Software Control Document

Project Title: Robotic Platform for 1kg Loads (RP1)
Project Team: P09204
Project Revision: 2
Document Revision: 1.1

Change Log

<table>
<thead>
<tr>
<th>Revision Number</th>
<th>Date of Change</th>
<th>Description of Change</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>23 Oct 2008</td>
<td>Added layering diagram, began populating empty sections.</td>
<td>Jason Jack</td>
</tr>
<tr>
<td>-</td>
<td>24 Oct 2008</td>
<td>Updated system block design layout and description.</td>
<td>Jason Jack</td>
</tr>
<tr>
<td>-</td>
<td>26 Oct 2008</td>
<td>Added complete API catalogue, up to date.</td>
<td>Jason Jack</td>
</tr>
<tr>
<td>-</td>
<td>08 Jan 2009</td>
<td>Added various sections of prose including compilation procedure, repository structure, and inserted TBD markers into incomplete sections</td>
<td>Jason Jack</td>
</tr>
<tr>
<td>-</td>
<td>09 Jan 2009</td>
<td>Changed Date-of-Change to DD “MMM” YYYY format</td>
<td>Jason Jack</td>
</tr>
<tr>
<td>-</td>
<td>12 Jan 2009</td>
<td>Updated Operational Software design layout</td>
<td>Jason Jack</td>
</tr>
<tr>
<td>-</td>
<td>10 Apr 2009</td>
<td>Updated API for operational software, updated the system level description and block diagram</td>
<td>Jason Jack</td>
</tr>
<tr>
<td>-</td>
<td>11 May 2009</td>
<td>Added detailed information about the PID controller</td>
<td>John Corleto</td>
</tr>
<tr>
<td>1</td>
<td>13 May 2009</td>
<td>Added GUI discussion, pictorial guide, and usage information; completed release 1.0-0 of document</td>
<td>Jason Jack</td>
</tr>
<tr>
<td>1.1</td>
<td>15 May 2009</td>
<td>Added more info about the PID controller, fixed some typos.</td>
<td>John Corleto</td>
</tr>
</tbody>
</table>
# Table of Contents

1 Overview ................................................................................................................................. 5
   1.1 Software Control Document ............................................................................................... 5
   1.2 General Information ............................................................................................................ 5
   1.3 System High Level Design ................................................................................................. 6
2 Software Repository .................................................................................................................. 7
   2.1 Software Versions ................................................................................................................. 7
      2.1.1 Version 1.0-0 ................................................................................................................ 7
   2.2 Operational Software ........................................................................................................... 8
      2.2.1 Software High Level Design ......................................................................................... 8
      2.2.2 Repository Directory Structure ................................................................................. 10
      2.2.3 Installing the Compiler Toolset .................................................................................. 11
      2.2.4 Compilation Procedure ............................................................................................... 11
      2.2.5 Programming the BDMicro MAVRICIB ATmega128 ................................................ 11
      2.2.6 Connecting to the BDMicro MAVRICIB ................................................................. 12
      2.2.7 Application Programmers Interface ........................................................................... 12
         2.2.7.1 Analog to Digital Converter Control Module ..................................................... 12
         2.2.7.2 EEPROM Tx/Rx Control Module ....................................................................... 14
         2.2.7.3 Timer Control Module ......................................................................................... 15
         2.2.7.4 Two Wire Interface (TWI, a.k.a. I^C) Communication Module ......................... 22
         2.2.7.5 USART Tx/Rx Control Module .......................................................................... 27
         2.2.7.6 LED Control Module ......................................................................................... 29
         2.2.7.7 External RAM Interface Module ......................................................................... 30
         2.2.7.8 Linked List Module .......................................................................................... 31
         2.2.7.9 EEPROM Tags (Rudimentary EEPROM File System) Protocol Module .......... 34
         2.2.7.10 Command Line Interface (CLI) Module ........................................................... 38
         2.2.7.11 CLI Support Functions Module ......................................................................... 40
         2.2.7.12 Error Log Control Module ............................................................................ 41
## Software Control Document

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.7.13</td>
<td>Robotic Platform Configuration Module</td>
<td>43</td>
</tr>
<tr>
<td>2.2.7.14</td>
<td>Proportional Integral Derivative (PID) Controller Module</td>
<td>47</td>
</tr>
<tr>
<td>2.2.7.15</td>
<td>Motor Module Control Module</td>
<td>51</td>
</tr>
<tr>
<td>2.3</td>
<td>Client Software</td>
<td>54</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Software High Level Design</td>
<td>54</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Repository Directory Structure</td>
<td>54</td>
</tr>
<tr>
<td>2.3.3</td>
<td>RPCTRL</td>
<td>56</td>
</tr>
<tr>
<td>2.3.4</td>
<td>RPGUI</td>
<td>57</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Compilation Procedure</td>
<td>58</td>
</tr>
<tr>
<td>2.3.6</td>
<td>Usage Instructions</td>
<td>59</td>
</tr>
<tr>
<td>2.3.7</td>
<td>Graphical User Interface Walkthrough</td>
<td>60</td>
</tr>
<tr>
<td>2.3.7.1</td>
<td>Navigation Panel</td>
<td>60</td>
</tr>
<tr>
<td>2.3.7.2</td>
<td>Script Processing and Recording</td>
<td>62</td>
</tr>
<tr>
<td>2.3.7.3</td>
<td>Motor Module Control Panel</td>
<td>63</td>
</tr>
<tr>
<td>2.3.7.4</td>
<td>TWI/I(^2)C Communication Interface Panel</td>
<td>64</td>
</tr>
<tr>
<td>2.3.7.5</td>
<td>Configuration Window</td>
<td>67</td>
</tr>
<tr>
<td>2.3.7.6</td>
<td>Motor Module Status Window</td>
<td>72</td>
</tr>
<tr>
<td>2.3.7.7</td>
<td>System Health Window</td>
<td>73</td>
</tr>
<tr>
<td>2.3.7.8</td>
<td>Console Text Viewer Window</td>
<td>75</td>
</tr>
<tr>
<td>2.3.7.9</td>
<td>About Dialog</td>
<td>76</td>
</tr>
<tr>
<td>2.4</td>
<td>PID Controller</td>
<td>77</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Software High Level Design</td>
<td>77</td>
</tr>
<tr>
<td>2.4.1.1</td>
<td>PID Controller/TWI Bus Manager</td>
<td>77</td>
</tr>
<tr>
<td>2.4.1.2</td>
<td>Message Handler</td>
<td>77</td>
</tr>
<tr>
<td>2.4.1.3</td>
<td>Motor and Servo Controller</td>
<td>77</td>
</tr>
<tr>
<td>2.4.1.4</td>
<td>PID Engine</td>
<td>78</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Repository Directory Structure</td>
<td>78</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Compilation and Programming Instructions using the Arduino IDE</td>
<td>78</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Command Set between Microcontroller and PID Controller</td>
<td>80</td>
</tr>
</tbody>
</table>

## Acronyms

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acronyms</td>
<td>81</td>
</tr>
</tbody>
</table>
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>System Level Block Diagram</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Operational Software Organization</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Software Repository</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Software Repository</td>
<td>56</td>
</tr>
<tr>
<td>Figure 5</td>
<td>NetBeans IDE</td>
<td>58</td>
</tr>
<tr>
<td>Figure 6</td>
<td>RPGUI Dist Directory</td>
<td>59</td>
</tr>
<tr>
<td>Figure 7</td>
<td>RPGUI Latest Build Archive</td>
<td>59</td>
</tr>
<tr>
<td>Figure 8</td>
<td>GUI Navigation Panel</td>
<td>61</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Script Control Menu Items</td>
<td>62</td>
</tr>
<tr>
<td>Figure 10</td>
<td>GUI Motor Module Control Panel</td>
<td>64</td>
</tr>
<tr>
<td>Figure 11</td>
<td>GUI TWI/I(^2)C Communication Panel</td>
<td>66</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Configuration Menu Item</td>
<td>67</td>
</tr>
<tr>
<td>Figure 13</td>
<td>GUI Configuration Window Serial Panel</td>
<td>68</td>
</tr>
<tr>
<td>Figure 14</td>
<td>GUI Configuration Window System Panel</td>
<td>69</td>
</tr>
<tr>
<td>Figure 15</td>
<td>GUI Configuration Window Motor Modules Panel</td>
<td>70</td>
</tr>
<tr>
<td>Figure 16</td>
<td>GUI Configuration Window Motor Groups Panel</td>
<td>72</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Motor Module Status Menu Item</td>
<td>72</td>
</tr>
<tr>
<td>Figure 18</td>
<td>GUI Motor Module Status Window</td>
<td>73</td>
</tr>
<tr>
<td>Figure 19</td>
<td>System Health Status Menu Item</td>
<td>73</td>
</tr>
<tr>
<td>Figure 20</td>
<td>GUI System Health Window</td>
<td>74</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Console Viewer Menu Item</td>
<td>75</td>
</tr>
<tr>
<td>Figure 22</td>
<td>GUI Console Viewer Window</td>
<td>76</td>
</tr>
<tr>
<td>Figure 23</td>
<td>GUI About Dialog</td>
<td>76</td>
</tr>
<tr>
<td>Figure 24</td>
<td>PID Controller I(^2)C Command Protocol</td>
<td>81</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Table of Acronyms</td>
<td>81</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Table of References</td>
<td>82</td>
</tr>
</tbody>
</table>
1 Overview

1.1 Software Control Document
The Software Control Document (SCD) is a collection of all Software related design and test material for the Robotic Platform for 1kg Payloads (RP1). The SCD outlines the Graphical User Interface (GUI) software design and usability as well as the Operational Software design and usability. Care is made to give a detailed overview of the design of the software, both client and operational software, to allow for future teams to develop on top of this platform with ease.

Contained within this document is a brief system level description of the RP1 platform which is mirrored in the Interface Control Document (ICD) and Hardware Control Document (HCD) for convenient accessibility to high level system organization. Further in this document the Operational Software design methodology and source code repository layout, section 2.2.1, as well as rudimentary Application Programmers Interface (API) is defined. Similar detail is given to the client software design and API. Operational Software programming, compilation procedures, and microcontroller flash programming procedures are all defined in sections 2.2.3 and 2.2.5 below.

1.2 General Information
The Robotic Platform for 1kg Payloads (RP1) is a robotic assembly and physical platform built for the purpose of expediting construction of robotics of a much higher complexity. Quite frequently rudimentary navigation and obstacle avoidance logic consumes a large portion of time when building any robotic device. This platform is intended for applications in which a robotic device needs navigation control but the builder does not want to focus a lot of time or money into designing the components which manipulate motion.

The RP1 system consists of two core assemblies: the RP1 Control System and the RP1 Mechanical Motor Module and chassis. The control system will interface with a payload which will have full control over the platform itself. This will allow the payload to control the navigation of the platform. The payload in this instance would be any robotic device which will build off of the RP1 platform as a basis of motion.

The platform does not rely solely on the payload to command navigation. The RP1 control system also comes equipped with a wireless communications device which will allow a user at any PC machine equipped with the Graphical User Interface (GUI) software and appropriate wireless communication hardware to control navigation of the robotic platform. In this scenario, the payload may rest idle and perform its own, separate tasks or it may poll the platform for encoder data, power data, or any peripheral sensor data for which the system may come equipped.
1.3 System High Level Design

The System High Level Design is the technical layout and design of the RP1 control system. The system is broken down into a number of subsystems which are each designed, implemented, and tested individually and tested during system integration.

There are a total of seven subsystems, the **Graphical User Interface**, the **Wireless Communications** subsystem, the **Power Distribution** subsystem, the **Processing** subsystem, the **Motor Module Controller** subsystem, and lastly the **Motor Module** subsystem. A block diagram of these subsystems and their interconnections is displayed in **Figure 1**.

The Graphical User Interface (GUI) is a software application written in JAVA and is thusly cross platform compatible. Any operating system running the JAVA JRE (Java Runtime Environment) will be able to run the GUI client software. The details of the GUI are described in the Interface Control Document (ICD). Please see section 4 below for the location of the ICD.

The Wireless Communications subsystem is a subsystem dedicated to maintaining wireless control over the robotic platform. The details of the communication properties and communication protocol of this subsystem are described in the ICD. Please see section 4 below for the location of the ICD.

The Processing subsystem is the computational heart of the robotic platform. This subsystem contains the processing core which computes all commands issued through the communication channel into motor controls and sensor feedback to the user. The details of this subsystem are not relevant within the scope of the ICD, for more information please reference the Customer Needs, Design Specifications, and any design documentation which may pertain to the Processing subsystem.

The Motor Module Controller subsystem and subsequent Motor Module subsystem are the logic and device systems for actuating a motor. The Motor Module Controller subsystem contains logic to actuate a motor, but does not contain the motor or driver circuitry itself. The controller subsystem simply generates the timing and control signals which are then fed into the Motor Module subsystem. There are a variety of Motor Module controllers for the various supported motors. DC and Stepper motors required extra controller and timing circuitry to operate, while Servos require only Pulse Width Modulation (PWM) input. DC and Stepper motors require PWM inputs as well. Moving PWM generation responsibility off chip onto a separate microcontroller, identified in the diagram as the PID Controller module will allow for increased system modularity and less load on the core processing system.

The Motor Module system is not part of the RP1 control system platform, but is mentioned here only for clarity of the design. The Motor Module will interface with the control system through the electrical interface defined in the ICD. Please see section 4 below for the location of the ICD. The Motor Module is expected to utilize the timing signals specified in this interface to control driver circuitry and the motor. The driver circuitry, whether this be a motor H-Bridge or some other device, shall be contained within the Motor Module itself and not within the RP1 control system.
2 Software Repository

2.1 Software Versions

Care should be made to Tag revisions of operational software code when an official new release is made. Label the tagged submission based on “Version N.M-K” where N, M, K are the major, minor, and revision numbers (i.e. “Version 1.0-0”).

2.1.1 Version 1.0-0

This is the first release version of the software package. Any further releases or updates to this version shall be clearly discussed in future versions or revisions. The contents and capabilities of the software are bulleted below:

- Core CLI supporting two simultaneous command interfaces over both RS-232 channels
- No multiple access mitigation for the command interfaces are provided
2.2 Operational Software

The Operational Software (OpSoft) is the software which resides on the processing subsystem of the RP1 and processes commands from the wired and wireless communication channels. These commands call upon specific functionality compiled into the OpSoft to control different aspects of the platform.

2.2.1 Software High Level Design

The Operational Software is a layered design, each layer abstracting the behavior of the layer below for the layer above. The diagram in Figure 2 displays the layered design of the software.

The Application Commands layer is the highest layer of abstraction, and is the layer the user will directly interface with. This software layer contains all commands listed in the ICD to which a payload or client may call upon.

The Command Line Interface (CLI) layer is an integral layer which contains all core logic for allowing swift implementation of new application commands. This layer need not be edited when adding new commands but must be thoroughly understood in order to develop new commands appropriately. This layer handles multiple access (communication with both wired and wireless channels on two RS-232 ports), and does so seamlessly without concerning the Application Commands layer above.

The Intelligence layer contains all advanced logic for “smart” navigation and advanced algorithms for processing and utilizing sensor feedback. This layer should contain all functionality which is more advanced than any rudimentary navigation control, such as accurate robotic positioning or extraneous sensor integration.

The Navigation layer contains all functionality to convert speed input, motor indexing, forward/reverse input, and any desired rudimentary motion commands into the appropriate signals to registered devices. This layer contains all the necessary logic to translate commands such as “move motor N forward at a speed of X” or “turn wheel N a number of degrees X clockwise” into commands to specific devices, abstract speed calculations, and incorporation of closed loop feedback values when utilizing encoder feedback signals.
The Device layer is a single layer of abstraction over the driver layer, but is very similar in behavior. The Device layer contains any I²C addresses, timer configuration parameters, knowledge of motor controller signals, and any device specific configuration or parameters and can convert abstract commands to control a device into a procedure of actions which are performed on devices attached to the system. Such devices may include but are not limited to motor controllers, I²C devices (PIC controllers over serial interface), Encoder Feedback circuitry, and any peripheral device attached to the microcontroller. The device layer also contains core operating system code including linked lists structures, threads control, semaphore logic, and additional operating system tools. These tools are placed here because they reside at a higher level of intelligence above the drivers but belong strictly below the navigation layers. Instead of making another layer separate to devices but residing in the same level, it was decided to put this responsibility within the device layer, despite these modules not being devices.

The Driver layer is the layer closest to the hardware itself and contains all code which controls microcontroller hardware functionality. Universal Synchronous/Asynchronous Receiver/Transmitter (USART) Input/Output (I/O) and configuration, timer and PWM configuration and operation control, EEPROM configuration and read/write functionality, I²C packet I/O, Analog to Digital Converter (ADC) functionality, and other such on-chip device functionality exists here. All interrupt management is handled strictly in this layer only and shall not be utilized in any layer of higher abstraction.
2.2.2 Repository Directory Structure

The operational software repository is located in the ‘software/operational software’ subdirectory within the public directory of the P09204 team document repository. The operational software directory has a src directory containing all compilable source code, and a templates directory containing Makefile, C source, and C header templates (see Figure 3).

Within the src directory are three subdirectories. The drivers subdirectory contains all driver code which resides in the Drivers Layer of the operating system layered schema (see Figure 2). The rp1 subdirectory contains all Command Line Interface and Application Commands code. Lastly, the rpos subdirectory, which stands for Robotic Platform Operating System, contains all Operating System code including Device code and Navigation and Intelligence code (see Figure 3). As the system develops, new subdirectories may be added within these three mentioned or in addition to these three; these changes will not be mentioned here.
2.2.3 Installing the Compiler Toolset
Note, this section assumes that the default installation procedures will be followed for each application install as defined on their respective websites.


- Download and install TortoiseSVN from http://tortoisesvn.net/.

- Checkout RP1 P09204 repository by creating a folder named P09204 somewhere on your system (i.e. My Documents in Windows). Click File -> SVN Checkout. Type in https://edge.rit.edu/dav/P09204 as the URL of repository and click OK. When prompted enter username and password and click OK.

- Download and install WinAVR from http://winavr.sourceforge.net/. The AVR tools can be used within cygwin through makefiles.

2.2.4 Compilation Procedure
Open a cygwin window. Please note that in Windows, cygwin does not recognize drive letters, but concerns these as “/cygdrive/drv-letter”. Navigate to the directory the repository was checked out. Navigate to subdirectory web/public/software/operational software/src/rp1.

From within this directory, type `make` to build the operational software binary. The binary and map files are stored in the bin subdirectory. All linkable object files are stored in the obj subdirectory.

NOTE: The makefiles for the RP1 project have already been created, but they will need to be modified as source files are added or removed from the project.

2.2.5 Programming the BDMicro MAVRICIIB ATmega128
Make sure the programming parallel cable is plugged in to the PC in use and into the Microcontroller or RP Control System. Power off the Microcontroller and once the system has fully shut down restore power while the programming cable is still plugged in.
Within `cygwin`, navigate to the build directory mentioned above in the Compilation Procedure. Type `make flash` to start the Microcontroller programming sequence. Allow the programmer to fully finish and verify the flash file before removing the programming cable.

### 2.2.6 Connecting to the BDMicro MAVRICIIB

Please reference the Interface Control Document (ICD) for details on communicating with the BDMicro MAVRICIIB (see Section 4).

### 2.2.7 Application Programmers Interface

Contained in this section is a detailed list of all API available to RP1 software developers including the header and source files associated with each module, the software layer to which it resides, and the directory location in the repository for which it resides. All directories are subdirectories within the `src` base directory of the Operational Software repository.

#### 2.2.7.1 Analog to Digital Converter Control Module

Source File: `adc.c`

Header File: `adc.h`

Description: Controls the on-chip ADC device

Directory: `drivers`

Layer: Drivers

**Typedef Data Defined:**

```c
void adc_fn_t( uint16_t )
    ADC response function template, used to register ADC interrupt response handlers
```

**Functions Defined:**

**adc_sample**

Definition: `uint16_t adc_sample( void )`

Description: Poll ADC module for sample value

Inputs: None

Outputs: None

Return: 16-bit Unsigned Integer ADC Value
adc_register
  Definition:    void adc_register( adc_fn_t * fn )
  Description:  Registers a function within the interrupt handler,
                 the function is of type: void adc_fn_t( uint16_t )
  Inputs:
    adc_fn_t * fn
    The interrupt handler function pointer
  Outputs:   None
  Return:     None

adc_unregister
  Definition:    void adc_unregister( void )
  Description:  Unregister the interrupt handler for the ADC
  Inputs:       None
  Outputs:      None
  Return:       None

Macros Defined:

ADC_INIT(mux,ps)
  Description: Initialize the ADC module with multiplexer value
               ‘mux’ and control register value ‘ps’.

ADC_INIT_INTERNAL(i)
  Description: Initialize ADC (indexed by ‘i’) to 125kHz sampling
               rate, right adjusted values, internal 1.25V
               reference, single sample only, interrupts disabled
               (ADC disabled)

ADC_INIT_EXTERNAL(i)
  Description: Initialize ADC (indexed by ‘i’) to 125kHz sampling
               rate, right adjusted values, external reference,
               single sample only, interrupts disabled (ADC
disabled)

ADC_INIT_AVCC(i)
  Description: Initialize ADC (indexed by ‘i’) to 125kHz sampling
               rate, right adjusted values, AVCC reference, single
               sample only, interrupts disabled (ADC disabled)

ADC_ENABLE()
  Description: Enable ADC module, should be called immediately after
               ADC_INIT_* macros are called
ADC_DISABLE()
Description: Disable/Deactivate the ADC module, call ADC_ENABLE to reenable

ADC_ISENABLED()
Description: Returns TRUE of ADC is enabled

ADC_START()
Description: Kicks off an ADC sample

ADC_ISBUSY()
Description: Returns TRUE if ADC is in the middle of sampling

ADC_INT_ENABLE()
Description: Enables ADC interrupts, interrupt will fire after sampling is complete, ADC module will call a registered interrupt response function

ADC_INT_DISABLE()
Description: Disable ADC interrupts

ADC_INT_ISENABLED()
Description: Returns TRUE if ADC is enabled

ADC_SINGLE()
Description: Sets ADC to perform only one sample when started

ADC_FREERUN()
Description: Sets ADC to continuously sample when started

ADC_ISFREERUN()
Description: Returns TRUE if ADC is set to “freerun” mode

2.2.7.2 EEPROM Tx/Rx Control Module
Source File: eeprom.c
Header File: eeprom.h
Description: Functions to control reading from and writing to on chip EEPROM
Directory: drivers
Layer: Drivers
Functions Defined:
Robotic Platform 1kg
Software Control Document

**eeprom_purge**

Definition: void eeprom_purge( void )
Description: Resets all bytes in EEPROM to NULL
Inputs: None
Outputs: None
Return: None

**eeprom_read**

Definition: uint8_t eeprom_read( uint16_t eeprom_addr )
Description: Read a single byte from EEPROM
Inputs:
  uint16_t eeprom_addr
  16-bit byte address in 4KB EEPROM
Outputs: None
Return: 8-bit byte data value read from EEPROM, if address is invalid
a zero is returned.

**eeprom_write**

Definition: void eeprom_write( uint16_t eeprom_addr, uint8_t data )
Description: Write a single byte to EEPROM at a specified address
Inputs:
  uint16_t eeprom_addr
  16-bit byte address in 4KB EEPROM
  uint8_t eeprom_data
  8-bit data value to write
Outputs: None
Return: None

2.2.7.3 Timer Control Module

Source File: timer.c
Header File: timer.h
Description: Contains all code to initialize and utilize on-chip timer
modules.
Directory: drivers
Layer: Drivers

Typedef Data Defined:
void timer_fn_t( uint8_t timerndx )
  Timer response function template, used for registering
timer interrupt response handlers
struct time_t {
    uint16_t milliseconds; /* max 1000 */
    uint8_t seconds;        /* max 60 */
    uint8_t minutes;        /* max 60 */
    uint8_t hours;          /* max 60 */
    uint8_t days;           /* max 41 */
};

time_t

Timer clock data, when a timer is initialized as a clock
this global data structure is allocated to memory and accurate
time data is recorded as time passes

Functions Defined:

timer_stop
 Definition:    void timer_stop( uint8_t timer_index )
 Description:   Disables an active timer
    Inputs:
        uint8_t timer_index
            TIMER0, TIMER1, TIMER2, TIMER3, the ATmega128 has four
            on chip timers, use one of the aforementioned defines
            for this input
    Outputs: None
    Return: None

timer_start
 Definition:    void timer_start( uint8_t timer_index )
 Description:   Starts a deactivated timer
    Inputs:
        uint8_t timer_index
            TIMER0, TIMER1, TIMER2, TIMER3, the ATmega128 has four
            on chip timers, use one of the aforementioned defines
            for this input
    Outputs: None
    Return: None
timer_freq_get
Definition: uint32_t timer_freq_get( uint8_t timer_index )
Description: Returns the programmed frequency of the timer
Inputs:
  uint8_t timer_index
  TIMER0, TIMER1, TIMER2, TIMER3, the ATmega128 has four
  on chip timers, use one of the aforementioned defines
  for this input
Outputs: None
Return: 32-bit frequency of the timer

time_diff
Definition:
  bool_t time_diff( time_t * time_a, time_t * time_b,
                  time_t * time_c )
Description: Finds the difference between times,
  time_c = time_a - time_b
  Returns FALSE IF time_b > time_a, time_c will
  contain invalid data.
Inputs:
  time_t * time_a
    Time structure A
  time_t * time_b
    Time structure B
Outputs:
  time_t * time_c
    Output time structure (carries result)
Return: TRUE if output is valid and operation was successful,
  FALSE otherwise

to_seconds
Definition: uint16_t to_seconds( time_t * time )
Description: Convert a time structure into a count of seconds.
  If there is an integer overflow, this function will
  return 65535.
Inputs:
  time_t * time
    Time structure to convert to seconds
Outputs: None
Return: 16-bit value, number of seconds
**to(milliseconds)**

**Definition:**

uint16_t to_milliseconds( time_t * time )

**Description:**

Convert a time structure into a count of milliseconds. If there is an integer overflow, this function will return 65535.

**Inputs:**

- time_t * time
  - Time structure to convert to milliseconds

**Outputs:** None

**Return:** 16-bit value, number of milliseconds

**timer_clock_get**

**Definition:**

void timer_clock_get( uint8_t timer_index, time_t * time )

**Description:**

Get timer clock data, useful for creating timestamps

**Inputs:**

- uint8_t timer_index
  - TIMER0, TIMER1, TIMER2, TIMER3, the ATmega128 has four on chip timers, use one of the aforementioned defines for this input

**Outputs:**

- time_t * time
  - Buffer to store output timestamp data

**Return:** None

**timer_clock_set**

**Definition:**

void timer_clock_set( uint8_t timer_index, time_t * time )

**Description:**

Set timer clock data, useful for setting to Epoch time if necessary

**Inputs:**

- uint8_t timer_index
  - TIMER0, TIMER1, TIMER2, TIMER3, the ATmega128 has four on chip timers, use one of the aforementioned defines for this input

- time_t * time
  - Buffer to of timestamp data, set the timer’s clock data

**Outputs:** None

**Return:** None
ms_sleep
Definition: void ms_sleep(uint16_t ms)
Description: Sleep for a user defined number of milliseconds
Inputs:
    uint16_t ms
    Number of milliseconds to sleep
Outputs: None
Return: None

timer_register
Definition:
    uint8_t timer_register( uint8_t timer_index, timer_fn_t * fn )
Description: Registers a function in the timer events handler, this function will be called by the timer interrupt on a clock tick
Inputs:
    uint8_t timer_index
    TIMER0, TIMER1, TIMER2, TIMER3, the ATmega128 has four on chip timers, use one of the aforementioned defines for this input
timer_fn_t * fn
    Timer interrupt response function, this function is called upon every clock tick of the timer
Outputs: None
Return: PASS on success, FAIL if the timer response function buffer is already full
timer_unregister

Definition:

    void timer_unregister( uint8_t timer_index, timer_fn_t * fn )

Description: Unregisters a function in the timer events handler, if the function is not registered with the timer, then this function will do nothing.

Inputs:

    uint8_t timer_index
    TIMER0, TIMER1, TIMER2, TIMER3, the ATmega128 has four on chip timers, use one of the aforementioned defines for this input

    timer_fn_t * fn
    Timer interrupt response function to remove from the response function list

Outputs: None

Return: None
timer_init

Definition:

```c
void timer_init( uint8_t timer_index, uint8_t prescaler,
                uint16_t ocr, bool_t clock )
```

Description: Initializes a timer. Timers initialized in this way do not make use of PWM functionality. Timers initialized in this way utilize the prescaled system clock and upon timer interrupt Output Compare Register (OCR) is reset.

Use safe macros instead of directly calling this function. Header and macros are defined to tune timer clocks based on chip frequency. Any hard coded prescaler and OCR values may be subject to error if chip frequency is changed or if platform is changed.

Safe macros are defined below:

```
TIMER_INIT_1MS(timer)
TIMER_INIT_100US(timer)
TIMER_INIT_10US(timer)
TIMER_INIT_1US(timer)
```

Inputs:

- `uint8_t timer_index` Index of timer (use TIMERn defines)
  - TIMER0 - 8bit General Purpose Timer (System clock)
  - TIMER1 - 16bit High Precision Timer
  - TIMER2 - 8bit General Purpose Timer
  - TIMER3 - 16bit High Precision Timer
  - TIMER_DISABLE - Disable all Timers

- `uint8_t prescaler` System internal clock prescaling (clock divider)
- `uint16_t ocr` Output Compare Register value, when the timer has counted this number of prescaled clock rising edges an internal interrupt will be fired and the OCR will be reset
- `bool_t clock` Will the timer run a system clock?

Outputs: None
Return: None

Macros Defined:
TIMER_INIT_1MS(timer)
Description: Initialize a timer for 1ms clock ticks

TIMER_INIT_400US(timer)
Description: Initialize a timer for 400us clock ticks

TIMER_INIT_200US(timer)
Description: Initialize a timer for 200us clock ticks

TIMER_INIT_100US(timer)
Description: Initialize a timer for 100us clock ticks

TIMER_INIT_50US(timer)
Description: Initialize a timer for 50us clock ticks

TIMER_INIT_10US(timer)
Description: Initialize a timer for 10us clock ticks

TIMER_INIT_1US(timer)
Description: Initialize a timer for 1us clock ticks

2.2.7.4 Two Wire Interface (TWI, a.k.a. FC) Communication Module
Source File: twi_ctrl.c
Header File: twi_ctrl.h
Description: Two-Wire Interface (TWI) control module, contains all necessary function calls to initialize, disable, read or write to the on-chip Inter-Integrated Circuit (I2C) serial data bus. This module works by first initializing the serial data bus, then enqueuing read or write operations, followed by a single call to twi_write_bytes. At this time all enqueued data will be sent over the bus to the devices they’ve addressed and if a response handler has been registered it will be called several times during each stage of the transmission.

Directory: drivers
Layer: Drivers
Typedef Data Defined:

```c
struct twi_packet_t {
    uint8_t addr;       /* address of packet (to or from) */
    uint8_t len;        /* length of data in packet */
    uint8_t* data;      /* data read or to be sent */
    uint8_t direction;  /* direction of data flow, 
                          INCOMING or OUTGOING */
    /* special packet type TWI_PT_ADDR, 
     for SLA packets */
    uint8_t cursor;     /* "cursor" of data where transmission 
                          or reception has arrived, keeping track 
                          of byte placement */
} twi_packet_t
```

TWI packet data, generated internally during write operations
or returned during a TWI response function call after a bus
operation has been performed

```c
uint8_t twirc_t
```

TWI return code:

- TWI_RC_OK - generic TWI OK, or pass code
- TWI_RC_FAIL - generic TWI failure code
- TWI_RC_PEND - waiting for appropriate interrupt
- TWI_RC_ALREADY_INIT - init cannot be performed when the TWI is enabled
- TWI_RC_NO_DATA - No data to write, write op abort
- TWI_RC_DISABLED - TWI module is disable, no ops can be performed
- TWI_RC_BUS_ERROR - TWI bus error, bus failed

```c
void (*twi_fn_t) ( twirc_t, twi_packet_t* )
```

TWI response function template, response code and TWI packet
as parameters; twirc_t can be one of the following:

- TWI_RSP_OPENED - TWI module finished initializing
- TWI_RSP_CLOSED - TWI module finished closing
- TWI_RSP_READ_DATA - TWI response, Rx operation complete
- TWI_RSP_WRITE_DATA - TWI response, Tx operation complete
- TWI_RSP_BUS_OPEN - TWI bus opened, entered Master mode
- TWI_RSP_BUS_CLOSE - TWI bus closed, leaving Master mode
- TWI_TSP_SLA_NACK - TWI no ACK from Slave Device
- TWI_RSP_BUS_ERROR - TWI response to a bus error

Functions Defined:
Robotic Platform 1kg
Software Control Document

**twi_init**

**Definition:**

twrc_t twi_init( uint8_t sladdr, uint8_t freq )

**Description:** Initialize the TWI serial data bus

**Inputs:**

- uint8_t sladdr
  - Optional slave address, leave as 0x00 if this device is not intended on being a bus slave
- uint8_t freq
  - Operational frequency selection: TWI_400KHZ, TWI_100KHZ

**Outputs:** None

**Return:** May return one of the following return codes (defined above):

- TWI_RC_OK – Operation OK
- TWI_RC_ALREADY_INIT – Bus already initialized

**twi_clean_queue**

**Definition:**

void twi_clean_queue( void )

**Description:** Purge the transmission queue, remove all data queued for transmission over the data bus

**Inputs:** None

**Outputs:** None

**Return:** None

**twi_destroy_packet**

**Definition:**

void twi_destroy_packet( twi_packet_t * pkt )

**Description:** Destroy an enqueued packet, releasing all memory (this function is used internally)

**Inputs:**

- twi_packet_t * pkt
  - The packet to destroy

**Outputs:** None

**Return:** None

**twi_pop_packet**

**Definition:**

twi_packet_t * twi_pop_packet( void )

**Description:** Pops a packet off the TWI transmission stack (this function is used internally)

**Inputs:** None

**Outputs:** None

**Return:** Returns a pointer to the packet which was popped from the TWI transmission stack, the packet is at this point no longer on the stack
twi_enqueue_write

**Definition:**
```
twirc_t twi_enqueue_write( uint8_t addr, uint8_t len, 
uint8_t* data )
```

**Description:** Enqueue data to be written to a slave device, this performs a write operation to whichever device is specified by addr

**Inputs:**
- `uint8_t addr`
  8-bit Byte address of slave device
- `uint8_t len`
  Length of data to transmit to slave device
- `uint8_t * data`
  Pointer to buffer of data to transmit

**Outputs:** *None*

**Return:** May return one of the following return codes (defined above):
- `TWI_RC_OK` - Operation OK
- `TWI_RC_FAIL` - Failed to place packet onto the stack, could be due to a memory allocation failure

**twi_write_bytes**

**Definition:**
```
twirc_t twi_write_bytes( bool_t wait )
```

**Description:** Kicks off write operation to the TWI bus, if Boolean value wait is TRUE will halt the currently running thread until the operation finishes.

**Inputs:**
- `bool_t wait`
  If TRUE, will force operation to block and wait until the bus returns to an inactive state and all transmissions are complete

**Outputs:** *None*

**Return:** May return one of the following return codes (defined above):
- `TWI_RC_OK` - Data completely sent, STOP sent, queue empty
- `TWI_RC_PEND` - Operation in progress, wait for interrupt
- `TWI_RC_DISABLED` - TWI Module is disabled, no operations can occur
- `TWI_RC_NO_DATA` - No data in write queue to send, operation aborted
- `TWI_RC_FAIL` - TWI module is in invalid state, close and reinitialize
### twi_enqueue_read

**Definition:** \(\text{twirc_t twi_enqueue_read( uint8_t addr, uint8_t len )}\)

**Description:** Enqueue data to be written to a slave device, this performs a read operation to whichever device is specified by \(addr\), if successful this will result in returned packet data from the response handler.

**Inputs:**
- \(uint8_t \text{addr}\)
  - 8-bit Byte address of slave device
- \(uint8_t \text{len}\)
  - Length of data to read from slave device

**Outputs:** None

**Return:**
- TWI_RC_OK – Operation OK
- TWI_RC_FAIL – Failed to place packet onto the stack, could be due to a memory allocation failure

### twi_close

**Definition:** \(\text{twirc_t twi_close( void )}\)

**Description:** Softly closes the TWI device, waiting for all operations to desist before disabling the TWI device.

**Inputs:** None

**Outputs:** None

**Return:** May return one of the following return codes (defined above):
- TWI_RC_OK - Module closed.
- TWI_RC_PEND - TWI module will close soon.

### twi_register

**Definition:** \(\text{void twi_register( twi_fn_t twi_fn )}\)

**Description:** Registers a function as a response function for pertinent TWI interrupts.

**Inputs:**
- \(twi_fn_t \text{twi_fn}\)
  - Function pointer of function to be called when an event occurs on the TWI bus

**Outputs:** None

**Return:** None
twi_unregister
Definition:   void twi_unregister( twi_fn_t twi_fn )
Description: Removes a registered function from the response queue
Inputs:
      twi_fn_t twi_fn
Function pointer of function to remove from response queue
Outputs:   None
Return:    None

2.2.7.5 USART Tx/Rx Control Module
Source File:  usart.c
Header File: usart.h
Description: This module contains routines to control the RS232 USART ports
Directory:   drivers
Layer:      Drivers

Functions Defined:

usart_rx_ready
Definition:   bool_t usart_rx_ready( uint8_t usart_index )
Description: Determine whether or not a USART channel is ready
for Rx operations
Inputs:
      uint8_t usart_index
Index of USART, either USART0 or USART1
Outputs:   None
Return:    TRUE if USART is not busy, FALSE if USART is busy
**usart_init**

Definition: void usart_init( uint8_t usart_index, uint32_t baud )

Description: Initialize USART device to operate at a user defined baud rate

Inputs:

- uint8_t usart_index
  - Index of USART, either USART0 or USART1
- uint32_t baud
  - Baud rate to initialize USART interface. Typical settings:
    - USART_BAUD_57600
    - USART_BAUD_38400
    - USART_BAUD_19200
    - USART_BAUD_9600

Outputs: *None*

Return: *None*

**usart_close**

Definition: void usart_close( uint8_t usart_index )

Description: Close the USART and cease all operations

Inputs:

- uint8_t usart_index
  - Index of USART, either USART0 or USART1

Outputs: *None*

Return: *None*
**usart_stdio_init**

**Definition:**

```c
FILE * usart_stdio_init( uint8_t usart_index, uint32_t baud )
```

**Description:** Initialize the system to use standard input/output function calls (printf, getchar, ...) on a given USART. Do NOT call this function if a call to fdevopen or this function has already been made UNLESS you call fclose() on the returned file descriptor before making the call.

**Inputs:**

- `uint8_t usart_index`
  - Index of USART, either USART0 or USART1
- `uint32_t baud`
  - Baud rate to initialize USART interface. Typical settings:
    - USART_BAUD_57600
    - USART_BAUD_38400
    - USART_BAUD_19200
    - USART_BAUD_9600

**Outputs:** None

**Return:** Returns standard input/output file descriptor, store this in case the USART channel needs to be closed and to free allocated resources.

**stdio_flush**

**Definition:**

```c
void stdio_flush( void )
```

**Description:** Flush the Tx buffer, block until all Tx operations have been completed.

**Inputs:** None

**Outputs:** None

**Return:** None

### 2.2.7.6 LED Control Module

**Source File:** None

**Header File:** led.h

**Description:** Contains macros to turn the BDMicro onboard LED on or off

**Directory:** drivers

**Layer:** Drivers

**Macros Defined:**

- `LED_ENABLE()`
  - Enable LED control (activate port)
LED_DISABLE()
    Disable LED control (activate port)

LED_ON()
    Turn LED on

LED_OFF()
    Turn LED off

LED.Toggle()
    Toggle LED on or off

2.2.7.7 External RAM Interface Module
Source File: xram.c
Header File: xram.h
Description: Contains routines to initialize or detach extended RAM modules, also known as “X-RAM”
Directory: drivers
Layer: Drivers

Functions Defined:

xram_enable
    Definition:
    void xram_enable (void) __attribute__ ((naked)) __attribute__ ((section (".init1")))
    Description: Called when the system initializes, specified by the __attribute__ directive, this function enables the system to utilize XRAM; simply including this module will enable the extended memory module, no other initialization code is necessary
    Inputs: None
    Outputs: None
    Return: None

xram_disable
    Definition: void xram_disable (void)
    Description: Disable the use of extended memory
    Inputs: None
    Outputs: None
    Return: None
2.2.7.8 Linked List Module

Source File: llist.c
Header File: llist.h

Description: Functions and data structures defined to create and manipulate linked list data structures

Directory: rpos
Layer: Operating System

Typedef Data Defined:

```c
volatile struct llist_t {
    volatile struct llist_t * next; /* Pointer to next list in linked list */
    volatile void * data; /* Pointer to data of any type */
} llist_t
```

Linked list data structure; this data structure represents a single node in the linked list; use function calls below to initialize, add nodes to, or remove nodes from linked list data

Functions Defined:

**llist_init**

**Definition:** llist_t * llist_init( void * data )

**Description:** Initialize a linked list, allocating memory for the new structure

**Inputs:**

- void * data
  - Pointer to data to be stored in initialize node

**Outputs:** None

**Return:** Returns the new root node of the linked list or NULL if a failure occurred
**llist_remove**

**Definition:**

```c
llist_t * llist_remove( llist_t * llist, void * data )
```

**Description:** Remove a node from a linked list that has the associated data, if NULL is passed in as 'llist' argument, NULL is returned

**Inputs:**

- `llist_t * llist`: Linked list to remove node from
- `void * data`: If a node exists with the same data pointer, it will be removed

**Outputs:** *None*

**Return:** Returns the new root node of the linked list or NULL if the list is now empty

---

**llist_find**

**Definition:**

```c
llist_t * llist_find( llist_t * llist, void * data )
```

**Description:** Locate a node in the linked list which contains the specified data

**Inputs:**

- `llist_t * llist`: The linked list to modify
- `void * data`: If a node exists with the same data pointer, it will be returned

**Outputs:** *None*

**Return:** Returns the node which contains this data, or NULL if none exists

---

**llist_push**

**Definition:**

```c
llist_t * llist_push( llist_t * llist, void * data )
```

**Description:** Push a new node onto the linked list at the bottom of the list, if NULL is passed into 'llist' argument, this function mimics the 'llist_init' function

**Inputs:**

- `llist_t * llist`: The linked list to modify
- `void * data`: The data associated with the new node to create

**Outputs:** *None*

**Return:** Returns a direct pointer to the newly created node
### llist_pop

**Definition:**  
llist_t * llist_pop( llist_t * llist, void ** data )

**Description:**  
Pop the first node (top node) off the list returning the new base node, if NULL is passed in as an argument to 'llist' then NULL is returned

**Inputs:**
- llist_t * llist
  - The linked list to modify
- void ** data
  - A pointer to the data pointer

**Outputs:** None

**Return:** Returns the new root node of the linked list or NULL if the list is now empty

### llist_peek

**Definition:**  
void * llist_peek( llist_t * llist )

**Description:**  
Peek into the linked list, and return the data of the next node which would be popped

**Inputs:**
- llist_t * llist
  - The linked list to probe

**Outputs:** None

**Return:** Returns the data pointer of the next node to pop

### llist_destroy

**Definition:**  
void llist_destroy( llist_t * llist )

**Description:**  
Destroy the linked list and free all resources, this is DANGEROUS as all data pointers WILL BE LOST! There is no attempt to free the data associated with each list node; only use if node data has already been freed before this call is made

**Inputs:**
- llist_t * llist
  - The linked list to destroy

**Outputs:** None

**Return:** None
2.2.7.9 EEPROM Tags (Rudimentary EEPROM File System) Protocol Module

Source File: eeprom_tags.c
Header File: eeprom_tags.h

Description: Template Tags and Tag control for EEPROM memory usage, these tags define the proper application use of the EEPROM memory hierarchy and layout

Directory: rpos
Layer: Operating System

Typedef Data Defined:

typedef struct eeprom_tag {
  uint16_t type;
  uint8_t length;
} eeprom_tag_t

  EEPROM tag data structure which gets written to or read from EEPROM non-volatile memory

define enum eeprom_id {
  EEPROM_ID_NULL = 0x00, // Invalid...
  EEPROM_ID_TLV_01 = 0x01, // First TLV based EEPROM configuration
  EEPROM_ID_LAST
} eeprom_id

  This enumeration is a list of all possible EEPROM tag revisions, essentially defines how EEPROM tags are created and ordered in EEPROM; also known as the EEPROM Tags module version number

Functions Defined:

eeprom_tags_init
  Definition: void eeprom_tags_init( void )
  Description: Sets the EEPROM tags version ID to input value, assuming it is a verified ID value

  Inputs: None
  Outputs: None
  Return: None

eeprom_tags_valid
  Definition: bool_t eeprom_tags_valid( void )
  Description: Checks to test if the EEPROM tag data is valid

  Inputs: None
  Outputs: None
  Return: TRUE if valid, FALSE if not
eeprom_tags_read
Definition:
    int8_t eeprom_tags_read( uint16_t tag_type, uint8_t * len,
    uint8_t ** data )
Description: Read an EEPROM tag of the desired tag type
    Exits with FALSE if tag cannot be found or EEPROM data is invalid for reading
    Upon EEPROM_TAGS_PASS data MUST be freed by the calling function!!
Inputs:
    uint16_t tag_type
    Tag type (0-255) of tag to read back
Outputs:
    uint8_t * len
    Length of data in read buffer ‘data’
    uint8_t ** data
    Data buffer to read in EEPROM data
Return: May be one of the following return codes:
    EEPROM_TAGS_PASS - Success
    EEPROM_TAGS_INVALID - EEPROM not initialized
    EEPROM_TAGS_NOFIND - EEPROM tag not found, could not read
    EEPROM_TAGS_BADTYPE - Tag type supplied in arguments is invalid
    EEPROM_TAGS_INVARG - NULL pointer arguments
**eeprom_tags_write**

**Definition:**

\[
\text{int8_t } \text{eeprom_tags_write( eeprom_tag_t * eeprom_tag_wr, uint8_t * data )}
\]

**Description:** Write data to EEPROM

**Inputs:**

- `eeprom_tag_t * eeprom_tag_wr` EEPROM tag header, containing tag type, length, etc...
- `uint8_t * data` EEPROM tag data to be written to EEPROM

**Outputs:** None

**Return:** May be one of the following:

- `EEPROM_TAGS_PASS` - Success
- `EEPROM_TAGS_NOMEM` - Out of memory
- `EEPROM_TAGS_INVALID` - EEPROM not initialized
- `EEPROM_TAGS_BADTAG` - User supplied tag exists but with invalid length
- `EEPROM_TAGS_BADTYPE` - Tag type supplied in arguments is invalid
- `EEPROM_TAGS_INVARG` - NULL pointer arguments.

**eeprom_tags_erase**

**Definition:**

\[
\text{int8_t } \text{eeprom_tags_erase( uint16_t tag_type )}
\]

**Description:** Erases a tag of data in the EEPROM. This does not reorder memory, but instead simply marks the tag as deleted so that it's memory space may be overwritten

**Inputs:**

- `uint16_t tag_type` Tag type of tag to be erased

**Outputs:** None

**Return:** May be one of the following:

- `EEPROM_TAGS_PASS` - Success
- `EEPROM_TAGS_INVALID` - EEPROM not initialized
- `EEPROM_TAGS_NOFIND` - EEPROM tag not found, could not erase
- `EEPROM_TAGS_BADTYPE` - Tag type supplied in arguments is invalid
### eeprom_tags_retag

**Definition:**

\[
\text{int8_t eeprom_tags_retag( uint16_t tag_type, uint16_t new_tag_type )}
\]

**Description:** Changes the tag type of data in the EEPROM

**Inputs:**

- `uint16_t tag_type`
  - The old tag type to search for
- `uint16_t new_tag_type`
  - The new tag type to set

**Outputs:** None

**Return:** May be one of the following:

- `EEPROM_TAGS_PASS` - Success
- `EEPROM_TAGS_INVALID` - EEPROM not initialized
- `EEPROM_TAGS_NOFIND` - EEPROM tag not found, could not retag
- `EEPROM_TAGS_BADTYPE` - Tag type supplied in arguments is invalid

### eeprom_tags_iter

**Definition:**

\[
\text{int8_t eeprom_tags_iter( bool_t start, eeprom_tag_t * eeprom_tag, uint8_t ** data )}
\]

**Description:** Read back next tag in EEPROM tag read iteration

- Exits with FALSE if tag cannot be found or EEPROM data is invalid for reading.
- Upon `EEPROM_TAGS_PASS` data MUST be freed by the calling function.

**Inputs:**

- `bool_t start`
  - If TRUE this resets iteration to first tag item, if FALSE this continues read iteration

**Outputs:**

- `eeprom_tag_t * eeprom_tag`
  - Data buffer to read EEPROM header data
- `uint8_t ** data`
  - Data buffer to read in EEPROM data

**Return:** May be one of the following:

- `EEPROM_TAGS_PASS` - Success, item read
- `EEPROM_TAGS_INVALID` - EEPROM not initialized
- `EEPROM_TAGS_NOFIND` - Reached end of tags, no more tags to read
- `EEPROM_TAGS_INVARG` - NULL pointer arguments
- `EEPROM_TAGS_BADTAG` - Found an erased tag
2.2.7.10 Command Line Interface (CLI) Module

Source File: cli.c
Header File: cli.h

Description: Heart of the command line, contains all code to read from simultaneous input devices providing a unique command line to each serial device and abstracting character input from application code

Directory: rp1
Layer: CLI

Typedef Data Defined:

```c
struct cli_cmd_t { /* Command parameters */
    const PGM_P const name; /* command name */
    prog_void (*entry)(int, char**); /* Function pointer to run process */
} cli_cmd_t

This is the structure used to define an application; these command structures are placed in a “master command table” which is used by the CLI system to locate and call application functions from the command line
```

```c
struct cli_cmd_data_t { /* Command parameters */
    int argc;   /* Argument count */
    char ** argv; /* Command arguments */
} cli_cmd_data_t

Command data structure used internally to generate arguments for the application code; the core CLI reads in serial data input and then tokenizes the string data to form this data structure whose elements then get passed to the application program
```

Global Data:

```c
extern CLI_CMD_LIST
User software MUST define this array. The array could be of any size, but should be terminated by a NULL entry. For example:

CLI_CMD_LIST = {
    CONFIG_CMD_ENTRY,
    MOVE_CMD_ENTRY,
    SENSOR_CMD_ENTRY,
    NAV_CMD_ENTRY,
    MISC_CMD_ENTRY,
    CLI_NULL_ENTRY
};
```
const char cli_cmd_ok[] PROGMEM;  
const char cli_cmd_badindex[] PROGMEM;  
const char cli_cmd_badport[] PROGMEM;  
const char cli_cmd_badfreq[] PROGMEM;  
const char cli_cmd_badtype[] PROGMEM;  
const char cli_cmd_baddegrees[] PROGMEM;  
const char cli_cmd_badduty[] PROGMEM;  
const char cli_cmd_badenc[] PROGMEM;  
const char cli_cmd_badwheelsize[] PROGMEM;  
const char cli_cmd_badspeed[] PROGMEM;  
const char cli_cmd_baddirection[] PROGMEM;  
const char cli_cmd_badaddr[] PROGMEM;  
const char cli_cmd_badlength[] PROGMEM;  
const char cli_cmd_nodvice[] PROGMEM;  
const char cli_cmd_busfail[] PROGMEM;  
const char cli_cmd_sysfail[] PROGMEM;  

Constant strings used in application programs; all strings to be used when replying to user commands are defined here, all commands report back data as specified in the ICD

Usage: printf_P( cli_cmd_ok ); // print “OK” to console

Macros Defined:

**CLI_CMD_LIST_ELEMENT(element)**

Description: Used in application code to declare a new application to be inserted into the master command table

**CLI_CMD_ENTRY(i)**

Description: Used in application code to define a constant program string (i.e. to define the application name)

**CLI_BADMALLOC_PANIC(var)**

Description: Used after any malloc in application code, will freeze the system if a malloc is performed and fails, this is useful in locating memory leaks in application code
2.2.7.11 CLI Support Functions Module

Source File: clisup.c
Header File: clisup.h

Description: CLI support functions, these routines are rudimentary utilities that can be used by all commands to expand the capabilities of the CLI.

Directory: rp1
Layer: CLI

Functions Defined:

clisup_parse_hexdec
Definition:

    int32_t clisup_parse_hexdec( char * str, uint8_t * valid )

Description: Parse a hexadecimal or decimal string

Inputs:

    char * str
    String to process

Outputs:

    uint8_t * valid
    Valid will be set to TRUE if this procedure succeeds,
    FALSE if it fails, or is left alone if pointer is NULL

Return: The number extracted from the string, or 0 if invalid

clisup_isascii
Definition:

    bool_t clisup_isascii( char * str )

Description: Determine if a string of characters is a text readable string

Inputs:

    char * str
    String to process

Outputs: None

Return: TRUE if ascii, FALSE otherwise
clisup_isascii_char
Definition: bool_t clisup_isascii_char( char data )
Description: Determine if a character is a text readable character
(if it is an ascii string)
Inputs:
char data
Character to process
Outputs: None
Return: TRUE if ascii, FALSE if otherwise

2.2.7.12 Error Log Control Module
Source File: err_log.c
Header File: err_log.h
Description: This module controls the recording and reporting of error log
data accrued as the system runs from power-up.
Directory: rpos
Layer: Operating System

Macros Defined:

ESTOP_GET()
Description: Gets the status of the system-wide emergency stop
flag, set by the user

ESTOP_SET()
Description: Sets the emergency stop flag preventing motor
activity until the flag is cleared

ESTOP_CLR()
Description: Clears the emergency stop flag resuming normal
motor activity

IS_WARNING()
Description: Will return TRUE if the system has encountered
a warning (indicates data is in log list)

IS_CRITICAL()
Description: Will return TRUE if the system has encountered
a critical error, under this condition no motor
activity is permitted until the system is reset

Functions Defined:
errlog_busfail
Definition:
    void.errlog_busfail( uint8_t addr )
Description: Adds a TWI BUSFAIL error message to the log
Inputs:
    uint8_t addr
        Address of TWI device which could not be addressed
Outputs:
    None
Return: None

errlog_nodevice
Definition:
    void.errlog_nodevice( uint8_t addr )
Description: Adds a TWI NODEVICE error message to the log
Inputs:
    uint8_t addr
        Address of TWI device which could not be addressed
Outputs:
    None
Return: None

errlog_clear
Definition:
    void.errlog_clear( void )
Description: Clear the error log of all messages
Inputs:
    None
Outputs:
    None
Return: None
errlog_add
   Definition:
       void errlog_add( uint8_t err_lvl, uint8_t err_code, uint8_t len,
                        uint16_t * data )
   Description: Adds an error message to the log
   Inputs:
       uint8_t err_lvl
           Criticality level of error
       uint8_t err_code
           Error code, use defines in header
       uint8_t len
           Length of unsigned short integer array
       uint16_t * data
           An array of unsigned short integers of length 'len'
   Outputs:
       None
   Return: None

errlog_print
   Definition:
       void errlog_print( void )
   Description: Prints the error log to the screen
   Inputs: None
   Outputs: None
   Return: None

2.2.7.13 Robotic Platform Configuration Module
Source File: rp_conf.c
Header File: rp_conf.h
Description: This module defines and controls the RP1/10/100 configuration
data static memory system. All non-volatile configuration data
is stored and accessed via this interface.
Directory: rpos
Layer: Operating System
Typedef Data Defined:

```
typedef union rp_uid_t {
    uint16_t v;
    uint8_t c[2];
} rp_uid_t;
```

The robotic platform unique identifier (UID) can be expressed as a two byte character array or a unsigned 16-bit integer.

```
typedef struct rp_data_t {
    uint8_t drvport; // external port of drive motor (Port: 0, 1)
    uint8_t strport; // external port of steer motor (Port: 0, 1)
    uint16_t accel; // num milliseconds necessary to reach top speed
    int16_t drmin; // drive motor minimum PWM frequency (typically 0)
    uint16_t drmax; // drive motor maximum PWM frequency
    uint16_t stmin; // steer motor minimum PWM frequency
    uint16_t stmax; // steer motor maximum PWM frequency
    uint8_t drtype; // drive motor type (0 = None, 1 = DC, 2 = Stepper)
    uint8_t sttype; // steering motor type (0 = None, 1 = Servo, 2 = Stepper)
    uint16_t minduty; // minimum duty cycle of a servo steering motor (in uS)
    uint16_t maxduty; // maximum duty cycle of a servo steering motor (in uS)
    uint16_t enc; // number of encoder pulses per revolution of the wheel
    uint16_t whlsz; // circumference of the wheel after gear ratios have been calculated
} rp_data_t;
```

This structure encapsulates all pertinent data for a single robotic platform motor module as defined in the Interface Control Document (ICD). This data is stored in EEPROM memory and local system cache for quick access and writing. This data affects how motors are driven and how motor modules steer.

```
typedef struct rp_groups_t {
    uint8_t type; // motor group type (0 = Drive Only, 1 = Steering Only, 2 = Driving and Steering)
    uint8_t motors[RP_CONF_MAX_MOTORS_PER_GROUP]; // motors within group
} rp_groups_t;
```
The robotic platform has motor groupings or linkings which allow the user to swiftly change the speed or turning angle of multiple motor modules simultaneously. For instance, a user may create a drive motor group, a steering motor group, and a drive/steering motor group and issue commands to these groups to make a unique system navigation profile.

Functions Defined:

**rp_conf_init**
Definition:
```
void rp_conf_init( void )
```
Description: Initialize RP non-volatile memory for use
Inputs: None
Outputs: None
Return: None

**rp_conf_getuid**
Definition:
```
rp_uid_t rp_conf_getuid( bool_t * valid )
```
Description: Reads from non-volatile memory and returns the robot Unique ID
Inputs: None
Outputs:
```
bool_t * valid
TRUE if output is valid, FALSE otherwise
```
Return: The robotic platform UID

**rp_conf_setuid**
Definition:
```
bool_t rp_conf_setuid( rp_uid_t rpuid )
```
Description: Set a robotic platform Unique ID
Inputs:
```
rp_uid_t rpuid
The robotic platform unique id (can be long value, i.e. "(rp_uid_t) 0x00000005")
```
Outputs: None
Return: TRUE if configuration of UID is successful, FALSE otherwise
**rp_conf_get_confdata**

**Definition:**

```
bool_t rp_conf_get_confdata( uint8_t ndx, rp_data_t * rpdata )
```

**Description:**
Reads from non-volatile memory and returns the RP configuration data set

**Inputs:**
- `uint8_t ndx`: Motor module index

**Outputs:**
- `rp_data_t * data`: On successful read, contains the configuration data for an RP1/10/100 Motor Module

**Return:** TRUE if output is valid, FALSE otherwise

**rp_conf_set_confdata**

**Definition:**

```
bool_t rp_conf_set_confdata( uint8_t ndx, rp_data_t * rpdata )
```

**Description:**
Sets configuration data for a robotic platform

**Inputs:**
- `uint8_t ndx`: Motor module index
- `rp_data_t * rpdata`: Robotic Platform configuration data, if NULL data is deleted

**Outputs:** None

**Return:** TRUE on success, FALSE on an operating system failure

**rp_conf_get_grpdata**

**Definition:**

```
bool_t rp_conf_get_grpdata( uint8_t ndx, rp_groups_t * rpgrps )
```

**Description:**
Reads from non-volatile memory and returns the requested motor group data

**Inputs:**
- `uint8_t ndx`: Motor module index
- `rp_groups_t * rpgrps`: On successful read, contains the motor group type and array

**Outputs:** None

**Return:** TRUE if output is valid, FALSE otherwise
rp_conf_set_grpdata
Definition:
bool_t rp_conf_set_grpdata( uint8_t ndx, rp_groups_t * rpgrps )
Description: Sets configuration data for a robotic platform
Inputs:
uint8_t ndx
Motor module index
rp_groups_t * rpgrps
RP1/10/100 Motor Groups data, if NULL group ndx is deleted
Outputs: None
Return: TRUE on success, FALSE on an operating system failure

2.2.7.14 Proportional Integral Derivative (PID) Controller Module
Source File: pid_talk.c
Header File: pid_talk.h
Description: This module defines the communication protocol between the
microcontroller and PID units. Any command to the PID unit
should be made through function calls defined in this module.
This is to ensure safe and controlled communication to any
internal RP1 component.
Directory: rpos
Layer: Devices

Functions Defined:

pid_set_speed
Definition:
twirc_t pid_set_speed( uint8_t addr, uint8_t motor,
int16_t speed )
Description: Set the speed in centimeters per second that the
specified motor should lock onto
Inputs:
uint8_t addr
Address of PID controller
uint8_t motor
PID_MOTOR0 or PID_MOTOR1
int16_t speed
16-bit signed integer speed value in cm/s
Outputs: None
Return: Result of TWI operation
pid_get_speed

Definition:

```c
twirc_t pid_get_speed( uint8_t addr, uint8_t motor, 
    int16_t * prgmspeed, int16_t * curspeed )
```

Description: Sends a request to the PID controller to get the current speed of a motor. This speed value is an averaged speed calculated from encoder Cycles Per Revolution (CPR) and wheel circumference in centimeters.

Inputs:
- `uint8_t addr`
  - Address of PID controller
- `uint8_t motor`
  - PID_MOTOR0 or PID_MOTOR1

Outputs:
- `int16_t * prgmspeed`
  - The desired 16-bit signed integer speed value in cm/s
- `int16_t * curspeed`
  - The current 16-bit signed integer speed value in cm/s

Return: Result of TWI operation

pid_set_dist

Definition:

```c
twirc_t pid_set_dist( uint8_t addr, uint8_t motor, int16_t dist )
```

Description: Set the distance motor should travel before stopping. This does NOT alter the speed of the motor. User should set distance to travel and THEN set motor speed using pid_set_speed to move the motor a specific distance given a specific speed. If the distance to travel is set to 0x0000 then the motor will turn indefinitely until a new speed or distance is set.

Inputs:
- `uint8_t addr`
  - Address of PID controller
- `uint8_t motor`
  - PID_MOTOR0 or PID_MOTOR1
- `int16_t dist`
  - 16-bit distance to travel in centimeters (positive is fwd)

Outputs: None

Return: Result of TWI operation
**pid_get_dist**

**Definition:**
```
twirc_t pid_get_dist( uint8_t addr, uint8_t motor,
                      uint16_t * dist )
```

**Description:** Returns the remaining distance to travel as calculated by the PID controller. If this function returns 0x0000 that only means that the motor has traveled its programmed distance or that no distance limit is set.

**Inputs:**
- `addr`: Address of PID controller
- `motor`: PID_MOTOR0 or PID_MOTOR1

**Outputs:**
- `dist`: 16-bit distance to travel in centimeters (positive is fwd)

**Return:** Result of TWI operation

**pid_set_angle**

**Definition:**
```
twirc_t pid_set_angle( uint8_t addr, uint8_t servo,
                       uint8_t angle )
```

**Description:** Sets a servo to rotate to a specified angle. The angle value may be any unsigned value between 0 and 180. Values over 180 will be capped to 180. An angle of 0 degrees represents a 90 degree counterclockwise rotation and an angle of 180 degrees represents a 90 degree clockwise rotation from center.

**Inputs:**
- `addr`: Address of PID controller
- `servo`: PID_SERVO0 or PID_SERVO1
- `angle`: 8-bit unsigned integer angle value

**Outputs:** None

**Return:** Result of TWI operation
pid_get_angle

Definition:

    twirc_t pid_get_angle( uint8_t addr, uint8_t servo,
                           uint8_t * angle )

Description: Returns the current programmed angle of the servo motor. The reported value may be any angle between 0 and 180. An angle of 0 degrees represents a 90 degree counterclockwise rotation and an angle of 180 degrees represents a 90 degree clockwise rotation from center.

Inputs: None

    uint8_t addr
             Address of PID controller
    uint8_t motor
             PID_SERVO0 or PID_SERVO1

Outputs:

    uint8_t * angle
             8-bit unsigned integer angle value

Return: Result of TWI operation
pid_configure

Definition:
```
twirc_t pid_configure( uint8_t addr, uint8_t motor, 
                      uint16_t whlcirc, uint16_t enccpr )
```

Description: Configures the wheel circumference and encoder cycles-per-revolution of a single drive motor in the PID controller. This must be done before any move operations are commanded on the bus. This must be done after the PID controller comes out of reset as the configuration state is volatile.

Inputs:
- `uint8_t addr`: Address of PID controller
- `uint8_t motor`: PID_MOTOR0 or PID_MOTOR1
- `uint16_t whlcirc`: Circumference of wheel in centimeters
- `uint16_t enccpr`: Number of Cycles Per Revolution (CPR) on drive motor encoder

Outputs: None
Return: Result of TWI operation

2.2.7.15 Motor Module Control Module

Source File: motor_ctrl.c
Header File: motor_ctrl.h

Description: This module defines the interface between the PID communication protocol and the user interface. Methods to move motors, motor groups, and more intuitive motor module functionality is placed here. This module bridges the gap between the "motor module" and individual motors which are addressable by a PID controller unit.

Directory: rpos
Layer: Devices

Functions Defined:
motor_config
Definition:
    void motor_config( void )
Description: Configures all PID controllers with internally stored motor module configuration data. Pulls information from non-volatile EEPROM and configures all PID controllers over the serial TWI bus.
Inputs: None
Outputs: None
Return: None

motor_start
Definition:
    void motor_start( uint8_t ndx )
Description: Enables a motor by turning on power to the motor power supply. DC motor drivers should be supplied with an ACTIVE-LOW enable pin, meaning when the enable pin is 0V the device should be active, when it is 5V or floating the device should shut off.
Inputs:
    uint8_t ndx
    Motor module index
Outputs: None
Return: None

motor_stop
Definition:
    void motor_stop( uint8_t ndx )
Description: Disables a motor by cutting off power to the motor power supply. DC motor drivers should be supplied with an ACTIVE-LOW enable pin, meaning when the enable pin is 0V the device should be active, when it is 5V or floating the device should shut off.
Inputs:
    uint8_t ndx
    Motor module index
Outputs: None
Return: None
motor_drive

Definition:

\[
\text{twirc_t motor_drive( uint8_t mmndx, int16_t speed, int16_t dist )}
\]

Description: Actuate a motor module's drive motor if applicable. If no drive motor exists for this motor module, function shall return TWI_RC_NO_DATA.

Inputs:

- \text{uint8_t mmndx}
  - Index of motor module (defined in RP1 ICD)
- \text{int16_t speed}
  - The speed of the drive motor in centimeters per second
- \text{int16_t dist}
  - The distance to travel before motor stops, or 0 for indefinite

Outputs: None

Return:

- TWI_RC_OK on success
- TWI_RC_NO_DATA if motor module has no drive motor or is not configured
- TWI_RC_FAIL or other error code on failure

motor_steer

Definition:

\[
\text{twirc_t motor_steer( uint8_t mmndx, uint8_t degrees )}
\]

Description: Actuate a motor module's steering motor if applicable. If no steering motor exists for this motor module, function shall return TWI_RC_NO_DATA.

Inputs:

- \text{uint8_t mmndx}
  - Index of motor module (defined in RP1 ICD)
- \text{uint8_t degrees}
  - The degrees of rotation (0 to 180, 90 being center)

Outputs: None

Return:

- TWI_RC_OK on success
- TWI_RC_NO_DATA if motor module has no drive motor or is not configured
- TWI_RC_FAIL or other error code on failure
motor_status_get

Definition:

twirc_t motor_status_get( uint8_t mmndx, int16_t * prgmspeed,
                          int16_t * curspeed, uint8_t * angle )

Description: Gets the current status of a motor module's drive and
steering motors.

Inputs:

  uint8_t mmndx
    Index of motor module (defined in RP1 ICD)

Outputs:

  int16_t * prgmspeed
    Programmed speed of drive motor
  int16_t * curspeed
    Current speed of drive motor
  uint8_t * angle
    Steering angle currently programmed and in use

Return:

  TWI_RC_OK on success
  TWI_RC_FAIL or other error code on failure

2.3 Client Software

The Client Software is synonymous with the Graphical User Interface (GUI) software. The GUI software, mentioned in section 1.3 above is the canonical method of communicating with the robotic platform. The user may use a terminal emulator to connect to the platform and send commands, but this is much slower and not as robust.

2.3.1 Software High Level Design

The RP1 GUI software is broken down into two core sections, the RPCTRL and the RPGUI. The RPCTRL block represents the RP1 Control block and contains all utilities for sending commands to the RP unit, reading back response data, controlling the flow of data between the GUI and the platform, data containers for quick processing and storage of data to and from the platform, and so forth. The RPGUI block represents the RP1 View block and is the graphical interface code of the application. This block provides all the window, panel, and dialog code and configuration data for the efficient display of RP control information.

2.3.2 Repository Directory Structure

The software repository for the client GUI interface JAVA application is located in subdirectory 'software\client user interface' of the P09204 ‘Public’ directory. Within this directory are the project files and directories for a NetBeans 6.5, JAVA 6, SingleFrameApplication project. build.xml and manifest.mf are project files used by the NetBeans IDE, and rxtx-2.1.7-bins-r2.zip is the archive file containing the necessary serial communications libraries this application needs to
communicate over RS232 channels. The contents of this archive have already been copied to the necessary folders and this archive is present merely for convenience.

Of the subdirectories, lib contains all additional libraries this project requires. The contents of this directory are Windows Dynamic Linked Libraries (DLL’s), Linux object files, or MAC OSX object files and any JAVA packages.

The nbproject subdirectory contains additional project configuration data.

The rpscripts subdirectory contains some example scripts which the GUI application can execute. These scripts are a byte for byte record of commands sent over the RS232 interface and can help expedite repetitive or mundane tasks (i.e. initializing a newly programmed RP1 microcontroller).

The src subdirectory contains all the project source code and will be described in further detail. Within this folder is any project media content (i.e. icons, images, form properties, etc.) as well as project source code. Subdirectory icons within src contain all the icon images used for this application. These include the application status icons, the directional key buttons of the navigation panel, health status images, and more. The subdirectory images within src contain larger images, such as the reference image of the RP1 platform layout (seen in the motor module control panel).

The rpctrl subdirectory of the src directory contains all JAVA source code pertaining to the communication interface between the GUI application and the RP1 platform. There is no graphical code within this subset of the application, it consists entirely of control code.

The rpgui subdirectory of the src directory contains all JAVA source code pertaining to the graphical windows, panels, and dialogs of the GUI application. Where rpctrl code is the control block of the application, rpgui code is the view/interface.
2.3.3 RPCTRL

Below is a list of source files within the \texttt{rpctrl} source directory and their purpose:

- \texttt{CommEvent.java}
  - Communication event generated by the \texttt{SerialDeviceController} and reported to all \texttt{CommEventHandlers} registered as an event listener.
- \texttt{CommEventHandler.java}
  - Graphical interface code implements this interface to receive events from the \texttt{SerialDeviceController}.
- \texttt{ConfigData.java}
  - Configuration data used by the \texttt{ConfigMngr} class.
- \texttt{ConfigMngr.java}
  - This static class handles all reading and writing of GUI configuration data. This class generates the “config” file located in the root of the project execution directory. Any saved data between executions of the application is passed over through this class.
- \texttt{MotorModuleGroup.java}
  - Data container class to store all pertinent data of a motor module group. Specified in further detail within the ICD (Section 4).
- \texttt{MotorModuleParams.java}
  - Data container class to store all pertinent data of a motor module. Specified in further detail within the ICD (Section 4).
- \texttt{RpCommandParser.java}
Robotic Platform 1kg
Software Control Document

- This is the only interface between the GUI and the serial communications link which the `rpgui` code uses to send commands and read back data from the RP1 platform. All command processing and communication is contained within this control code. New commands can be implemented here, and old commands may be modified.

  - **SerialDeviceController.java**
    - This is a wrapper class for the interface and method calls of the JAVA implementation RXTX libraries. Contains all methods to open, close, and manipulate a serial communication channel and internal classes for reading and processing incoming serial data.

2.3.4 **RPGUI**

Below is a list of source files within the `rpgui` source directory and their purpose:

- **MotorModuleStatus.java**
  - Data container class to store any pertinent motor module status information. This code is contained with the `rpgui` directory rather than the `rpctrl` because of its proximity to the display of data rather than the storage and manipulation of data.

- **RpClientGui.java**
  - This class contains the `main()` method of the application and simply instantiates the `RpClientGuiView` object.

- **RpClientGuiAboutBox.java**
  - This class is the dialog box displayed when the user clicks the **About** menu item.

- **RpClientGuiConfigView.java**
  - This class is the configuration window. The configuration window contains all utilities to setup the serial communication interface and configure the platform for use.

- **RpClientGuiConsoleView.java**
  - This class is the console text area window to view and control RS232 traffic between the GUI and the platform.

- **RpClientGuiHealthView.java**
  - This class is the health window which displays RP1 system health and provides utilities to view and manipulate system health status codes and error logs.

- **RpClientGuiMotorModuleStatusView.java**
  - This class is the status window for displaying the individual motor module activity on a per module basis. This class utilizes two JLists using the `RpMotorModuleCellRenderer` class to render MotorModuleStatus data for the “left” and “right” sides of the robot. Four motor modules are currently instantiated at runtime, but more motor modules can be added at a later date.

- **RpClientGuiView.java**
  - This is the main window of the

- **RpMotorModuleCellRenderer.java**
  - This class overrides the default cell renderer for a JList or a JTree and renders MotorModuleStatus data of a list or tree in a graphically appealing fashion. The motor module status data is stored in array form so that adding additional motor modules (more than four) is easy and simple.
2.3.5 Compilation Procedure

The compilation procedure relies on the use of NetBeans 6.5 or higher and the JAVA SDK version 6 or higher. NetBeans can be downloaded here: [http://www.netbeans.org/](http://www.netbeans.org/).

To compile, run, and debug the project download the project source from the SVN directory and the NetBeans 6.5 (or higher) binary from their website. The SVN repository is located here: [https://edge.rit.edu/dav/P09204/web/public/software/client user interface](https://edge.rit.edu/dav/P09204/web/public/software/client user interface).

Once checked out, install and run the NetBeans IDE. Install all updates before proceeding. Open the project checked out from the subversion repository. Click **Run->Run Main Project (F6)** to execute the application. Click **Debug->Debug Main Project (Ctrl+F5)** to debug the application, step through code, monitor thread activity, etc. See Figure 5 for a screencapture of the NetBeans IDE.

Click **Run->Clean and Build Main Project (Shift+F11)** to build new binaries and package the code in a JAVA .jar file. When the code is packaged in a .jar file it is located in a subdirectory called **dist** within the project root directory. See Figure 6 for a screencapture of the distribution (dist) directory.
2.3.6 Usage Instructions

There are precompiled and packaged RPGUI binaries at: 
https://edge.rit.edu/dav/P09204/web/public/software/latest_builds/client user interface. Download the latest version and unpack it to a local directory.

Within the archive are two directories and a handful of files. The archive contents are displayed in Figure 7. The file named ‘client user interface.jar’ is the RPGUI JAVA package. The ‘config’ file is the configuration file produced by the GUI when the application is closed and read when the application begins. The ‘config’ file contains window parameters and serial channel configuration data. The ‘rungui-...’ files are execution scripts for various operating system platforms.

If you are running windows double click or run the ‘rungui-windows.bat’ to run the project.

If you are running Linux open a shell prompt and navigate to the directory of the ‘rungui-linux.sh’ script. At the prompt type “.\rungui-linux.sh” without the quotes and hit Enter.

Please read the ‘README.TXT’ for more details.
2.3.7 Graphical User Interface Walkthrough

The RP Graphical User Interface (GUI) comprises multiple tabbed panels and windows to configure and control the RP unit. The intention of the GUI is to develop more advanced control schemes for the RP unit which the small microcontroller cannot otherwise provide. It is also intended to provide the full range of control offered by the platform in a simple, elegant, and graphical interface. Lastly, it abstracts the platform implementation and size from the control, therefore enabling this GUI to be used by RP1, RP10, RP100 and other platform variants so long as they all implement the same communication protocol defined in the Interface Control Document (ICD, see Section 4).

2.3.7.1 Navigation Panel

The Navigation Panel is the first panel displayed when the GUI is opened. A screenshot of this panel can be seen in Figure 8. Immediately seen is the menu bar at the top of the main window, the panel of tabs labeled “Robot Navigation”, “Motor Module Control”, and “External Device Rx/Tx”. The Navigation Panel is the first tab, labeled “Robot Navigation”. The Motor Module Control Panel (Section 2.3.7.3) is the tab labeled “Motor Module Control”. Lastly, the TWI/I^2C Communication Interface Panel (Section 2.3.7.4) is the tab labeled “External Device Rx/Tx”.

Below this, within the Navigation Panel itself, are four subpanels. The first located in the upper left is a visual display of the currently selected speed setting and angular turn. The top most slider represents the FWD/REV throttle. The slider below represents the angular turn from center. The checkbox selects the mode of navigation. When checked, left clicking (and holding) the Up, Down, Left, or Right arrows or pressing ‘W’, ‘S’, ‘A’, ‘D’ (respectively) will cause the robot to move full throttle forwards, backwards, 75 degrees counterclockwise, or clockwise (respectively). When the key is released or the user releases the mouse button then the robot will send a new command to return the robot to neutral (0 speed, centered).

When the checkbox is unchecked, then pressing ‘W’, ‘S’, ‘A’, ‘D’ or clicking the arrow buttons will increase/decrease speed or angular turn incrementally. This means that the user need not press and hold the button or keys and can gradually increase or decrease angular turn and speed. In this mode of navigation, pressing the STOP button or pressing the ‘X’ key will bring the robot to a full stop. Pressing the ‘C’ key will recenter the motor modules.

The panel below the speed and angular turn display is the panel to select the drive and steer motor module groups. A motor module group is a configurable collection of motor modules which will receive the same command to move and steer in one ASCII command from the GUI. For example, if the user configures the platform’s motor group 0 to hold motor modules 0 and 1 and configures the group to accept commands for both drive and steer then sending a movegroup command for group 0 will set the speed and angular turn of both motor modules simultaneously. In this manner, the platform can be configured to have separate groupings of drive modules, steering modules, or “omni” (both drive and steer) modules.
The panel below the “Separate Drive and Steer” panel is the “Linked Drive and Steer” panel. If the user has configured the platform such that all motor modules in a group drive and steer simultaneously then this configuration should be used (i.e. forward wheel drive car). Otherwise, if the user has configured the platform to have one motor group reserved for steering and one for drive then the “Separate Drive and Steer” panel should be used (i.e. rear wheel drive car).

Lastly, the text field labeled “Top Speed: (cm/s)” is necessary for the GUI to calculate the maximum possible speed to command the platform to move. This should be found through experimentation, but for release 1 of the RP1 hardware the top speed was found to be approximately 50cm/s.

![Figure 8 - GUI Navigation Panel](image-url)
In the status panel at the bottom of the main window is a textual description of the activity status of the system (i.e. “Offline”, “Online”, “Busy”, “Error”). Next to the text field is a graphical, animated icon visually representing the status of the system. The various pictorial status messages are described below in detail.

When the system is ready for use, but not yet connected to a serial channel, Mog will stand still ( ). When connected, Mog begins to walk ( ). When the system is busy, Mog will wave his arms ( ). When the system encounters a recoverable error, Mog becomes surprised ( ). Lastly, when the system encounters a non-recoverable error, Mog will faint ( ) indicating that the GUI needs to be shut down and restarted before communication may continue.

### 2.3.7.2 Script Processing and Recording

![Figure 9 - Script Control Menu Items](image)

The GUI has the capability of running prerecorded command scripts, hand-made command scripts, or to record scripts on the fly. To run a saved script click **Edit->Run Command Script...** and select the script in the Open File dialog box that appears. If an error appears instead this means the system does not have that specific dialog box and scripts cannot be run from that machine.

The command script will be processed line by line, each line of the script file will be sent to the microcontroller as commands. Unrecognized commands will be ignored. Commands starting with “#” will be recognized as script comments and will not be sent to the microcontroller.

There are two special commands, **PAUSE** and **END**. **PAUSE** will stop script parsing for a specified number of milliseconds (i.e. “pause 1000” will stop script processing for 1 second). **END** will cause script parsing to stop, and all commands following will not be processed.

An example script is provided here:

```
# Move forward
movegroup 0 255 90
```

62 of 82
pause 1000

# Full stop
movegroup 0 0 90

# Script end
end

# Nothing down here will be processed
movegroup 0 -255 90

This script will command the motorgroup 0 to move forward at 255 cm/s for 1 second and then stop.
Script parsing ends when “end” is processed so the next command to move in reverse is never processed.

To record a script, click Edit->Record Script… and select a filename and directory to save the new script. When “Save” is clicked the script will start recording. Any action performed in the GUI, i.e. configuring the device ID, commanding movement, etc. will be recorded in the script. Pauses between commands will not be recorded however.

When the user is finished recording, click Edit->Stop Recording and the script will save and recording will stop. If the user wishes to pause recording to perform a set of actions that is not intended to be in the script, he may click Edit->Pause Recording and perform these actions. To resume recording simply click the menu item again.

2.3.7.3 Motor Module Control Panel
The Navigation Panel provides a means to easily control the free motion of the platform as a whole. This panel provides a diagnostician’s means of commanding individual motor modules to move a specific speed or angular turn. This panel is displayed in Figure 10. In addition the user may command motor groups for specific speeds and turns. This panel was not meant to offer a convenient way to navigate the robot but rather an expert control to diagnose errors and possibly improve efficiency. For example, when fine tuning PID parameters for DC motor control feedback this panel is invaluable for that effort.

The “Motor Module” and “Motor Group” radio buttons and associated combo boxes serve to select particular motor modules or groups to command. Once selected the speed in cm/s may be set along with the degree angular turn in the text boxes below. Further below this are the buttons to program the motor module or group speed and angular turn and a button to bring the robot into full stop. For convenience to the user there is a pictorial reference of the RP1 system chassis view and respective motor module indexes. This is a “birds-eye-view” graphical representation of the orientation of motor modules in the platform chassis.
The RP unit has an internal TWI/I²C serial data bus which can allow the microcontroller to communicate between devices internal and external to the control system. The only current device internal to the RP system is the PID controller whose device addresses are defined in the ICD (Section 4).

This document will not describe the functionality or behavior of the TWI/I²C data bus but will give a cursory description of how to use this GUI panel to communicate over the serial data bus.
The text field visible at the top of the panel labeled “Device ID” is the 7-bit slave device ID for which to communicate. This value is represented as a decimal value, by prepending a “0x” to this value it will be interpreted by the platform as a hexadecimal value.

Below this is the immutable “TWI Communication Log” which will accumulate messages as data is written to or read from the slave device.

The text field and slider below the communication log are the pertinent controls for writing to or reading from the slave device. The text field labeled “TWI Write Bytes” contains byte data to be sent to the slave device interpreted as decimal values unless prepended by “0x”. For example, writing “00 01 2 0xa 55” will write 5 bytes to the slave device interpreted as “0 1 2 10 55” in decimal. When the user presses the button labeled “Write” the data in the “TWI Write Bytes” text field is interpreted by the microcontroller and sent to the slave device on the data bus. If the device is not present then an error message will appear in the log, otherwise a description of what data was sent over the bus is logged in the communication log.

The slider labeled “TWI Read Bytes” selects the number of bytes which the microcontroller should read from the slave device. This performs a single read operation for the number of bytes specified. Upon a successful read the data will appear in the communication log, otherwise a communication error will be logged.
Figure 11 - GUI TWI/I^C Communication Panel
2.3.7.5 Configuration Window

The configuration window contains four panels. The tabs are labeled “Serial Communication”, “RP System”, “RP Motor Modules”, and “RP Motor Groups”. Each panel will be addressed separately below.

The “Serial Communication” panel, displayed in Figure 13, provides the user with every configuration parameter necessary to communicate with the RP unit over RS232 serial. The configurable items match what is provided with the de-facto standard Windows HyperTerminal application. The communication parameters for the RP platform are described in detail in the ICD (Section 4).
The “RP System” tab seen in Figure 14 is a configuration panel reserved for system specific configurable data. Each RP unit is programmable with a unique ID. This is useful if a company or work group has multiple robotic platforms which require unique identification or separation. To configure the RP UID, type in a decimal value (or prepend “0x” to specify hexadecimal) in the “Robotic Platform UID” textfield and press the “Send Configuration to RP Unit” button. Press the “Receive Configuration from RP Unit” button to retrieve the RP UID from the microcontroller static memory. Pressing this button will also retrieve the version of Operational Software on the microcontroller.
The “RP Motor Modules” panel seen in Figure 15 allows the user to configure each motor module connected to the platform. Each motor module can have a drive motor, a steer motor, both, or neither. If a motor module has neither drive nor steer motors then it is considered an “Idler” and should be deleted from the list. Otherwise the module may be configured below.

The full list of modules can be retrieved from the robotic platform static memory (EEPROM) by pressing the “Retrieve list from RP Unit”. When a motor module is programmed or received from the platform then the module will appear in the “Modules” combo box. A motor module may be deleted, or removed from the system, by selecting it in the combo box and pressing “Delete Motor Module Entry”.

Figure 14 - GUI Configuration Window System Panel
Below these controls is a panel for all configurable items of a motor module. Included is the motor module index (See ICD, Section 4), the drive and steering motor ports, and the operating parameters for both motors. Once selected, pressing “Save Motor Module Config” will store the motor module configuration information to the platform. By pressing “Restore Motor Module Config” any local changes will be overwritten and the textfields and combo boxes will be reset to whatever data is currently stored on the platform.

The “RP Motor Groups” panel, shown in Figure 16, allows the user to link one or more motor modules together in a group. A maximum of 16 groups can be defined as specified in the ICD (Section 4). Similar
to the “RP Motor Modules” the groups can be pooled from the platform by pressing “Retrieve List from RP Unit” and deleted by pressing “Delete Motor Module Group”.

Furthermore, the group type can be selected as “Drive”, “Steer”, or “Omni”. A “Drive” group is a collection of motor modules whose purpose in this group configuration is only to drive. A “Steer” group, similarly, is a collection of motor modules whose purpose is only to steer. If a group is defined as a “Steer” group yet it contains motor modules which have drive motors, these drive motors will be ignored and only the steering motors will be commanded.

Lastly, the text field labeled “RP Motor Modules” is a space delimited list of motor module indexes which the group will store. For example, specifying “0 1 2 3” will select all four motor modules of the platform, or “0 1” will specify only the two forward motor modules, excluding the two modules in the rear of the platform.
2.3.7.6 Motor Module Status Window

This field is a space-separated list of item items, each item being a motor module index, for example: "0 1 3" would link motors 0, 1, and 3 together in the motor group named by the 'RP Group Index' selection.
The Motor Module Status Window is a panel useful for displaying the current speed and programmed angular turn of each motor module. When “Update All Modules” is pressed, all motor modules will be selected for status updates. The microcontroller will attempt to provide information for all motor modules, and those not attached to the system will be marked as not present.

By pressing “Updated Selected Modules” all motor modules whose “Update” checkboxes are checked will be polled for status data. Please reference Figure 18.

By providing an interval in seconds in the “Interval (seconds)” text field and checking the “Auto-Update” checkbox the motor module status will automatically update in this interval without further user intervention. This allows the user to use the navigation panel to move the robotic platform and receive continuous status updates from all selected motor modules simultaneously.

![Figure 18 - GUI Motor Module Status Window](image)

### 2.3.7.7 System Health Window

The robotic platform comes with a built in error log manager. Each time an error is encountered in the system it is logged in the error manager in the microcontroller. The GUI may poll the system to retrieve
all the error logs and the overall system status. The system may be in one of three states: OK, MINOR, MAJOR with an optional ESTOP flag set. When the system encounters a trivial error or a burp in communications the status is set to MINOR and an error message is logged with timestamps. If a critical error is encountered that cannot be recovered from until the system is shut down for repairs the status is set to MAJOR. If for any reason the user triggers an EMERGENCY STOP, the ESTOP flag is set and no motor control commands will be sent from the microcontroller until this is lifted. Similarly, when the status is MAJOR then no commands will be set until the system is repaired and reset.

![Figure 20 - GUI System Health Window](image)

The thumbs up icon, ![ok](image), represents an OK status. The thumbs down icon, ![minor](image), represents a MINOR or MAJOR error status. MINOR errors can be cleared by pressing “Clear Error Log”. However, MAJOR errors will not be cleared until the system is power cycled. Pressing the “Check System Health” button will update the GUI to the current microcontroller system health status.
2.3.7.8 Console Text Viewer Window

The last window the GUI offers is the Console Text Viewer Window seen in Figure 22. All RP commands are defined in the ICD (Section 4) as ASCII commands sent over a terminal session. Therefore, these commands can be read by the user and if the user so wishes the command terminal can be directly accessed by user input, bypassing all GUI controls.

The Console Viewer window provides a HyperTerminal clone internal to the RP GUI. The viewer fully emulates a standard terminal client and is active immediately upon connection to the serial channel. The viewer disables user access when the GUI is executing a command, but will re-enable access when the system is not busy.

This panel is very useful for diagnostic purposes or to monitor communication activity manually. It is exceptionally useful for education purposes for those trying to familiarize themselves with the command protocol and the various uses of the terminal client.
2.3.7.9 About Dialog
The About dialog shows a brief description of the RP system as well as the version of the command protocol which this GUI adheres to. The Operational Software version (mentioned in Section 2.3.7.5) must match the Command Protocol Version specified in this dialog or communication between the GUI and microcontroller will most likely fail. Details on the Command Protocol Versions are specified in the ICD (Section 4).

Figure 22 - GUI Console Viewer Window

Figure 23 - GUI About Dialog
2.4 PID Controller

2.4.1 Software High Level Design
The Proportional-Integral-Derivative (PID) controller is the mediator between the OpSoft and the Motor Modules. It directly controls the servos, dc motors and encoder feedback for each motor module. Each PID controller can manage two motor modules with drive and steer capabilities, or a total of two dc motors and two servos.

The PID controller is separated into four different classes: the PID Controller/Bus Manager, the Message Handler and the Motor and Servo Controller, and the PID Engine.

2.4.1.1 PID Controller/TWI Bus Manager
The main class in the PID code structure is the bus handler. This class initializes runs and monitors the TWI (an I2C bus) on the PID controller. Because this is the main project file in the Arduino environment, interrupts for the encoder feedback ticks must be set up in this class and the passed down to the motor and servo controller for processing. It also contains a special loop function that is used to repeatedly poll the PID controller for updated output values.

Once the TWI has been initialized as a slave device on the bus, it waits for any incoming data that is addressed to it. All data from the microcontroller is formatted in a particular way that the message handler processes into meaningful commands. Once a full message has been received, it is executed. Some commands require the PID controller to return information back to the microcontroller. These commands place the wanted data in a special array that is sent back to the microcontroller whenever the next TWI read command is sent.

2.4.1.2 Message Handler
The message handler parses the incoming data stream from the TWI. The first byte received from a new TWI write command is assumed to be the message id. Based on the message id, the message handler will determine the size of the payload, or how many bytes to read after the message id is received.

2.4.1.3 Motor and Servo Controller
Once a full message is received and executed, the message handler will call on particular methods in the motor and servo controller to do the actual communication with the devices. The motor and servo controller is able to generate PWM signals on the analog pins of the Arduino Nano to drive DC motors, as well as pulses to turn the servos.

Encoder feedback is repeatedly polled by the TWI bus manager, and this information is trickled down through the message handler to the motor and servo controller. Each tick is placed in a circular queue that keeps a running average of how fast the ticks are being generated. This is used to determine how far the robot has traveled and how fast it is moving.
2.4.1.4 PID Engine

The PID engine is the actual intelligence behind all speed smoothing operations of the robot. A desired speed and a current speed are fed into the engine, which takes these values and determines how quickly the robot should attempt to reach its desired speed. This directly depends on how “aggressive” the PID engine is programmed to be.

These parameters can be easily tweaked in the source code of the Motor and Servo Controller class. In the class constructor, there are two commands set to initialize the PID engine for each motor. The ‘ConstructorCommon’ method is called in the PID Engine class with six arguments. The first three arguments are the input (current value), output, and set point (desired value). The final three arguments are the Proportional, Integral and, Derivative values. For more information on how to affectively tweak these values, the following page provides a thorough explanation: [http://en.wikipedia.org/wiki/PID_controller](http://en.wikipedia.org/wiki/PID_controller). If one simply wants to make the PID controller more aggressive or passive in its speed corrections, only the Proportional variable must be changed. Larger values for the Proportional variable make the engine more aggressive, and smaller values make it more passive.

2.4.2 Repository Directory Structure

The operational software repository is located in the ‘software/pid_controller/pid_controller’ subdirectory within the public directory of the P09204 team document repository. The ‘pid_controller’ directory contains all compilable source code, including the project file (pde) needed by the Arduino IDE.

2.4.3 Compilation and Programming Instructions using the Arduino IDE

The latest version of the Arduino software environment can be found here: [http://arduino.cc/en/Main/Software](http://arduino.cc/en/Main/Software)

Once you have downloaded the latest version of the PID controller code from the P9204 website ([https://edge.rit.edu/content/P09204/public/software/latest_builds/pid%20controller/](https://edge.rit.edu/content/P09204/public/software/latest_builds/pid%20controller/)); you may extract the entire contents of the zip into a directory of choice. The ‘pid_controller’ folder contains the project file and all class files associated with the project.

The Arduino IDE does not need to be formally installed on a pc, and can simply be unzipped to a directory and run. Once unzipped, run the ‘arduino.exe’ executable to launch the IDE. Once opened, select File>Sketchbook>Open (or Ctrl+O) to open a sketchbook (a project file). Select the previously downloaded ‘pid_controller.pde’ file in the ‘pid_controller’ folder to open the sketchbook project. You will see the following screen:
The Arduino IDE uses a multiple-tabbed interface with a centered code view and compiler output below. The PID controller code is written entirely in a special subset of C/C++ used by all Arduino products. Makefiles are created and managed by the IDE, and there is a sizeable amount of Arduino library code available. More information about the Arduino C/C++ programming language can be found at their website: [http://arduino.cc/en/Reference/HomePage](http://arduino.cc/en/Reference/HomePage). Thorough documentation on all included library functions, some of which are used in the PID controller, can be found here: [http://arduino.cc/en/Reference/Libraries](http://arduino.cc/en/Reference/Libraries).

Before programming, ensure that Arduino Nano is selected under Tools>Board. Also, be sure to select a valid COM port under Tools>Serial Port.

Once all parameters are properly set and the code is ready to be compiled and uploaded, click the “Upload to I/O Board” button to both compile and upload the current sketchbook to the Arduino Nano.

To toggle debug messages on or off, simply open ‘pid_controller.h’ and uncomment the ‘#define DEBUG’ line. Debug messages are sent over the serial port of the Arduino Nano and can be read by a program such as HyperTerminal. The Arduino IDE also contains its own internal serial monitor, which
can be launched by clicking the ‘Serial Monitor’ button ( ), which will begin displaying serial messages in the bottom of the window where compiler output is usually shown. The serial monitor must be turned off each time the Arduino Nano is programmed.

### 2.4.4 Command Set between Microcontroller and PID Controller

The PID controller and microcontroller communicate over the I\(^2\)C (TWI) bus using a custom protocol. Transmissions over the TWI bus are a binary encoded byte stream with several messages in a single transmission. Each message consists of a unique Message ID (MID) and a one or two byte payload. The protocol is described below.

<table>
<thead>
<tr>
<th>MID (hex)</th>
<th>Message Name</th>
<th>Payload Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Set Speed Motor 1</td>
<td>2</td>
<td>Sets the drive speed of motor 1 in cm/s.</td>
</tr>
<tr>
<td>11</td>
<td>Set Speed Motor 2</td>
<td>2</td>
<td>Sets the drive speed of motor 2 in cm/s.</td>
</tr>
<tr>
<td>12</td>
<td>Set Distance Motor 1</td>
<td>2</td>
<td>Sets the desired distance for motor 1 to travel in cm.</td>
</tr>
<tr>
<td>13</td>
<td>Set Distance Motor 2</td>
<td>2</td>
<td>Sets the desired distance for motor 2 to travel in cm.</td>
</tr>
<tr>
<td>20</td>
<td>Request Speed Motor 1</td>
<td>0</td>
<td>Returns the current and desired (set) speed of motor 1.</td>
</tr>
<tr>
<td>21</td>
<td>Request Speed Motor 2</td>
<td>0</td>
<td>Returns the current and desired (set) speed of motor 2.</td>
</tr>
<tr>
<td>22</td>
<td>Request Distance Motor 1</td>
<td>0</td>
<td>Returns the current and desired (set) distance of motor 1. The upper two bytes contain the current speed and the lower two contain the desired speed.</td>
</tr>
<tr>
<td>23</td>
<td>Request Distance Motor 2</td>
<td>0</td>
<td>Returns the current and desired (set) distance of motor 2. The upper two bytes contain the current speed and the lower two contain the desired speed.</td>
</tr>
<tr>
<td>30</td>
<td>Set Angle Servo 1</td>
<td>1</td>
<td>Sets the angle of servo 1.</td>
</tr>
<tr>
<td>31</td>
<td>Set Angle Servo 2</td>
<td>1</td>
<td>Sets the angle of servo 2.</td>
</tr>
</tbody>
</table>
### Robotic Platform 1kg
#### Software Control Document

<table>
<thead>
<tr>
<th>MID (hex)</th>
<th>Message Name</th>
<th>Payload Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Request Angle Servo 1</td>
<td>0</td>
<td>Returns the current angle of servo 1.</td>
</tr>
<tr>
<td>41</td>
<td>Request Angle Servo 2</td>
<td>0</td>
<td>Returns the current angle of servo 2.</td>
</tr>
<tr>
<td>50</td>
<td>Set Wheel Size Wheel 1</td>
<td>2</td>
<td>Sets the wheel size of wheel 1 in cm.</td>
</tr>
<tr>
<td>51</td>
<td>Set Wheel Size Wheel 2</td>
<td>2</td>
<td>Sets the wheel size of wheel 2 in cm.</td>
</tr>
<tr>
<td>52</td>
<td>Set CPR Encoder 1</td>
<td>2</td>
<td>Set the Cycles Per Revolution of encoder 1.</td>
</tr>
<tr>
<td>53</td>
<td>Set CPR Encoder 2</td>
<td>2</td>
<td>Set the Cycles Per Revolution of encoder 2.</td>
</tr>
<tr>
<td>54</td>
<td>Set Min Pulse Width Servo 1</td>
<td>2</td>
<td>Sets the minimum pulse width for servo 1.</td>
</tr>
<tr>
<td>55</td>
<td>Set Max Pulse Width Servo 1</td>
<td>2</td>
<td>Sets the maximum pulse width for servo 1.</td>
</tr>
<tr>
<td>56</td>
<td>Set Min Pulse Width Servo 2</td>
<td>2</td>
<td>Sets the minimum pulse width for servo 2.</td>
</tr>
<tr>
<td>57</td>
<td>Set Max Pulse Width Servo 2</td>
<td>2</td>
<td>Sets the maximum pulse width for servo 2.</td>
</tr>
</tbody>
</table>

**Figure 24 - PID Controller I2C Command Protocol**

### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RP1</td>
<td>Robotic Platform 1</td>
</tr>
<tr>
<td>ICD</td>
<td>Interface Control Document</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>JRE</td>
<td>JAVA Runtime Environment</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>OpSoft</td>
<td>Operational Software</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer (Desktop or Laptop)</td>
</tr>
<tr>
<td>SCD</td>
<td>Software Control Document</td>
</tr>
</tbody>
</table>

**Figure 25 - Table of Acronyms**

### Document References

<table>
<thead>
<tr>
<th>Document</th>
<th>Location or Link</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

81 of 82
## Robotic Platform 1kg
### Software Control Document

<table>
<thead>
<tr>
<th>Document</th>
<th>Location or Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer Needs</td>
<td><a href="https://edge.rit.edu/content/P09204/public/documentation/Needs%20Assessment.xls">https://edge.rit.edu/content/P09204/public/documentation/Needs%20Assessment.xls</a></td>
</tr>
<tr>
<td>Design Specifications</td>
<td><a href="https://edge.rit.edu/content/P09204/public/documentation/Design%20Specifications.xls">https://edge.rit.edu/content/P09204/public/documentation/Design%20Specifications.xls</a></td>
</tr>
<tr>
<td>Interface Control Document</td>
<td><a href="https://edge.rit.edu/content/P09204/public/documentation/Interface%20Control%20Document.docx">https://edge.rit.edu/content/P09204/public/documentation/Interface%20Control%20Document.docx</a></td>
</tr>
<tr>
<td>Hardware Control Document</td>
<td><a href="https://edge.rit.edu/content/P09204/public/documentation/Software%20Control%20Document.docx">https://edge.rit.edu/content/P09204/public/documentation/Software%20Control%20Document.docx</a></td>
</tr>
</tbody>
</table>

---

Figure 26 - Table of References