Photocopy and Use Authorization

X

Kenneth Stone
Author

A Glance at Augmented Reality in Robotics

by

Kenneth Stone

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

Master of Science in the Department of Mechanical Engineering

Idaho State University

August 2020

Committee Approval

To the Graduate Faculty:

The members of the committee appointed to examine the thesis of KENNETH STONE find it satisfactory and recommend that it be accepted.

X

Marco Schoen, Ph.D.
Major Advisor

X

Kenneth Bosworth, Ph.D.
Committee Member

X

Wenxiang Zhu, Ph.D.
Graduate Faculty Representative

Acknowledgements

I would firstly like to thank my primary advisor, Dr. Marco Schoen, for his guidance in research and writing this thesis. His help has been instrumental in the success of this project and documentation of the results thereof. Additionally, the original principle investigator of this project, Dr. Alba Perez-Gracia, deserves special recognition of her effective organization and training of myself and other participants early in the project's duration. Committee members Dr. Ken Bosworth and Dr. Wenxiang Zhu deserve recognition for their willingness to assist in the completion of this thesis.

The project lead was Omid Heidari, whose brilliance and determination produced the most significant parts of the project's structure and substance. Colleagues Shovan Chowdhury, Tyler Hedgepeth and others contributed critical components to the project that aided substantially in its function and usability.

The project would have been impossible without the equipment and training provided by the partner company House of Design. Expertise and guidance critically came from a few different members of their company, including Shane Dittrich and Mike Luna. Training in the usage of ABB Robots and RAPID™ Programming given by Trevor Hall made possible much of the robot interaction featured in the current version of the application.

I would also like to thank Idaho Global Entrepreneurial Mission for their funding of this project and our participation with House of Design. Finally, Idaho State University provided the research laboratory and infrastructure which gave us the ability to perform the project.

A Glance at Augmented Reality in Robotics

Thesis Abstract—Idaho State University (2020)

This thesis details the theory and implementation of software on the augmented reality platform Microsoft HoloLens® which communicates and interacts with ABB robotic systems. Recent advancements in augmented reality technology have provided new unrealized potential for use in industrial robotic development. This project uses the exposure of ABB's controller information by HTTP web requests and displays the information in a specially built application in HoloLens®. System information is displayed to the user on a virtual panel, and a virtual robot model with path targets can be displayed on the actual robot to create a new intuitive development environment. Though the project is in its infant stages and has room for many improvements, the realized concepts from this project and further avenues of development show great promise.

Key Words:

ABB, Industrial Robots, FlexPendant, HoloLens, Unity, Augmented Reality, IRC5, IRB-120, C#, HTTP, RAPID

Table of Contents

List of Figures

List of Tables

List of Equations

1 Introduction

Objective

The objective of this work is to describe the foray into developing software to communicate between an AR headset and an ABB industrial robot controller. Some of the topics introduced and presented in this thesis deal with the programming of AR headsets and their communication with robot control hard and software, as well as capabilities and impact of such systems in industrial settings.

Augmented Reality

The world of virtual reality and augmented reality is a young one, with computational equipment only recently becoming powerful and compact enough to bring them to practical use.

Virtual Reality (VR) is a concept that brings computer-rendered graphics a new experience by allowing the user to view the virtual surroundings as if they were truly in that world. Augmented Reality (AR), by contrast, brings some virtual elements into the real world that the user is viewing, superimposing those elements on the physical surroundings.

AR provides an ideal tool to join the experiences of the physical world and electronic data in a way never-before possible. This technology has great potential in a variety of settings, including automated manufacturing.

Robotics in Industry 4.0

Robots are at the heart of the modern manufacturing plant, and the advances of Industry 4.0 bring about methods of robotic implementation that a short time ago would have seemed overwhelming.

Industry 4.0 is the changes brought about to the manufacturing world due to the fourth industrial revolution, which took place in the last decade, [1]. It takes advantage of recent concepts such as the Internet of Things (IoT) and applies them in an industrial setting.

The increasing ability for individual manufacturing components to take and transmit data, and act according to data received by other components, is a phenomenon that is referred to as the Industrial Internet of Things (IIoT). This trend allows for much more complex and nuanced control of automated processes than previously attainable.

This interconnectedness of manufacturing devices allows unprecedented automation of record keeping, and access to virtually unlimited statistics about a given manufacturing process. The advance of processing technology such as powerful PLCs and machine learning algorithms across the world has led to widespread use of smart automation processes, as the machinery can now make complex decisions without human intervention, [1].

Combination of Robotics and AR Technology

Modern industrial robots are typically controlled and monitored via software on a computer nearby the system. These computers provide useful and practical interfaces to control robot operations; however, the connection between the planned robot workspaces in the software and the robot’s physical environment can be unintuitive.

The burgeoning world of augmented reality presents a logical combination with manufacturing robotics. AR can provide a new tool to developers and customers of industrial systems.

2 State of the Art

There is a multitude of solutions currently on the market for automated manufacturing. Numerous companies compete to provide products and services for this demand, such as ABB, Siemens, Rockwell Automation, Eaton, Honeywell, and Mitsubishi, just to name a few. In addition, tech-savvy prospective producers can venture into open-source software for running their custom-built robots, such as Robot Operating System (ROS).

RobotStudio®

This work focuses on robots, controllers, and software produced by ABB. The company's flagship robotic software is RobotStudio®, a powerful program with wide-ranging capabilities, [2]. It facilitates the design of robotic systems by furnishing ready-to-use virtual versions of the entire catalog of ABB robots. It allows the setup of a simulated workspace for the robot and provides straightforward planning of the robot's path during each production cycle. It also provides direct access to write and edit controllers' code. It has capabilities to interface with Computer-Aided Design (CAD) software, such as SolidWorks, [3].

RobotStudio® also has a feature allowing the robotic systems built within the program to be viewed from a VR headset. RobotStudio® provides simulation of its virtual robot following the designated paths, and this simulation can be viewed from a VR headset to better feel the scale and operation of a production cycle, [3].

As powerful as RobotStudio® is, there are issues imposed by its scope and the hardware it uses. The software currently has no implementation for AR glasses, a useful new tool as previously stated. This second issue is the focus of this thesis. Ideally, the AR software would ultimately feel natural to any developer already familiar with RobotStudio®, [3].

AR Hardware

There are multiple AR platforms developed by companies. For example, Microsoft™ HoloLens® and Google™ Glass are two competing projects to build state-of-the-art eyewear for commercial use. Additionally, Leap Motion develops Project North Star, a device which has a fully 3D printable frame, and has a much lower cost than the products from Google and Microsoft, at the expense of developmental support.

3 Proposed Approach

The main goal of this project is to provide an example of the potential of AR technology applied to the world of industrial robotics. The result is an application that can communicate with an ABB system to provide developers and customers a new way to interface with their production technology.

Microsoft HoloLens®

The project's hardware of choice is Microsoft's AR eyewear product, HoloLens®. It has numerous advantages that facilitate development of specialized software. HoloLens® has a simple group of predefined controls, and fully functional motion tracking algorithms allowing developers to focus on their products [4].

In the center of the user's field of vision is a white dot called the "gaze," which is analogous to the cursor on a regular computer. When the user places the gaze on something that they wish to interact with, there are a variety of hand gestures; a simple click activates any button. Clicking and dragging with one hand can move objects around in the virtual space and doing so with two hands also allows rotation and scaling of objects. Finally, if one needs to exit the program and return to the main menu, a bloom gesture can be done from any application, [4].

HoloLens® uses a range of different sensors to calculate the movements of the headset within its real surroundings, using algorithms that are known as Simultaneous Location And Mapping (SLAM) algorithms. This procedure allows developers to place objects in their application and have those objects stay in place relative to the physical surroundings, giving the user the impression that they can move freely in this hybrid world, [4].

Since HoloLens® is an active commercial development, it is likely that programs developed for the current HoloLens® should be relatively easily ported to HoloLens® 2 and further versions of the eyewear.

Experimental Setup

The final product should be able to handle any system the user needs, but for development and testing, the proposed system consists of a single ABB brand IRB 120™ six-degree-of-freedom robot, which is attached to an aluminum table surrounded by an aluminum cage for safety. Mounted under the table is an IRC5™ Controller connected to the robot above, [5].

ABB's controllers come with a teach pendant called FlexPendant™. The FlexPendant™ is the most direct route to editing a robot controller's program. Any other smart equipment trying to modify the program must be given explicit permission through the FlexPendant™, [5].

Either the FlexPendant™ or RobotStudio® can edit the controller's code, which is scripted in a high-level language called RAPID™, [3]. The controllers code is made up of structures called tasks, which are further subdivided into modules containing symbols and routines.

4 Mathematics of Robotics

Position vectors

Positions are represented by three-dimensional vectors showing the position in the basis of some coordinate frame. Relative position between two objects can be found by adding and subtracting these vectors from each other, [6].

It should be noted that these position vectors are closed under addition and scalar multiplication, as stated in Equations (4-1) and (4-2). Successive positions can be added to other positions for translations between each. Scalar multiplication (as the name implies) can scale any vector to any size. Vector addition and scalar multiplication are both commutative.

$$\forall \vec{u}, \vec{v} \in W : \vec{u} + \vec{v} \in W \tag{4-1}$$

$$\forall \vec{u} \in W, \forall c \in \mathbb{R} : c\vec{u} \in W \tag{4-2}$$

Rotation matrices and quaternions

Rotation matrices

There are a couple computational approaches to rotation. The first is a 3-by-3 rotation matrix that encodes the three basis unit vectors given by the current rotated coordinate frame. These rotated unit vectors are the columns of the rotation matrix. This matrix is therefore an orthonormal skew-symmetric matrix that can be premultiplied to any position vector to rotate that vector correspondingly, [7].

$$\boldsymbol{R}_x(\theta) := \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{4-3}$$

$$\boldsymbol{R}_y(\theta) := \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{4-4}$$

$$\boldsymbol{R}_z(\theta) \coloneqq \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4-5}$$

$$\vec{p}_{rotated} = \boldsymbol{R}\vec{p}_{original} \tag{4-6}$$

Just as rotations can be multiplied onto vectors to rotate them, they can be premultiplied to another rotation matrix. Since these are matrices, their multiplication is generally not commutative. The matrix product resulting is the second rotation factor, followed by the first. A chain of rotations can be made in this manner, with the right-most factor being the first rotation, with each next premultiplied matrix being next rotation, [7].

$$\boldsymbol{R}_{total} = \prod_{i=1}^{\overset{\curvearrowleft}{n}} \boldsymbol{R}_i = \boldsymbol{R}_n\boldsymbol{R}_{n-1} \ldots \boldsymbol{R}_2\boldsymbol{R}_1 \tag{4-7}$$

A rotation matrix can be inverted to give the opposite rotation from the original matrix. Since the rotation matrix is an orthonormal skew-symmetric matrix, a rotation matrix will always be nonsingular, and its inverse is its transpose. A rotation pre- or postmultiplied by its inverse rotation will give the identity matrix, which signifies no net rotation, [6].

$$\boldsymbol{R}\boldsymbol{R}^{-1} = \boldsymbol{R}^{-1}\boldsymbol{R} = \boldsymbol{I}\ (No\ rotation) \tag{4-8}$$

Quaternions

The other typical computational representation of a rotation is the quaternion. It is an extension of complex numbers to include three different imaginary dimensions. It encodes both some axis of rotation away from the parent orientation, and the angle of rotation about that axis. The quaternion only has four elements, and only has three-degrees of freedom, since the quaternion has the constraint of having an absolute value of one, [7].

$$\vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}: \quad Rotation\ axis$$

$$i^2 = j^2 = k^2 = ijk = -1: \quad Imaginary\ constants$$

$$q \coloneqq e^{\frac{\theta}{2}\vec{u}\cdot\begin{bmatrix} i \\ j \\ k \end{bmatrix}} = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)\vec{u}\cdot\begin{bmatrix} i \\ j \\ k \end{bmatrix} \tag{4-9}$$

Calculating the inverse of a quaternion, which is the opposite rotation just as it is with rotation matrices, is calculated by simply substituting $-\theta$ for θ, which negates all imaginary components of the quaternion. Also, like rotation matrices, pre- or postmultiplying a normal quaternion by its inverse yields the real number 1, corresponding to no rotation, [7].

$$q^{-1} = e^{-\frac{\theta}{2}\vec{u}\cdot\begin{bmatrix} i \\ j \\ k \end{bmatrix}} = \cos\left(\frac{\theta}{2}\right) - \sin\left(\frac{\theta}{2}\right)\vec{u}\cdot\begin{bmatrix} i \\ j \\ k \end{bmatrix} \tag{4-10}$$

These objects can be used to calculate the rotation of a point, but its calculation is somewhat less straight-forward compared to rotation matrices. The vector must be modified by using an inner product with a vector of the imaginary constants i, j, and k. The corresponding ‘position quaternion’ has a real component of zero and is not bound by the typical normalization rule of quaternions. To find the rotated vector, the position quaternion is premultiplied by the rotation quaternion and postmultiplied by the inverse rotation. The quaternion product can then be returned to vector form by taking the coefficients of the three imaginary constants as respective elements, [7].

$$p' = \vec{p}\cdot\begin{bmatrix} i \\ j \\ k \end{bmatrix} \tag{4-11}$$

$$p'_{rotated} = qp'q^{-1} \tag{4-12}$$

Quaternions, like rotation matrices, can be premultiplied onto other quaternions to perform successive rotations. Using identities of products of the imaginary unit constants and multiplication of each right-most pair of quaternions, the single quaternion representing the result of the rotation sequence is achieved.

$$q_{total} = \prod_{i=1}^{\overset{\curvearrowleft}{n}} q_i = q_n q_{n-1} \cdots q_2 q_1 \tag{4-13}$$

Transformation Matrices

A transformation matrix is a structure that combines a translation vector and a subsequent rotation matrix. It is a 4-by-4 matrix with its rotation matrix in the upper-left elements with zeros in the row below, and its translation vector in the top three elements of the right-most column with a one below, [6].

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{R} & \vec{t} \\ \boldsymbol{0} & 1 \end{bmatrix} = \begin{bmatrix} r_{xx} & r_{xy} & r_{xz} & t_x \\ r_{yx} & r_{yy} & r_{yz} & t_y \\ r_{zx} & r_{zy} & r_{zz} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4-14}$$

Like rotation matrices, it can be premultiplied for successive combined transformations and inverted to give the opposite transformation. However, unlike them, the transformation matrix is not generally orthonormal or skew-symmetric due to its incorporation of the translation vector; thus, to find the inverse, one must manually implement a specialized inversion algorithm or resort to using numerical linear algebra methods.

Comparison of methods

The methods above all provide similar ways of calculating successive combined transformations, or separate positions and rotations, including rotations of points. The choice of which methods to use generally false to computational simplicity and performance, as well as compactness of data storage. The main methods for rotation computation are rotation matrices,

Table 1: Number of floats required to save a rotation in a specific form

Representation	Floats	Comments
Rotation Matrix	9	
Angle-axis	4	no precompute of sin θ or 1 - cos θ
Angle-axis	6	precompute of sin θ or 1 - cos θ
Quaternion	4	

quaternions, and explicit angle-axis 4-vectors (like quaternions, but not discussed here). Rotation matrices and translation vectors could be further combined in transformation matrices, [8].

In most cases, quaternions proved the most efficient method of representing rotations, both in computation steps and data storage [8]. The data storage of both quaternions and angle-axis structures is four elements (and can be reduced to three at the expense of computing the fourth for subsequent operations), but composing multiple rotations together is drastically more

Table 2: Operations required to convert between each rotation format

Conversion	Additions	Multiplications	Divisions	Function Calls	Comparisons
Axis-angle to matrix	13	15		2	
Matrix to axis-angle	8	7	1	2	
Angle-axis to quaternion	1	5	1	2	
Quaternion to angle-axis		4		2	
Quaternion to matrix	12	12			
Matrix to quaternion	6	5	1	1	1 or 3

efficient with quaternions. Quaternions are also under half the data size of rotation matrices (four numbers to nine) and compose rotations somewhat faster.

The only case where quaternions are not ideal is when a position is being transformed, where a rotation matrix wins out. Fortunately, the conversion between quaternions and rotation matrices is relatively inexpensive, and that conversion followed by transformation via the resultant matrix is somewhat cheaper than direct rotation by quaternion. As a result, poses in most computer systems, including ABB's robotic controllers, are implemented by a vector position and a separate quaternion rotation, [8].

Table 3: Operations required for a transformation of a vector with each rotation format.

Representation	Additions	Multiplications	Comparisons
Matrix	6	9	
Axis-angle	12	18	
Quaternion	24	32	Using generic quaternion multiplies
Quaternion	17	24	Using specialized quaternion multiplies
Quaternion	18	21	Convert to matrix, then multiply

Kinematic chain

Most industrial robots, including the ones at the focus of this project, are six-degree-of-freedom serial robots, meaning that each degree of freedom is represented by a joint attached to the previous one. If transformation matrices are used to attain this transformation, a simple chain of transformation matrices would give the pose of the end-effector, [6].

With separate vectors and quaternions representing poses, however, the process is slightly different. Finding the rotation of the end-effector is simple, as the rotation of each link on the robot can be successively premultiplied on each other to get the total rotation. Finding the position, however, is slightly more complicated. Since all joints on this robot are revolute joints, each link has a vector of constant magnitude pointing from its base joint to its end joint. The direction of this vector is determined by its default configuration rotated by that link's total rotation from the base. Therefore, the end-effector's position is given by the sum of those link vectors rotated by their corresponding global rotations, [6].

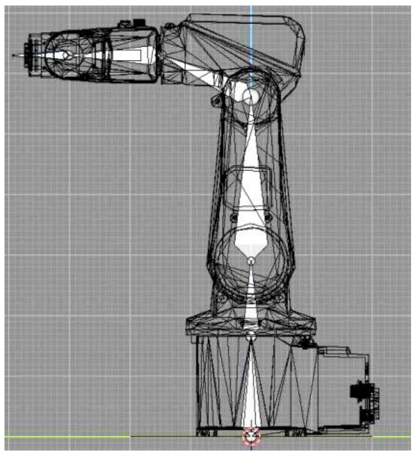


Figure 4-1:IRB 120 Wireframe showing the robot being animated and handled as a kinematic chain.

Kinematic calculation

Forward Kinematics

Forward kinematics is the process of taking a known set of robot coordinates, referred to as its configuration, and finding the resulting pose of the end-effector. The process of forward kinematics is a straight-forward one, as it is simply an addition of joint vectors and a composition of rotations to find the end-effector's pose. Forward kinematics is useful when a robot is first constructed, as it can be used to find the reference vectors and orientations of the robot for later use in inverse kinematics, [6].

$$\begin{aligned}\vec{t}_{total} = \vec{t}_1 + \boldsymbol{R}_1\vec{t}_2 + \boldsymbol{R}_2\boldsymbol{R}_1\vec{t}_3 + \boldsymbol{R}_3\boldsymbol{R}_2\boldsymbol{R}_1\vec{t}_4 + \boldsymbol{R}_4\boldsymbol{R}_3\boldsymbol{R}_2\boldsymbol{R}_1\vec{t}_5 + \boldsymbol{R}_5\boldsymbol{R}_4\boldsymbol{R}_3\boldsymbol{R}_2\boldsymbol{R}_1\vec{t}_6 \\ + \boldsymbol{R}_6\boldsymbol{R}_5\boldsymbol{R}_4\boldsymbol{R}_3\boldsymbol{R}_2\boldsymbol{R}_1\vec{t}_{tool}\end{aligned} \tag{4-15}$$

Inverse Kinematics

Since forward kinematics gets a pose from joint, predictably, inverse kinematics is the opposite function, taking a desired end-effector position and getting a set of joint variables that satisfies that position. As the objective of most robotic systems are targets in physical space, inverse kinematics is used much more in typical operation of robots. Typical numerical solvers treat this as a root-finding problem, with a function of one or more arguments evaluating to zero with a suitable selection of those arguments, [6].

$$\begin{gathered}\vec{t}_1 + \boldsymbol{R}_1\vec{t}_2 + \boldsymbol{R}_2\boldsymbol{R}_1\vec{t}_3 + \boldsymbol{R}_3\boldsymbol{R}_2\boldsymbol{R}_1\vec{t}_4 + \boldsymbol{R}_4\boldsymbol{R}_3\boldsymbol{R}_2\boldsymbol{R}_1\vec{t}_5 + \boldsymbol{R}_5\boldsymbol{R}_4\boldsymbol{R}_3\boldsymbol{R}_2\boldsymbol{R}_1\vec{t}_6 \\ + \boldsymbol{R}_6\boldsymbol{R}_5\boldsymbol{R}_4\boldsymbol{R}_3\boldsymbol{R}_2\boldsymbol{R}_1\vec{t}_{tool} - \vec{t}_{target} = \vec{0}\end{gathered} \tag{4-16}$$

$$\boldsymbol{R}_6\boldsymbol{R}_5\boldsymbol{R}_4\boldsymbol{R}_3\boldsymbol{R}_2\boldsymbol{R}_1 - \boldsymbol{R}_{target} = \boldsymbol{0} \tag{4-17}$$

Chapter Summary

This chapter discussed the mathematics that are involved in the poses and kinematic calculations of robotics. Three dimensional poses are composed of positions and rotations. Positions are described as a three-dimensional vector describing the position in millimeters with respect to a specific reference frame. Rotations are described as unit quaternions, a scalar numerical construct with a real component and three imaginary components, which encode a rotation axis and the angle of rotation about that axis. Robot joint configurations are stored as six-dimensional vectors representing the angle of each joint in degrees.

Robot kinematics can be divided into forward kinematics, which transform a vector of joint angles into an axis pose, and inverse kinematics, which perform the opposite mapping. The kinematic calculations are not one-to-one, so a least-squares calculation can be done to find the closest inverse kinematic solution. Forward kinematics can initially be done by a large matrix equation to capture the reference pose and joint vectors of the robot. Inverse kinematics themselves can be performed by a numerical root finding method on matrix equations with terms obtained from the forward kinematics.

These mathematical concepts are integral in the function of robots, including the ABB Robots focused on in this project. Recognizing how to transfer the poses between coordinate systems, such as between the RAPID™ and Unity® coordinate systems, is crucial for display of objects as done in this project.

5 IRB 120™ Operation Overview

Safety

Cage

The robot setup used during development of the application is enclosed by an aluminum cage with plexiglass panels allowing safe visibility into the cage. Industrial assemblies may or may not have safety cages built around the robot workspaces, so additional precautions may be necessary on a per-case basis.

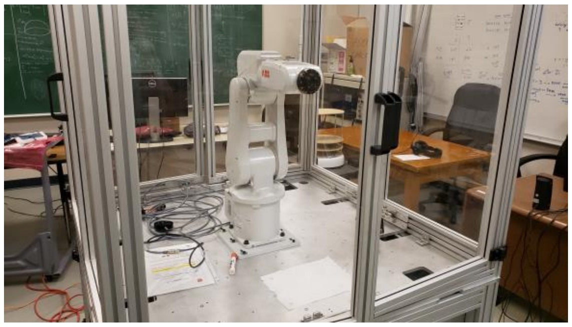


Figure 5-2: View from the right-front of the experimental robot setup

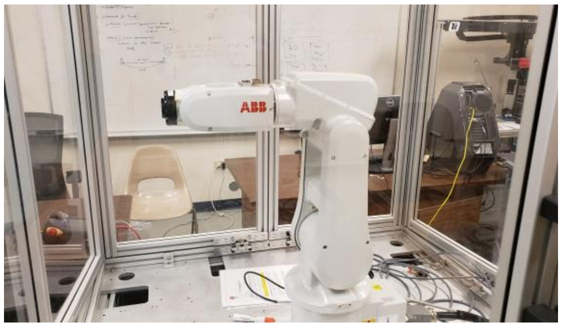


Figure 5-1: Setup from the left side

Force Sensors

The robot is equipped with force sensors in its joints, which will deactivate any motion that would otherwise cause the robot to damage itself.

Although the robot is sensitive enough to protect itself, most ABB robots, including the IRB 120™, are not able to prevent injury to external objects or humans. Therefore, care must be taken to remain clear of the robot's workspace while it is active.

Manual Mode and Automatic Mode

When programming the robot's movements for the first time, the robot will be in *manual reduced speed mode,* typically referred to as simply *manual mode;* in this mode the robot is limited to movement no faster than 250 mm/s, [5].

Additionally, when moving the robot in *manual mode*, the operator must be holding the *enabling device*, a button on the FlexPendant™ meant to assure that the operator is fully aware of the current robot operation. The button must be depressed half-way, as either undepressed or fully depressed will not allow the robot to move. When the *enabling device* is not engaged while in manual mode, the words *Guard Stop* will appear at the top of the FlexPendant™ screen, [5].

The other primary operating mode is *automatic mode.* Here the operator does not have direct authority to change the system or the code in any direct way. However, the system can run at full speed and can be set to run and repeat continuously. The *enabling device* does not need to be depressed in *automatic mode,* as the workspace is assumed to be clear of any obstructions and personnel, and the scripted movements have already been defined and tested, [5].

Manual full-speed mode can be thought of as an intermediate operating mode between the *manual and automatic* modes, available on some controllers. It has most of the fine control and abilities of the regular *manual mode,* but allows operation at full speed, and as such expects that external obstructions and personnel are safely away from its workspace. This mode is primarily meant for testing a system before proceeding to *automatic mode,* [5]. (Note: *Manual full-speed mode* is not available in the controller used in this test.)

Switches

Emergency Stop Buttons

There are emergency stop buttons on both the IRC5™ controller and the FlexPendant™, both being large red buttons. In the event of some failure or emergency near the robot system, this button can be used to immediately cease all operation, cutting power to all components except the brake release circuits. The controller must be rebooted to resume operation after an emergency stop, [5].

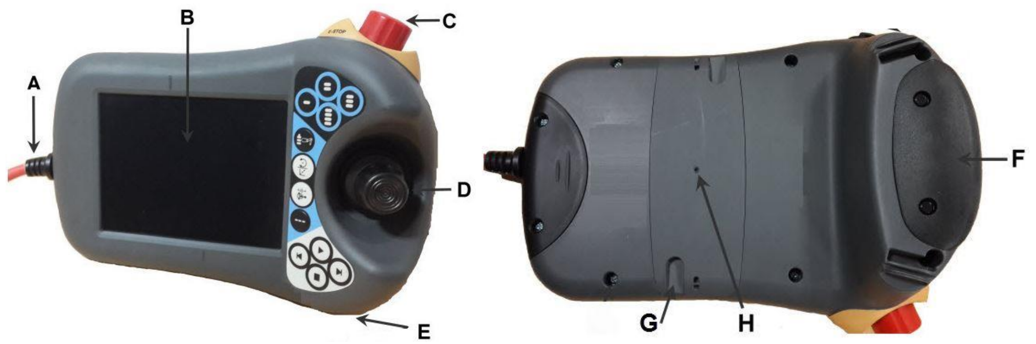


Figure 5-3: FlexPendant™ Front and Back.
A: Connector, B: Touch Screen, C: Emergency Stop Button, D: Joystick, E: USB Port, F: Three-Position Enabling Device (Guard Stop), G: Stylus Pen, H: Reset Button. Source: [5]

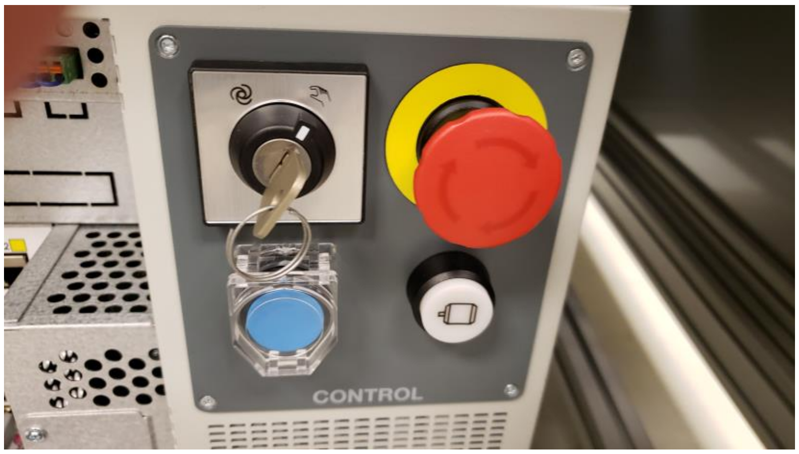


Figure 5-4:IRC5 Controls. Clockwise from Top-Left; Manual-Auto Key, Emergency Stop Button, Motors-On Button, Brake Release

On/Off switch

The On/Off switch, as the name suggests, controls the power state of the system. It is a stiff rotating switch to protect against accidental toggling. When the system is first booting up, it will require several moments to load all software before the FlexPendant™ will show the main screen and any alerts, showing that the system is operational, [5].

Manual/Auto Key

There is a key on the controller that switches between manual and auto mode (and manual full-speed mode on some controllers). This key can be left in the controller for experimental setups like this or can be removed for safety and security purposes, [5].

Motors On button

The Motors On button allows the motors to be moved in auto mode. When automatic mode is first enabled, the motors will be switched off by default. This button must be pressed so that it is illuminated for automatic robot operation. Since, in manual mode, motor control is done through the FlexPendant™ this button becomes irrelevant, [5].

Brake Release

The brake release button allows the robot to become limp and manually adjustable if it gets stuck in a certain position. Larger models of robots typically also have brake releases for individual joints. These individual switches can be found on the robot body if present, [5]. (CAUTION: The robot must be fully externally supported before the brake release is engaged; otherwise, the robot may fall and cause injury or equipment damage)

Manual Operation

Jogging

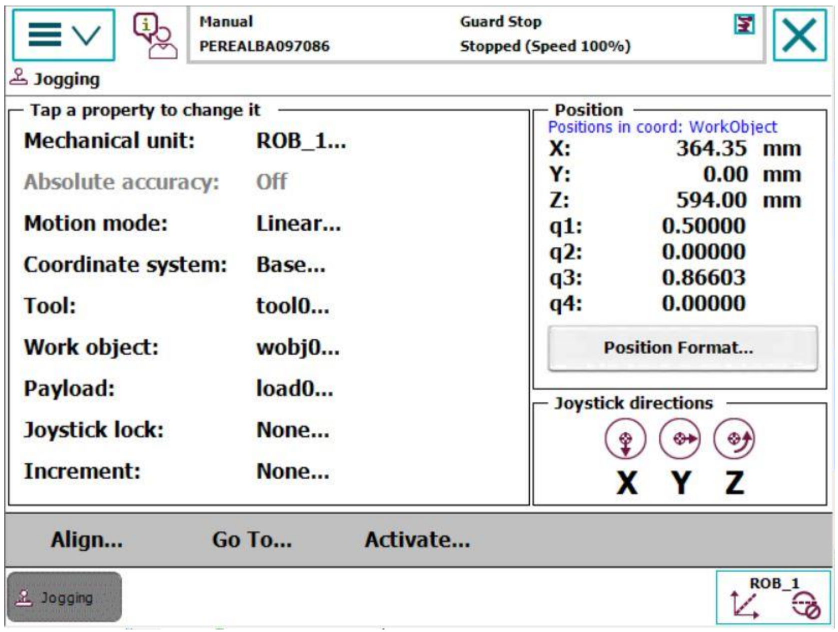
Jogging the robot is a procedure that allows the operator to directly control the robot's axes or end-effector pose by a joystick, allowing empirical definition of robotic elements in the controller's script, [5].

Jogging requires the robot be in *manual mode*. To find the Jogging menu in the FlexPendant™ open the ABB menu and find *Jogging* in the left-hand column.

In the left side of the Jogging menu, the is a variety of properties about the robot's configuration, such as the current mechanical unit, motion mode, and various objects used to calculate the end-effectors current position and orientation.

In the upper-right corner of the menu, the *Position* panel shows the joint angles if jogging in axis mode, or the end-effector position and orientation if jogging in linear or reorient mode.

In the lower-right corner of the menu the *Joystick Directions* panel shows the how movements of the joystick will affect the robot, again depending on the Motion mode.



Manual
PEREALBA097086
Guard Stop
Stopped (Speed 100%)

Jogging

Tap a property to change it

Mechanical unit:	ROB_1...
Absolute accuracy:	Off
Motion mode:	Linear...
Coordinate system:	Base...
Tool:	tool0...
Work object:	wobj0...
Payload:	load0...
Joystick lock:	None...
Increment:	None...

Position
Positions in coord: WorkObject

X:	364.35	mm
Y:	0.00	mm
Z:	594.00	mm
q1:	0.50000	
q2:	0.00000	
q3:	0.86603	
q4:	0.00000	

Figure 5-5: Jogging Menu on FlexPendant™

Per-axis jogging

To jog axes individually, select *Motion Mode* in the Jogging menu and select *Axis 1-3* or *Axis 4-6*, depending on which axis is desired, and press *OK*. The *Joystick directions* panel will show how each selected axis can be moved by the joystick, [5].

Linear Jogging

To jog the end-effector on a linear path in physical space, select *Motion Mode* in the Jogging menu and select *Linear*. The *Joystick directions* panel will show which joystick movements correspond to each cartesian direction, [5].

Inverse Kinematics

Linear movements use Inverse Kinematics to calculate the specific joint movements required to produce the desired end-effector movement. Inverse kinematics are further described in Chapter 4.

Since, when jogging, the controller only receives information on its commanded move in the instant it makes that move, the inverse kinematic calculation must be redone for each discrete computation cycle.

There are places in the robot’s workspace where joints are aligned in such a way that movements between the joints become dependent of each other. This represents a singularity in the kinematic matrix, where inverse kinematics are impossible. Linear movements should avoid these singularities in the robot workspace, and an error message will interrupt operation should a singularity get too close, [6].

Reorientation

To jog the end-effector in a rotating path around the Tool Center Point, select *Motion Mode* in the Jogging menu and select *Reorient*. The *Joystick directions* panel will show which joystick movements correspond to each Euler-angle movement.

As with linear jogging, reorientation relies on inverse kinematics to calculate the joint values necessary for the new position. Like linear jogging, it is also vulnerable to approaching singularities, and will throw an error if one is approached, [5].

Coordinate Frames

A coordinate frame exists for any object in space which has its own absolute position and rotation and can possibly have other coordinates as children. There are multiple coordinate frames available for use by controller. The coordinate frame selected in the jogging menu determines the location and orientation of the cardinal axes used for jogging, [5].

Poses

Position

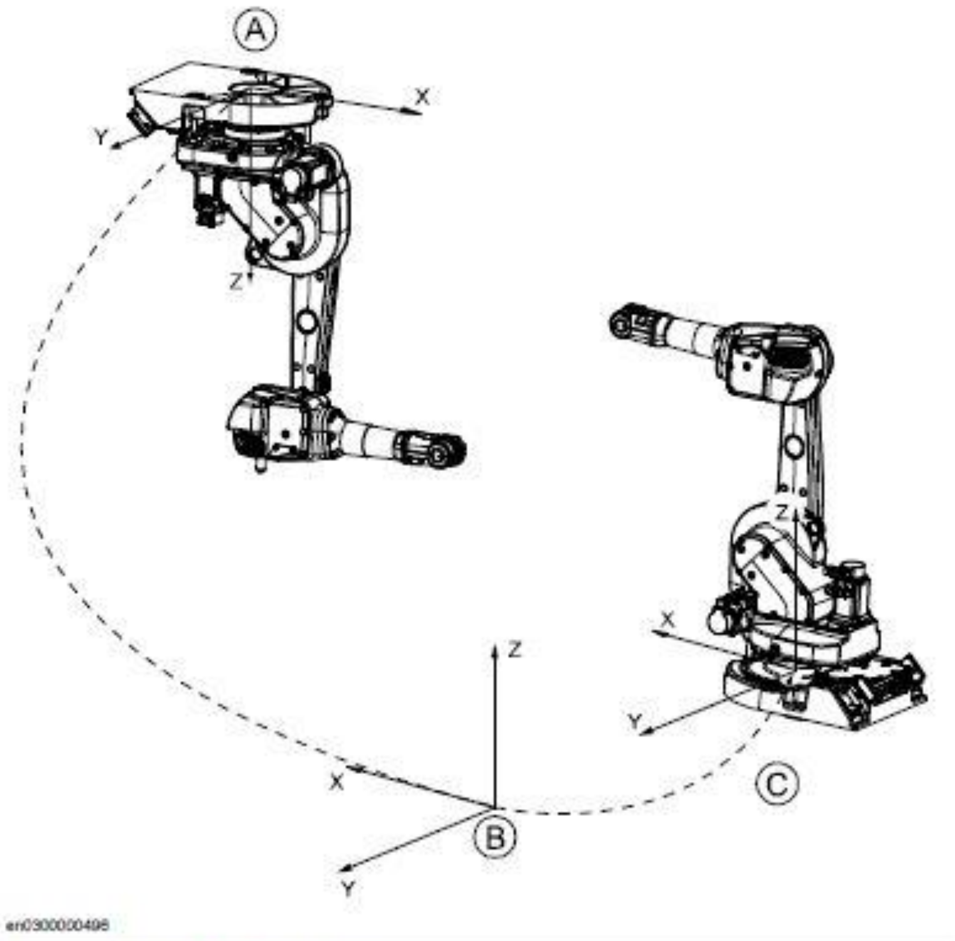
Position vectors in RAPID™ are represented by pos, a 3D vector with each element of type num. In robot base coordinates, the X-axis is designated as forward and the Z-axis is up; consequently, the Y-axis is left, signifying that RAPID™ uses a right-handed coordinate system, [9].

Rotation

Orientations of objects in RAPID™ are recorded as orient, a quaternion derived from a rotation axis in the same coordinate frame as the position, [9].

World

The most basic coordinate system in a robot's controller is called the world frame. This frame is system specific and is therefore defined once per robot controller. Even if a system has multiple robots running inside it, the world coordinate frame is common among all of them.



A	Base coordinate system
B	World coordinate system
C	Base coordinate system

Figure 5-6:World Coordinates and Base Coordinates. Source: [5]

Base

The base frame is specified on a per-robot basis and is by default at the bottom center of a robot's base. In a system with a single robot where neither world frame nor base frame have been explicitly defined, the world and base frames are identical to one another.

Work Object

In addition to the world and base frames, each work object has its own coordinate system. A work object can be any specific element in a system which the developer feels that warrants its own definition. Work objects are useful to serve as parents of certain smaller items in the system. The default work object is wobj0, defined in the default BASE system module.

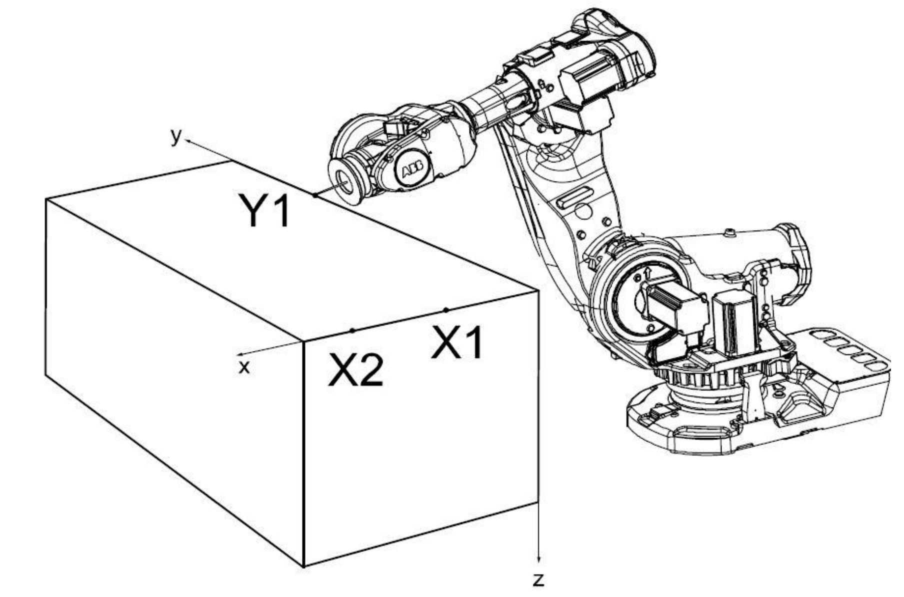


Figure 5-7:Work object coordinates and definition process. Source: [5]

Tool (End-effector)

The active tool has its own moving coordinate frame. It can be useful to jog the robot using the tool coordinates if certain diagonal geometry needs to be traversed that is difficult to navigate in other coordinate systems. Just like work objects, a default tool is given in the BASE module called tool0, which has a tool center point at the end of link 6, and no inertia properties, [9].

Robot Target

Robot targets are places where the robot moves during operation. Robot targets do not have coordinate systems in themselves, but they are always defined with respect to the current jogging work object’s coordinate system. The robot target can be seen to have its own local position set compared to whatever work object is specified in each move instruction, [9].

Calibration

Calibration of the robot ensures that the zero positions the controller sees match with the reference configuration of the robot. The robot axes are measured at each joint with a small rotary encoder. Each rotary encoder views the angle that it is currently in, and the number of rotations the encoder has gone through to calculate the joint angle. If the number of complete rotations is incorrect however, the joint angle calculation is inaccurate, and the joint must be recalibrated.

At each joint, a large mark is located on one joint and a small mark is located nearby on the other joint. The reference position is defined as where the two marks overlap. Therefore, when calibrating, the joints must be moved such that the small mark is entirely within the large mark at each joint. If the small mark is completely inside the large mark, the encoder will recognize this as within half a revolution of the reference value and will select the zero point automatically.

If all axes are to be recalibrated, it is typically best to align the joints closest to the end-effector first, using the lower joints to place the upper joints in a location convenient for inspection, [5].

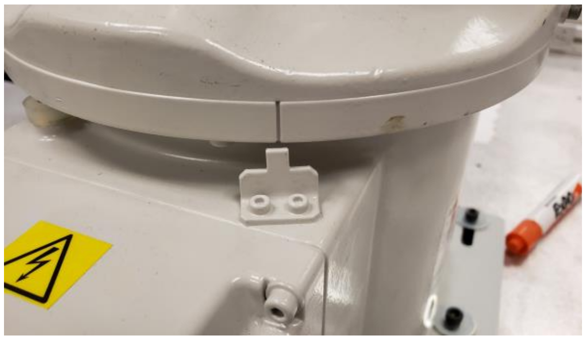


Figure 5-9: Calibration marks for Axis 1

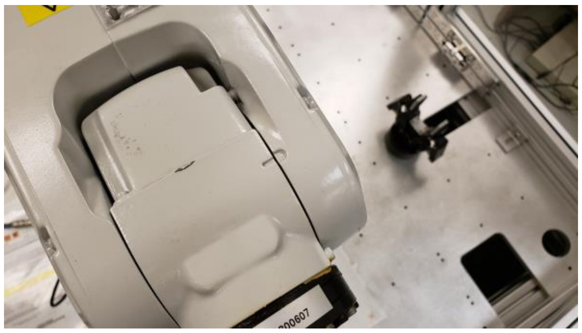
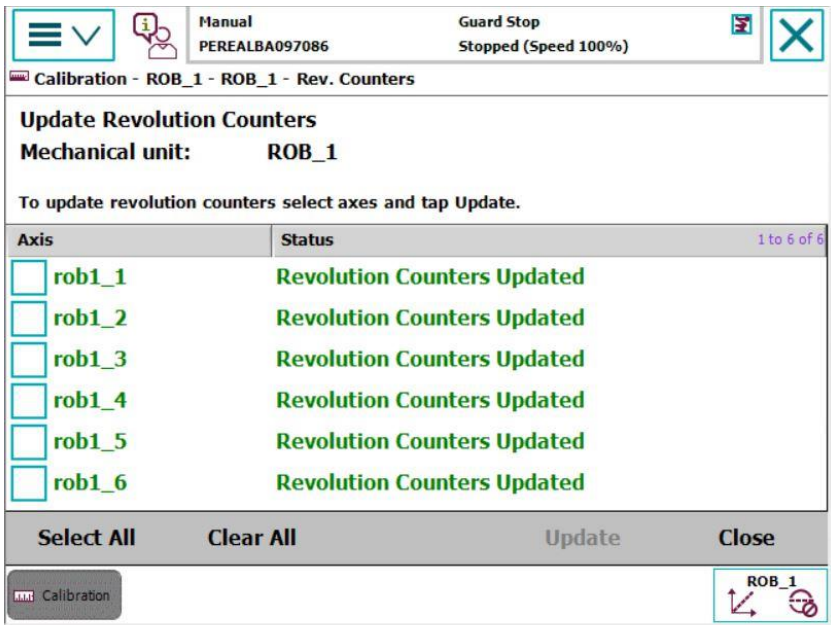


Figure 5-8: Calibration marks for Axis 5

To start calibration, open the ABB menu and find *Calibration.* Select the mechanical unit corresponding with the robot being serviced and tap *Rev Counters.* A list of axes will be displayed; select which axis or axes has been aligned and click *Update.*



Manual
PEREALBA097086
Guard Stop
Stopped (Speed 100%)

Calibration - ROB_1 - ROB_1 - Rev. Counters

Update Revolution Counters
Mechanical unit: ROB_1

To update revolution counters select axes and tap Update.

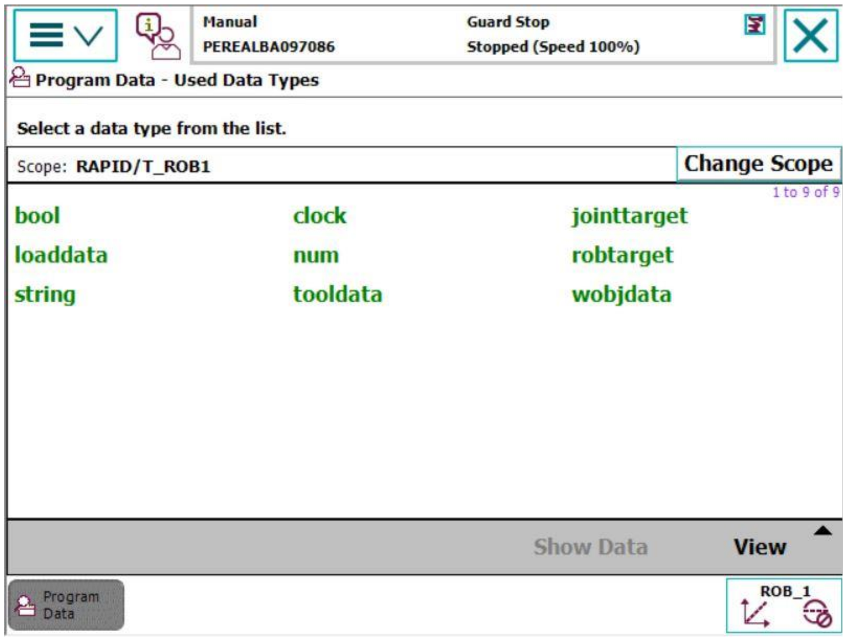
Axis	Status
rob1_1	Revolution Counters Updated
rob1_2	Revolution Counters Updated
rob1_3	Revolution Counters Updated
rob1_4	Revolution Counters Updated
rob1_5	Revolution Counters Updated
rob1_6	Revolution Counters Updated

Figure 5-10: Calibration Menu showing option to update rev counters

Note that this is a simple operation for misaligned axes; if the robot needs more in-depth calibration, these procedures should be left to a certified technician.

Data Editing

During the programming process, it may be necessary to create variables on the Module (or Task) level. On the FlexPendant™, this is achieved by using the *Program Data* view.



Manual
PEREALBA097086
Guard Stop
Stopped (Speed 100%)

Program Data - Used Data Types

Select a data type from the list.

Scope: RAPID/T_ROB1

bool	clock	jointtarget
loaddata	num	robtarget
string	tooldata	wobjdata

Figure 5-11: Program Data Menu

To access the *Program Data* view, open the ABB menu, and find *Program Data* in the left column. This menu shows all data types listed in the current scope, depending on some settings, [5].

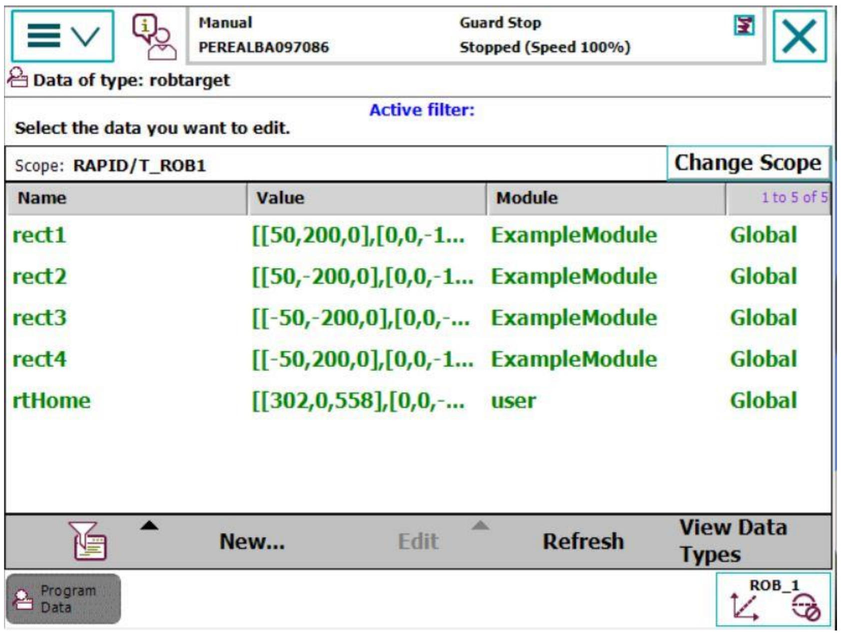
Scope and Viewing

To change the current scope of variables listed under, press the *Change Scope* button at the upper-right corner of the menu. The first two options are *Built-In Data Only* (this option does not allow addition of any data to the program; it is read-only) and *Current Execution*, which looks at all data marked as *Global*, or are in the current Task, Module, and Routine that currently has the *Program Pointer*. Other selectable scopes are any Task, Module, or Routine in the selected Module.

At the lower right of the *Program Data* view there is a button labeled *View*. Using this button, one can choose between all available data types, and only the data types that currently exist in the selected scope. This option allows creation of new data of types that have not been introduced in the program yet, while keeping the data menu uncluttered the rest of the time, [5].

Editing data

Double press, or press and select *Show Data*, on any desired data type. A list of type-specific data with their values are then displayed on the screen. Selecting any of these data and


Manual
PEREALBA097086
Guard Stop
Stopped (Speed 100%)

Data of type: robtarget

Active filter:

Select the data you want to edit.

Scope: RAPID/T_ROB1

Name	Value	Module	1 to 5 of 5
rect1	[[50,200,0],[0,0,-1...	ExampleModule	Global
rect2	[[50,-200,0],[0,0,-1...	ExampleModule	Global
rect3	[[-50,-200,0],[0,0,-...	ExampleModule	Global
rect4	[[-50,200,0],[0,0,-1...	ExampleModule	Global
rtHome	[[302,0,558],[0,0,-...	user	Global

Figure 5-12: Program Data Menu, searching all robot target saved in the controller

using the *Edit* menu at the bottom of the screen allows changing, copying or deleting these variables. New variables can be declared and given values via the adjacent *New* menu, [5].

The *View Data Types* button at the lower right returns to the top-level *Program Data* screen.

Complex data

Some complex data, such as tools, robot targets and work objects, can be defined empirically rather than explicitly. New robot targets default their positions to the current end-effector position, and existing robot targets can do the same using the *Modify Position* command in the *Edit* menu, [5].

Tool data and Work Object definition is slightly trickier. Once the data has been declared, the *Define* command in the *Edit* menu loads a method of defining several poses that, for a Tool, bring the Tool Center Point to the same point in world space, or for a Work Object, define the X-Y plane of the object.

Creating a tool

Creating a tool is a process that involves taking the desired tool center point to a fixed point in space, usually some object that the tool is supposed to touch. This is done from several different orientations, most commonly, and by default, four. The algorithm finds the center of rotation between each pair of points and averages them for better accuracy, [5].

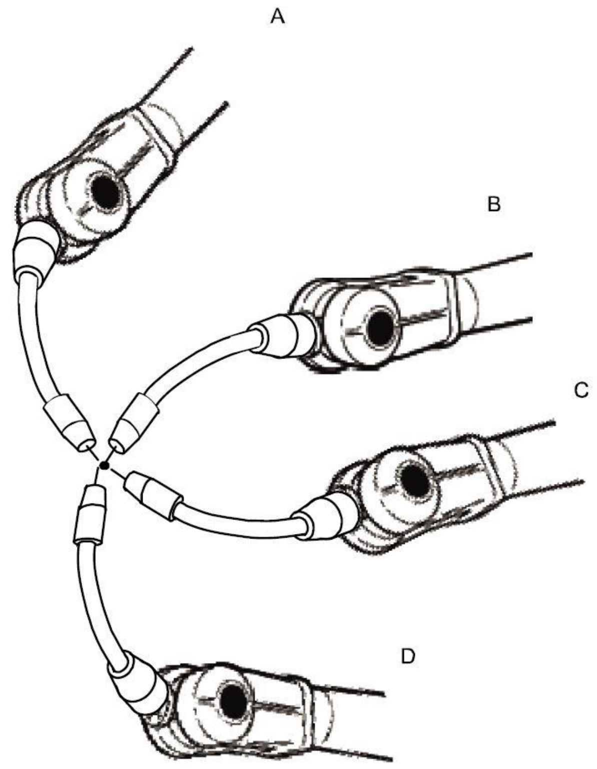


Figure 5-13: Tool definition via the 4-Point process. Source: [5]

Creating a work-object

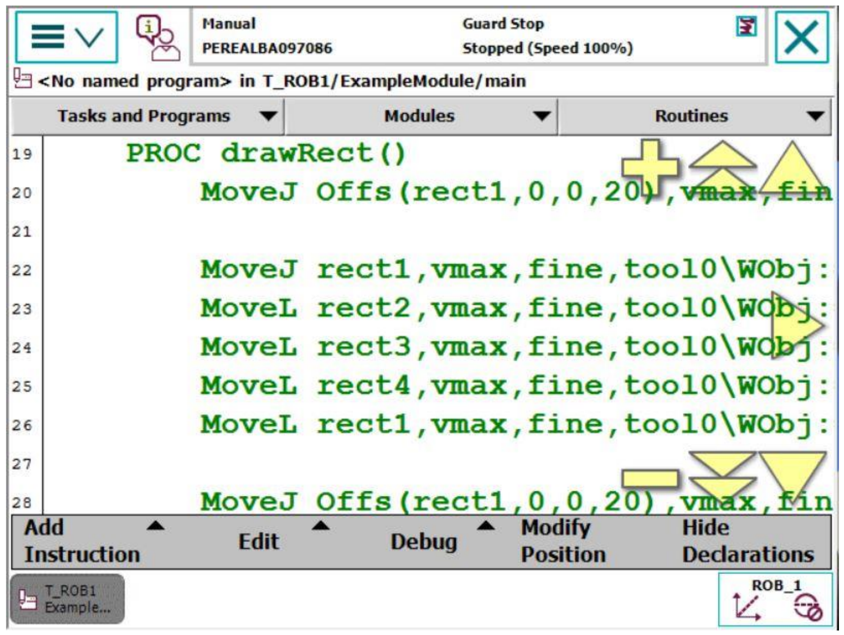
Defining a work object is a straight-forward manner and is as accurate as the placement of the tool center point used in its definition. Work objects are defined in world space by picking three points on the work object. The first two defined points form a line along which the work object's X-axis is defined. The third point sets the direction of the Y-axis, which is defined as the shortest line projecting from the X-axis to the Y-point. The work-object's position is consequently the intersection between these new X and Y-axes, with its orientation defined by the axis directions, [5].

Creating a robot target

The process of creating a robot target is simple. The robot is jogged to the pose which is the desired robot target. The position will be defined relative to the active jogging tool and work object; however, those components are not stored in the target's data. When creating a robot target, it will automatically default to the robot's current pose, [5].

Program Editing

The main scripting for the robot is accessed from *Program Editor*. This is found in the ABB menu on the left-hand side. Upon opening, it will open the active *PROC main* on the control, [5].



Manual
PEREALBA097086
Guard Stop
Stopped (Speed 100%)
<No named program> in T_ROB1/ExampleModule/main

```
    PROC drawRect()
        MoveJ Offs(rect1,0,0,20),vmax,fin

        MoveJ rect1,vmax,fine,tool0\WObj:
        MoveL rect2,vmax,fine,tool0\WObj:
        MoveL rect3,vmax,fine,tool0\WObj:
        MoveL rect4,vmax,fine,tool0\WObj:
        MoveL rect1,vmax,fine,tool0\WObj:

        MoveJ Offs(rect1,0,0,20),vmax,fin
```

Figure 5-14: Program Editor, showing a procedure

Editing routines

To edit a routine, one can use three buttons at the top, *Tasks and Programs, Modules,* and *Routines,* to navigate to the desired routine. When the desired routine is selected, the *Show Routine* button at the bottom opens the routine for editing.

Adding routines

To make a new routine, navigate to the desired module to contain the routine, then use *File > New Routine…* to add a custom routine to that Module.

Instructions

Routines are made up of a series of instructions. These instructions vary widely in their purpose; examples include procedure calls, movement instructions, and assignment instructions. Chapter 6 goes into more detail about the types and requirements of instructions.

Adding an instruction

When viewing a routine in the *Program Editor,* the instruction menu is activated by the *Add Instruction* button at the bottom left. The instructions are organized into various categories, which may be selected by a dropdown at the top of the instruction menu.

Editing instructions

To edit an existing instruction, one can select the instruction's handle (the name of the instruction call, e.g. *MoveL, WaitTime*) and go to *Edit > Change Selected.* A menu showing all required instruction arguments, and some optional, each of which may be changed however desired.

Testing and debugging

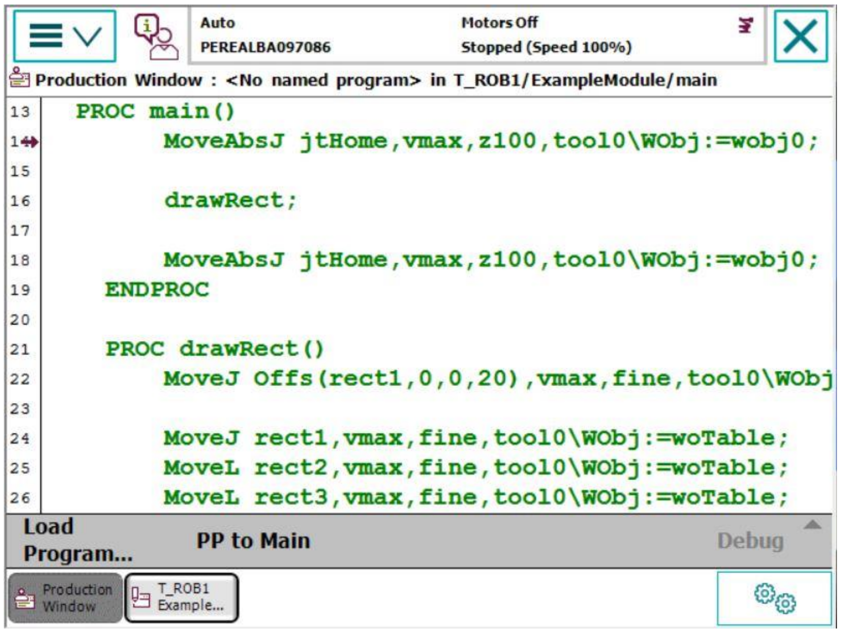
The debug menu shows a variety of tools for testing a program in manual mode. It allows movement of the Program Pointer, a marker showing the next statement that will be executed by the program if it is run. The program pointer can be moved to multiple places, such as the selected instruction, the beginning of a routine, or to the beginning of the main routine as the program would execute in automatic mode. Holding the guard-stop and play buttons will allow the program to slowly execute each instruction sequentially, moving the program pointer to the next instruction when the current one is completed, [5].

Module Text

Notice that in the *Program Editor* view, there is a button that toggles declarations, labeled *Show Declaration* or *Hide Declaration.* When this is set to show declarations, not only the current routine, but other routines and variables currently in the module will also be displayed. This is the module's complete RAPID™ code and is how all aspects of the robot script are stored on the controller.

Automatic Operation

When the mode key is set to automatic mode, the Production Window will automatically appear on the FlexPendant™. From the Production Window, real-time automatic mode execution of the RAPID™ code can be viewed. The robot runs at full speed during automatic mode, and customization of the program at this point is limited to some path tweaking via the Hot Edit menu. Hot Editing is an operation for tuning robot targets to optimize production operations without reverting to manual mode for large changes. This operation is beyond the scope of this project, however, [5].


Auto
PEREALBA097086
Motors Off
Stopped (Speed 100%)
Production Window : <No named program> in T_ROB1/ExampleModule/main

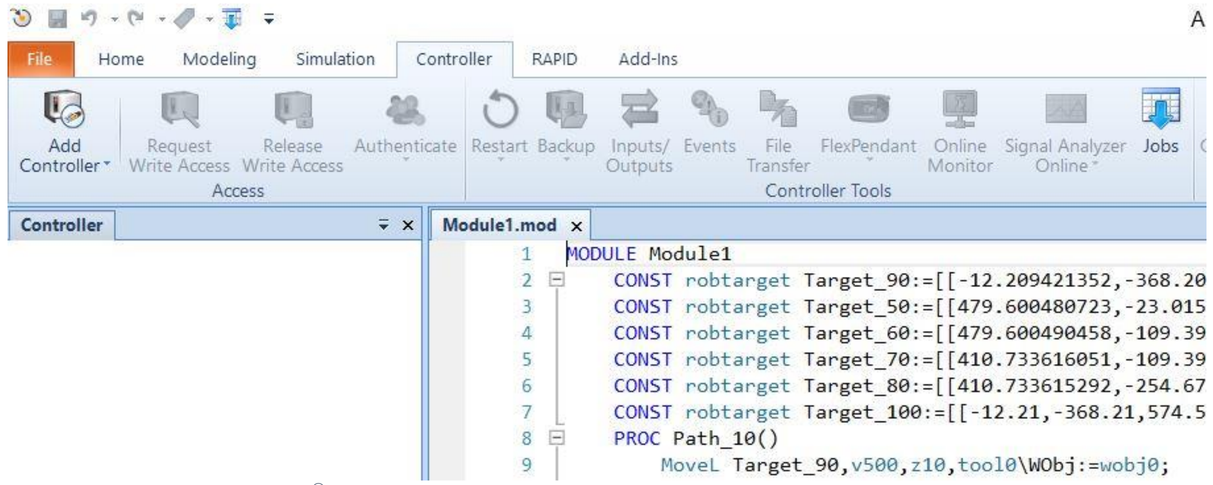
```
  PROC main()
      MoveAbsJ jtHome,vmax,z100,tool0\WObj:=wobj0;

      drawRect;

      MoveAbsJ jtHome,vmax,z100,tool0\WObj:=wobj0;
    ENDPROC

    PROC drawRect()
      MoveJ Offs(rect1,0,0,20),vmax,fine,tool0\WObj

      MoveJ rect1,vmax,fine,tool0\WObj:=woTable;
      MoveL rect2,vmax,fine,tool0\WObj:=woTable;
      MoveL rect3,vmax,fine,tool0\WObj:=woTable;
```

Figure 5-15: Production Window, showing the program progressing through PROC main

RobotStudio® RAPID™ Editing

RobotStudio® has a couple different main features, of which the most relevant to this paper is its ability to remotely access and change the RAPID™ code on a robot controller set to manual mode. The tools for this feature in RobotStudio® can be accessed by the Controller and RAPID™ tabs. The computer running RobotStudio® must be on the same Local Area Network (LAN) as the Robot Controller, or the computer and controller must be directly connected with an Ethernet cord, [3].

To connect to a controller, switch to the Controller tab in RobotStudio®. The Add Controllers drop-down menu should show available controllers to connect to. However, if the



```
MODULE Module1
    CONST robtarget Target_90:=[[-12.209421352,-368.20
    CONST robtarget Target_50:=[[479.600480723,-23.015
    CONST robtarget Target_60:=[[479.600490458,-109.39
    CONST robtarget Target_70:=[[410.733616051,-109.39
    CONST robtarget Target_80:=[[410.733615292,-254.67
    CONST robtarget Target_100:=[[-12.21,-368.21,574.5
    PROC Path_10()
        MoveL Target_90,v500,z10,tool0\WObj:=wobj0;
```

Figure 5-16: RobotStudio®'s Controller Tab, showing buttons used to connect to a controller. Since a controller is not currently connected, most controls are greyed out.

controller is not listed as online or at all, one can connect manually by entering the IP address assigned to the controller, which can be found in the FlexPendant™'s System Info menu.

Once the controller is connected, one can view the controller's RAPID™ code, broken into tasks and modules, by opening the RAPID™ icon in the left-hand menu. To edit the code, however, one must request write access. The button for this is in the Controller tab, next to the Add Controllers drop-down. A notice will appear on the FlexPendant™ that an external source has requested write-access, giving the operator a choice to grant or deny. Upon granting write access, the RobotStudio® client has privileges to edit the code and apply it to the controller. The FlexPendant™ can revoke write-access at any time, or the RobotStudio® client can release write-access by a button next to the Request Write-Access button.

With write-access, the user is free to edit the RAPID™ variables and routines in existing modules or add/delete modules and tasks from the controller. When the user wishes to apply changes to the controller, they can find the Apply button in the RAPID™ tab, at the top-center of the screen, [3].

Note that debug testing and jogging must be done from the FlexPendant™ and write-access must be released or revoked for the FlexPendant™ to become interactable again.

Chapter Summary

This chapter gives an overview of the operation of the IRB 120™, controlled by an IRC5™ Robot Controller equipped with the FlexPendant™. It begins by outlining some safety features of the system; namely the workspace cage, force sensors, emergency stop buttons, and mode of operation. The IRC5™ controller also has several switches on its panel which control the power, operation mode, motors and joint brakes.

It then outlines some basic features and operation of the robot. Jogging is performed by accessing the eponymous menu on the FlexPendant™ and using the joystick to directly maneuver the robot. It also covers basic calibration and editing of RAPID™ code directly on the FlexPendant™. It shows the differences in capability and function between manual and automatic modes.

The chapter concludes with an introduction to editing the RAPID™ code on the controller using RobotStudio® on an external computer. As supreme control authority rests with the FlexPendant™, write access must be given to the RobotStudio® client before it can edit the code. Any controller connected to the same local area network as the computer or connected directly by ethernet cable can be connected. The client itself has efficient coding tools to create and verify the RAPID™ code prior to deployment on the robot and is the typical place where most professionals write their programs.

6 Introduction to RAPID™

ABB systems use the RAPID™ language to set instructions and data that their robotic systems use in each run cycle.

RAPID™ Structure

Tasks

Tasks are the most essential structure of RAPID™ code, with code execution happening on a per-Task basis. Tasks contain one or more Modules, one and only one of which must have a main procedure (PROC main) as a place for the Task to begin its code execution. Most

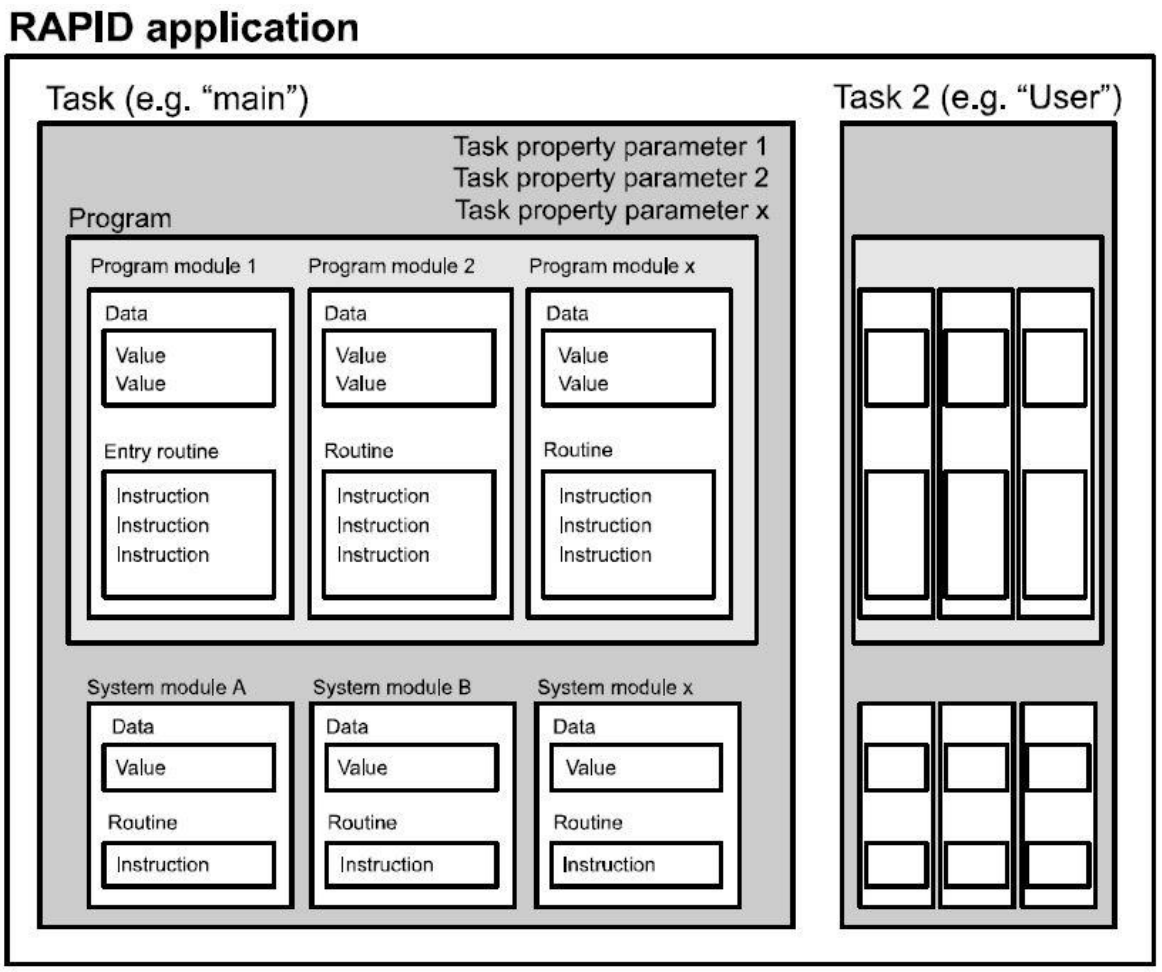


Figure 6-1: Diagram showing the hierarchy of RAPID™ Code. Source: [5]

controllers support some form of multitasking, where the controller contains multiple Tasks that all run in parallel, [5].

Modules

A module is a constituent unit of a Task. Modules are the objects where the actual RAPID™ text is stored, both variables and routines. Any number of Modules can be used in a Task, which can be used for organizational purposes.

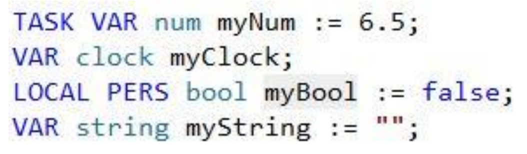
Variables

Data at the module level can be created via the Program Data menu outlined in Chapter 5 or typed directly into the module text, either in RobotStudio® or on the FlexPendant™ (directly editing RAPID™ code by typing is not recommended on the FlexPendant™). For the physically defined objects such as tools, work objects and robot targets, the typical manual definition processes should be used.

Simpler variables, such as those without poses, or whose poses are known from external sources such as Computer Aided Design (CAD) models, are good candidates for directly declaring and initializing in the RAPID™ code, [10].

Variable Syntax

Variable declarations are statements that create a variable and initialization is assigning the variable a value before the program starts. The typical syntax of a variable declaration is a scope (optional), a symbol type, a datatype, and finally the variable's name, all separated by spaces. The initialization, required in RAPID™ for most data types, follows with the assignment



```
TASK VAR num myNum := 6.5;
VAR clock myClock;
LOCAL PERS bool myBool := false;
VAR string myString := "";
```

Figure 6-2: Several variable declarations. Scopes include GLOBAL (default, implicit), LOCAL and TASK. Symbol Types include VAR, PERS, and CONST. After these elements follow the data type and name of the variables.

operator “:=” followed by a datatype-specific permutation of data, finally ended with a semicolon, [10].

Scopes

Variables in the RAPID™ language are declared with a scope. The scope of a variable controls the locations from which the variable is accessible. GLOBAL, the default scope of a variable which is used if an explicit scope is omitted, allows a variable to be accessed from anywhere in a controller’s RAPID™ code, including other Tasks. The scope TASK allows a variable to be accessed from any Module in the same task as the variable definition. The scope LOCAL further restricts the variable to only be accessible from the same Module, [10].

Symbol Types

Variables are also defined with a symbol type, which governs how it is interacted with by the program. The simplest type, CONST, represents a constant in the code, which cannot be changed anywhere other than its original definition; therefore, it is baked into the code at runtime. VAR and PERS, by contrast, are variables which can be changed at runtime from anywhere within their scopes. The primary difference between VAR and PERS is what happens to the data during a system reboot. A VAR loses its current value and will be redeclared with its default value when the system restarts, while a PERS is persistent through a reboot and will hold its value across sessions, [10].

Datatypes

There are seventy-eight datatypes in RAPID™, the vast majority of which are not covered in this document. More specific details on each datatype can be found by referring to this technical reference manual on RAPID™, [10]. Some examples of commonly used datatypes in the language are given below.

Primitives: bool, num, string

The most basic data types in the RAPID™ language are the *bool*, *num*, and *string*. Like in most other languages, a *bool* represents a binary value of either *True* or *False,* and a *string* represents a sequence of characters. In RAPID™, a *num* is a 32-bit numeric type that can variably represent an integer or floating-point number, depending on the value assigned to it.

Other simple types may have slightly different implementations of the same concept, (e.g. the type *dionum* is a binary type which is represented by *0* or *1*, rather than *True* or *False*, and *dnum* is a 64-bit numeric value that works similarly to *num*).

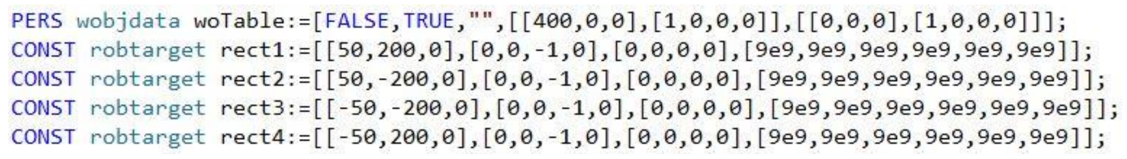
Composite data: robtarget, tooldata, wobjdata

There are many compound datatypes in RAPID™ which comprise of several primitive data. Three prominent examples are *robtarget*, *tooldata*, and *wobjdata.* All three of these objects use further compound datatypes like *pos* and *orient*, which are 3D position vectors and quaternions made of multiple *num* elements.

The type *tooldata* is made when the user trains a new tool on the robot. It consists of the *Tool Center Point*, which is a *pose* (Position and Orientation pair), as well as a *loaddata* consisting of a mass, center of gravity, and axis of moment. The tool can serve as a coordinate frame by which the robot can move. When a move statement is given, the robot calculates the path to take so that the *tooldata* has the same position and rotation as the chosen robot target.

Work objects, with their RAPID™ type name *wobjdata*, are objects defined in the space around a given robot which serve as parents to robot targets in the path. Work objects can be stationary or moving within the robot's workspace.

Robot targets, of RAPID™ type name *robtarget*, are places defined in 3D space that can be assigned to move statements in a robot's path. They are typically made by a robot's jogging position relative to the active work object. Hence, care must be taken when using move statements to specify the same work object that was used to create the robot target, even if the work object's pose has changed.



```
PERS wobjdata woTable:=[FALSE,TRUE,"",[[400,0,0],[1,0,0,0]],[[0,0,0],[1,0,0,0]]];
CONST robtarget rect1:=[[50,200,0],[0,0,-1,0],[0,0,0,0],[9e9,9e9,9e9,9e9,9e9,9e9]];
CONST robtarget rect2:=[[50,-200,0],[0,0,-1,0],[0,0,0,0],[9e9,9e9,9e9,9e9,9e9,9e9]];
CONST robtarget rect3:=[[-50,-200,0],[0,0,-1,0],[0,0,0,0],[9e9,9e9,9e9,9e9,9e9,9e9]];
CONST robtarget rect4:=[[-50,200,0],[0,0,-1,0],[0,0,0,0],[9e9,9e9,9e9,9e9,9e9,9e9]];
```

Figure 6-3: Complex data variables. Included in both wobjdata and robtarget are position vectors and rotation quaternions.

Special objects: clock, zonedata

There are some data types that do not have explicit primitive types but are self-contained data that have specific functions. One example of this is the *clock* data type. The *clock* has no exposed field types but can be passed as an argument to numerous predefined routines and stores an elapsed time since the clock was started.

Another example is *zonedata*, a type that encodes the behavior of the robot on intermediate targets in its path. There are several predefined *zonedata*, named by the letter ‘z’ prefixed to a distance from a robot target in millimeters. This distance is the proximity the tool must be to the designated robot target before it changes paths to the next target.

Routines

Routines are procedural methods in RAPID™. They can be called from other routines, be passed variables as arguments, and can return variables if necessary. They allow compartmentalization of specific tasks to simplify code and repeat complicated procedures. There are three routine types in RAPID™, each with a slightly different purpose.

Routine Syntax

Routines begin with a definition containing the routine type first, either *PROC, FUNC*, or *TRAP*. If the routine is a function, the return type follows. Next is the routine’s name, followed by parentheses unless the routine is a trap. Inside these parentheses is a list of all arguments of the procedure or function, each being specified by a type and a name, with the arguments separated by commas.

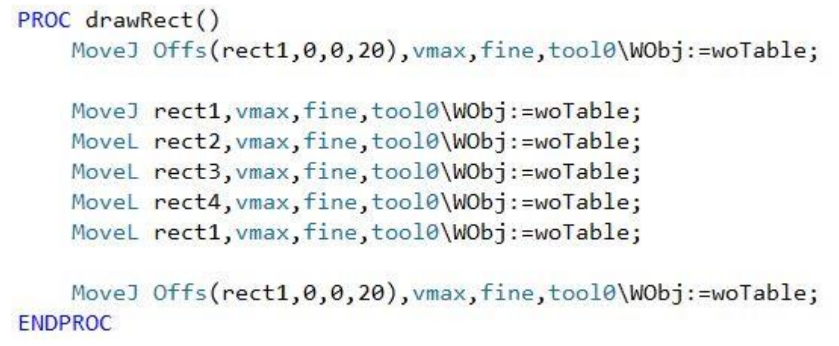
The end line of the routine is simply *END* concatenated with the routine type, e.g. a *PROC* would have an end line labeled *ENDPROC*, and a *FUNC* (of any return type) would end with *ENDFUNC.*

All non-comment lines between these beginning and end lines are either program flow controls such as if blocks, while loops, or error handlers, or they are instructions, which could be any predefined instruction, such as a move statement, a call to another user-made procedure, or an assignment to a variable.

Routine Types

PROC

PROC, or a procedure, is the primary routine type, which can take in any number of arguments and does not return anything. Any scope permitted *PROC* can be called from code in execution at any point. The program pointer will move to the new *PROC* and execute all lines.



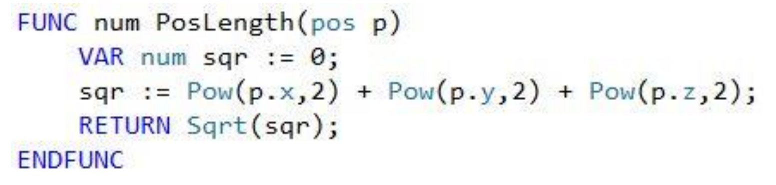
```
PROC drawRect()
    MoveJ Offs(rect1,0,0,20),vmax,fine,tool0\WObj:=woTable;

    MoveJ rect1,vmax,fine,tool0\WObj:=woTable;
    MoveL rect2,vmax,fine,tool0\WObj:=woTable;
    MoveL rect3,vmax,fine,tool0\WObj:=woTable;
    MoveL rect4,vmax,fine,tool0\WObj:=woTable;
    MoveL rect1,vmax,fine,tool0\WObj:=woTable;

    MoveJ Offs(rect1,0,0,20),vmax,fine,tool0\WObj:=woTable;
ENDPROC
```

Figure 6-4: Example of a PROC declaration, with several move instructions. This procedure has no arguments.

When all statements have been executed or the instruction *Return* is declared, the *PROC* ends execution and returns the program pointer to the code which called it.

FUNC

If a routine must return some data when execution ends, it is called a *FUNC*, or function. These are called wherever the return type serves a purpose, e.g. as an argument to another routine, or as an assignment to a variable. Execution of a *FUNC* works in a similar way to a *PROC*; when it is called, the program pointer shifts to the *FUNC* and executes its statements sequentially until it returns its value; then the value is used in-place of the function call and the external program continues normally.



```
FUNC num PosLength(pos p)
    VAR num sqr := 0;
    sqr := Pow(p.x,2) + Pow(p.y,2) + Pow(p.z,2);
    RETURN Sqrt(sqr);
ENDFUNC
```

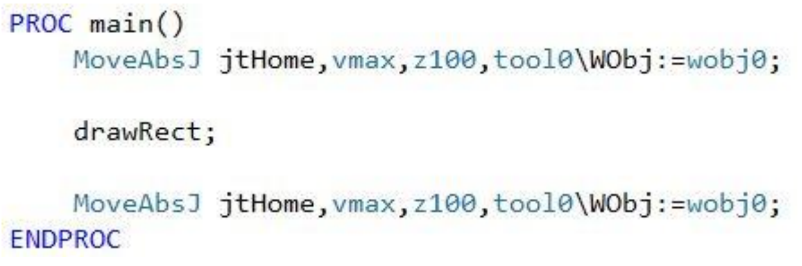
Figure 6-5: Example of a FUNC declaration, with a single position vector argument returning a number corresponding to the length of the vector

TRAP

The final routine type is a *TRAP*. Trap routines are like procedures in that they are simply sequential executions of code. However, *TRAP* serves the special purpose of handling certain interrupts in the code. When the main code is operating as normally, but a certain interrupt signal is engaged, it can stop the program at any point and run the *TRAP* instead.

PROC Main

The main procedure, shown in RAPID™ as *PROC* main, is the starting point for any task execution. Every RAPID™ task must have one, and only one, main procedure defined in its program modules. The task's execution begins with the first instruction given in the main procedure. If a system uses multiple tasks, each task will operate independently of the others, and they will run asynchronously. However, operation on a specific mechanical unit (robot) can only have one task.



```
PROC main()
    MoveAbsJ jtHome,vmax,z100,tool0\WObj:=wobj0;

    drawRect;

    MoveAbsJ jtHome,vmax,z100,tool0\WObj:=wobj0;
ENDPROC
```

Figure 6-6: PROC main, showing two joint movements before and after a ProcCall to another procedure.

Instructions

Each line in a routine that performs some sort of task is called an instruction, or statement. RAPID™, like most text-based programming languages, evaluates instructions in the order of their appearance, starting in *PROC* main. Instructions can be broadly separated into several categories.

Move Statements

Move statements are a kind of instruction that physically moves the robot toward a target in some manner. The three basic movement types are *MoveJ*, *MoveL*, and *MoveC*. Each move type additionally has different related instructions that can do other things, such as set a digital output signal or run a procedure in parallel with the move.

MoveJ

MoveJ is the simplest kind of movement, which only does inverse kinematics of the movement target to find the desired joint values, then interpolates each joint rotation toward that value; this is a linear interpolation in joint space. This is the quickest movement between points as inverse kinematics for the path between are not required, and the joint rotations are direct and efficient. MoveJ should be used whenever the motion taken between current and target positions is not path sensitive.

A variant of this movement mode, MoveAbsJ, takes a jointtarget instead of a robtarget as its first argument. A jointtarget is a target defined by its axis angles rather than a pose in physical space. If the angles are explicitly known by the robotic developer, this instruction is useful to avoid kinematic solving for this movement. MoveExtJ performs similarly but is meant to move the external axes connected to the system.

MoveL

MoveL is a slightly more complex movement, which is a straight line in physical space. *MoveL* is most useful when the end-effector needs to move tangent to some object surface, or axial to a narrow channel in an object. Inverse kinematics need to be traced at intermediate points on the linear path; therefore, the individual joint movements are not strictly linear, and may take arcing values to keep the end-effector on its linear path. *MoveL* is also susceptible to singularities in the kinematic matrices, so care should be taken to avoid these when possible.

MoveC

MoveC is a move instruction that interpolates a circular path in physical space. It takes in two targets as arguments, moving in a circular path from starting pose, through an intermediate target, and ending at the final target. This movement type is useful when needing to move around circular objects or run a path concentric with them. Like *MoveL,* it needs to calculate inverse kinematics for its path, and is vulnerable to singular configurations.

Other movement types

Search

The instructions *SearchL*, *SearchC,* and *SearchExtJ* perform similarly to the move statements of the same suffix; however, these statements will cease movement when the digital input argument of the search instruction becomes true. This is typically useful for an unknown precise location of an object to be picked up, as sensors can tell the system when the object has been found.

Trigg

TriggJ and *TriggL* are also similar to their Move statement counterparts, but they are able to trigger a parallel running event at a fixed point during its movement; for example, if the end-effector vacuum tool should engage when the tool is 10 millimeters from the target during a linear movement, TriggL should be used with a corresponding triggdata object as an argument.

Move Statement Components

Robot target

All move instructions accept robot targets (or joint targets) as their first arguments. Joint and Linear movements accept a single target, while Circular movements accept two. Functions that return robtarget, such as the *Offs()* function, can be used in place of a named robot target.

Speed data

The speed data argument controls the speed of a given movement while the robot operates. Predefined speeddata are specified with a small 'v' followed by its speed in millimeters per second, e.g. "v100". Other predefined speeddata are specified for movement of external linear and rotational axes by the *MoveExtJ* instruction.

Custom speeddata can be defined as a variable in a module. It is defined as a tuple of four numbers, corresponding to the linear and rotational speeds of the TCP and of linear axes, respectively.

Note that the speeds specified in these move statements are still subject to the Manual Mode limitation, i.e. the TCP will never move faster than 250 millimeters per second while in Manual Mode, even if a faster speed data is specified.

Zone data

The zone data argument specifies the behavior of the robot when a target is neared by a move statement. The predefined zonedata "fine" forces the TCP to stop at (within 0.2 millimeters of) the robot target before proceeding to the next move statement. Other predefined zonedata start with a 'z' followed by the zone tolerance in millimeters, e.g. "z40". These fly-by zonedata specify that the TCP must come within this zone; then it will begin veering off toward the next path segment, without stopping.

Like speed data, custom zone data can be specified as a variable at the module level. Its first element is a Boolean specifying if the zonedata represents a stop point or a fly-by point. The remaining elements are sizes of the path zone and zoning data for external axes and translation and rotation of the TCP.

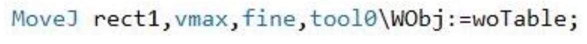
Tool

The Tool argument says which Tool the robot is using to approach the robot target. The target is considered reached when the tool's pose in world space is within the specified zone distance of the target's world pose.

Optional: Work object, others

There are certain optional arguments that can be given in a move statement, the most common of which is a work object. This argument specifies the parent work object of the specified robot target. Although this argument is optional, it is highly recommended as its use can simplify component relocation considerably, by simply changing the work object's pose rather than those of all child targets.

Other optional arguments can modify the behavior of the move statement, and some can substitute for the speed or zone data by specifying different parameters. All optional arguments are preceded with a backslash, followed by an assignment statement of the optional argument.



```
MoveJ rect1,vmax,fine,tool0\WObj:=woTable;
```

Figure 6-7: A move statement showing the movement type, robot target, speed data, zone data, tool, and work object used in defining the move.

Other Instructions

Program Flow Control

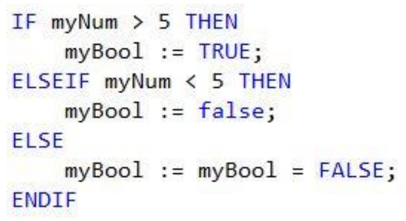
The structured programming paradigm intrinsic to most modern programming languages provides several code structures that can control the flow of the program. These can be controlled by Boolean conditions for execution and repetition, repetition using an incremental numeric value in a certain range, or handlers of certain triggered events.

If, elseif, else

The simplest flow structure in a program is an *If* statement. The *if* statement takes in a single Boolean argument, and if the argument case is true, it executes its code block. Once the code block is executed, the program pointer leaves the block and executes further down the script.

If statements can be extended with an *ElseIf* statement, which is evaluated if the preceding *If* case is false; if its own argument is true, its code block is executed. An *Else* block takes a false case from a proceeding *if* statement and executes its own code block.

With a chained *If…ElseIf…Else* statement, once a true case is achieved, the entire chain is exited after executing the appropriate block. If the programmer wishes to have each case evaluated regardless of previous cases, serial *If* statements are preferable. These *If* statements and chains are useful for evaluation of a wide variety of complex logic. They can also be useful for catching special cases which might cause bugs in regular operation.

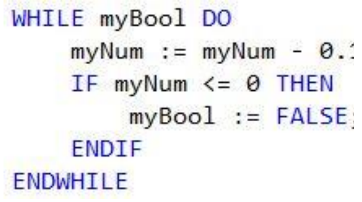


```
IF myNum > 5 THEN
    myBool := TRUE;
ELSEIF myNum < 5 THEN
    myBool := false;
ELSE
    myBool := myBool = FALSE;
ENDIF
```

Figure 6-8: An If-ElseIf-Else structure, used for handling logic

While

While loops function similarly to *If* statements, in that one takes a single Boolean argument and executes its code block if the argument case is true. Unlike *If* statements however, once a *While* loop's code block is completed, the argument case is reevaluated; if the case is true, the code block is executed again. This repetition can happen an infinite number of times and is only terminated by an argument that evaluates as false. *While* loops are useful when a procedure



```
WHILE myBool DO
    myNum := myNum - 0.
    IF myNum <= 0 THEN
        myBool := FALSE
    ENDIF
ENDWHILE
```

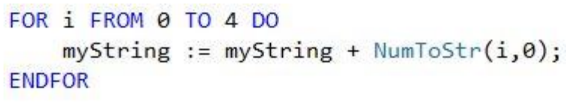
Figure 6-9: A While loop

must be repeated an undetermined number of times for a condition to be met.

<u>*For*</u>

For loops repeat a certain code block, like *while* loops. However, *for* loops use a predetermined number of steps, with the current step represented by an iteration variable, usually an integer. In the most common configuration of a *for* loop, this variable starts at one and ends after its specified maximum value is reached.

For loops are the best kind of loop to use when there is a known number of iterations to run. Due to their consecutive counting nature, as opposed to the *While* loop's check-and-run nature, they are significantly less prone to infinite loop errors.

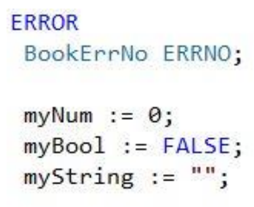


```
FOR i FROM 0 TO 4 DO
    myString := myString + NumToStr(i,0);
ENDFOR
```

Figure 6-10: A For loop

<u>*Special Case Handlers*</u>

There are structures that can be defined at the end of procedures which can handle special cases; the most common type is an *error* handler, which is jumped to if a runtime error is encountered in the program. This error handler has special properties including *ERRNO*, which allows case logic to determine what error occurred and to proceed accordingly.



```
ERROR
 BookErrNo ERRNO;

 myNum := 0;
 myBool := FALSE;
 myString := "";
```

Figure 6-11: An error handler at the end of a procedure

Calls to other routines

When one wishes to pass execution to a different procedure somewhere in the program, they invoke a special instruction called a procedure call, or *ProcCall*. It is invoked by simply stating the name of the desired procedure, followed by any arguments the procedure may take separated by commas, and ended with a semicolon. Figure 6-6 shows a *ProcCall* to a custom procedure.

Chapter Summary

This chapter gave a broad overview of the elements that make a RAPID™ program. A program deployed to a controller is made of one or more Tasks, with each Task having one or more Modules. Modules contain variables and routines. Each Task must have one and only one PROC main, which is where the Task begins execution. If multiple Tasks are present on a controller, all Tasks run in parallel.

Variables can be listed at a field level in a Module, or locally in a routine. Each field level variable is designated a scope, symbol type, and data type. Its scope determines what locations in code the variable is exposed to. The symbol type determines the rules the variable uses for value assignment and retention. There is a long list of data types available for use in RAPID™; however, this project focuses on a few that are specific to robot path planning, such as work objects, robot targets, tools, and poses.

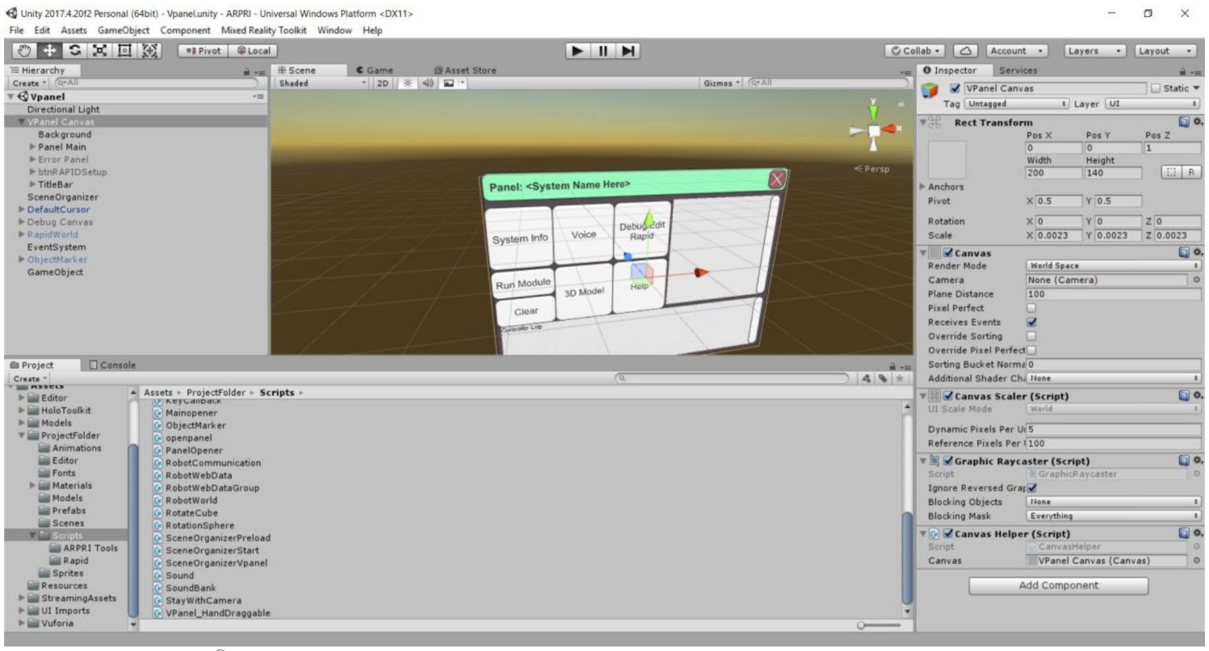
7 Introduction to Unity® and C#

Unity®

While there are multiple ways of developing AR software for HoloLens®, one sensible method is to use a ready-made engine that already supports the target hardware. This project uses Unity®, a publicly available game and software engine with a large support base and extensive documentation, [11]. Microsoft has an introduction to Unity® development in their own online development manual and provides a toolset for application building called Mixed-Reality Toolkit, [12].

An application built into Unity® is divided into parts called scenes. Each scene loads separately and independently of other scenes. Within a scene, there is a hierarchy of objects in 3D space called GameObjects. Each GameObject has a position, rotation, and scale, which is stored as its transform. Some GameObjects can be parent to other GameObjects, and a parent's transform is applied to all its child objects before their own transforms are calculated, [11].

GameObjects can have one or more attachments called components which alter the behavior of these GameObjects. These components inherit from a special Unity Engine® class,



Unity 2017.4.20f2 Personal (64bit) - Vpanel.unity - ARPRI - Universal Windows Platform <DX11>

```
VPanel Canvas                         Static
Tag  Untagged            Layer  UI
Rect Transform
                 Pos X      Pos Y      Pos Z
                 0          0          1
                 Width      Height
                 200        140
Anchors
Pivot            X 0.5      Y 0.5
Rotation         X 0        Y 0        Z 0
Scale            X 0.0023   Y 0.0023   Z 0.0023
Canvas
Render Mode              World Space
Camera                   None (Camera)
Plane Distance           100
Pixel Perfect            ☐
Receives Events          ✓
Override Sorting         ☐
Override Pixel Perfect   ☐
Sorting Bucket Norma     0
Additional Shader Cha    None
Canvas Scaler (Script)
UI Scale Mode            World
Dynamic Pixels Per Ui    5
Reference Pixels Per U   100
Graphic Raycaster (Script)
Script                   GraphicRaycaster
Ignore Reversed Gra      ✓
Blocking Objects         None
Blocking Mask            Everything
Canvas Helper (Script)
Script                   CanvasHelper
Canvas                   VPanel Canvas (Canvas)
```

Figure 7-1: Unity® Editor interface, showing the VPanel Scene

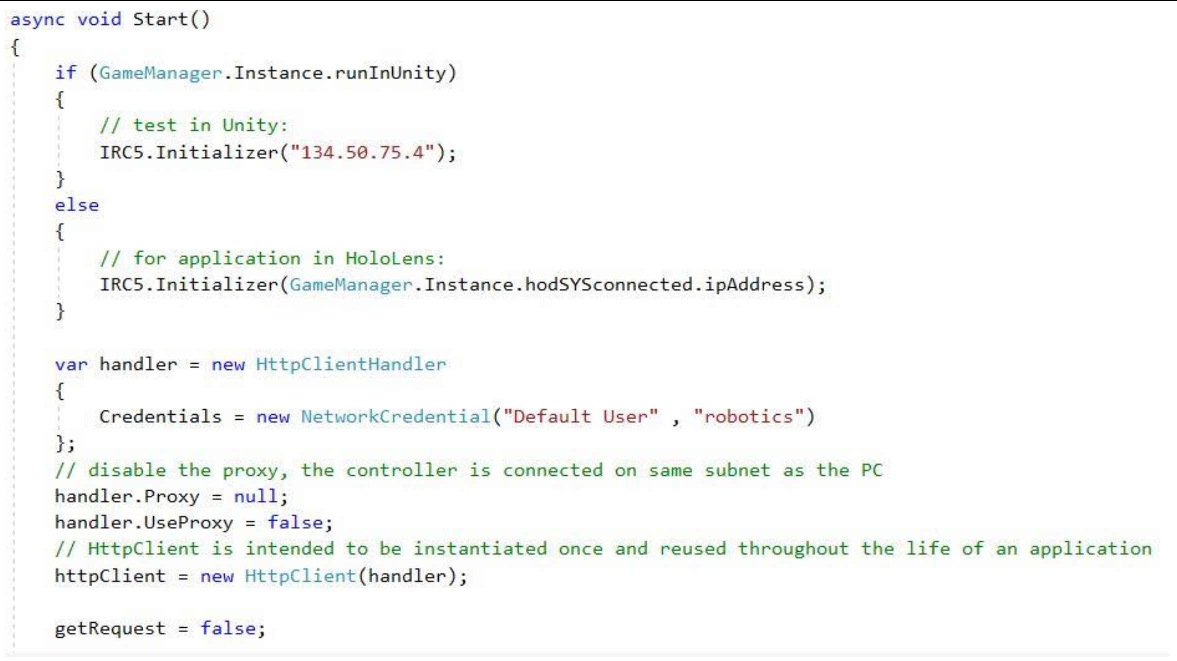
MonoBehaviour, and can be scripted in the engine for nearly any purpose. Due to the open-ended versatility of this scripting, most Unity Engine® custom scripting is done in these components, [11].

There are several events triggered by the engine that components can take advantage of. Two examples of these events are Start and Update. Start is an event that is triggered each time the scene is loaded, which is useful for various initialization processes required in the scene. Update is an event called once per frame; most repeatable and continuous tasks are performed in a component method called by the Update event.

C#

Unity® uses C# as its primary language for developer scripting. This provides a secure but straightforward language to get developers off the ground. The use of C# also means it is relatively effortless to include references to .NET assemblies in the project. It is a massive subject, and an implicit look at its features is far beyond the scope of this paper. However, some general characteristics of the language and its implementation in the Unity Engine® are discussed here.

C# is a language developed and maintained by Microsoft Corporation in its .NET initiative. It is meant as a high-level object-oriented programming language building upon some fluid characteristics of C++ while implementing features of managed languages, such as garbage collection, found in related languages such as Java.



```
async void Start()
{
    if (GameManager.Instance.runInUnity)
    {
        // test in Unity:
        IRC5.Initializer("134.50.75.4");
    }
    else
    {
        // for application in HoloLens:
        IRC5.Initializer(GameManager.Instance.hodSYSconnected.ipAddress);
    }

    var handler = new HttpClientHandler
    {
        Credentials = new NetworkCredential("Default User" , "robotics")
    };
    // disable the proxy, the controller is connected on same subnet as the PC
    handler.Proxy = null;
    handler.UseProxy = false;
    // HttpClient is intended to be instantiated once and reused throughout the life of an application
    httpClient = new HttpClient(handler);

    getRequest = false;
```

Figure 7-2: Sample C# Code, taken from the app's RobotCommunication script. Source: [15]

Characteristics

Classes and Objects

Its chief characteristic as an object-oriented programming language is the use of classes and objects. A class is a template for an object that may have any number of different things built into it. A class can be instantiated into any number of objects with the use of constructors, a special method that returns a new object of its type. Classes may inherit from other classes to signify the subclass also being the type of the base class, [13].

Member: Fields, Methods, Properties

A class contains members, programmed parts which can be interacted with by the program. The types of members that hold the data of the class are called fields. Methods are the routines of C# and can have any number of arguments and may or may not return a variable value. Properties are a clever way to create a method which is used as a variable in code, [13].

Modifiers

Classes and their members have numerous modifiers that can be added in their declarations to alter their behaviors. The most common modifiers alter the scope of the variable; private members can only be accessed from within the class, while protected members can be accessed by any subclass and public members can be accessed elsewhere in code. Other modifiers include static, which makes the member tied to the class itself rather than an object instance, or abstract, which indicates the implementation of the class or member is incomplete in that state and must not be directly used, [13].

Generics

Generics are types that are meant to be associated with another type that is not specified when the class or member is created. An example is the List<> class, which is implemented as List<T>, and T can be replaced by a true type when a list instance is created, [13].

Reflection

In native C#, classes can search themselves and others for members that have a name only decided at runtime. This is a complicated process that has some very useful applications; however, the UWP implementation which is used on the HoloLens® is missing certain assemblies related to reflection, so this concept has been avoided in this paper after much experimentation, [13].

Garbage Collection

A feature that makes C# relatively beginner-friendly is its garbage collection. Unmanaged languages such as C++ require the programmer to destroy all objects manually when they are no longer needed, and failure to do so causes memory leaks which are harmful to performance and security. The garbage collection feature of C# automatically destroys objects in the heap when they are no longer referenced from the stack, removing a tedious chore that a programmer can easily neglect, [13].

Use in Unity®

Unity® uses C# as its primary scripting language for development in the engine. This provides a good combination of versatility and ease of learning critical for ground-breaking projects. It allows simple access to the vast libraries written to be compatible with the .NET assemblies; it therefore can accomplish a wide variety of Windows-specific computational operation and data management. This proves useful for development in the HoloLens®, which uses the Universal Windows Platform, [12].

The Unity Engine® classes, such as GameObject and various MonoBehaviour derivatives, expose a wide variety of properties and methods for use in the engine's scripting. A typical MonoBehaviour script contains references to certain events called by the engine, such as Start, which is called once when a scene is finished loading and beginning to operate, and Update, which is called once per frame thereafter. These events can be planned to dictate behavior of any object in the application, [11].

.NET Assemblies to use

There are several .NET assemblies used in the creation of this application. Microsoft provides a large system-specific API for use with basic computing resources, while the Unity Engine® has libraries useful for application function. Other libraries need to be used for specific functions, such as Microsoft's Mixed Reality Toolkit for Unity®, which provides the tools necessary to allow the HoloLens® to interface properly with an application made in Unity®. Other API's include NewtonSoft.Json, which provides robust compilation of JSON objects to parse for information, and System.Net.Http, which is used for the communication between the application and robot which is central to the project, [13].

Chapter Summary

This chapter covers the game engine Unity® and the programming language C#, which are the basic tools used to build the application. Unity® organizes its structure as scenes, and scenes have a collection of GameObjects which may or may not be visually represented in the application. GameObjects have components attached to them which modify the behavior of the parent GameObject.

Unity® uses C# as its primary scripting language. C# provides several advantages in versatility and ease of use in the development of an application using Unity®. C# is an object-oriented programming language, which allows custom classes and instantiated objects which are unique in form and function, allowing the storage and use of data as the application demands. It contains many of the typical elements of similar programming languages, including class members such as fields, methods and properties. Most such symbols can be given modifiers which change their accessibility or function. Some classes can be specified as generic, to be made specific to a type when used in code. C# uses automated garbage collection, which destroys heap objects which are no longer referenced in code. These features all play significant roles in the development of the project.

8 Application Description and Function

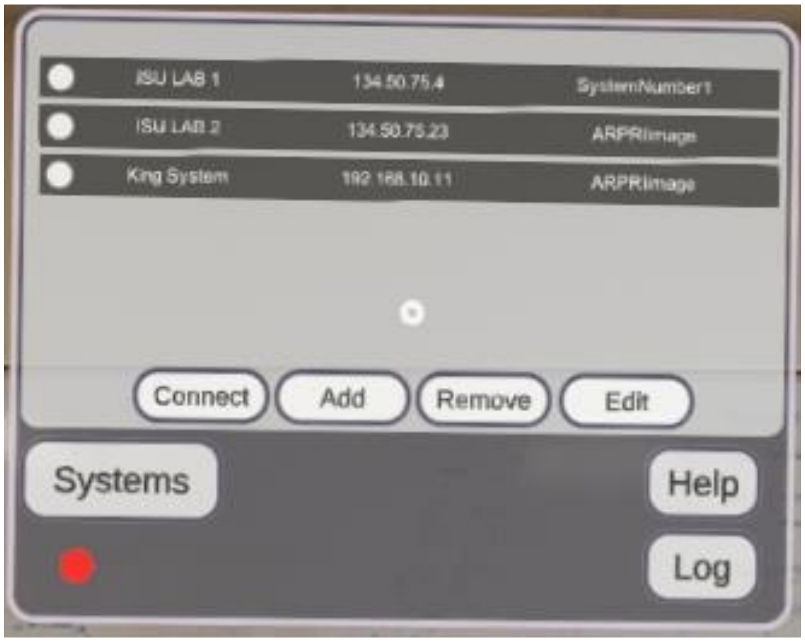
Application Overview

The proposed application is divided into two functional scenes, Start and VPanel. The Start scene provides a console where the user selects a system they wish to connect to. Once connected, the application switches to the VPanel scene, where they are shown data and numerous options for interacting with the connected robot system. The VPanel can be closed, which will return the user to the Start scene to optionally choose another system.

Start Scene

The first scene, called “Start,” shows a small console that follows the user when moving. It displays some instructions on connecting to new or previous systems. The HoloLens® must be connected to the same Local Area Network (LAN) as the controller for the application to function properly.

To connect to an unrecognized controller, the user must enter the controller’s LAN IP address. If a controller is found at that address, a message will be displayed asking to confirm



○	ISU LAB 1	134.50.75.4	SystemNumber1
○	ISU LAB 2	134.50.75.23	ARPRImage
○	King System	192.168.10.11	ARPRImage

Figure 8-1: VPanel, displaying system information received from the robot

connection. That controller is then recorded and saved. If the user wishes to a previously saved controller, they must simply select that controller from a list of previously saved ones.

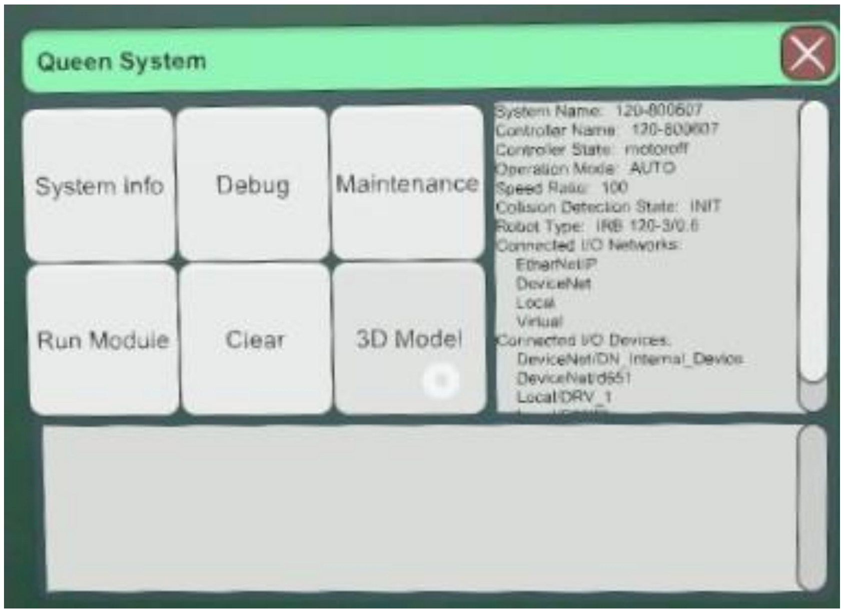
VPanel

Overview

Upon successful connection to a controller, the application switches to a scene with a virtual panel that displays information and interaction options for this robotic system. This “VPanel” scene has numerous features providing examples of possible uses of the software; the main features are described in the paragraphs below.

In addition to displaying raw data about the controller’s status, the ability to directly access the unchanged RAPID™ code provides a detailed picture of the controller’s script. This is broken apart and parsed to identify each element in the lines of code. When these are positively identified, for example, as a variable of type num, or a routine with no return type (called a procedure), a reflected object is created in the application to reflect this element. One use of identifying each of these elements, is reconstructing the path that the robot will take during the execution of its main procedure.

While planning the RAPID™ features in the application, the reflected objects were referred to as “mirror objects”; hence, the objects made in the application to reflect the actual



```
System Name:  120-800607
Controller Name:  120-800607
Controller State:  motoroff
Operation Mode:  AUTO
Speed Ratio:  100
Collision Detection State:  INIT
Robot Type:  IRB 120-3/0.6
Connected I/O Networks:
    EtherNet/IP
    DeviceNet
    Local
    Virtual
Connected I/O Devices:
    DeviceNet/DN_Internal_Device
    DeviceNet/d651
    Local/DRV_1
```

Figure 8-2: VPanel, displaying system information received from the robot

ones may be called mirror objects to distinguish them from the true RAPID™ objects on the controller.

System Info

ABB provides an API exposed in each of their controllers called Robot Web Services, [14]. This interface uses Hypertext Transfer Protocol (HTTP) to allow access to various controller information and data via web connections by external clients. The application uses this service for all interactions with the robot, getting controller states, RAPID™ module text, the current running task or module, maintenance, and calibration information and more upon request. Much of this information is directly displayed to the user.

3D Model

The VPanel Scene has the option to enable a virtual model of the robot world, which appears near the panel. This robot world model has the robot model itself, which was imported from RobotStudio® (and possibly simplified to ease the rendering load on the HoloLens®).

The robot lies at the center of the virtual world. The robot's stationary base can be clicked and dragged around to move the virtual robot world to the desired section of physical space. A sphere lies before the robot, which can be dragged anywhere, and the world will pivot such that the robot always faces that sphere.

Also present are small coordinate frames representing the Work Objects and Robot Targets present in the main procedure in the current program module on the controller and are

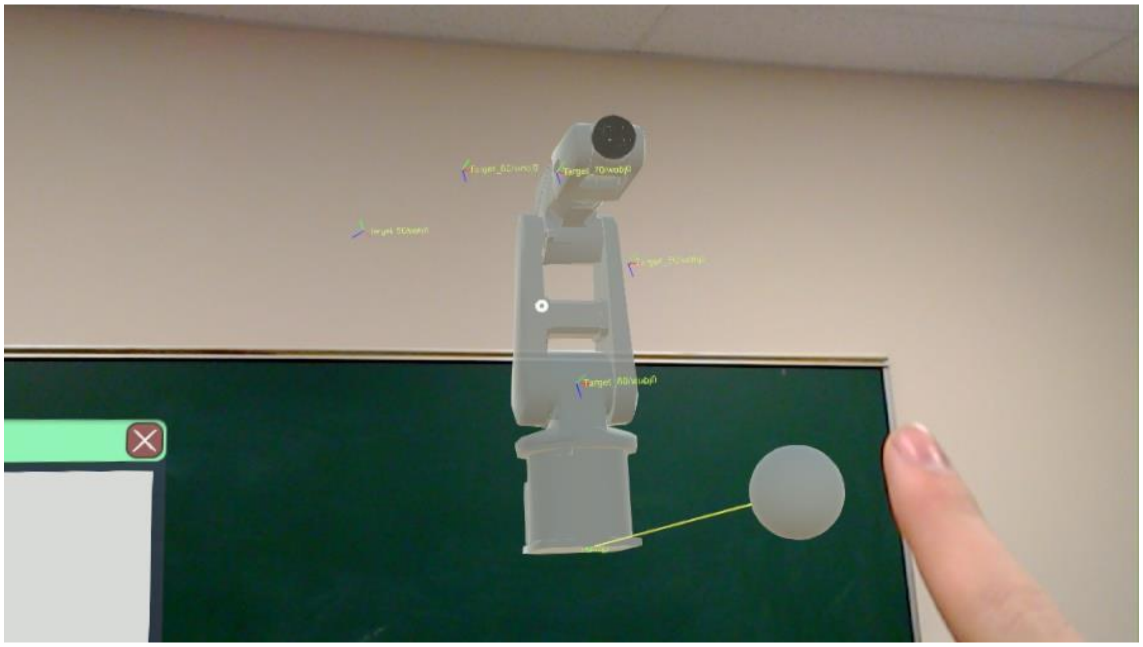


Figure 8-3: Robot 3D Model, with targets displayed around it

translated and rotated as such. This gives a visualization of these objects highly demanded by robotic developers.

Run

The application also can use the Robot Web Services to run the robot remotely. When the robot is set to automatic mode with motors on, a message can be sent for the robot to start or stop its main routine. This provides the ability to test out robotic operation even after the development of the system is complete.

Communication

HTTP Connection

The app's main purpose is to directly interact with the robot controller. The controller has built in support for external sources to communicate with it via HTTP web requests. .NET libraries include an HTTP assembly which provides the necessary tools to establish an HTTP client (with a class name *HttpClient* in C# from the namespace *System.Http.Net*) and send messages out from this client to addresses. The address used for this project is the LAN IP address of the robot, followed by the path of the specific information required from the controller, which is specified in Robot Web Services, and certain request parameters appended onto the end of the URL (parameters are shown behind the question mark in a URL), [14].

The default format of a response to HTTP request is an XML document transmitted as a string. The library *System.Xml* has a handler for reconstructing the XML document and extracting the desired information from the document. However, XML documents can often be cluttered and hard to read. If the parameter *json=1*, the returning document will be in the JSON format also transmitted as a string, which can be reconstructed and sorted with tools from numerous libraries; this project uses *Newtonsoft.Json*. The JSON layout is generally easier to read through to find the desired information, [13].

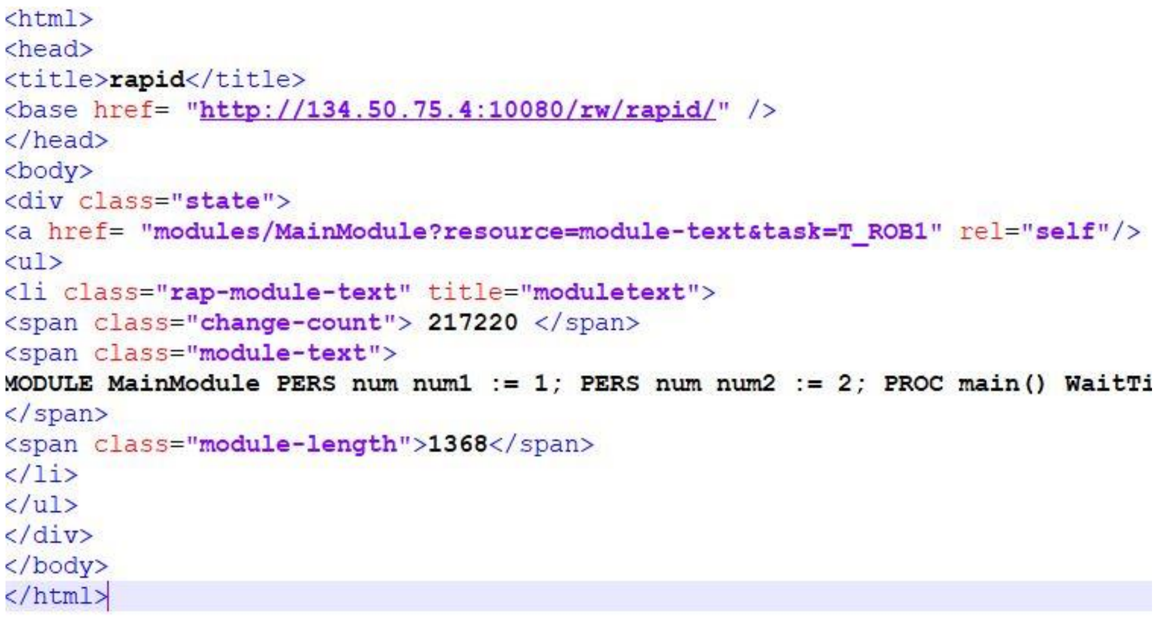
A GET request to the following address allows the acquisition of Module text in XML:

```
http://<SystemIPAddress>/rw/rapid/modules/<ModuleName>?task=<TaskName>&resource=module-text
```

GET

The simplest type of HTTP message is a GET request. These only require the specific address of the desired page. Numerous data are exposed to get requests, typically ones which only access data on the controller without otherwise interacting with it. In C# code from the

System.Net.Http assembly, the *HttpClient* has a method named *GetAsync()*, for which the only argument is the URL with the parameters appended as described above.



```
<html>
<head>
<title>rapid</title>
<base href= "http://134.50.75.4:10080/rw/rapid/" />
</head>
<body>
<div class="state">
<a href= "modules/MainModule?resource=module-text&task=T_ROB1" rel="self"/>
<ul>
<li class="rap-module-text" title="moduletext">
<span class="change-count"> 217220 </span>
<span class="module-text">
MODULE MainModule PERS num num1 := 1; PERS num num2 := 2; PROC main() WaitTi
</span>
<span class="module-length">1368</span>
</li>
</ul>
</div>
</body>
</html>
```

Figure 8-4: Sample response to the address that gets the Module Text. Source: [14]

POST

For most communication meant to interact with the robot in a direct manner, or a resource that requires a large amount of input information, a POST request is required by Robot Web Services. These requests are similar to get requests as they need a specific IP address, resource path, and certain parameters passed in as a URL, but unlike get requests there are also data parameters that are passed in key-value pairs in the body of a POST request. C# code from the same assembly also provides a *PostAsync()* method, which takes two arguments; the first is a URL in the same manner as a GET request, and an object of type *FormUrlEncodedContent* made from a *Dictionary<string,string>* with the key-value pairs of parameters.

Robot Web Services

ABB Controllers expose the information to HTTP requests via an Application Programming Interface (API) titled Robot Web Services (RWS). The API has online documentation detailing the specific resources that can be retrieved from the controller. The information useful to this project can be divided into two broad categories, [14].

System information

There are many details and quantities that are exposed in the system, such as motor and controller states, speed ratios, connected networks and devices, calibration data, elapsed time since maintenance, running time, service lists, file system details, I/O channels, log entries, user and mastership statuses, and much more. All information not directly related to the RAPID™ code deployed to the controller is treated as *System Information* by the project. This information is not generally used by the current version of the application but is simply displayed on the VPanel for developer or user reference, [14].

RAPID™ service

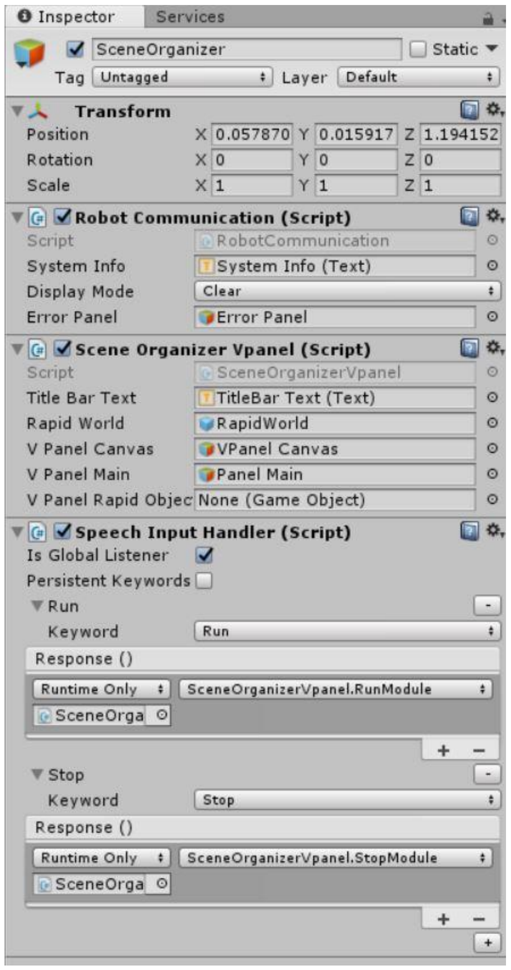
Information on the RAPID™ code is central to features of the application and is therefore treated separately from the other information exposed by the API. Chapter Introduction to RAPID6 states that each controller's RAPID™ code is comprised of one or more Tasks, and each Task contains Modules, which in turn contain all RAPID™ elements declared in the Module text. In Robot Web Services, a list of Tasks can be retrieved, a list of Modules can be obtained from each task, and each Module's RAPID™ text can be recovered by GET requests to each respective resource, [14].

Subscription service

It should be noted that Robot Web Services has a subscription service on many different resources. Subscribing to a resource establishes a WebSocket connection with the controller, which will send an event to the WebSocket client when the resource has its value changed. This would significantly reduce the computational expense of keeping track of frequently changing resources, as the update process could be tied to the WebSocket event rather than regular HTTP checkups. However, at time of writing the team is still researching the function of WebSocket connections, and the current understanding is insufficient to include in this paper, [14].

Code Structure

The VPanel scene is set up to make efficient use of both types of resources described in the previous section. The scene has a GameObject named SceneOrganizer, which has the components SceneOrganizerVPanel, RobotCommunication, and SpeechInputHandler.

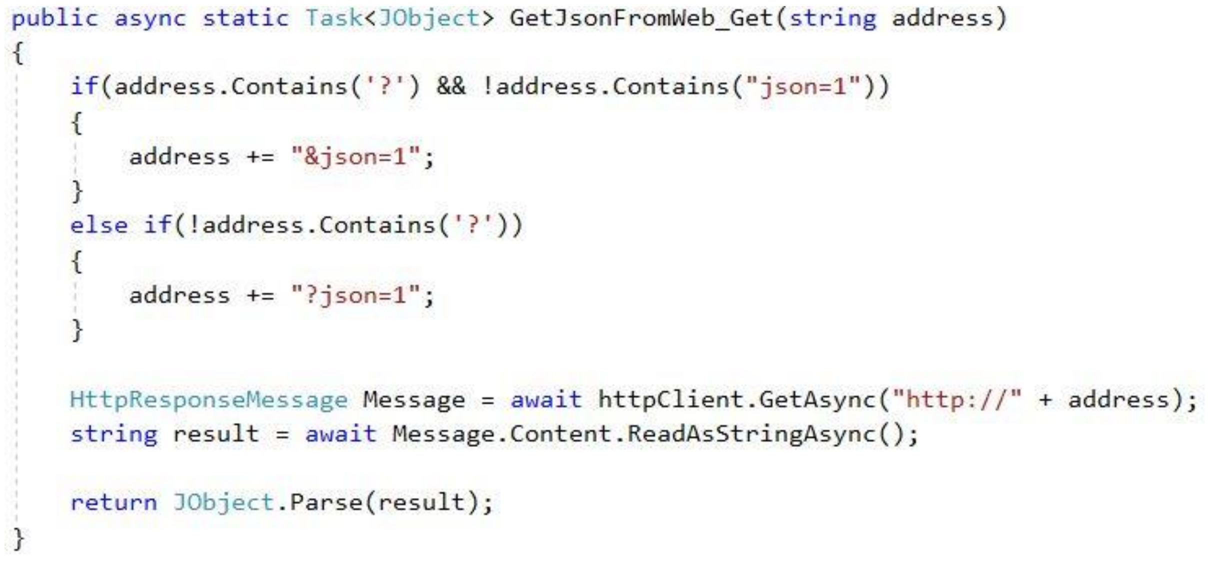


```
Inspector   Services
[✓] SceneOrganizer                         [ ] Static
Tag  Untagged            Layer  Default

Transform
Position   X 0.057870   Y 0.015917   Z 1.194152
Rotation   X 0          Y 0          Z 0
Scale      X 1          Y 1          Z 1

[✓] Robot Communication (Script)
Script                 RobotCommunication
System Info            System Info (Text)
Display Mode           Clear
Error Panel            Error Panel

[✓] Scene Organizer Vpanel (Script)
Script                 SceneOrganizerVpanel
Title Bar Text         TitleBar Text (Text)
Rapid World            RapidWorld
V Panel Canvas         VPanel Canvas
V Panel Main           Panel Main
V Panel Rapid Objec    None (Game Object)

[✓] Speech Input Handler (Script)
Is Global Listener     [✓]
Persistent Keywords    [ ]
Run
  Keyword              Run
  Response ()
  Runtime Only   SceneOrganizerVpanel.RunModule
  SceneOrga
Stop
  Keyword              Stop
  Response ()
  Runtime Only   SceneOrganizerVpanel.StopModule
  SceneOrga
```

Figure 8-5: The scripts attached to the SceneOrganizer object, which runs the primary operations of the VPanel scene.

RobotCommunication

The *RobotCommunication* component is a handler for all communications between the application and the robot controller. It contains a static *HttpClient* and an *IRC5Services* object to store information obtained from responses from the controller. Methods it has include handlers to return the reconstructed XML or JSON documents from a given address, or to even go straight through and get the desired string from a path in the returned document. It takes button events from the *SceneOrganizerVPanel* to trigger communications at desired times, and place



```
public async static Task<JObject> GetJsonFromWeb_Get(string address)
{
    if(address.Contains('?') && !address.Contains("json=1"))
    {
        address += "&json=1";
    }
    else if(!address.Contains('?'))
    {
        address += "?json=1";
    }

    HttpResponseMessage Message = await httpClient.GetAsync("http://" + address);
    string result = await Message.Content.ReadAsStringAsync();

    return JObject.Parse(result);
}
```

Figure 8-6: A method in RobotCommunication to create a JSON object from a web GET resource. Source: [15]

information received into *RobotWebData* objects and RAPID™ *Tasks*.

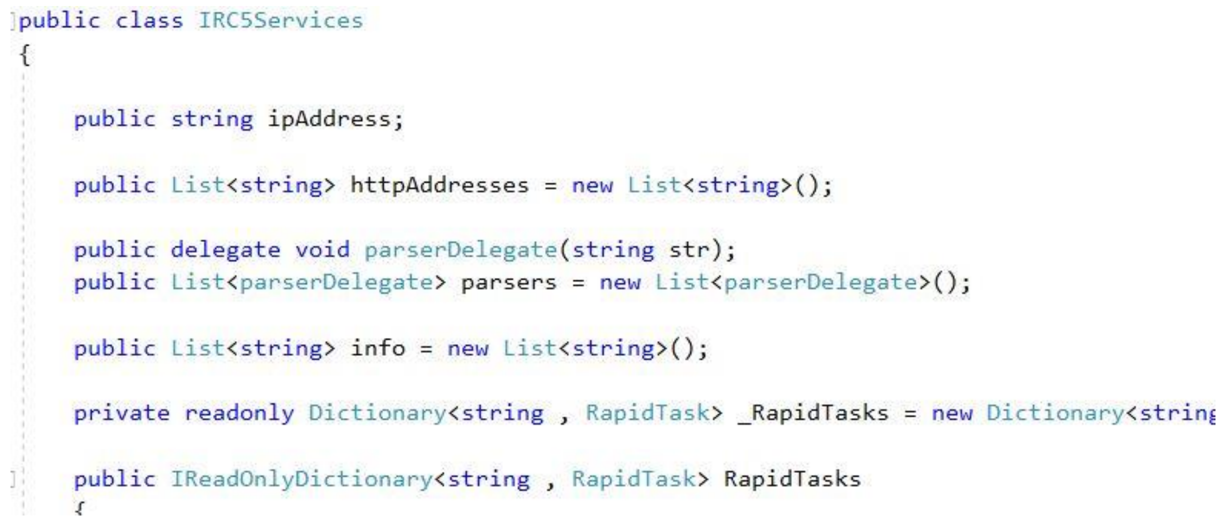
SceneOrganizerVPanel

SceneOrganizerVPanel is a script that handles interaction between the *RobotCommunication* component and the VPanel scene. It handles the various buttons on the VPanel and reads the RAPID™ objects in the *IRC5Services* object to create the virtual *RobotWorld*, with the robot's model and targets as children to the world *GameObject*.

While planning the RAPID™ features in the application the reflected objects were referred to as "mirror objects"; hence, the objects made in the application to reflect the actual ones may be called mirror objects to distinguish them from the true RAPID™ objects on the controller.

IRC5Services

IRC5Services is the class create to contain information received from a robot controller, hence the title. It contains multiple groups of *RobotWebData* objects meant to capture all significant data to use on the VPanel display. It also serves as the host of the RAPID™ mirror objects, having a list of *Tasks* on the controller that is referred to by *SceneOrganizerVPanel*, and



```
public class IRC5Services
{

    public string ipAddress;

    public List<string> httpAddresses = new List<string>();

    public delegate void parserDelegate(string str);
    public List<parserDelegate> parsers = new List<parserDelegate>();

    public List<string> info = new List<string>();

    private readonly Dictionary<string , RapidTask> _RapidTasks = new Dictionary<string

    public IReadOnlyDictionary<string , RapidTask> RapidTasks
    {
```

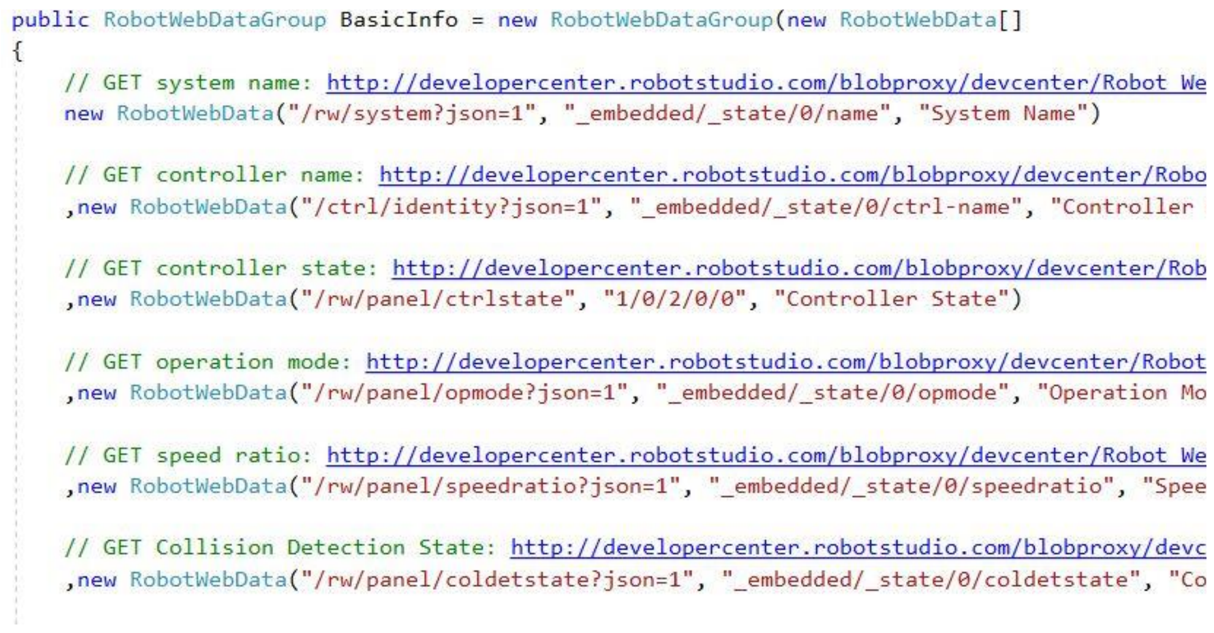
Figure 8-7: Beginning of IRC5Services class declaration; note the fields for addresses and information, as well as its collection of RAPID™ Task objects. Source: [15]

other classes. It uses an asynchronous method that uses the controller's listed IP address and searches the RWS directory for the *Task* list to populate the objects own collection of *Tasks* and begins the process of searching for *Modules* in each *Task*.

RobotWebData

The class *RobotWebData* is designed to be a self-contained reference to data available from RWS and store the most recent value returned from the controller via HTTP. The *RobotWebData* class contains the user-specified identifier, full HTTP address, and data path of the desired data in the returned XML or JSON file, depending on the passed HTTP address parameters. When the data specific to this object is needed, the object is added to the queue of data to be updated in *IRC5Services*.

RobotWebData are arranged in specialized collections of type *RobotWebDataGroup*, which are similarly identified by the developer for the nature of *RobotWebData* objects they contain. The *RobotWebDataGroup* objects can be passed directly to the System Info screen on the VPanel, where they will be listed sequentially, with the data they contain listed subordinately to the group for ease of reference.



```
public RobotWebDataGroup BasicInfo = new RobotWebDataGroup(new RobotWebData[]
{
    // GET system name: http://developercenter.robotstudio.com/blobproxy/devcenter/Robot We
    new RobotWebData("/rw/system?json=1", "_embedded/_state/0/name", "System Name")

    // GET controller name: http://developercenter.robotstudio.com/blobproxy/devcenter/Robo
    ,new RobotWebData("/ctrl/identity?json=1", "_embedded/_state/0/ctrl-name", "Controller

    // GET controller state: http://developercenter.robotstudio.com/blobproxy/devcenter/Rob
    ,new RobotWebData("/rw/panel/ctrlstate", "1/0/2/0/0", "Controller State")

    // GET operation mode: http://developercenter.robotstudio.com/blobproxy/devcenter/Robot
    ,new RobotWebData("/rw/panel/opmode?json=1", "_embedded/_state/0/opmode", "Operation Mo

    // GET speed ratio: http://developercenter.robotstudio.com/blobproxy/devcenter/Robot We
    ,new RobotWebData("/rw/panel/speedratio?json=1", "_embedded/_state/0/speedratio", "Spee

    // GET Collision Detection State: http://developercenter.robotstudio.com/blobproxy/devc
    ,new RobotWebData("/rw/panel/coldetstate?json=1", "_embedded/_state/0/coldetstate", "Co
```

Figure 8-8: A collection of RobotWebData, complete with addresses and data labels, to be aggregated and recorded by the IRC5Services object. Source: [15]

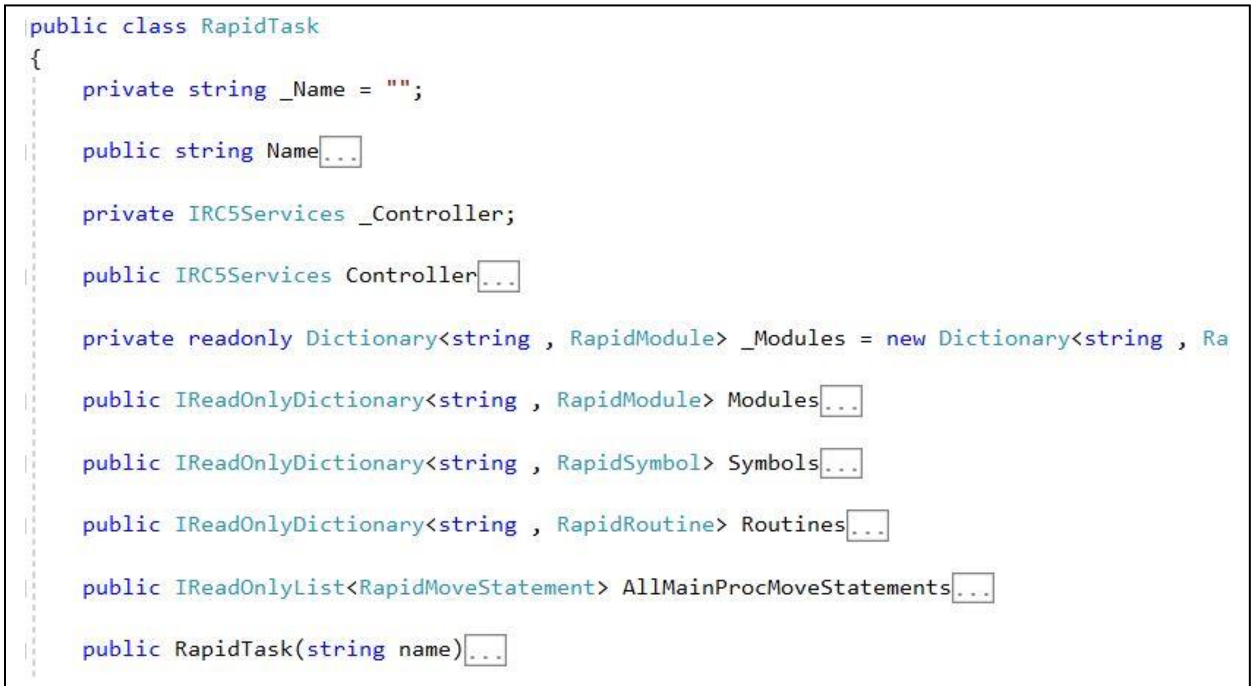
Rapid class set

The classes collectively named Rapid have been developed specially for this application to serve as the set of all classes specially designed to use the RAPID™ code from the controller directly for various purposes. They contain all types of directly mirrored symbols, including *Tasks*, *Modules*, *Routines* and *Variables* of all types. They also contain some utility classes and interfaces such as the Symbol class, which is the base class of all variables and contains important methods and data-specific fields.

Task

The Task class is a definition for the mirror objects for *Tasks* found on the controller in RAPID™. It contains fields and properties that give its name, a reference to its parent controller, and a dictionary of its child *Modules*. It contains an asynchronous method which uses its name and searches via RWS to find all *Modules* in the Task and add them into its *Module* dictionary.

Other properties it has includes dictionaries of all routines and data in its child modules to provide task-wide data search functionality. It also performs a rudimentary search for movement instructions beginning in PROC main.



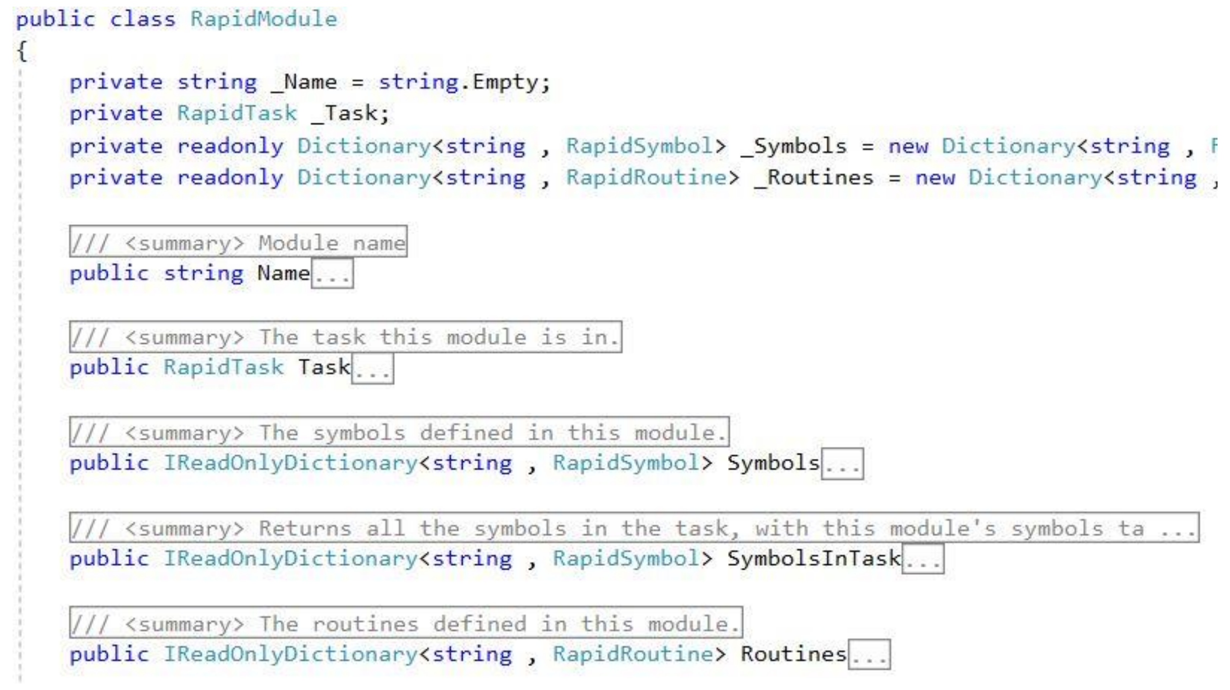
```
public class RapidTask
{
    private string _Name = "";

    public string Name...

    private IRC5Services _Controller;

    public IRC5Services Controller...

    private readonly Dictionary<string , RapidModule> _Modules = new Dictionary<string , Ra

    public IReadOnlyDictionary<string , RapidModule> Modules...

    public IReadOnlyDictionary<string , RapidSymbol> Symbols...

    public IReadOnlyDictionary<string , RapidRoutine> Routines...

    public IReadOnlyList<RapidMoveStatement> AllMainProcMoveStatements...

    public RapidTask(string name)...
```

Figure 8-9: Declaration of the Task mirror class; holding a collection of Modules and references to the variables and routines present in those modules. Source: [15]

Module

The mirror class *Module* is made similarly to that of *Task* and is the structure of the Rapid classes that holds the mirror objects of data and routines. It has fields for its name and a reference to its parent *Task*, as well as dictionaries of references to its data and routines. It contains an asynchronous routine that gets the *Module* text from the controller and parses it into each variable, routine and instruction present in the controller. The code used for parsing the RAPID™ code from each module is crude and incomplete at time of writing, and significant work into better code interpretation would benefit functionality considerably.

It also has wrapper properties and routines that call the dictionaries of its parent *Task*, allowing a Task-wide search for variables and routines, which is useful for finding references in routines that may call variables that are out of the same *Module*.

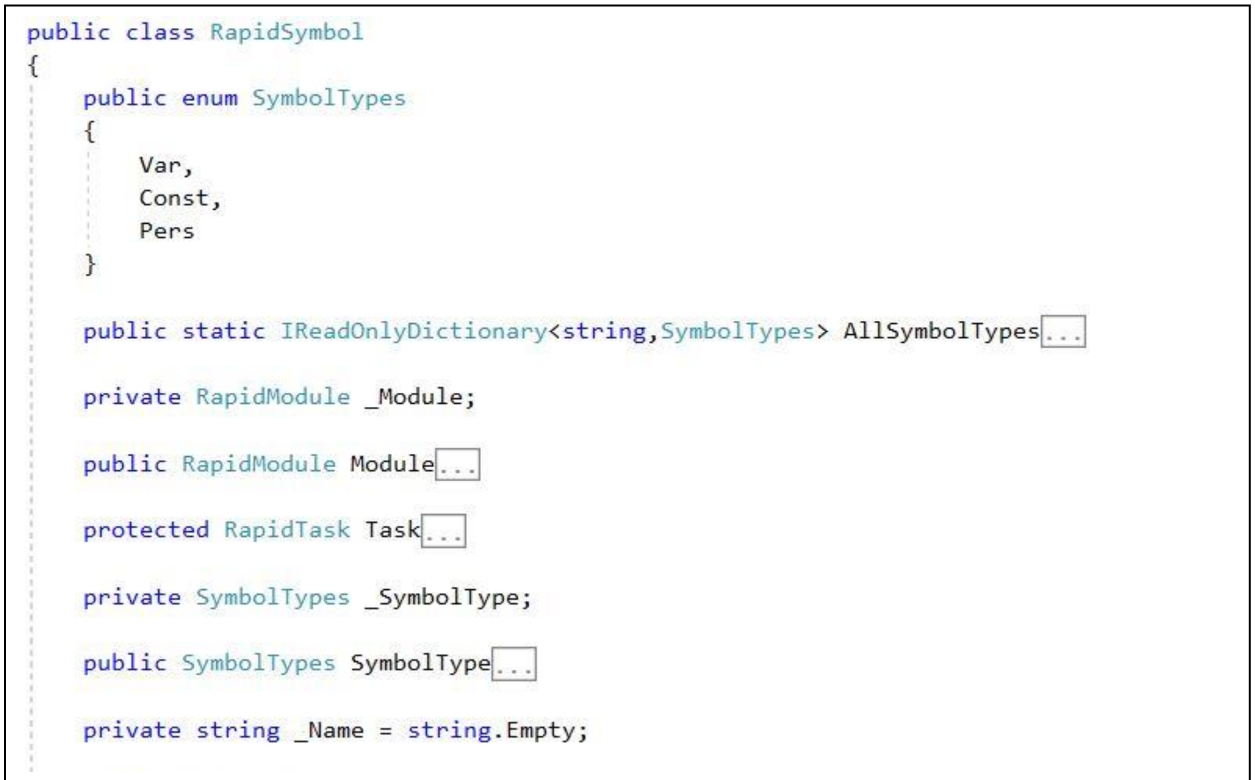


```
public class RapidModule
{
    private string _Name = string.Empty;
    private RapidTask _Task;
    private readonly Dictionary<string , RapidSymbol> _Symbols = new Dictionary<string , F
    private readonly Dictionary<string , RapidRoutine> _Routines = new Dictionary<string ,

    /// <summary> Module name
    public string Name...

    /// <summary> The task this module is in.
    public RapidTask Task...

    /// <summary> The symbols defined in this module.
    public IReadOnlyDictionary<string , RapidSymbol> Symbols...

    /// <summary> Returns all the symbols in the task, with this module's symbols ta ...
    public IReadOnlyDictionary<string , RapidSymbol> SymbolsInTask...

    /// <summary> The routines defined in this module.
    public IReadOnlyDictionary<string , RapidRoutine> Routines...
```

Figure 8-10: Declaration of the Module mirror class; which reads its text and converts them into mirror symbols and routines. Source: [15]

Symbol

The Symbol class is the base class for all variables held in a *Module* at field level. Additionally, it contains a symbol type field which specifies whether the RAPID™ object being mirrored is a *CONST*, *VAR*, or *PERS*. Since the specific data classes inherit from Symbol, the type of the object found by *GetType()* determines the datatype of the object. The class contains static collections of all symbol types and datatypes defined in the current Rapid classes. Additionally, there is a static method which takes in a RAPID™ line from the Module class and returns a symbol, which is deposited into the *Module's* data dictionary.



```
public class RapidSymbol
{
    public enum SymbolTypes
    {
        Var,
        Const,
        Pers
    }

    public static IReadOnlyDictionary<string,SymbolTypes> AllSymbolTypes...

    private RapidModule _Module;

    public RapidModule Module...

    protected RapidTask Task...

    private SymbolTypes _SymbolType;

    public SymbolTypes SymbolType...

    private string _Name = string.Empty;
```

Figure 8-11: Declaration of the Symbol mirror class, holding details on the symbol represented and designed to be inherited by more sophisticated data classes. Source: [15]

Symbol-inheriting classes

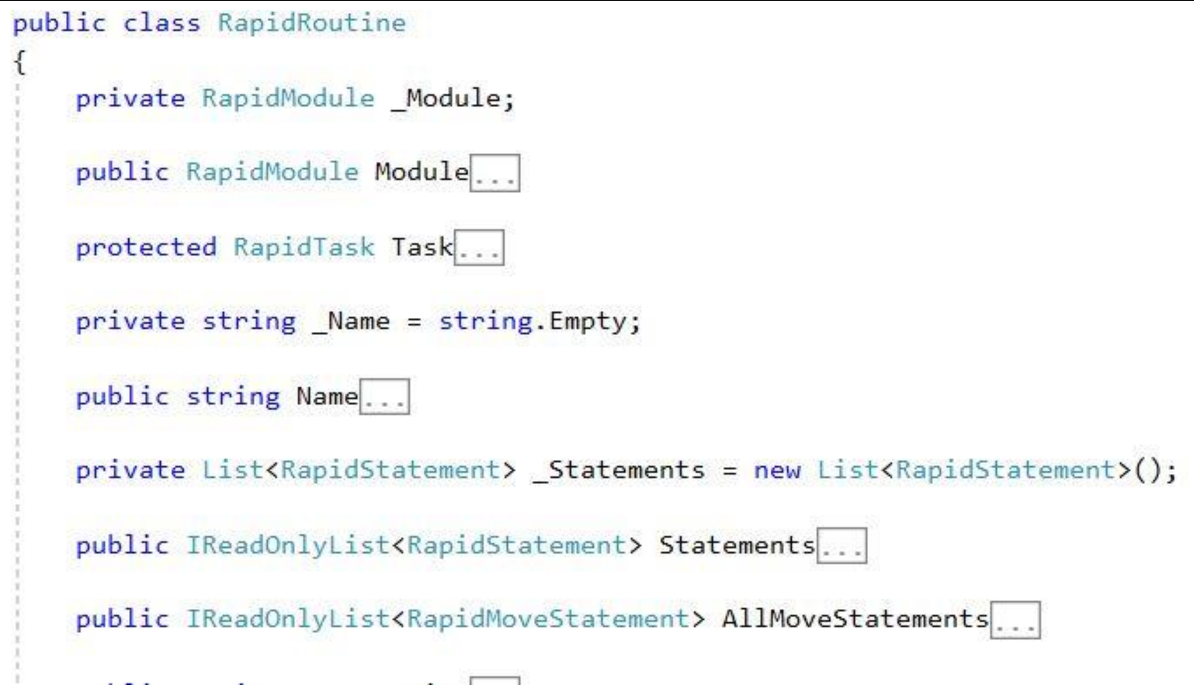
There are several types that inherit from the *Symbol* class, representing datatypes that have been implemented into the application so far. Each type has different properties, such as a different type of value, in the case of compound types, different component structures.

Just like the categories of datatypes in the RAPID™ language, the mirror variable datatypes can be classified as primitive, compound, or special types. Primitive datatypes are just like the ones in RAPID™; namely, *Bool*, *Num* and *String*. Compound data are defined similarly, such as *Pos*, *Orient*, *ToolData*, *WObjData*, and *RobTarget*, while special types include the *Clock*. Each of these are created while breaking down declaration statements, with their constructors being called by reflection from the *Symbol* class.

Routines

The *Routine* class is a base class for all mirror objects representing procedures, functions, and traps in RAPID™. Each *Routine* holds fields that identify its routine handle (or name), references to its parent *Task* and *Module*, and a list of instructions that the routine would execute on the controller.

One of the primary properties in the routine class is its list of move statement, which combs through its instructions, finding move statement both in the current routine, and in any routines referenced by *ProcCalls*. This allows the program to generate absolute target poses for viewing with the virtual targets.

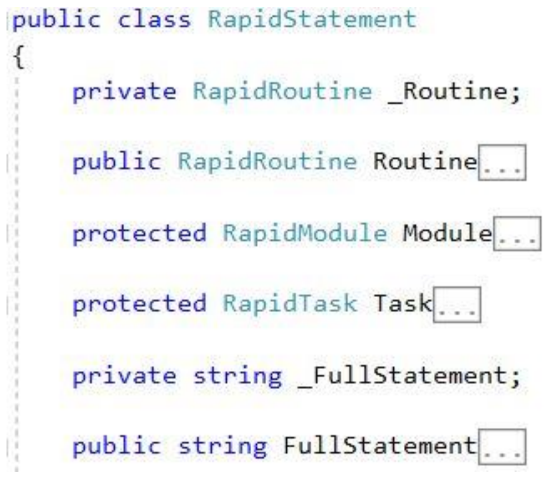


```
public class RapidRoutine
{
    private RapidModule _Module;

    public RapidModule Module...

    protected RapidTask Task...

    private string _Name = string.Empty;

    public string Name...

    private List<RapidStatement> _Statements = new List<RapidStatement>();

    public IReadOnlyList<RapidStatement> Statements...

    public IReadOnlyList<RapidMoveStatement> AllMoveStatements...
```

Figure 8-12: Declaration of the Routine mirror class, holding information about the instructions executed within and, similar to the Symbol class, designed to be inherited by specific routine types. Source: [15]

Statements

The Instruction object holds a statement extracted from the routine text. Its job is to sort out the kind of instruction, i.e. *ProcCall*, move statement, flow control, assignments, or others. The program only implements the use of *ProcCalls* and move statements, although plans for a proper interpreter would imply more complete emulation of all statements and variable assignments.



```
public class RapidStatement
{
    private RapidRoutine _Routine;

    public RapidRoutine Routine...

    protected RapidModule Module...

    protected RapidTask Task...

    private string _FullStatement;

    public string FullStatement...
```

Figure 8-13: Declaration of the Statement mirror class. Retains statement text but does have the ability to tell its own statement type. Source: [15]

Chapter Summary

This chapter dove into the structure and function of the project in its current state. The application is divided into two different scenes. The start scene displays a panel for the user to connect to a new or previous system. The VPanel scene coordinates with a connected system to display information and interaction options to a connected system. It does this via an eponymous panel that can be manually placed by the user. A *RobotWorld* object shows a draggable 3D robot model and all movements in the systems path.

The application uses HTTP web communication to transfer information between the robot controller and the application. The information used consists of various system information useful for currently running systems, as well as the loaded RAPID™ code in the controller. This

RAPID™ code is parsed by the application to allow specific analysis and visualization in the application.

The RAPID™ code is broken down and classified by the objects represented in RAPID™ code; each object found in the RAPID™ code is represented by a mirror object in the application. The controller's Tasks are represented by Task objects within the *IRC5Services* object handler, and the child Modules and dependent variables and routines are stored in corresponding collections within. The code sorts through the RAPID™ code beginning with PROC main to find all the move statements in the Task's path, and these move statements are then displayed as coordinate frame targets at the correct positions relative to the virtual robot model.

The application remains buggy and limited in features which will be detailed in the next chapter. However, the visibility of targets in the robot world and accessibility to many of the robot's functions meets desires put forth by robotics engineers and provides an example of the potential of this combination of technology.

9 Issues and Future Work

Issues

Performance

Certain issues have troubled the development process, both in performance and user experience. As the interface with the robot controller is through HTTP web requests, much of the performance relies on the time between request and response, which has proven to be slow enough to make continuous updates to the information unfeasible.

Other performance issues have arisen due to the limited graphics and central processing power in the HoloLens®, especially using 3D models originally meant for use on powerful PCs. Unity®'s support for parallel processing is also quite limited, especially when the development target is a device such as HoloLens®. Significant work needs to be done for model and rendering simplification to optimize the application.

Programming

A significant source of issues in the development process was inconsistencies between the programming libraries available in the development environment and those deployed to UWP. The environment specifically allowed use of the *System. Reflection* library consistently with the API documentation available on Microsoft's website; however, Unity® does not successfully build the application due to certain elements in assemblies, especially the Reflection namespace, being different or entirely absent in UWP's libraries.

The source of this issue is, however, unclear. The project was hindered by a lengthy development process using an older version of Unity® designated for long-term support. This resulted in outdated versions of C# and its assemblies, and early versions of the Mixed-Reality Toolkit being used. This may have resulted in incompatible software preventing clean application-building and running. This issue is a candidate for fixing in future versions, although as the likely fix would require changing game engine and language versions, it would only be prudent as part of a major migration update.

User Interface

User-interface dilemmas also frustrate efforts to create an effortless experience in the application. Currently, text input requires a virtual keyboard to be custom-built within the application. This keyboard can only be interacted with by tapping with the gaze. This is a very inefficient method of interaction and must be fixed if the application is to be comfortably used.

The process for connecting to robot systems could be better streamlined as well. The current awkward method of finding and entering an IP address is a tedious method that requires a technical expert who knows how to find this information. If the system's IP address is not static, it may change and require the initial connection process to be repeated. A feature to allow the application to scan LAN IP addresses for robotic systems would alleviate this problem considerably.

Future Work

The scope of this project can slowly widen as progress is made, with the vast possibilities of features that may be desired for an application such as this. However, certain next steps from the current abilities are evident.

Display tool on robot model

A trivial addition to the current functionality would be to display the tool center point at the robot's end-effector. This object marker, instead of being parented directly to the robot world, should be parented to axis-6 of the robot. This would cause the tool to move with the virtual robot's joints should they be changed during program execution.

RAPID™ Code Editor Within the Application

The ability to access and change the RAPID™ module text through Robot Web Services, direct editing of the code is theoretically possible within the application. An interface for code editing this code and updating it in the application would provide a crucial feature to the functionality of this software.

Better Input

An in-application code editor would be nearly useless without some way to better interact with the code itself than is currently implemented. Two different methods are proposed to tackle the input issue for coding. One is being able to use an external keyboard for input into the HoloLens®, either attached to an external computer, or a wireless keyboard that can communicate directly to the HoloLens®. Keyboard input would allow an interface comparable to traditional IDEs, and to the RobotStudio® coding interface.

The other method of improved interaction with the code is to develop an interface like that on the FlexPendant™. Each variable, routine and instruction has menus for editing fields and arguments, showing varieties of selections, and automatically changing the RAPID™ code based on those selections. This allows the application to minimize necessary use of a virtual keyboard and allow the application to run as smoothly as possible with that restriction.

More Proper and Comprehensive RAPID™ Interpreter

The application interprets the RAPID™ code to find and display the move instructions reached after starting the main procedure. While the current application can display predefined robot targets from relative to their work objects, it does not currently support any dynamically changed variables, including work objects and robot targets, and thus will incorrectly display any such movements.

Writing a more complete interpreter to truly simulate a task's progression is an important objective for the future of the application, as it will allow more correct function and predictions of robot movements. Properly writing a RAPID™ interpreter, however, is a very difficult task which requires extensive knowledge on the function of language interpreters in general.

This task will involve creating "tokens" which represent every element found in RAPID™ definitions, values, routines, and statements. Sequential execution of the code in simulation is typically done with an "Abstract Syntax Tree", which sorts through the tokens and determines the order that operations are performed in throughout the code.

Simulate Robot Movement

A development of a more complete interpreter would be the simulation of robot actuation by the virtual model. As the interpreter would be able to walk through execution of the RAPID™ code, it should be able to recreate the movements of the robot and show an approximation in the application.

This would require the use of inverse kinematics. There would be two ways that this could be achieved; the web services made available via HTTP requests includes an address which makes the controller perform the inverse kinematic calculations and sends them back to the application. The other method would be to develop the parameters by forward kinematics and make the calculations manually in the manner described in Chapter 4. For linear movements, a reasonable approximation of the path could be made by a cubic interpolation of two intermediate points and the beginning and end points, requiring only four inverse kinematic calculations. Joint movements would only require the beginning and end points and could be linearly interpolated accordingly.

Dynamically edit robot targets/work objects

The current visible coordinate frames of the work objects and robot targets are helpful to developers getting a feel for the system, but the ability to move, rotate, add, delete, or otherwise interact with targets in a path is a feature that certainly belongs in software bringing the potential of Augmented Reality to practical application.

Chapter Summary

This final chapter discusses problems encountered within early development of the application, and possible avenues to fix many of these issues and improve the features of the application in future project operations. The main issues discussed include performance problems stemming from inefficiencies of the HTTP web request system and limited graphics processing capabilities in the first version of HoloLens®. Additionally, programming issues surfaced from unknown incompatibilities between the .NET libraries available in the Unity® environment versus those deployed to the Universal Windows Platform. The third major issue is the difficult user interface, which is hampered by a poor text input method in the HoloLens®. These features must be addressed in a satisfactory way before such a project becomes viable for industrial use.

There is a vast supply of potential features this application could incorporate to improve its utility. Several which were at various stages of conceptualization or development at time of writing include displaying objects such as a robot's tool center point on the virtual model. RAPID™ code features could be added which include the ability to edit RAPID™ code directly in the application, in a manner like the FlexPendant™'s Program Editor. With the app's RAPID™ interaction comes the prospect of a true RAPID™ interpreter, which will allow the application to more completely mirror the behavior of the physical robot. With the app's interpreter comes the visual simulation of the robot's movement, obtainable by kinematic calculations detailed in Chapter 4. One of the best goals when it comes to useful interaction by Augmented Reality, however, is to be able to physically manipulate, or drag-and-drop, various RAPID™ objects around the app's robot world. This ability would allow much more intuitive editing of the RAPID™ environment when compared to editing code directly, regardless of coding environment.

Final Thoughts

The process of creating an application to work with industrial robots has provided valuable insight into multiple worlds of development. Development of an application that bridges these technologies requires enough extensive knowledge on the part of each developer to know how information from the robot's controller and code can be received and used by unrelated code on the HoloLens® platform. Training from a robotics company partnered with this project provided priceless information, described in Chapters 4, 5 and 6 which would otherwise be too mercurial to navigate for the project otherwise. The skills learned about Unity® and the C# language proved equally critical in developing a product that showed any function at all. These skills must be retaught to anyone wishing to pick the project up beyond its current state.

The issues and goals laid out in Chapter 9 provide a good roadmap for future development. This thesis will ideally help the next developer in multiple areas; the spirit of the project is to pioneer a unique blend of technologies, and any project wishing to prove useful for their target audience must have time and care dedicated by researchers and students beyond any initial survey, as this paper describes. The combination of AR and industrial robotics is one with enormous potential, and its realization will impart knowledge on its developers which will give them extensive credentials for a career in either field.

References

[1] B. Sneiderman, M. Mahto and M. J. Cotteleer, "Industry 4.0 and manufacturing ecosystems".

[2] ABB Group, "RobotStudio SDK: Getting Started," 2016. [Online]. Available: http://developercenter.robotstudio.com/robotstudio/api_reference?Url=html%2F4ffca351-267a-4b49-a512-81638c3b21bf.htm.

[3] ABB Group, "Operating Manual - Robot Studio," ABB, Västerås, 2008-10.

[4] "Microsoft HoloLens," Microsoft, 14 October 2019. [Online]. Available: https://docs.microsoft.com/en-us/hololens/. [Accessed April 2020].

[5] ABB Group, "Operating Manual - IRC5 with FlexPendant," ABB, Västerås, 2004-08.

[6] K. M. Lynch and F. C. Park, Modern Robotics: Mechanics, Planning and Control, Cambridge University Press, 2017.

[7] J. M. McCarthy, Introduction to Theoretical Kinematics, MIT Press, 1990.

[8] D. Eberly, "Rotation Representations and Performance Issues," Geometric Tools, Redmond WA, 2002.

[9] ABB Group, "Application manual - RAPID development guidelines for handling applications," ABB, Västerås, 2013.

[10] ABB Group, "Technical reference manual - RAPID Instructions, Functions and Data types," ABB, Västerås, 2004-10.

[11] Unity Technologies, "Unity User Manual (2017.4)," [Online]. Available: https://docs.unity3d.com/2017.4/Documentation/Manual/.

[12] Microsoft, "Unity development overview: Microsoft Mixed Reality," 25 October 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/mixed-reality/unity-development-overview.

[13] "Get started with C#," Microsoft, 5 April 2019. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/. [Accessed April 2020].

[14] ABB Group, "Robot Web Services," [Online]. Available: http://developercenter.robotstudio.com/blobproxy/devcenter/Robot_Web_Services/html/index.html.

[15] O. Heidari, S. Chowdhury, T. Hedgepeth and K. Stone, *ARPRI App*, Pocatello, Idaho: Idaho State University, 2019-20.

Index