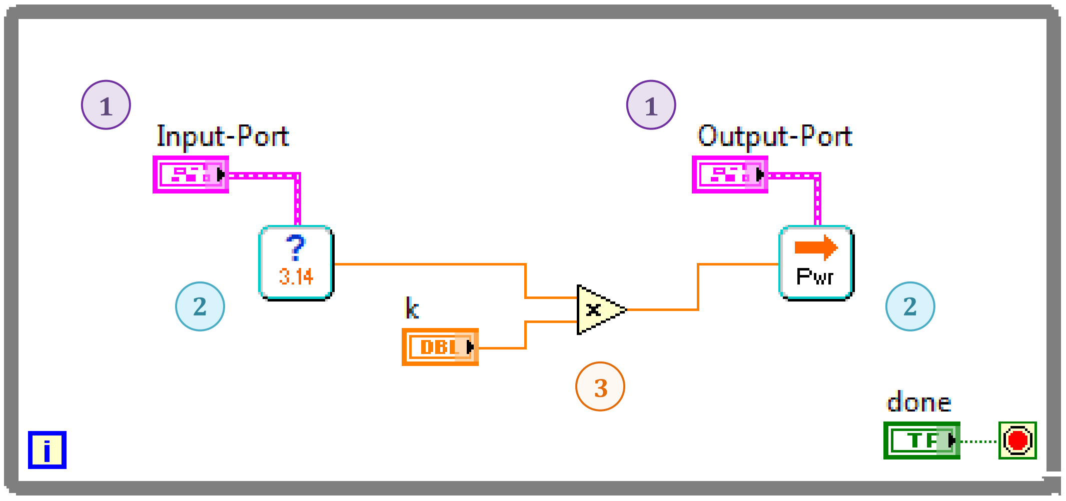


Figure 3.1: URAPI Code for Demo including (1) Universal Devices, (2) Universal Command VIs, and (3) LabVIEW code.

The Sensing and Acting tasks of the teacher's activity are reading an analog Input to sense the "predator warning" signal, and writing to an actuator Output to make the animal run away. These two tasks take the form of the Universal Command VIs (2) Read Analog Input and Motor Power Output, respectively. The parts of the code are also illustrated in more detail in Figure 3.2. For each of these Commands, the teacher adds a Universal Device control (1), one for the "Light Sensor" and one for the "Motor". The rest of the teacher's program is made up of native LabVIEW code (3). This handles most of the Thinking behaviors in the program, including repeating the Input and Output actions and multiplying the gain on the sensor value before using it as the motor power. All of these parts together form the universal behavior that the teacher intends, regardless of the robot chosen by the students. For this example, we will assume that the teacher is using an NXT. If a student runs her program with some other hardware, all the code will remain the same from his perspective, but what happens behind the scenes inside (2) will change based on the new Manifest for the student's hardware.

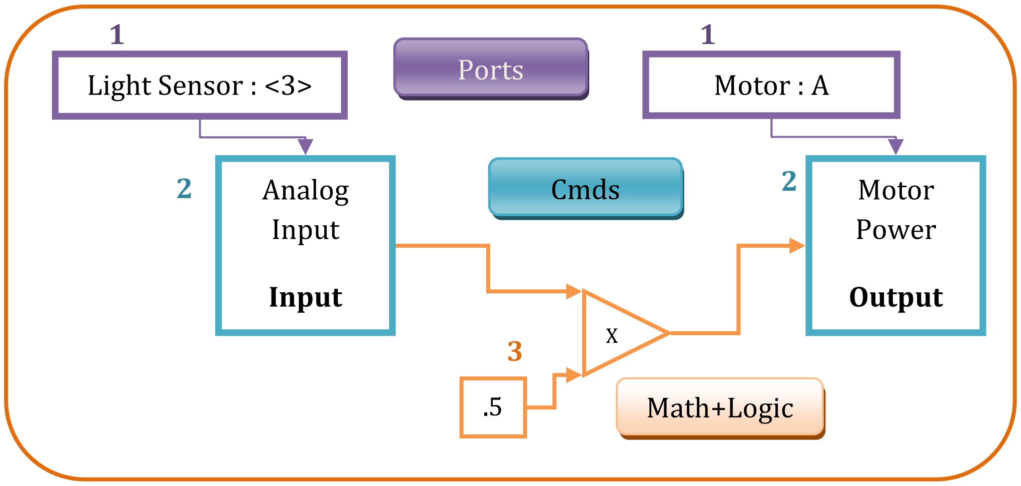


Figure 3.2: Illustration of the same URAPI Demo Code for Demo showing (1) Universal Devices, (2) Universal Command VIs, and (3) LabVIEW code.

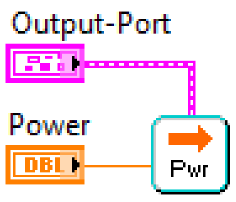
The definitions and execution of the URAPI code for the Sensor task and for the Motor task are nearly identical with the exception of the sensor return value and the motor power argument. In stages of the following demonstration that focus on only one of the tasks, the reader may assume the process is similar, just slightly less interesting, for the task without the relevant parameter.

2) VIs based on the Universal Command

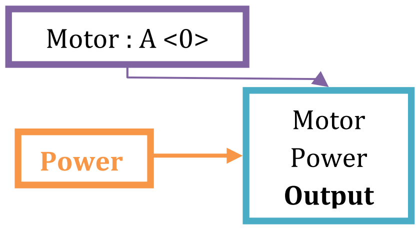
In order to complete the "run away" task, the teacher uses the Motor Power VI, as detailed in Figure 3.3. This VI corresponds directly to the MotorPower Universal Command, and is one of a set of URAPI VIs generated for each Universal Command in the URAPI language. The VI's input terminals and output terminals correspond directly to the Input and Output arguments of the Command, whose bytecode is hardcoded within the VI. The Motor Power VI internally defines the MotorPower bytecode [`0x3302`] and has two input arguments: One is a Universal Port to specify the Actuator to use and the other is a (generic) analog Power to set. The Output Motor Power VI corresponds to the URAPI declaration:

```
MotorPower(U-Device U-Port{Motor, 0, Effector}, double Power);
```

The teacher wires a Universal Device control into the U-Port[4] input to specify which motor on the NXT to move. The default Motor Device for the U-Port argument in MotorPower() allows her to not have to change any values and use the default NXT Definitions. The default values can be seen inside the terminal if the VI is opened.



(a)

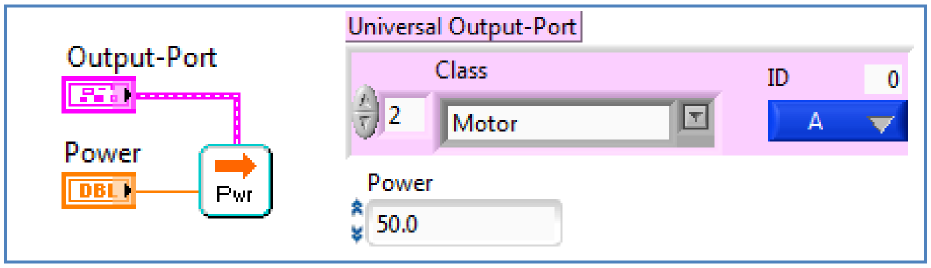


(b)

Figure 3.3: The (a) Motor Power VI, linked to the URAPI Command declaration, with (b) an illustration of components.

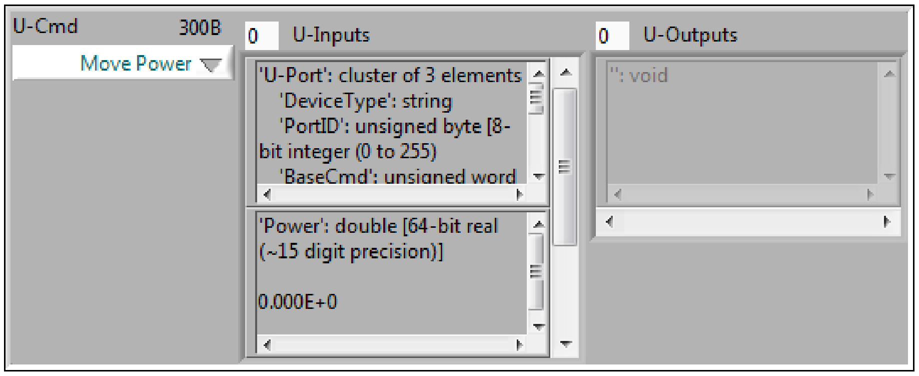
There is a VI generated for each of the Universal Commands in the URAPI Instruction Set. While the user may always use any of the URAPI Commands in any program, the robot hardware developers can setup customized sets of VIs, known in LabVIEW as palettes, for their particular robot; the palettes can include more physically representative icon displays and the most relevant VIs for the hardware. The VIs in the customized palettes still have the same foundation of URAPI Universal Commands and simply serve as a 'skin' or superficial covering for the URAPI Commands underneath.

[4] U-Port == Universal-Port; Likewise, U-___ in any code label is short for Universal-_____ or URAPI-_____. If there is an 'R' instead of a 'U', it differentiates the Robot version of the object as opposed to the URAPI or universal version.



Universal Output-Port
Class 2 Motor
ID 0 A
Power 50.0

(a)


U-Cmd 300B
Move Power

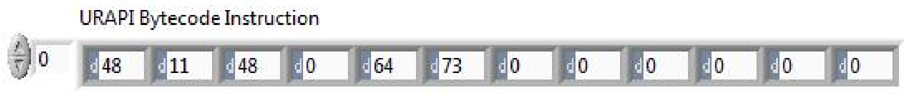
0 U-Inputs
'U-Port': cluster of 3 elements
'DeviceType': string
'PortID': unsigned byte [8-bit integer (0 to 255)
'BaseCmd': unsigned word

'Power': double [64-bit real (~15 digit precision)]

0.000E+0

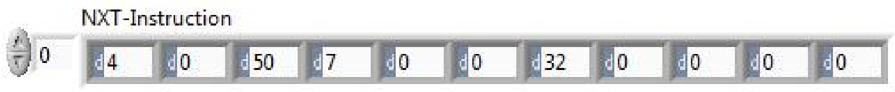
0 U-Outputs
'': void

(b)


URAPI Bytecode Instruction

0 | 48 | 11 | 48 | 0 | 64 | 73 | 0 | 0 | 0 | 0 | 0 | 0

(c)


NXT-Instruction

0 | 4 | 0 | 50 | 7 | 0 | 0 | 32 | 0 | 0 | 0 | 0

(d)

Figure 3.4: The progression from (a) user VI and controls, to (b) URAPI Command, to (c) bytecode, to (d) robot.

Universal Input and Output Arguments

The Read Analog VI always returns a double precision number, which lets the teacher use it directly as an input to the Motor Power VI that also takes a double for the Power argument. These generic-analog "Universal Values" enable compatibility by always having a double type and value within the range of [0.0, 100.0], or [-100.0, +100.0] in the case of the Motor Power input. By ensuring the same format of the inputs and output, URAPI VIs can transfer across robots with different bit-size data representations. In other words, [0.0, 100.0] will be the whole range of possible analog output values on the NXT, which uses integer values [0, 100] as motor speeds, as it is on the Arduino that sets [0, 255] for PWM duty-cycles. Similarly, the [0, 1023] integer values returned from the robots' 10-bit Analog-Digital input converters will be scaled to [0.0, 100.0] so these values can directly feed into generic-analog outputs. The generic-analog Universal Value essentially implements a normalized, or "percentage", representation of the sensor's value for simple compatibility between different devices.

All the Universal Commands have strict data types for the inputs and outputs and common ranges for these values that all devices scale to or from, thereby letting the user count on the expected meaning of a given "numeric" value because she knows that "0.50" means 50% of the sensor's possible range, regardless of hardware. Since it is sometimes desirable, even at introductory levels, to retain the scales of the sensed phenomenon, Devices such as a distance sensor that reads centimeters are often used with the Count-based Universal Commands whose implied function is to return an answer to questions of "how much?" The current implementation limits the specificity of the value to the data type of the variable. While it would be possible to define and use units, the general types allow easier compatibility between robots, especially the less advanced platforms in classrooms that often do not have precise unit-based sensors. The VIs corresponding to each Universal Command have predefined input and output terminals for each variable, which enables LabVIEW to handle all variable declarations.

3) Universal Device Port Input

a) Class (Function)

Once the teacher has created the basic Input and Output Commands, she further defines the behavior of the Analog Input task in a Universal Device. While the Universal Port inputs of the VIs provide default Device Classes capable of mapping to compatible transducers on the NXT, the teacher more precisely identifies her Light Sensor by using the Universal Device control seen in Figure 3.5. The teacher creates a Universal Device control off of the default U-Port argument and uses the Class field to describe the prey's "threat warning signal". The default Class for the U-Port is "Analog Input", which is built on top of the "Input" base Class. When the user adds a Universal Command, the Device Type associated with that URAPI Command is used as the starting value of the Class. The teacher can then further describe the functionality of the Universal Device by building a hierarchy of Classes on top of the defaults.

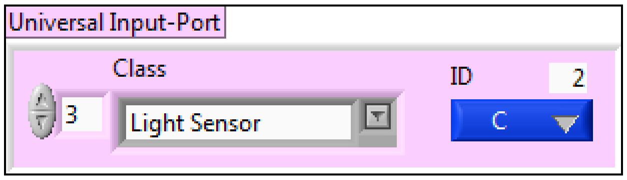


Figure 3.5: Light Sensor Universal Device

The teacher can specify here to use a sensor that senses light brightness by inputting the "Light Sensor" class. For this NXT example, this means that the NXT light sensor will be plugged in and used to gather this brightness value. If another robot, say, the Finch, were used, it would instead use its built in light sensor. The teacher can also specify an intermediate class, the "Intensity Sensor." If a student in the class wanted to have his prey robot run away from loud sounds and therefore built his robot with a sound sensor instead of a light sensor, the URAPI code would detect that there were no light sensors, fall back to the Intensity Sensor class, and choose to use the sound sensor instead.

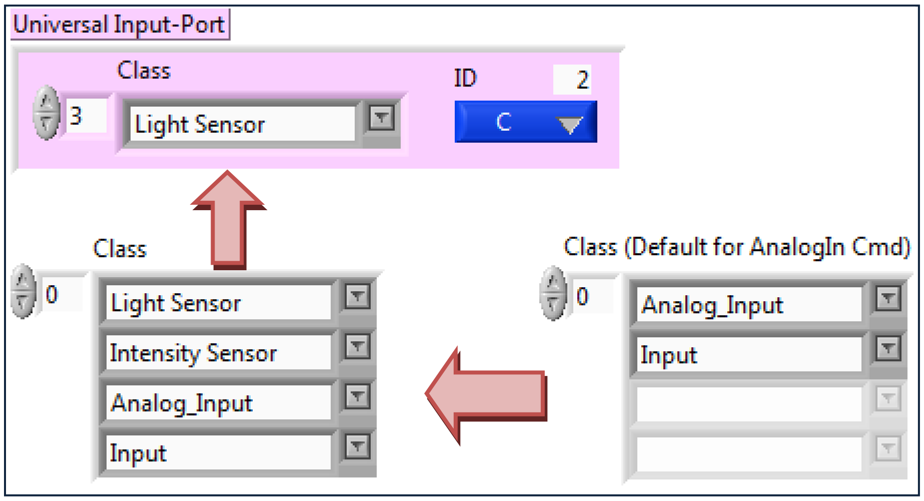


Figure 3.6: The Light Sensor Universal Device with the teacher's added "threat warning" Classes and the Default Classes provided by the Analog Input Command.

To do this, the teacher takes the default Analog Device control and adds a new line to the top of the Device Class multi-line string with the words "Light Sensor." Now, when the URAPI system matches the control to the Universal Devices on a robot, the system will first look for Universal Devices with the "Light Sensor" functionality defined in their Device Class. As seen in the depiction of the Device Class functionality chain in Figure 3.6, the teacher is able to add multiple layers of contingency to her functionality definition by also inserting the line "Intensity" after "Light Sensor."

b) Identity (ID)

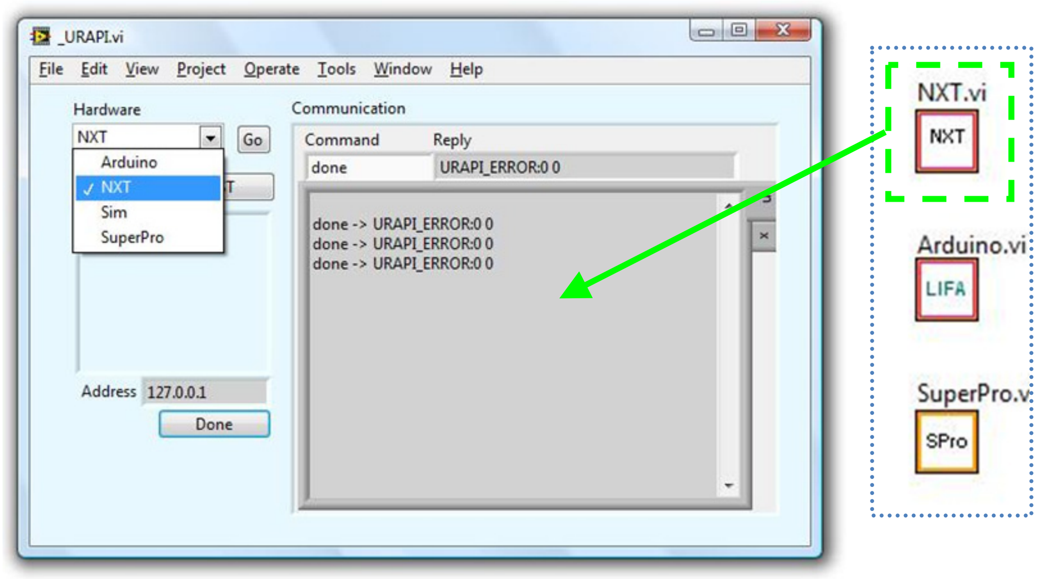
Along with the functionality of the Universal Devices, the teacher assigns a numeric tag, or ID, to the specific Device in order to identify it in the future. In this case, she only has one instance of each Universal Device; however, if she were to add a second light sensor the ID number would be used to differentiate the two sensors.

Writers of software can link unique text names to the ID field so that they correspond to hardware port labels on the current robot (e.g. ‘Right Button’ on the NXT, or ‘Nose LED’ for the Finch). In the simple case, the value of the ID corresponds to the list of Universal Devices that match the user’s Class specifications. For the teacher’s “Light Sensor”, she uses an ID of ‘2’ to simply indicate she has plugged her Light Sensor into the NXT’s Input “Port 3” (indexed starting at 0).

3.A.iii. Program Execution

1) Select Robot Hardware for Loading Definitions

The teacher can select the NXT robot as her target hardware using the URAPI Server menu Figure 3.7. This action will load all the target definitions for the NXT that include the supported Devices, the NXT Dictionary and Look-Up Tables, and the NXT implementations of dynamic robot targeting VIs, such as the communication handler.



```
done                URAPI_ERROR:0 0

done -> URAPI_ERROR:0 0
done -> URAPI_ERROR:0 0
done -> URAPI_ERROR:0 0
```

Figure 3.7: Target Selection on URAPI Server loads NXT custom code and Definitions.

In advanced software applications, once the NXT is targeted, the Port Configurator and other background processes become aware of NXT hardware capabilities and can update elements like the ID control or the VI palettes to present NXT oriented labels and VIs. These processes demonstrate how a dynamic, specialized environment may be built on top of the universal URAPI platform. In the current implementation, NXT Definitions may be viewed by the user during Edit time once the NXT is targeted; however, matching is done in real-time during execution.

2) Running the User Program

When the teacher plugs in her NXT and hits the "Run" button, she initiates the main stage of turning the universal code she wrote into NXT targetable code. That is to say, the majority of what she has created so far has no direct connection to the NXT language or robot. The code is translated and adapted to the NXT in real-time by a backend URAPI Server where all robot-specific operations are handled. The Universal Command VIs, Analog Input and Motor Power, pack the Commands and arguments into the URAPI bytecode instruction format and then use a core intermediary VI to communicate with the URAPI Server via TCP/IP.

3) VI Converted to Bytecode Instruction Format

Inside the teacher's Universal Command VIs the Universal Device and Power (for MotorPowerOutput()) arguments are combined with the URAPI Command bytecodes to form URAPI Universal Instructions. When the program executes, the VIs will flatten all the input values into a standard byte format for sending the Universal Instructions to the URAPI server. The goal of the top-layer of the block diagram – the primary view of the user – is to hold all the universal code and allow the user to define hardware independent behaviors; once inside the VI, the transition begins from universal definitions towards context dependent robot actions and hardware. The Commands and Instructions are still in a generic format, but the Devices are matched to the robot's capabilities.

The inside of the Motor Power Output VI, shown in Figure 3.8, illustrates the overall steps taken by all Universal Command VIs: Universal Ports feed into the Port Registry to be matched to unique NXT-supported Devices; the (Motor Power) Command Bytecode and all Inputs are packed into a Universal Instruction packet; and the Universal Instruction is sent to the URAPI Server. The Instruction is sent to the URAPI Server using a specified Wait Behavior and expected Return Size. In the case of the Analog Input Command where the NXT (via the Server) responds with a sensor value, the flattened return value in the readBuffer is cast back to a double precision Universal Variable.

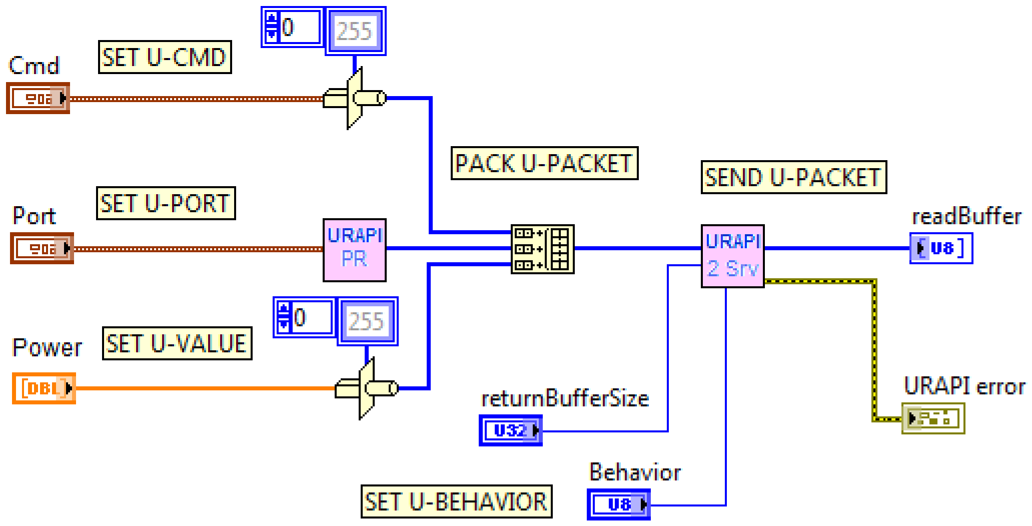


Figure 3.8: Inside the Motor Power Output Universal VI

4) Port Registry

a) Class Matching to Full Universal Device Definition

The first step in generating code the NXT understands is matching the functional description of a light intensity sensor Universal Device in the U-Port of the teacher's Read Analog Input Command to a unique, supported Universal Device definition on the NXT. When the teacher added the "Light Sensor" Class specification, she was describing the purpose of the "warning signal" sensor and the topic of information the sensor should provide (the brightness of the light seen by the robot), but she was not mandating the specific hardware or even Device Type of the sensor that the NXT should use to read the light level.

Inside the Analog Input VI, the teacher's Universal Port specification feeds into the Port Registry, which sits alongside the URAPI Server and is loaded with the NXT manufacturer's hardware Manifest; the Manifest includes a list of Universal Devices defined for each of the different sensor and actuator Types the NXT supports. Among other information, each Universal Device in this list has an attached Device Class hierarchy. The Port Registry reads the top-most Class "Light Sensor" from the teacher's U-Port input, and then searches for a Device matching that Class specification in the set of Universal Devices defined for the NXT. In this case, the Port Registry matches the definition for the 'NXT Light Sensor' Universal Device, which also has a top-level Device Class of "Light Sensor" Figure 3.9. Once a device match has been made, the Port Registry now uniquely identifies the teacher's Block Diagram "Light Sensor" control as the 'NXT Light Sensor' Universal Device on Port 3. The Port Registry registers this unique Universal Device with the run-time Port Registry Table and passes out a transformed Universal Device that now defines the Device Type and an identifier.

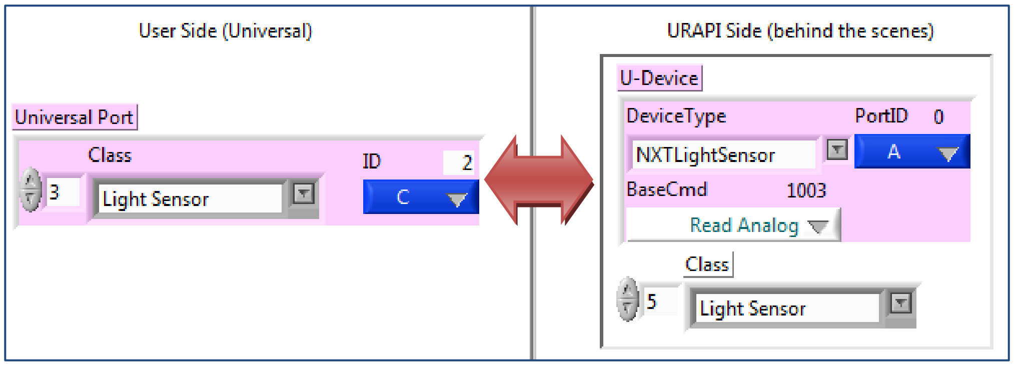


Figure 3.9: Port Matching User Universal Specifications to an available Universal Device from the NXT Robot Definitions

b) Device Type

The Universal Device Definition for the 'NXT Light Sensor', seen in Figure 3.10, shows that the sensor returns Analog values via its Base Command, ReadAnalog(), but has a unique Device Type of `NXTLightSensor`, rather than the default `Analog_Input` Type; this is because the NXT uses a specialized setting for interpreting the analog voltages as relative light intensity. This form of the

Universal Device is focused on the properties needed to uniquely identify the full Universal-to-NXT translation. Every Device Type used in the program will have an entry in the Robot LUTs or custom stored in the Port Registry. The Port Registry takes care of storing some functions like SetupDevice Commands.

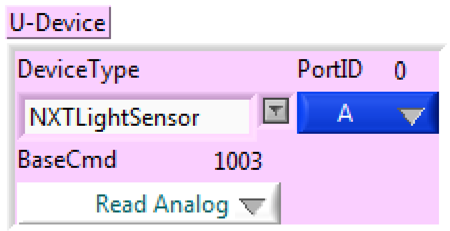


Figure 3.10: NXT Light Sensor Universal Device

There are other attached Device properties that are inherently defined in the system, such as compatible hardware slots and custom function definitions, and will appear later in the process. These properties are predefined by the manufacturer of the robot, though the user has the option to configure their own, building on top of the base and standard device types. The teacher is using the Universal Device for the physical NXT Light Sensor which is one of the included standard Devices provided with the NXT Definitions. These standard Devices are intended to cover all the different methods (functions) the robot has to get values from a sensor. For instance, the NXT has different native functions for reading the states of the integrated user-interface buttons than the functions it uses to read an external Touch Sensor attached to a Sensor Port, even though they are both digital inputs read using the URAPI Digital Input() Command. Note that the purpose of the Device Type is found mostly in the translating and identification of the Device that the given robot supports. Thus, this form is used to pass the Identifier for the Universal Device to the Server.

5) Communication of Instruction with URAPI Server

After processing the Universal Port input, the VI flattens the values of all the arguments into a plain byte buffer format. For the teacher's Motor Power Output Command, the format of the packed values begins with the 16-bit Motor Power Bytecode, the Motor Device type and instance identifier, and then the flattened double for Power. The example values for a Power of 50% are seen below.

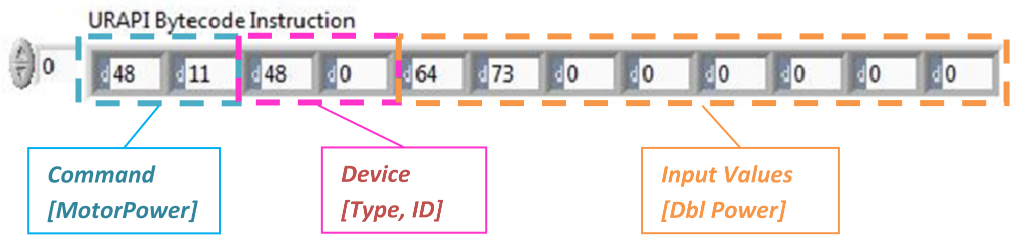


Figure 3.11: Bytecode Command and flattened Inputs in byte value representation.

This buffer is then sent from the user's executing program to the backend URAPI Server via the Server Communicator VI, URAPI.Server2Cmd.vi. The Server Communicator handles all the communication between universal programs and the URAPI Server using a standard instruction protocol inside TCP/IP packets. Using LabVIEW reentrancy allows communication with the server to be contained within one functional block of code inside all Universal Commands while top-level VIs are still able to execute in parallel. The Command VIs can stipulate whether to immediately move to the next instruction in the program or Wait for a reply from the Server. In the case of the Read Analog Input Command, the VI will wait for the expected return Value read from the Light Sensor. The protocol specification of the URAPI ReadAnalog Command declares the single return variable and its type of Double, so the VI can simply cast back the expected bytes and output a LabVIEW Double. Table 3.1 shows the template of the server protocol. The actual Bytecode

Instruction is contained in the grayed out section. The first two fields specify the waiting behavior and the byte-size of the return values.

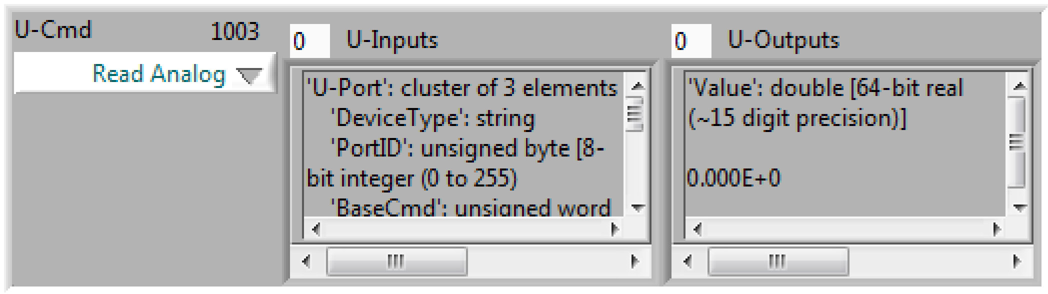
[Behavior]x 1	**(Opt: [Return Size]x 4)**				**[Command]x2**		**(Opt:[Port]x2)**		**([Values] x*)**
WaitResponse	Hi.Hi	Hi.Lo	Lo.Hi	Lo.Lo	Class	Specifier	Type	ID	*Bytes

Table 3.1: URAPI Server Communication Protocol

3.A.iv. Server Processing

Reading from and writing to the URAPI Server is done with simple subVIs that handle sending packets with length headers over TCP/IP between the user code and the Server. The TCP/IP packet simply allows the packed URAPI Bytecode Instructions to be sent as complete messages. Thus, when the command for Motor Power Output is sent to the server as a string of flattened values, the URAPI VIs on the server are able to unpack the first two bytes to indicate the bytecode for the MotorPower URAPI Command. By looking up the declaration for the MotorPower Command in the URAPI Dictionary, the rest of the buffer can be parsed properly into data typed variables. Parsing the variables ahead of time merely allows the server to structure the indistinguishable buffer of bytes into Values with quantitative meaning that map directly to arguments of a URAPI Command.

The URAPI server now has a complete, strictly-typed, and value-filled Universal Instruction, which should have a single corresponding translation into a complete NXT instruction. The LabVIEW Variant Array provides a common container for all the different Instructions. For Input Commands like ReadAnalog, shown in Figure 3.12, the URAPI Dictionary entry also provides the return format of the Universal Output Variables.



'U-Port': cluster of 3 elements
'DeviceType': string
'PortID': unsigned byte [8-bit integer (0 to 255)
'BaseCmd': unsigned word

'Value': double [64-bit real (~15 digit precision)]

0.000E+0

Figure 3.12: Universal Read Analog Input Command in Variant Format

3.A.v. Target Robot Handlers

Up to this point, the URAPI Server has been performing NXT-independent operations and using VIs that are always the same. The next set of processes and VIs are NXT-dependent, and will vary with the target robot chosen. The NXT implementation is shown in Figure 3.13. There are three general steps each Command goes through. The first step is to lookup the translation for the Universal Instruction in the URAPI-NXT Look-Up Table (LUT), which includes the NXT opcode and how to convert the Universal arguments to variables in the NXT command. Once the Instruction is full translated into an NXT instruction, the Command must be sent to the NXT using the Robot Communicator NXT.Communicator.vi. The final process of the target robot-related code is to translate the bytes returned by the NXT, back into Universal Values, for each of the Command's Output Variables. The Outputs are sent back out to the teacher's code and the execution of the URAPI program continues.

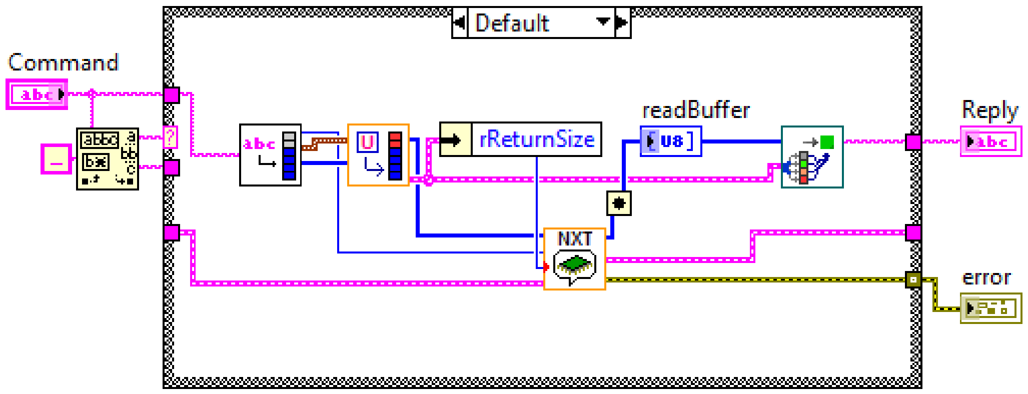


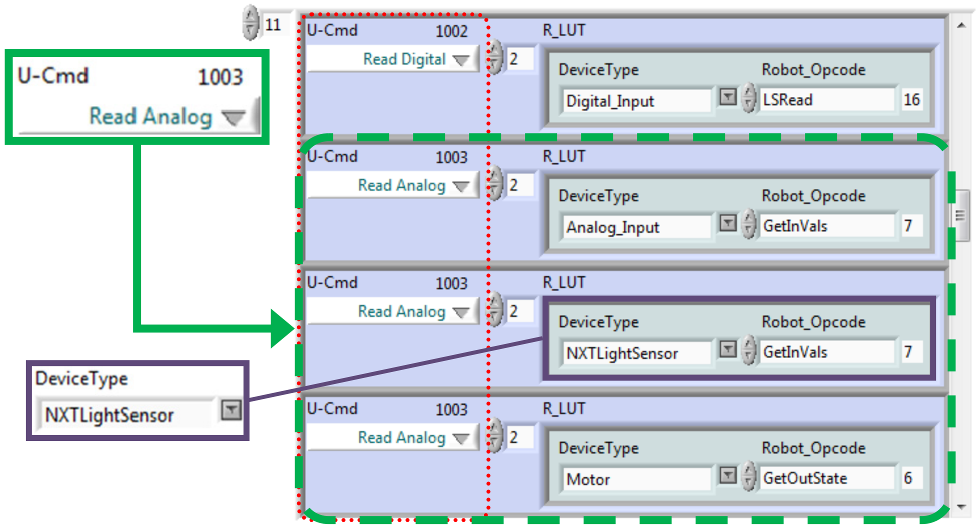
Figure 3.13: Target Robot Translation Code

1) Robot Translator – Look-up Translation in NXT LUT

a) By Command

The translation of the Universal Read Analog Instruction into an NXT instruction begins in the NXT's 'Robot Translator' VI, NXT.Translator.vi. The Translator indexes the LUT of NXT definitions of URAPI Commands, first by Universal Command, and then by Device Type, until it finds a unique entry. The entry will tell the Translator if and how the NXT supports the Universal Instruction, including the NXT-specific command to call, how to convert the Universal Variables to the proper range and format for each of the NXT command's arguments, and how to extract return value from the NXT command and correctly convert it back into a Universal Output Variable. In the event that the Command is not supported, the Server is able to send back specific URAPI_ERROR information that simply allows custom error code transferring.

The Translator searches for the proper translation entry in the URAPI-NXT LUT by first looking down the ordered list of URAPI Commands for the ReadAnalog Command Bytecode. Since the LUT is sorted by Bytecode hex value, this is a relatively straight forward process to linearly find the first entry equal to the hex value of ReadAnalog, `0x1003`, as depicted in Figure 3.14. In addition, since there will often be multiple entries defined for a single Universal Command, sorting by Bytecode value means that all possible translations can be found consecutively.



U-Cmd 1003
Read Analog
DeviceType
NXTLightSensor
11
U-Cmd 1002 Read Digital 2 R_LUT DeviceType Robot_Opcode Digital_Input LSRead 16
U-Cmd 1003 Read Analog 2 R_LUT DeviceType Robot_Opcode Analog_Input GetInVals 7
U-Cmd 1003 Read Analog 2 R_LUT DeviceType Robot_Opcode NXTLightSensor GetInVals 7
U-Cmd 1003 Read Analog 2 R_LUT DeviceType Robot_Opcode Motor GetOutState 6

Figure 3.14: Universal Command indexing beginning of available ReadAnalog Commands in NXT LUT.

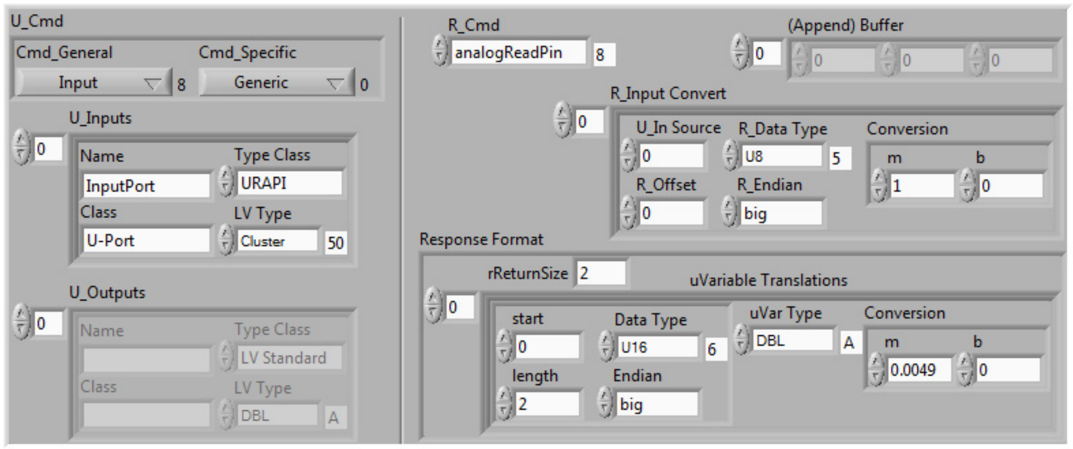
b) By Device

The multiple entries for ReadAnalog exist because the NXT has a number of different sensors that can give analog readings, and the NXT uses different native functions for reading different sensors and will return different value ranges depending on the sensor. In other words, it is unlikely that all the possible analog sensors on a robot use the exact same instruction and return the same range of values – that is half the point of URAPI.

In order to use the proper command and format, the subset of ReadAnalog Commands are then themselves indexed by the `NXTLightSensor` Device Type. In this way, it is the combination of both the Universal Command Bytecode and the Universal Device Type that defines a unique translation on any robot. As seen in

Figure 3.15, the `NXTLightSensor` has its own ReadAnalog entry in the NXT LUT, with its own set of conversions.

The entry shows that the ReadAnalog(`NXTLightSensor`) Universal Instruction is supported directly with an NXT instruction. Direct Robot translations specify a corresponding robot command to call - including the native opcode and corresponding parameters - and the scaling functions to convert Universal Variables into valid arguments for any variable parameters in the NXT instruction. The Direct Robot translation for the ReadAnalog command uses the NXT command GetInVals (`0x17`) which takes the NXT Input Port to read as a parameter. The NXT Port user argument happens to also be the only parameter, so the LUT translation defines one conversion for the Port (variable) parameter and an empty (constant) parameter buffer.


U_Cmd
Cmd_General: Input 8
Cmd_Specific: Generic 0
U_Inputs: 0
Name: InputPort
Type Class: URAPI
Class: U-Port
LV Type: Cluster 50
U_Outputs: 0
Name:
Type Class: LV Standard
Class:
LV Type: DBL A

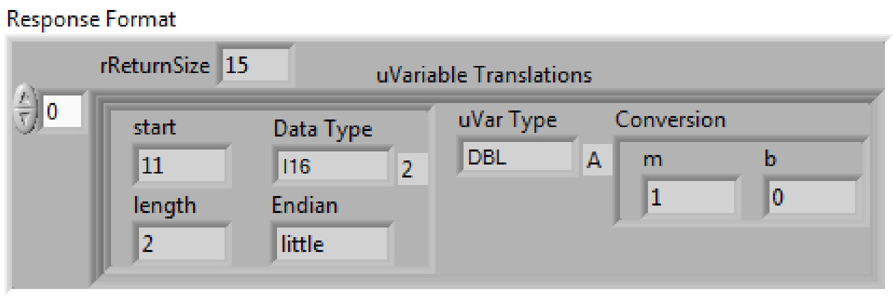
R_Cmd: analogReadPin 8
(Append) Buffer: 0 | 0 | 0 | 0
R_Input Convert: 0
U_In Source: 0
R_Data Type: U8 5
R_Offset: 0
R_Endian: big
Conversion: m 1, b 0
Response Format
rReturnSize: 2
uVariable Translations: 0
start: 0
Data Type: U16 6
length: 2
Endian: big
uVar Type: DBL A
Conversion: m 0.0049, b 0

Figure 3.15: ReadAnalog(`NXTLightSensor`) entry

Note that not every Device Type must be specified like this for every Command, just the lowest common denominator of devices that take different commands and formats to read or on the NXT. Device Types not defined for a specific Command can still be used through the function of the final property of the Universal Devices, the Base Command Type. The URAPI system can automatically call the Base Command used to read the Device (via the Port Registry where it is stored), and index the LUT for the original Command by the Base Command Type, which is equivalent to the Default Device Type for the Base Command.

The value of '1' in the Source field indicates that the conversion for the Port argument is applied to the first Variable of the URAPI Instruction, the "Light Sensor" Universal Device. When Universal Devices are indicated as the Source Variable, the Port Registry takes over most of the conversion and the only meaningful values in the rest of the conversion definition are those required by all conversions; this includes the data type and endian (if not implicit) for the argument, where in the parameter buffer to insert the argument, and the Source Variable to apply the conversion to. These values are all outlined in the entry in the diagram Figure 3.15: ReadAnalog(`NXTLightSensor`) entry. The more common and interesting URAPI Input-NXT Input conversions occur primarily in the Setup Device and 'Act' Commands, such as for the Power argument in the Motor Power Output Command.

Input commands also define reverse conversions for the values returned by the sensor. The reverse conversions indicate the relevant bytes in the robot's return buffer and the scaling constants to use in order to format the values back into proper Universal Output data types. The corresponding conversion factors for the ReadAnalog Command are shown in Figure 3.16: ReadAnalog(`NXTLightSensor`) return translation below.


Response Format

rReturnSize	15
index	0

uVariable Translations

start	11
length	2
Data Type	I16 (2)
Endian	little
uVar Type	DBL (A)
Conversion m	1
Conversion b	0

Figure 3.16: ReadAnalog(`NXTLightSensor`) return translation

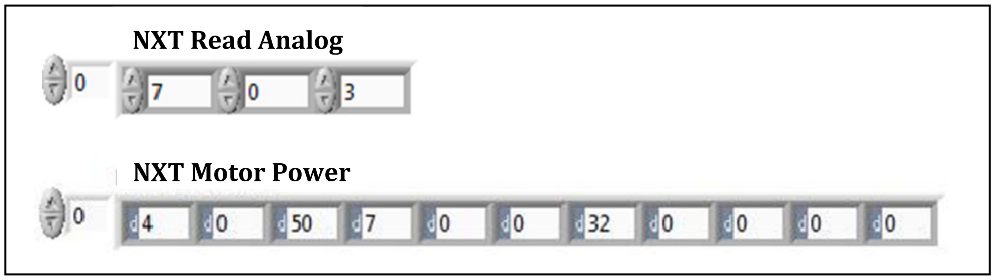
c) Port Registry

The last background process that happens at this stage is a check through the Port Registry to see if the "Light Sensor" Universal Device has been called and setup previously. Since this is the first call, the Port Registry implicitly calls a SetupInput Instruction for the "Light Sensor".

2) Apply Conversions

The Translator does not actually execute the conversions on the Universal Arguments. This is done separately through an intermediate process when the translation command, in this case the robot instruction, is called. The Translator passes out the translation information and the conversions and once the Command is called, the conversions are applied to produce the final, completely translated instruction.

The translation of the Power in the MotorPower Command is relatively simple in this example, as it is just a 1 to 1 conversion since Motor Power and the NXT Power both use values within the range [0, 100]. When translating the "Light Sensor" Universal Device to a physical Input Port on the NXT, and likewise a value for the 'Port' parameter in the NXT instruction, the URAPI system triggers a special case that uses the Port Registry to process and return the proper values. This is essentially just a special way of applying conversions to a Universal Input if it happens to be a Universal Device. The complete translated instructions for both the ReadAnalog and MotorPower Commands are shown below in Figure 3.17.


NXT Read Analog

0	7	0	3

NXT Motor Power

0	4	0	50	7	0	0	32	0	0	0	0

Figure 3.17: NXT final translations for both commands

3) Robot Communicator – Execute Command

At this point the URAPI Universal instruction is completely translated into a full NXT bytecode instruction. The URAPI Server passes the buffer of bytes to custom code unique to the NXT that will handle all communication with the physical robot. This VI, called the Robot Communicator, is implemented uniquely for each robot as every robot uses different physical means of communication. All URAPI needs to do is pass along an instruction the robot can understand along with the "Wait" Behavior for returning values, and the manufacturer can handle all the mess of communicating with their robot. For the Read Analog Command, the return variable Value specifies that the URAPI Server will wait for a response from the NXT Communicator.

While LabVIEW implements the Virtual Instrument Software Architecture (VISA) API [40] to serve as a common mechanism to handle a range of communication protocols, many robots require unique communication patterns and formats. Allowing robot-specific communication code means that manufacturers can handle all the eccentricities required to guarantee a reliable connection to the robot, while the URAPI system does not need to become over complicated.

4) Robot Rx Translator – Receive and Convert Return Values

Once the NXT Communicator passes back the received buffer communication, the return relevant bytes in the robot-specific format must be extracted and converted back into an Analog Double for the Input Value. The buffer returned from the NXT is shown below. Similar to the NXT instruction translations, there are bytes and fields particular to the NXT that are not directly relevant to the return value. Along with the NXT function to use, the NXT LUT entry for ReadAnalog() also specifies the location where the return variable's value is stored, in the 11th and 12th bytes of the expected return buffer; the format of the byte representation as a little-endian U16, and the conversion factors to use to scale the Unsigned Integer value to a Double-precision number. The URAPI Receive-Translator simply uses the location to extract the bytes for each Universal Output Variable, and the format to convert the bytes into a LabVIEW U16 Integer Figure 3.18. Then the Receive-Translator uses the conversion information, to turn the Unsigned Integer into a number from [0, 100] that corresponds to the Universal Output Value that will be passed back to the universal program.

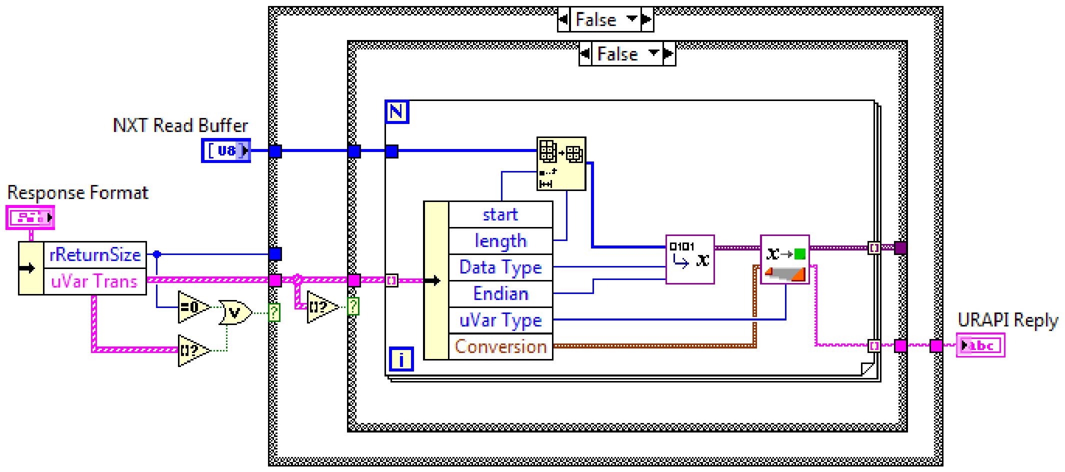


Figure 3.18: Robot Rx Translator

5) Send back to User Code VIs (URAPI)

The final stage of the URAPI Server is to take the Output Value and send it back to the user's program. The variables are flattened and packed along with any errors that may have occurred during the Translation and Communication process, and then sent back down using the same TCP/IP VIs. The program-server communication VI then passes out the flattened bytes of the variable, and the URAPI Command VI simply casts the bytes back to the expected data type and outputs the Value as a LabVIEW Double precision number.

3.B. Basic Switching Hardware

To demonstrate how the URAPI system is able to smoothly transfer the same universal program to new hardware, consider the situation of the teacher in the first scenario, now giving her code to a student who is using a Finch robot. The Finch has only hard-wired sensors and actuators that are controlled through device-specific commands. The student is able to run the same exact program, despite these differences.

3.B.i. Finch Code

The program displayed in Figure 3.19 below is the URAPI code once the student targets the Finch. The user front-end VIs and controls are all the same; however, the Finch Definitions are loaded and used to match different functions and components on the robot.

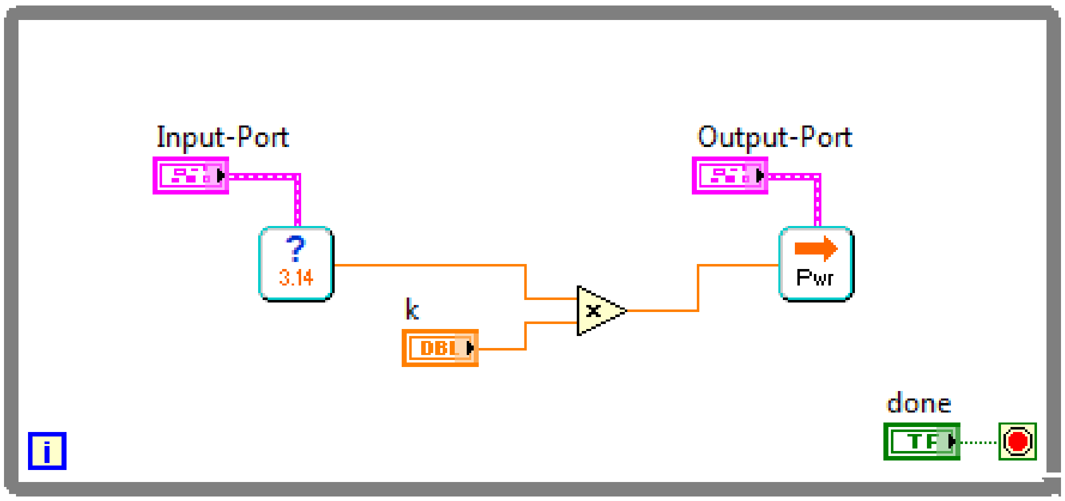


Figure 3.19: URAPI Code targeted to Finch.

3.B.ii. Devices

When the student runs the program, initial execution proceeds exactly as described previously; the first difference occurs inside the Analog Input Command VI when the system needs to match the Universal “Light Sensor” Device to available sensors on the Finch. Since the Finch obviously does not have the “NXTLightSensor” Device Type, URAPI uses the Class definition of the Universal Device to find an appropriate sensor. In this case, URAPI is able to match the first specification of “Light Sensor” directly to the Universal Device Type for the photoresistor light sensors on the Finch. Similarly, the NXT Motors used originally are now matched to two wheels of the Finch.

3.B.iii. Commands

The next difference occurs in the actual functions to use for reading Analog Inputs and writing out Motor Powers. For the Finch, the functions are closely tied to the hardware device used as there are many analog inputs on the robot, each with their own command. When URAPI indexes the Finch LUT for the Read Analog Command and finds multiple entries, it will then use the Device Type to find the specific Finch command to read the value of the light sensors. This command, as shown in Table 3.2, is indicated by the bytecode character 'L' and will be the entirety of the instruction sent to the Finch Communicator.

Universal Command		Finch Command	Device Type
Read Digital	1002	I	Digital_Input
Read Analog	1003	A	Analog_Input
Read Analog	1003	T	Temp_Sensor
Read Analog	1003	L	FinchLight

Table 3.2: Finch LUT

As also indicated in the Finch LUT, the command will return two bytes, one for each sensor. This is where the Port Registry is able to specify not just the conversion factors but which byte contains the relevant information about the specific Light Sensor Device Type being used. In this case, the left light sensor was matched to the instance of the Universal Device and thus the Port Registry indicates that the first byte returned should be extracted and converted to a Double precision number for the Output Value.

In the end, the student has the same Universal Commands and Universal Devices as the teacher, but the URAPI system is able to use different commands when running on each robot. Where the Light Sensor command would normally take a completely different communication format, opcode and even translation of inputs and outputs to comparable values, these are all handled in the robot definitions and the standard URAPI commands, so the teacher's code doesn't change a bit.

3.C. Indirect Command & Device Handling

Macro Commands & Hierarchical Definitions

In cases where there is not a clean translation between two robots, certain adaptations are made to allow the code to run as coherently as possible. Simple cases may exist where the user wants to adjust one of the automatic URAPI matches. For instance, the student in the last example, may have expected to use the right light sensor instead of the left, and could easily configure the program to the Finch's exact hardware. The more significant cases are when the URAPI system automatically accounts for differences between complex and basic systems.

Take for example a user who writes a URAPI program for the iRobot Create. One of the native functions to move the Create around is the Drive Direct command that allows the programmer to set velocities of both wheels. If the user writes a program using the Universal Command Move Motors, which provides similar functionality, then the Create can use its single corresponding function directly. If this user then wants to transfer their code over to an SRV-1 based robot, issues arise as there are no functions for controlling the velocities or movement of external motors. Thus, URAPI provides Macros for the manufacturer, so that on the SRV, the Move Motors Universal Command can be translated into a couple of lower-level Pulse-Position Modulation (PPM) Output functions on the SRV Figure 3.20. In this way, users can still transfer robotic code between the platforms while maintaining the same important behavior of the robot.

Move Command on Create:

[Drive-bytecode, Left Velocity (mm/s), Right Velocity (mm/s)]

Move Command on SRV-1:

[Write i^2c Address (`motor controller`), Pin (`left motor`), Duty Cycle (`left velocity`)]
[Write i^2c Address (`motor controller`), Pin (`right motor`), Duty Cycle (`right velocity`)]

Figure 3.20: Move Command Macros

3.D. Conclusion

The URAPI system enables the teacher to write code that is entirely universal and defines the robotic behaviors by meaningful tasks. The implementation demonstrated here shows how the backend URAPI system is then able to handle all the robot-dependent communication and translation whether it is an NXT, Finch, or Create being used. By using the URAPI Server as a middle-man, the robots can be swapped in and out with no difference to the teacher's code or its effectiveness. The Server is able to automatically make sure appropriate commands are sent and are suitable for devices used. When there is a compatibility inequality, there are macros and generic functions for graceful adaptation, based on keeping as much of the meaning and purpose of the teacher's activity as possible.

The current implementation of the URAPI system is only a basic version to prove the fundamental concepts and exhibit the core components in use. The set of URAPI Commands implemented in this thesis (see Figure 2.11) is designed to capture the majority of robotic behaviors in educational activities at the broadest level. While basic Commands like Read Analog may not take full advantage of complex hardware such as a Color Sensor, the goal of the implementation set is to provide a working base set of universal Commands that enable functionality of some form across robots. Future development of the URAPI Command set will be able to build upon this base set to further explore meaningful branches of simple robot behaviors, such as the initial development on the Move and Display/Indicate Universal Command categories.

4. Conclusion

The Universal Robotics API design presented in this thesis demonstrates an effective system for allowing teachers and other users of simple robots to write universal programs capable of being transferred to any URAPI robot. By implementing the abstracted interfaces of the URAPI layer on the basis of robot function and behavior, programs that are hardware-independent can be written using Commands based on basic robotic tasks.

In this way, the URAPI Commands provide a simple means of cross-robot compatibility for teachers interested in the benefits of robotic systems as a teaching aid. Abstracting by the tasks to be performed allows robotic programs to be generalized without losing the important relationships between the programming and the robots physical interaction with the world. Even using the most fundamental low-level URAPI Commands demonstrated in the examples, the teacher is able to create basic robotic activities that get at the main concepts of their curriculum. The URAPI Command set further enables the transference of this curriculum to other teachers and robots because it allows tasks to be completed by individual robots in the way best suited for that robot's particular capabilities.

The classification features of the URAPI Universal Devices allow users to not only describe the robot behaviors, but also describe the purpose of the tasks in a more meaningful way. The hierarchical Device Class list enables users to describe this purpose in a way that ensures the greatest success across platforms. While other possibilities exist to provide standard designations for hardware connections, enforcing a set layout or numbering across all robotic platforms would complicate matters for manufacturers and diminish the benefits of unique robotic systems. A system that only used numeric identification for input and output ports would require users to independently determine the purpose of each port and reconfigure every program for each robot. Since there are a number of different phenomena that can be measured with any given sensor, and a number of sensors that can give information about any given event, even using the type of sensor alone would not provide the clean adaptability desired in a universal system. Ultimately, there will need to be a larger, standard set of common functionalities defined so that users can speak with a common vocabulary. Open-source contributions could even enable a growing, relevant library of terms.

While successful implementations based on the lowest-level of robotic tasks were demonstrated in this paper, extending the hierarchy of the URAPI Command set to higher-level actions and behaviors, would allow for a broader range of both robots and applications. The meaningful abstraction of the robotic behaviors could then not only allow overcoming the compatibility inequalities between low-level, pin-control based platforms like the Arduino and more wrapped up platforms like the Create, but also allow more complex algorithms to be transferred between fundamentally different types of robots; tracking algorithms for colored blobs could be easily shared between a student developing on an SRV-1 tank robot and another student playing with a flying AR-Drone quad-rotor helicopter.

Similarly, teachers are able to keep and share curriculum that fully utilizes robotic aids across differences in platforms because activities can retain their focus on specific robotic concepts or physical interactions by referring directly to the abstraction tasks. The URAPI design and further expansions of its Command set and robot specifications, would not only let teachers share despite differences in platforms, but in fact allow teachers to take advantage of the unique traits of the various robots. By keeping the environment and fundamental behaviors in programming the code the same, teachers and students can smoothly switch to new robots that might be better suited for certain units or learning objectives without the difficulties of learning a new system. Activities can be simply fit to the platform of choice or the basic robotic programs taught on one platform could then be used as the foundation on new robot platforms that explore its more complex features. A robotics teacher in Tasmania envisions the benefits of such a flexible system:

> At present there are a number of different hardware platforms that are targeted at the education market but have different strengths. For example, the modularity of Mindstorms that means that everyone's robot can be different. There's the simplicity of the Finch that means hardware does what's it meant to do so the student can concentrate on programming. The iRobot Create that makes the base of a robust, commercially available robot chassis available as a building platform. …
>
> Imagine being able to run the same program on multiple platforms. Wouldn't it be wonderful if they all played nice together? Having learned basic programming techniques on the Finch, the student might use the same code to program an NXT that they had built themselves. [10]

The LabVIEW environment demonstrates how URAPI can provide users with a single language to learn that they can carry with them to any new robot. While implementation differences in hardware components or task realization may require users to tweak programs, the transference of main ideas and basic structure, along with the consistent overall scheme means that the users are already familiar with the notation and can easily identify and make necessary adjustments. Even when transferring between entirely different types of robots, they still are presented with a familiar environment and only need to learn the new capabilities and mannerisms of the robot itself, without the added complication of teaching themselves an entirely new language.

The Universal Data API provides an example of how the universal generalization of the inputs and outputs to the physical world can expand the range of devices useable by classrooms without retraining. It also shows how generalized standard interfaces enable data-focused software to be more easily developed even as they become more powerful by opening up compatibility to a literally indefinite range of new devices. This lets the software developers worry about the quality of their software and not the quirks of interfacing with each specific device driver. Likewise, device manufacturers can now focus on doing what they do best by spending their resources on improving the performance of their physical sensor. In this way, the UDAPI proves useful as a means of applying the basic Economics principle of 'comparative advantage' – hardware developers are able to specialize in their expertise of sensor development while the necessary complimenting software tools can be "traded" off to software developers more proficient at creating classroom oriented applications. Finally, teachers – who are usually best at teaching – are able to worry simply about their curriculum and student learning.

The focus of the URAPI design presented in this thesis is on the 'Sense' and 'Act' portions of robotic programming. In the global URAPI system, the universal definitions extend to the 'Think' category of operations that are more necessary when compiled, completely autonomous programs are being run on the robotic platform alone. Integration of the Input-Output driven work of this thesis and multiple demonstrations and prior work of robot-based interpreters of universal commands would allow a full realization of the global URAPI design. The Universal Command set can be rounded out with the low-level, virtual assembly language commands demonstrated in previous work [37]. In combination with the LabVIEW compiler to transform the Universal Programs into URAPI bytecode programs, these compiled URAPI programs can be downloaded onto any platform running an interpreter firmware. Demonstrations in [37] across different chip architectures show this is a feasible goal for mid- to low-level microprocessors, and even partial input and output functionality has been demonstrated on the simplest of boards.

Future development will be able to integrate the more extensive UDAPI hardware definitions found in the Manifest.xml file with the URAPI robot command and hardware definitions. Along with the benefit from an XML general definition scheme, this integration in URAPI will enable robots to be completely independent programming platforms. Computers, and even other robots, will be able to discover everything about the robot's capabilities and how to program the robot through dynamic queries and communication. Even first steps of implementing URAPI hardware definitions in the manner of the Manifest.xml will enable more complex sensor and actuator information and control while still retaining universal compatibility. The UDAPI design focuses on giving context and description to the input and output data; integration into the URAPI return values would enable implementation of unit-typed variables and extension to higher precision control robotic applications.

A complete Universal Robotics API would do more for educational robotics areas than simply ease the pain of learning new robotic systems or being trapped to a single integrated solution. Developing a standardized way to describe the tasks, data, and behaviors common to all robots, opens the possibility of using this universal language as a means for all robots to communicate with each other. Robotic systems can now interact and share information no matter the architecture or brand, as easily as any computer can use the internet or TCP/IP to talk to another computer, regardless of make or power. Efforts such as the Robotic Web [38] can now be taken to an even lower level as simpler robots and simpler tasks are described and shared across platforms. Larger robotics projects can be broken into functional components, each one carried out by a different small robot, uniquely proficient at the task. Independent robots, such as distributed swarms, could communicate as a whole while some robots in the swarm could be flying and others could be ground based. In the classroom, students could work on robotics projects using platforms suited to their various interests or strengths, and larger term projects could take advantage of groups focused on various modular parts of a larger robotic system. Sharing information through the common URAPI language will not only allow development of robotics in the classroom, but also enable innovative robotics applications. A standard, universal protocol for robotics allows for the sharing of curriculum, content, and knowledge that will advance robotics in the classroom and research.

Appendices

Appendix A
Educational Robot List

Robot	Variants / Company	Link
iRobot Create	Base Command Module	http://store.irobot.com/shop/index.jsp?categoryId=3311368
Bioloid		http://www.robotis.com/xe/bioloid_en
Surveyor SRV-1	Base +Tank; +RCM	http://www.surveyor.com/SRV_info.html
Finch	BirdBrain Technologies	http://csbots.wetpaint.com/page/Finch
BrainLink	BirdBrain Technologies	http://www.brainlinksystem.com/
Nao	Aldebaran	http://www.aldebaran-robotics.com/
PicoCricket	Scratch	http://www.picocricket.com/
AR-Drone	Parrot	http://ardrone.parrot.com/parrot-ar-drone/usa/
Rovio	Wowee	http://www.wowwee.com/en/products/tech/telepresence/rovio/rovio
LSMaker	LaSalle University Tank	http://blogs.salleurl.edu/LSMaker/
Hexapod	Lynxmotion	http://www.lynxmotion.com/
NXT	LEGO MINDSTORMS	http://mindstorms.lego.com/en-us/Default.aspx
RCX		
TETRIX		http://www.tetrixrobotics.com/
WeDo		http://www.legoeducation.us/
DaNI Robot Starter Kit	National Instruments	http://sine.ni.com/nips/cds/view/p/lang/en/nid/208010
VEX		http://www.vexrobotics.com/
SuperPro	HiTechnic	http://www.hitechnic.com/cgi-bin/commerce.cgi?preadd=action&key=SPR2010
ArduPro	TangibleK / CHEERP	
Arduino	LIFA Firmata	http://arduino.cc/en/Main/Hardware
Wiring Board		http://wiring.org.co/hardware/
Teagueduino		http://teagueduino.org/
BOE-bot		http://www.parallax.com/
Basic Stamp		http://www.parallax.com/
LaunchPad	MSP430 TI	http://e2e.ti.com/group/msp430launchpad/w/default.aspx

Robot	Variants / Company	Link
MSP430 Variants	Experimenter's board	
BeagleBoard		http://beagleboard.org/
BeagleBone		
HawkBoard		
BUG		http://www.buglabs.net/
Machine Science		http://www.machinescience.com/
FlexStacks	Boston Engineering	http://www.analog.com/en/content/FlexStack/fca.html
TurtleBot	ROS	http://www.ros.org
PR2	ROS	
Scribbler		http://www.parallax.com/
Beebot		http://www.terrapinlogo.com/bee-bot.php
Probot		http://www.terrapinlogo.com/pro-bot.php
ooBug		http://www.superdroidrobots.com/shop/category.asp?catid=75
MA-VIN		http://www.robonova.com/manager/product.php?send_cateSeqid=26
Spykee		http://www.spykeeworld.com/spykee/US/index.html
mbed		http://mbed.org/
Pololu 3pi		http://www.pololu.com/
Sumo-bot		http://www.parallax.com/
POB-bot		http://www.pob-technology.com/web/index.php
Asuro		http://www.arexx.com/arexx.php?cmd=goto&cparam=p_asuro
Robotino		http://www.festo.com/ext/en/5022.htm
AIBO		http://support.sony-europe.com/aibo/
Kondo		http://kondo-robot.com/EN/
PICAXE		http://www.rev-ed.co.uk/picaxe/
ooPic		http://www.oopic.com/
Botball		http://www.botball.org/
KUKA youBot		http://kuka-youbot.com
STM32F4 Discovery	STMicroelectronics	http://www.st.com/internet/evalboard/product/252419.jsp
Roboids	Duck Hamster	http://www.roboidstudio.org/en/downloads/packages/

Appendix B
UDAPI Commands

Legend	
,	Commas are optional and can be excluded
(vN)	Parameter N, required as part of the command
Parameters (Params)	Each row is a possible pattern for the command
"TC"	Transducer Channel (individual Sensor, Actuator, or Event Sensor)

B.1. UDAPI Simple Command Protocol

Cmd	Bytecode	Params	Returns	Description
Get XML	X		X, (r1), {Manifest.xml[text]} >> r1 = length X[-1] >> [-1] is a byte value	Request the Manifest.xml file from the Sensor. Sends the content in chunks.
Clear	+++			Resets buffer and clears temp data
Read	R		R, Sensor value	Reads the sensor value using the current configurations and calibration
Set	S	M, (v1) C, (v1)		Sets the possible options (M - for mode, C - for conversion) to the given value/selection (v1). (v1) selection should correspond to the index in the Manifest.xml for the desired configuration
Get	G	M C	G,M,(r1) >> r1=Current Mode G,C,(r1) >> r1=Current Conversion	Returns the current selections for the given option
Write	W			Opposite of Read, but for Actuators
(Error)			!!!,(bad command)	

B.2. UDAPI Data Logging Commands

B.2.i) Command Structure

START{Signal, End, Transmit}, **SAMPLING**{Trigger}

- **For Actuators:**
  - **START**{ " " }, **SAMPLING**{*Trigger*, *WriteDataset*[...], *EndDataset Behavior* }
- **For Events:**
  - **START**{ " " }, **SAMPLING**{*Trigger*, CONDITION(s)[{*Value*, *Range*, *LogType*}] }

B.2.ii) Command Details

- **START**- Experiment Setup Portion:
  - **START**{*Signal*, *End*, *Transmit*}

Argument	Description	Options	Option Descriptions
Start Signal	Signal to start taking samples for this experiment	1) Now 2) Time 3) Trig. Cmd 4) TC	1) Start taking samples ASAP 2) Start after certain delay 3) Wait to receive the Trigger Command 4) Wait for another TC to broadcast a Trigger Signal
End Condition	What determines when the experiment is over (end of data samples)	1) N samp. 2) TC (End) 3) Inf.	1) Finish after taking N sample points 2) Stop taking samples after a given TC broadcasts its End signal 3) No Explicit end condition - (streaming)
Data Transmission	When the dataset (or partial dataset) is sent back from the Sensor	1) R. Cmd 2) S done 3) S Ts 4) S P(n)	1) Send the dataset (or however much is done) after receiving the Read Command 2) Send dataset when End Condition is reached 3) Send dataset at regular time periods 4) Send dataset every time a certain number of samples are collected

- **SAMPLING -** Per TC(i) options:

  - **SAMPLING**{*Trigger*}

Argument	Description	Options	Option Descriptions
Trigger	What trigger signals that the next sample should be taken	1) Ts 2) TC(S)	1) sample each time interval (*Ts*) 2) sample when TC(S) sends out a Sample signal (e.g. a button TC pressed or motor TC angle reached)

  - For Actuators:

  SAMPLING{*Trigger*, *WriteDataset*[...], *EndDataset Behavior* }

Argument	Description	Options	Option Descriptions
Trigger	What triggers a sample (from the WriteDataset[]) should be written (output to the Actuator)	1) Ts 2) TC(S)	1) write (output) next sample each time interval (*Ts*) 2) write sample when TC(S) sends out a Sample signal (e.g. a button TC pressed or motor TC angle reached)
***WriteDataset*[]**	Set of points (samples) to iterate over (write out) to the actuator	[*actuator output values*]	Array of outputs for the actuator that the Sensor should iterate over for each Data Point in the data set
EndDataset Behavior	What output the actuator should do once it reaches the end of its *WriteDataset*[]	1) Hold 2) Repeat	1) The actuator should hold the last output value once it reaches the end (ex.: Heater that turns up to a certain power and stays there, while a diff TC measures temp.) 2) Restart from the beginning of the WriteDataset, and continue iterating

  - For Events:

  SAMPLING{*Trigger*, CONDITION(s)[{*Value*, *Range*, *LogType*}] }

Argument	Description	Options	Option Descriptions
Trigger	What should signal the next sample to be taken	1) Ts 2) TC(S)	1) sample each time interval (*Ts*) 2) sample when TC(S) sends out a Sample signal (e.g. a button TC pressed or motor TC angle reached)
Condition(s)	The conditions for the Event sensor to record and/or trigger a Sample/Event signal	[{*Value*, *Range*, LogType}, ...]	Possibility for array of conditions
|----> Value	Primary threshold value that defines the event occurring	sample value	Value inside set of possible input values for the TC (ex.: angle of 30 degrees, button LOW, etc.)
|----> *Range*	Range to sample when inside		
|----> *LogType*	Single sample when threshold is met or sample continuously		

Appendix C
UDAPI Manifest.xml Structure

C.1. General Manifest.xml

```
<sensor lang="English">
    <identity>
        <name>My Sensor</name>
        <uuid>0FF0ABCD</uuid>
        <version></version>
        <info_uri>http://ceeo.tufts.edu/sensor</info_uri>
        <hardware>
            <num_chans></num_chans>
            <timeout></timeout>
        </hardware>
        <manufacturer>
            <name></name>
            <serial_no></serial_no>
            <documentation></documentation>
            <home_page></home_page>
            <etc></etc>
        </manufacturer>
    </identity>
    <transducers>
        <transducer>
            <chan_num></chan_num>
            <chan_type>
                <option>sensor</option>
                <option>actuator</option>
                <option>event_sensor</option>
            </chan_type>
            <desc>
                <name></name>
                <parameter></parameter>
                <documentation></documentation>
            </desc>
            <conversions>
                <conversion>
                    <units>
                        <type>
                            <option>raw</option>
                            <option>si_product</option>
                            <option>digital</option>
                            <option>percent</option>
                            <option>arbitrary</option>
                        </type>
                        <label></label>
                        <si_units>FORMULA</si_units>
                        <magnitude></magnitude>
                        <states></states>
                    </units>
                    <method>
                        <function>
```

```
                            <option>none</option>
                            <option>piecewise_lin</option>
                            <option>polynomial</option>
                            <option>exponential</option>
                            <option>log</option>
                            <option>normalize</option>
                        </function>
                        <parameters>
                            <coord></coord>
                        </parameters>
                    </method>
                    <scale>
                        <min></min>
                        <max></max>
                        <resolution></resolution>
                        <uncertainty></uncertainty>
                    </scale>
                    <data_model>
                        <datatype></datatype>
                        <sample_size></sample_size>
                    </data_model>
                </conversion>
            </conversions>
            <modes></modes>
            <capabilities>
                <min_period></min_period>
                <trig_delay></trig_delay>
            </capabilities>
        </transducer>
    </transducers>
    <datalogging>
        <max_datasize></max_datasize>
        <time_source></time_source>
        <modes>
            <commands></commands>
            <options></options>
        </modes>
    </datalogging>
</sensor>
```

C.2. Sample Manifest: NXT Light Sensor

Below is an example implementation for an NXT Light Sensor UDAPI device.

```
<sensor lang='English'>
    <identity>
        <name>NXT Light Sensor</name>
        <uuid>LEGO.NXT.Light.9844</uuid>
        <version>9844</version>
        <info_uri>http://mindstorms.lego.com/en-
          us/products/default.aspx#9844
        </info_uri>
        <hardware>
            <num_chans>1</num_chans>
            <timeout>10000</timeout>
        </hardware>
        <manufacturer>
            <name>LEGO</name>
            <serial_no>749J7</serial_no>
            <documentation>The Light Sensor assists in helping your robot
                to "see." Using the NXT Brick (sold separately), it
                enables your robot to distinguish between light and dark,
                as well as determine the light intensity in a room or the
                light intensity of different colors.
            </documentation>
            <home_page>http://www.lego.com</home_page>
        </manufacturer>
    </identity>
    <transducers>
        <transducer index='1'>
            <chan_num>1</chan_num>
            <chan_type>sensor</chan_type>
            <desc>
                <name>Light Sensor</name>
                <parameter>Brightness</parameter>
                <documentation>Measures light received at sensor diode,
                    with option to turn on floodlight. Higher Scaled values
                    indicate a greater amount of light sensed, while higher
                    Raw values indicate less light sensed.
                </documentation>
            </desc>
            <conversions>
                <conversion index='1'>
                    <units>
                        <type>raw</type>
                        <label>Raw</label>
                    </units>
                    <method>
                        <function>none</function>
                    </method>
                    <scale>
                        <min>0</min>
                        <max>1023</max>
                        <resolution>1</resolution>
                        <uncertainty></uncertainty>
                    </scale>
                    <data_model>
                    <datatype>uint16</datatype>
                    <sample_size>1</sample_size>
                </data_model>
            </conversion>

            <conversion index='2'>
                <units>
                    <type>percent</type>
                    <label>Light</label>
                </units>
                <method>
                    <function>piecewise_lin</function>
                    <parameters>
                        <lut>
                            <in>0</in><out>100</out>
                            <in>200</in><out>100</out>
                            <in>900</in><out>0</out>
                            <in>1023</in><out>0</out>
                        </lut>
                    </parameters>
                </method>
                <scale>
                    <min>0</min>
                    <max>100</max>
                    <resolution>1</resolution>
                    <uncertainty>0.143</uncertainty>
                </scale>
                <data_model>
                    <datatype>uint16</datatype>
                    <sample_size>1</sample_size>
                </data_model>
            </conversion>
        </conversions>

        <modes>
            <mode index='1'>
                <type>state</type>
                <name>Floodlight</name>
                <states>
                    <option index='1'>On</option>
                    <option index='2'>Off</option>
                </states>
            </mode>
        </modes>
        <capabilities>
            <min_period>3.004</min_period>
            <trig_delay>3.004</trig_delay>
        </capabilities>
    </transducer>
</transducers>
<datalogging>
    <max_datasize></max_datasize>
    <time_source></time_source>
    <modes>
        <commands></commands>
        <options></options>
    </modes>
</datalogging>
</sensor>
```

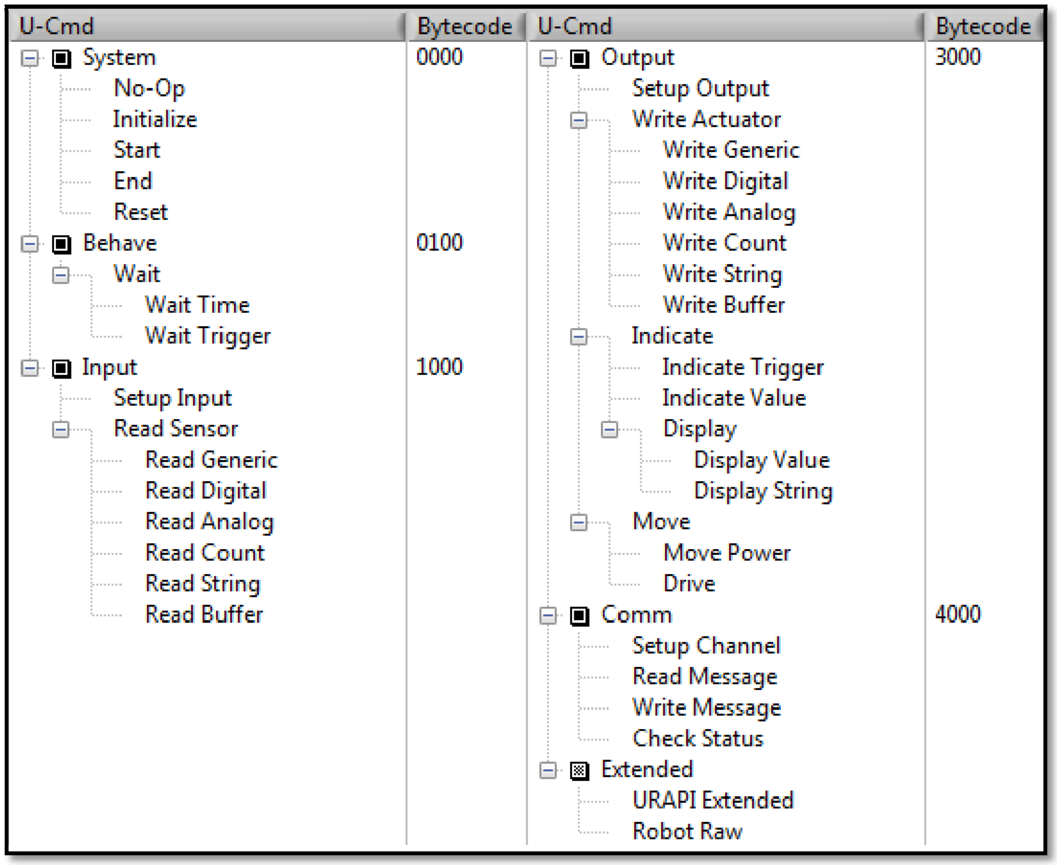
C.3. Basic Tag Descriptions

Some of the more notable elements in the UDAPI Manifest.xml files are described below. Element tags with a '+' annotation denote elements with children.

Element	Description
`<sensor lang="English">`	XML root element for UDAPI sensors. Standard `lang` attribute allows for localization.
Section: **Identity**	
+ `<identity>`	The identity of the UDAPI sensor involves manufacture identification information and other sources of information about the sensor itself (as opposed to the data or capabilities of the sensor).
`<name>`	Name is a human-readable label for the sensor.
`<uuid>`	The Universally Unique Identifier (uuid) is a number assigned specifically to the company or specific hardware.
`<info_uri>`	The info Universal Resource Identifier (URI) is the location of a website or other information resource where users and systems can access extra information about the sensor or updates and current resources.
+ `<manufacturer>`	The manufacturer subsection provides an area where hardware companies can add custom information.
Section: **Transducers**	
+ `<transducers>`	A transducer channel (or transducer) describes everything about the options and capabilities of reading or writing data to the Input or Output.
+ `<transducer>`	There can be multiple transducer channels (or transducers) on a single sensor, from simple 3-axis accelerometers to 3-sensor depth cameras.
+ `<desc>`	The description tag for the transducer indicates identifying characteristics about what the sensor element measures.
`<parameter>`	The parameter indicates the physical world phenomenon that the sensor measures.
+ `<conversions>`	For each sensor there can be various conversion modes it can run in.
+ `<conversion>` `<units>` `<type>` `(<si_units>)` `<method>` `<function>` `<scale>`	Each conversion mode specifies the units or type of data the sensor will return in the end (ex: inches, percentage, TRUE/FALSE, etc.), the scaling function used in that conversion to obtain these units (from raw electrical signals to measurements), and the parameters and resolution of those functions and output.
+ `<modes>`	The modes element group lists all the settings capable of being configured on the sensor. It also lists the various options that can be set.
Section: **Data Logging**	
`<datalogging>`	The data logging subsection contains relevant logging capabilities when used in a larger sampling experiment, especially in conjunction with other UDAPI sensors and sampling configurations.

Appendix D
URAPI Protocol

D.1. URAPI Commands Hierarchy Tree



U-Cmd	Bytecode
System	0000
No-Op	
Initialize	
Start	
End	
Reset	
Behave	0100
Wait	
Wait Time	
Wait Trigger	
Input	1000
Setup Input	
Read Sensor	
Read Generic	
Read Digital	
Read Analog	
Read Count	
Read String	
Read Buffer	
Output	3000
Setup Output	
Write Actuator	
Write Generic	
Write Digital	
Write Analog	
Write Count	
Write String	
Write Buffer	
Indicate	
Indicate Trigger	
Indicate Value	
Display	
Display Value	
Display String	
Move	
Move Power	
Drive	
Comm	4000
Setup Channel	
Read Message	
Write Message	
Check Status	
Extended	
URAPI Extended	
Robot Raw	

D.2. URAPI Command Dictionary

URAPI Command	Inputs	Default Values	Outputs
No-Op			
Initialize			
Start	'timeout': unsigned word [16-bit integer (0 to 65535)]	1000	
End			
Reset			
Wait Time	'timeout': unsigned word [16-bit integer (0 to 65535)]	1000	
Wait Trigger			
Setup Input	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "General" 'PortID' -> 0 'BaseCmd' -> 0	
Read Generic	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "Input" 'PortID' -> 0 'BaseCmd' -> 4097	'Value': double [64-bit real (~15 digit precision)]
Read Digital	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "Digital_Input" 'PortID' -> 0 'BaseCmd' -> 4098	'Value': boolean (TRUE or FALSE)
Read Analog	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "Analog_Input" 'PortID' -> 0 'BaseCmd' -> 4099	'Value': double [64-bit real (~15 digit precision)]
Read Count	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "Input" 'PortID' -> 0 'BaseCmd' -> 4100	'Count': long [32-bit integer (-2147483648 to 2147483647)]

URAPI Command	Inputs	Default Values	Outputs
Read String	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "Input" 'PortID' -> 0 'BaseCmd' -> 4101	'String': string
Read Buffer	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "Input" 'PortID' -> 0 'BaseCmd' -> 4102	'Buffer': string
Setup Output	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "Output" 'PortID' -> 0 'BaseCmd' -> 12288	
Write Generic	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] --- 'Value': double [64-bit real (~15 digit precision)]	'DeviceType' -> "Output" 'PortID' -> 0 'BaseCmd' -> 12289 --- 0.000E+0	
Write Digital	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] --- 'Value': boolean (TRUE or FALSE)	'DeviceType' -> "Digital_Output" 'PortID' -> 0 'BaseCmd' -> 12290 --- FALSE	

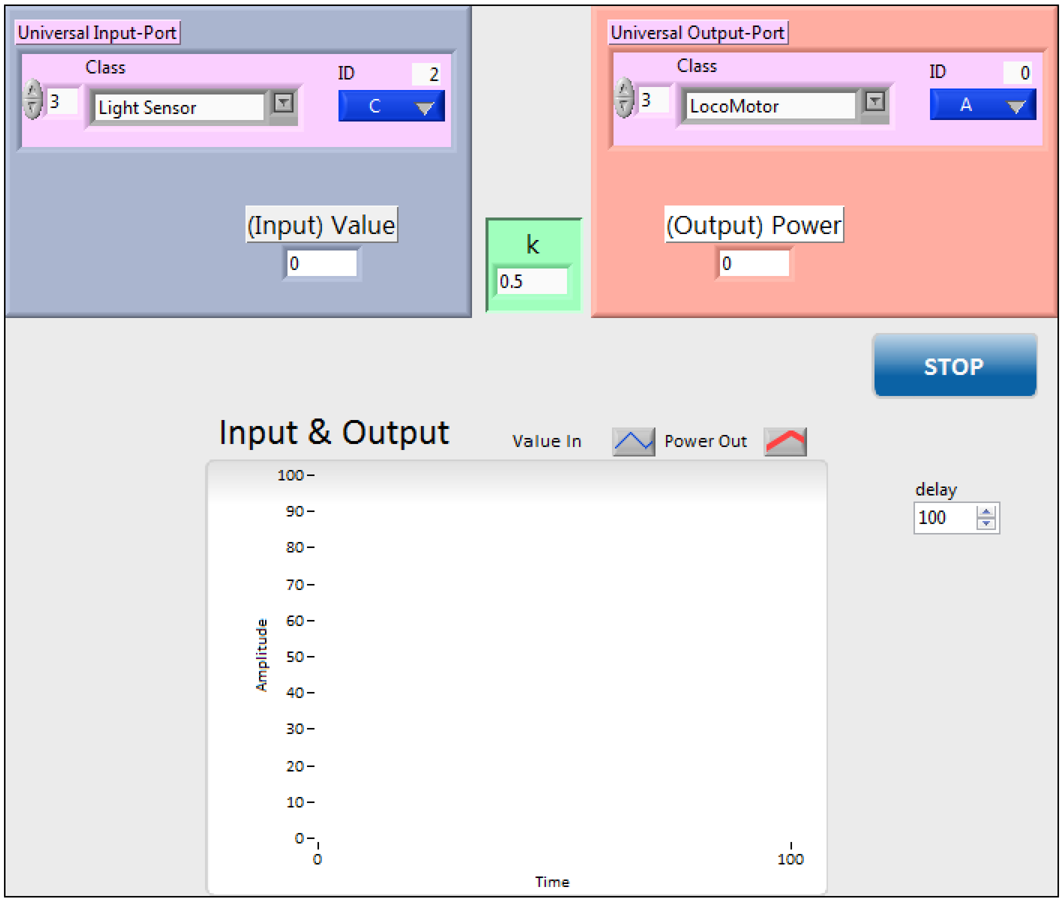
URAPI Command	Inputs	Default Values	Outputs
Write Analog	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] --- 'Value': double [64-bit real (~15 digit precision)]	'DeviceType' -> "Analog_Output" 'PortID' -> 0 'BaseCmd' -> 12291 --- 0.000E+0	
Write Count	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] --- 'Count': long [32-bit integer (-2147483648 to 2147483647)]	'DeviceType' -> "Output" 'PortID' -> 0 'BaseCmd' -> 12292 --- 0	
Write String	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] --- 'String': string	'DeviceType' -> "Output" 'PortID' -> 0 'BaseCmd' -> 12293 --- ""	
Write Buffer	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] --- 'Buffer': string	'DeviceType' -> "Output" 'PortID' -> 0 'BaseCmd' -> 12294 --- ""	

URAPI Command	Inputs	Default Values	Outputs
Indicate Trigger	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] 'Triggered': boolean (TRUE or FALSE)	'DeviceType' -> "Output" 'PortID' -> 0 'BaseCmd' -> 12290 FALSE	
Indicate Value	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] 'Value': double [64-bit real (~15 digit precision)]	'DeviceType' -> "Analog_Output" 'PortID' -> 0 'BaseCmd' -> 12291 0.000E+0	
Display Value	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] 'Value': double [64-bit real (~15 digit precision)]	'DeviceType' -> "Analog_Output" 'PortID' -> 0 'BaseCmd' -> 12291 0.000E+0	
Display String	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] 'String': string	'DeviceType' -> "Output" 'PortID' -> 0 'BaseCmd' -> 12298 ""	

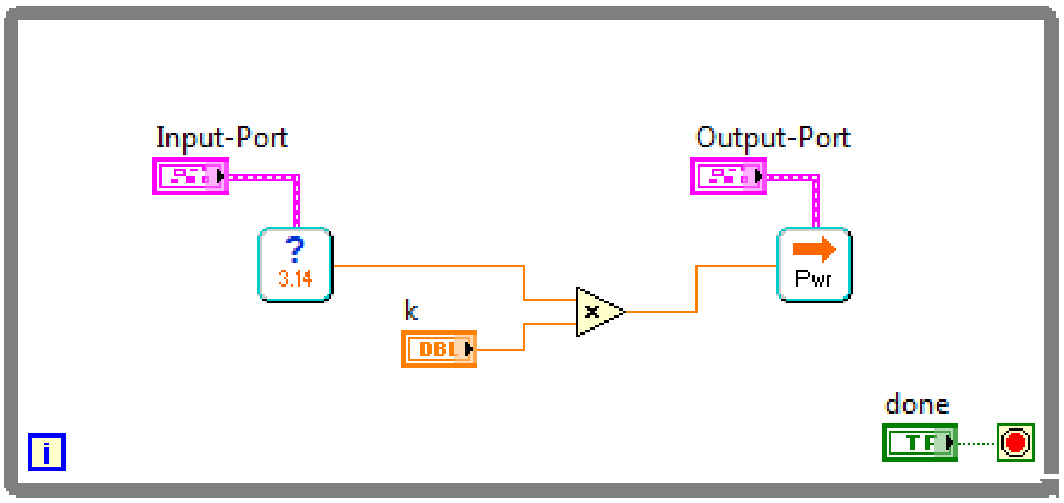
URAPI Command	Inputs	Default Values	Outputs
Move Power	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] ——— 'Power': double [64-bit real (~15 digit precision)]	'DeviceType' -> "LocoMotor" 'PortID' -> 0 'BaseCmd' -> 12299 ——— 0.000E+0	
Drive	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] ——— 'Power': double [64-bit real (~15 digit precision)]	'DeviceType' -> "DriveAxis" 'PortID' -> 0 'BaseCmd' -> 12300 ——— 0.000E+0	
Setup Channel	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "Output" 'PortID' -> 0 'BaseCmd' -> 16384	
Read Message	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "Input" 'PortID' -> 0 'BaseCmd' -> 16385	'Message': string
Write Message	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)] ——— 'Message': string	'DeviceType' -> "Output" 'PortID' -> 0 'BaseCmd' -> 16386 ——— ""	

URAPI Command	Inputs	Default Values	Outputs
Check Status	'U-Port': cluster of 3 elements 'DeviceType': string 'PortID': unsigned byte [8-bit integer (0 to 255) 'BaseCmd': unsigned word [16-bit integer (0 to 65535)]	'DeviceType' -> "General" 'PortID' -> 0 'BaseCmd' -> 16387	'Status': long [32-bit integer (-2147483648 to 2147483647)]
URAPI Extended	'Buffer': string	""	'Buffer': string
Robot Raw	'Buffer': string	""	'Buffer': string

Appendix E
URAPI Sample User Program

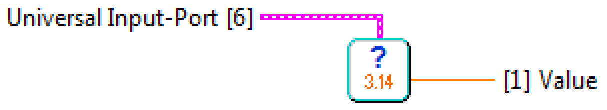
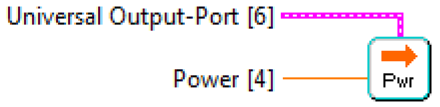


LabVIEW VI Front Panel



LabVIEW VI Block Diagram

SubVIs

Read Analog "UR.Input.ReadAnalog.vi"	
Move Power "UR.Output.MovePower.vi"	

References

[1] M. M. Hynes, "Teaching middle-school engineering: An investigation of teachers subject matter and pedagogical content knowledge," ProQuest Dissertations and Theses, United States -- Massachusetts,

[2] D. Kee, "Educational Robotics—Primary and Secondary Education [Industrial Activities]," *Robotics & Automation Magazine, IEEE,* vol. 18, no. 4, pp. 16 -19, Dec. 2011.

[3] A. Birk, "What is Robotics? An Interdisciplinary Field Is Getting Even More Diverse [Education]," *Robotics & Automation Magazine, IEEE,* vol. 18, no. 4, pp. 94 -95, Dec. 2011.

[4] P. Salvine, M. Nicolescu and H. Ishiguro, "Benefits of Human–Robot Interaction [TC Spotlight]," *Robotics & Automation Magazine, IEEE,* vol. 18, no. 4, pp. 98-99, Dec. 2011.

[5] M. U. Bers, Blocks to robots: learning with technology in the early childhood classroom, Teachers College Press, 2008, p. 154.

[6] iRobot Corporation, "iRobot: Education & Research Robots," 2011. [Online]. Available: http://store.irobot.com/shop/index.jsp?categoryId=3311368.

[7] ROBOTIS Inc., "Bioloid," 2011. [Online]. Available: http://www.robotis.com/xe/bioloid_en.

[8] BirdBrain Technologies, LLC, "The Finch," 2011. [Online]. Available: http://www.finchrobot.com/.

[9] LEGO Group, "LEGO.com MINDSTORMS," 2011. [Online]. Available: http://mindstorms.lego.com/en-us/Default.aspx.

[10] R. Torok, "Reflections on LEGO Engineering Symposium 2011 (Tufts University's CEEO, 24-26 May 2011) - Part 1 of 3," 03 June 2011. [Online]. Available: http://robtorok.blogspot.com/2011/06/lego-engineering-symposium-2011-part-1.html.

[11] Surveyor Corporation, "Surveyor SRV-1 Open Source Mobile Robot," 2011. [Online]. Available: http://www.surveyor.com/SRV_info.html.

[12] "IEEE Standard for a Smart Transducer Interface for Sensors and Actuators Wireless Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats," *IEEE Std 1451.5-2007,* pp. C1-236, 5 Oct 2007.

[13] D. Wobschall, "IEEE 1451-- A Universal Transducer Protocol Standard," Unpublished, 2007.

[14] Free Software Foundation (FSF), "GNU Compiler Collection," 2011. [Online]. Available: http://gcc.gnu.org.

[15] LLVM Team, "The LLVM Compiler Infrastructure Project," University of Illinois at Urbana-Champaign, 2011. [Online]. Available: http://llvm.org/.

[16] LLVM Team, "Projects built with LLVM," University of Illinois at Urbana-Champaign, 2011. [Online]. Available: http://llvm.org/ProjectsWithLLVM/.

[17] National Instruments, "The LabVIEW Compiler - Under the Hood," *NI Instrumentation Newsletter,* 3 Aug. 2010.

[18] VMKit Team; LLVM Team, "VMKit: a substrate for virtual machines," 2011. [Online]. Available: http://vmkit.llvm.org/.

[19] "Wiring," 2011. [Online]. Available: http://wiring.org.co/.

[20] "Arduino - HomePage," 2011. [Online]. Available: http://arduino.cc/.

[21] "Arduino playground," 2011. [Online]. Available: http://arduino.cc/playground/.

[22] H. Barragán, "Wiring: Prototyping Physical Interaction Design," Ivrea -- Italy, 2004.

[23] "Supported Platforms - Wiring," Wiring.co., 2012. [Online]. Available: http://wiki.wiring.co/wiki/Supported_Platforms.

[24] H.-C. Steiner, "Firmata: Towards Making Microcontrollers Act Like Extensions of the Computer," in *9th International Conference on New Interfaces for Musical Expression*, Pittsburgh, Pennsylvania, USA, 2009.

[25] "What Is Sanguino? - Sanguino.cc," [Online]. Available: http://sanguino.cc/.

[26] "ROBOTC.net," Robomatter, Inc., 2011. [Online]. Available: http://www.robotc.net/.

[27] "VEX Educational Robotics Design System," VEX Robotics, 2011. [Online]. Available: http://www.vexrobotics.com/.

[28] Microsoft Robotics, "Microsoft Robotics Development Studio (MRDS)," 2011. [Online]. Available: http://www.microsoft.com/robotics/.

[29] "The Player Project," 2011. [Online]. Available: http://playerstage.sourceforge.net/.

[30] Gostai Technologies, "UrbiForge," Gostai Technologies, 2011. [Online]. Available: http://www.urbiforge.org/.

[31] Willow Garage, "The Robot Operating System (ROS)," Willow Garage, 2011. [Online]. Available: http://www.ros.org/wiki/.

[32] J. Baillie, "URBI: towards a universal robotic low-level programming language," in *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, 2005.

[33] "Nao for Education," Aldebaran Robotics, 2011. [Online]. Available: http://www.aldebaran-robotics.com/en/Solutions/For-Education/introduction.html.

[34] A. Barco and J. A. Canals, "LSMaker| El blog de la plataforma robótica de Campus La Salle," LaSalle University, 2011. [Online]. Available: http://blogs.salleurl.edu/LSMaker/.

[35] iRobot Corporation, "iRobot Create Open Interface (OI) Specification," 2006.

[36] "AR.Drone Parrot," Parrot, 2011. [Online]. Available: http://ardrone.parrot.com/parrot-ar-drone/usa/.

[37] J. Palmer, "Application of a universal language for low-cost classroom robots," ProQuest Dissertations and Theses, United States -- Massachusetts, 2010.

[38] M. B. Blake, S. L. Remy, Y. Wei and A. M. Howard, "Robots on the Web," *Robotics & Automation Magazine, IEEE,* vol. 18, no. 2, pp. 33--43, 2011.

[39] J. L. Jones and D. Roth, Robot programming: a practical guide to behavior-based robotics, McGraw-Hill, 2004.

[40] National Instruments, "NI-VISA Overview," National Instruments, 19 Feb 2009. [Online]. Available: http://zone.ni.com/devzone/cda/tut/p/id/3702.