Kaizen Relay Control Module

Programming Manual

Table of Contents

Introduction:

The Kaizen Relay Control Module at it’s basic level is an Arduino Leonardo on a custom PCB with some specialized circuits and dressed in fancy plastic. Kaizen Speed chose to use this platform for it’s open source nature and wealth of information already available online.

In order to use this API (Application Programming Interface) we ask that you understand the basics of programming a microcontroller and have experience with AVR, STM32, PIC, or similar architectures.

If you do not have experience with any of the previously mentioned architectures, we ask that you contact one of Kaizen Speed’s development partners found on Kaizen Speed.

For questions or comments on the contents of this document,
please contact: Programs@KaizenSpeed.com

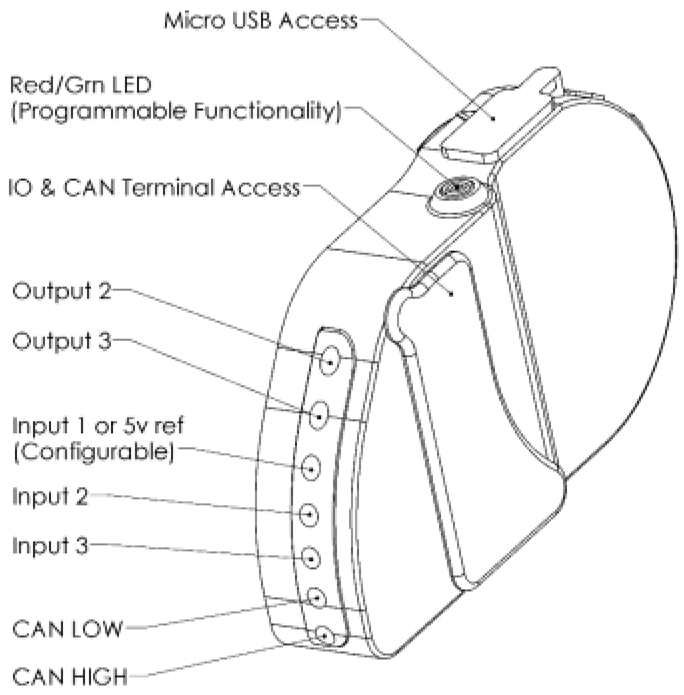
Getting Started

This API will be based on code written in the official Arduino IDE, using mostly C/ C++.
We will outline the major features of the IDE (Integrated Development Environment) and it's uses with the Kaizen Relay Control Module.

Please download and install the latest Arduino IDE before proceeding. As far as this API goes, we will be using the Windows Desktop version, but Mac and Linux versions are also available. The online version has not been tested, and is not recommended for use.

Connecting to the Kaizen Relay Control Module

Connection to the Kaizen Relay Control Module is done via a standard Micro-USB connector. Please ensure the cable is a data cable, and not just a charging cable. If you require a cable, an official Kaizen Relay Control Module Micro-USB cable can be purchased from the Kaizen Speed website.



Hardware Overview

The Kaizen Relay Control Module contains the following features:

- USB Programmability
- Dual Red/ Green LED - Individually Controllable
- 1x CANbus Network
    - Software Selectable Termination Resistor
- 3x Analog Inputs (0-5V)
    - Each input has two software selectable resistors that can be enabled: a pullup resistor, and a pulldown resistor.
    - Input #1 can be repurposed as a fused 5V supply that can provide a reference voltage to an external sensor, if required.
        - Common on many “3-wire” sensors such as a MAP sensor.
- 3x Low Side Outputs
    - Each output is designed to activate a Kaizen Relay using the Low Side Trigger pin.
- Power Consumption
    - The microcontroller in its idle state consumed about 42mA
    - Each Green LED on a Kaizen Relay consumes about 6mA
    - The Green LED on a Kaizen Relay Control Module consumes about 4mA
    - The Red LED on a Kaizen Relay Control Module consumes about 5mA
    - Because the Control Module is powered by the attachment to a Kaizen Relay, be wary of current draw when powered using a Constant Battery Connection, and vehicles sitting for long periods of time.

Kaizen Relay Template

External Software Note

The Kaizen Relay Control Module is dependent on one piece of external software which is included in the starting template, and that is for the CANbus Transmit and Receive functionality. This piece of software is public information and available on Github from SeeedStudio

Description of Files

KaizenRelay_Template_x.ino - This is where we will focus our attention and actually write the code that the control module will execute.

KaizenRelay.h - This is where the setup and helper functions are contained. Hardware Setup, CAN Setup, and various helper function definitions are all defined here.

KaizenRelayCAN.h - This is where the majority of the CAN functionality is contained and defined. The CAN Output Stream definition is defined here as well.

KaizenRelayEEPROM.h - This file is mostly for future use and stores the EEPROM positions of the variables required to store Control Module configurations when they become available for the end users to change -- “Tuner Software” aka InfinityTuner from AEM, Haltech ESP, PCLink, etc.

KaizenRelayPins.h - This is where the pin mapping definitions are stored. **DO NOT MODIFY**

KaizenRelaySerial.h - This is where the USB communication with the Kaizen Relay Manager software is defined. It is not advised to modify this file.

KaizenRelayVariables.h - This is where the variables used elsewhere in the template are defined. All of these variables are GLOBAL and can be used anywhere else in the template.

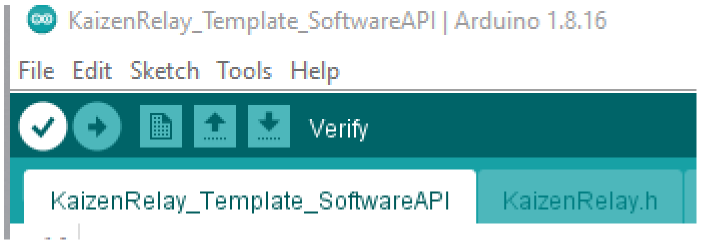
Mcp_can_x - These are the files provided by SeeedStudio to control the CAN interface. **DO NOT MODIFY**

Program Structure

The arduino programming uses two functions to runs its program:
There is a SETUP function that runs only once after each power up, and there is a LOOP function that runs on repeat continuously from start to end as long as the Kaizen Relay Control Module is powered. The Setup function is traditionally used to configure the Kaizen Relay Control Module, while the Loop function is traditionally the place where your application code lives.

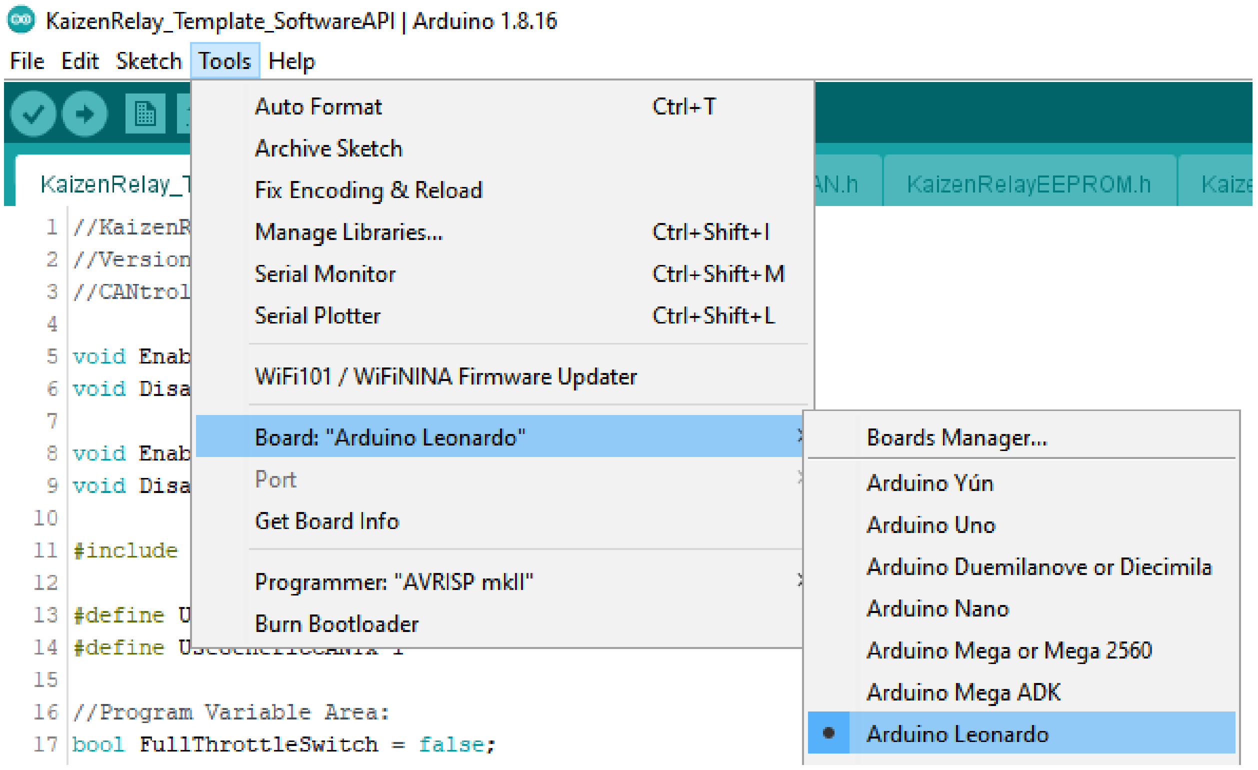
Verifying Code

Once the code is written, it is compiled and verified against errors using this button:


KaizenRelay_Template_SoftwareAPI | Arduino 1.8.16
File Edit Sketch Tools Help
Verify
KaizenRelay_Template_SoftwareAPI
KaizenRelay.h

Uploading to the Kaizen Relay Control Module

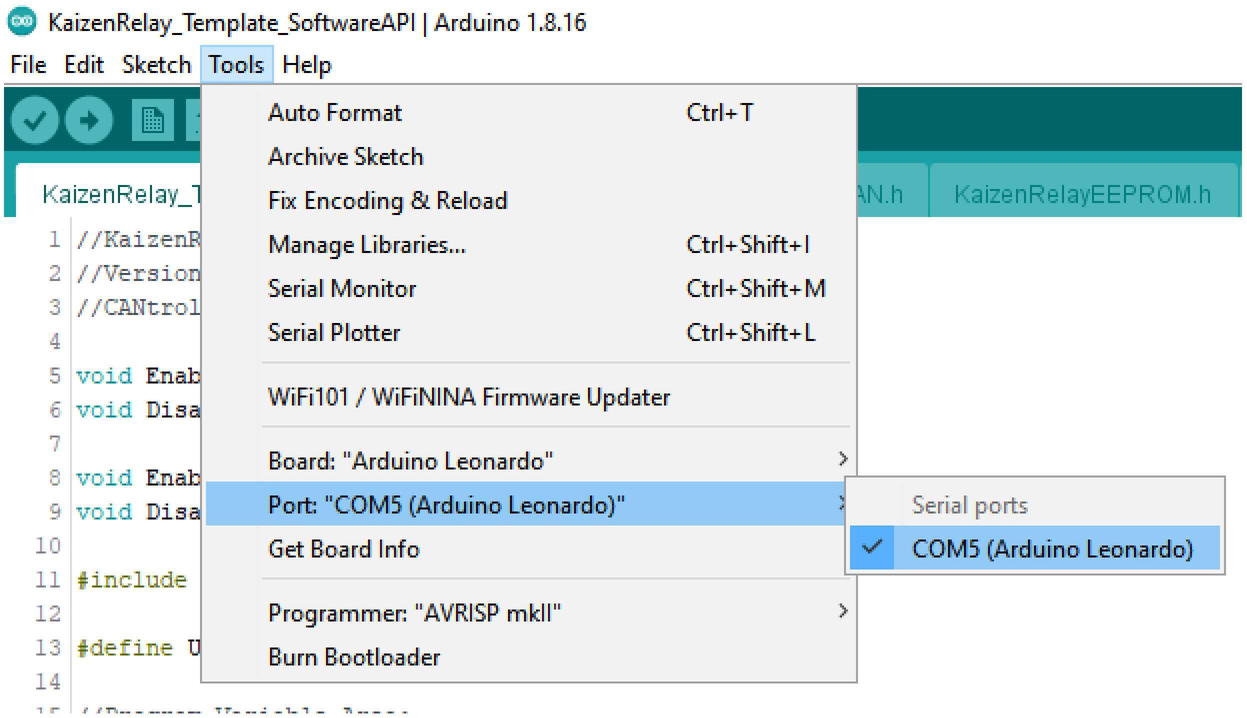
Once the code is verified to be error free, we can begin the upload process:
First, the correct board must be chosen from the Tools menu: Arduino Leonardo


KaizenRelay_Template_SoftwareAPI | Arduino 1.8.16

File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Board: "Arduino Leonardo"
Port
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Boards Manager...
Arduino Yún
Arduino Uno
Arduino Duemilanove or Diecimila
Arduino Nano
Arduino Mega or Mega 2560
Arduino Mega ADK
● Arduino Leonardo
KaizenRelayEEPROM.h

```
//Program Variable Area:
bool FullThrottleSwitch = false;
```

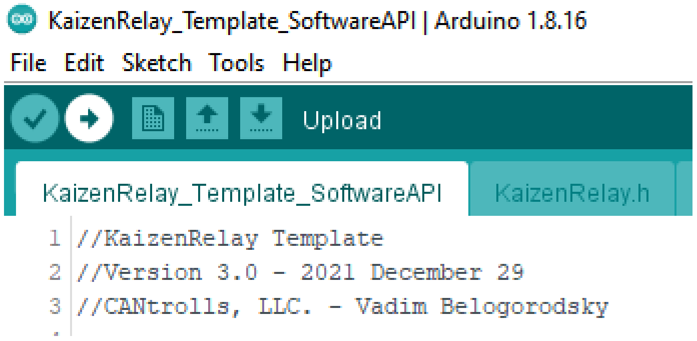
Next, the correct PORT must be selected from the Tools menu: COMx (Arduino Leonardo)
NOTE: Control Module must be connected to the computer via Micro-USB cable


KaizenRelay_Template_SoftwareAPI | Arduino 1.8.16

File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Board: "Arduino Leonardo"
Port: "COM5 (Arduino Leonardo)"
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Serial ports
COM5 (Arduino Leonardo)
KaizenRelay_T
AN.h
KaizenRelayEEPROM.h

```
//KaizenR
//Version
//CANtrol

void Enab
void Disa

void Enab
void Disa

#include

#define U
```

Lastly, click the Upload button:


KaizenRelay_Template_SoftwareAPI | Arduino 1.8.16

```
//KaizenRelay Template
//Version 3.0 - 2021 December 29
//CANtrolls, LLC. - Vadim Belogorodsky
```

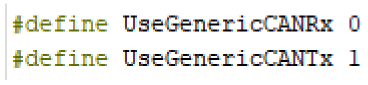
Programming

From here on out, we will be diving into the actual programming. Unless otherwise noted, the screenshots below are from the KaizenRelay_Template.ino file. The goal of many of the predefined variables and function names are to enable the USB communication between the Kaizen Relay Control Module and the Kaizen Relay Manager software.

These predefined variables and functions can be found in the index of this document.

Custom Firmware & CANbus Communication

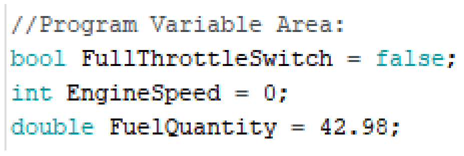
If you are planning to implement your own CANbus control scheme and do not want the default receive and transmit messages to be used, as defined in the Kaizen Relay CANbus Specification, then switch the “1” in the lines below to a “0” for the respective choice:



```
#define UseGenericCANRx 0
#define UseGenericCANTx 1
```

Adding Program Variables

To add variables to the program that are application specific to your needs, there is a space allotted here, shown with example variable names:

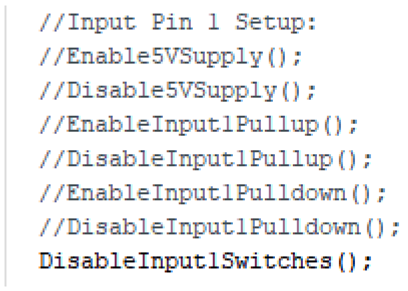


```
//Program Variable Area:
bool FullThrottleSwitch = false;
int EngineSpeed = 0;
double FuelQuantity = 42.98;
```

Setting up Input Pins

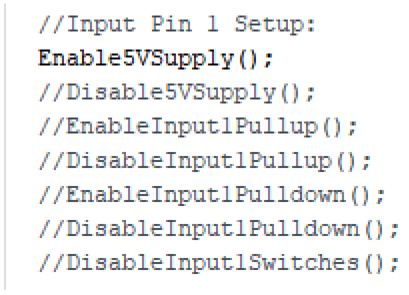
Due to the nature of the microcontroller used, each input can be configured to read a digital signal, returning a ‘true’ or ‘false’ reading, or can be configured to read an analog signal, returning a voltage from 0 to 5 volts. Additionally, each input has the ability to activate a 10kohm pullup resistor, or a 10kohm pulldown resistor. These are useful when reading switched digital inputs, to keep the input from floating. Lastly, Input 1 has an additional functionality built-in to provide a fused 5V reference to external sensors. Only Input 1 has this ability. Only one configuration per input is allowed: pullup, pulldown, or 5V supply, if available. Choosing configurations is a matter of uncommenting the configuration that you would like:

Default Configuration on Input 1:



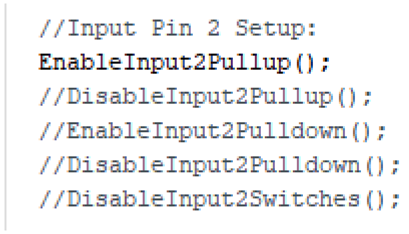
```
//Input Pin 1 Setup:
//Enable5VSupply();
//Disable5VSupply();
//EnableInput1Pullup();
//DisableInput1Pullup();
//EnableInput1Pulldown();
//DisableInput1Pulldown();
DisableInput1Switches();
```

Choosing 5V Supply on Input 1:



```
//Input Pin 1 Setup:
Enable5VSupply();
//Disable5VSupply();
//EnableInput1Pullup();
//DisableInput1Pullup();
//EnableInput1Pulldown();
//DisableInput1Pulldown();
//DisableInput1Switches();
```

Choosing Pullup Resistor on Input 2:

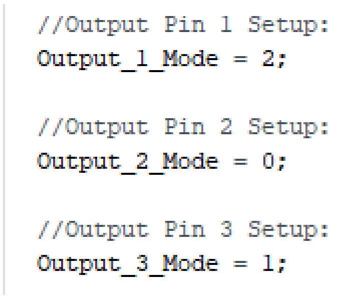


```
//Input Pin 2 Setup:
EnableInput2Pullup();
//DisableInput2Pullup();
//EnableInput2Pulldown();
//DisableInput2Pulldown();
//DisableInput2Switches();
```

NOTE: These input pin configurations can be changed throughout the program, not just at startup, by calling the required function(s).

Setting up Output Pins

The output pins are configured in a similar method, but using a variable instead of a function. All three outputs are identical, and can be configured for digital, on/ off control, or configured for analog, PWM control. The variables “Output_x_Mode” should be set to: 0 if the output is not used, 1 for analog PWM mode, and 2 for digital On/Off mode. See below where Output 1 is set to Digital Mode, Output 2 is not used, and Output 3 is set to Analog PWM Mode.



```
//Output Pin 1 Setup:
Output_1_Mode = 2;

//Output Pin 2 Setup:
Output_2_Mode = 0;

//Output Pin 3 Setup:
Output_3_Mode = 1;
```

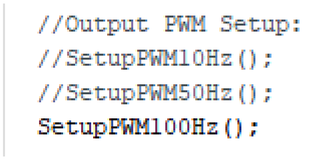
Digital outputs are good for triggering a Kaizen Relay full-on when a certain threshold or voltage is reached. For example, nitrous solenoids that need to be banged on, or transbrakes that need to be ‘bumped’.

Analog outputs are good for triggering a Kaizen Relay to smoothly turn on, when commanded. For example, soft-starting a cooling fan, or running a fuel pump slower when maximum fuel flow is not required so as to not introduce excess heat into the fuel system.

NOTE: These output pin configurations can be changed throughout the program, not just at startup, by changing the variables required.

Setting up Output PWM Frequency

For outputs that are set up as analog PWM outputs, a frequency for the PWM must be chosen. This is done similarly to how the Inputs are configured, by uncommenting a function corresponding to the frequency desired. See below where the output frequency is selected to be 100Hz.



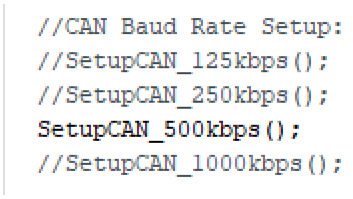
```
//Output PWM Setup:
//SetupPWM10Hz();
//SetupPWM50Hz();
SetupPWM100Hz();
```

Currently, 10Hz, 50Hz, and 100Hz are the only selectable frequencies. If a frequency you require is not listed, please contact Programs@KaizenSpeed.com for instructions on how to include custom PWM Frequencies.

NOTE: All outputs configured as analog PWM will share the PWM frequency chosen. It is currently not possible to configure individual PWM frequencies per output.

Setting up the CANbus Baud Rate

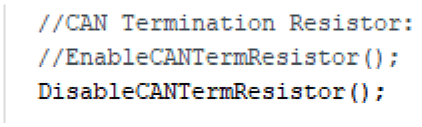
If CANbus is required in your firmware, the first thing to configure is the Baud Rate at which the Kaizen Relay will operate at. The default choices are 125kbps, 250kbps, 500kbps, and 1000kbps (1Mbps). Selection is done similar to the Inputs again, by uncommenting which Baud Rate you require. See below where 500kbps has been selected:



```
//CAN Baud Rate Setup:
//SetupCAN_125kbps();
//SetupCAN_250kbps();
SetupCAN_500kbps();
//SetupCAN_1000kbps();
```

Setting up the CANbus Termination Resistor

The Kaizen Relay Control Module has a built-in software selectable Termination Resistor that can be enabled. Without getting into details, every CANbus network must have 2 termination resistors, preferably one at each physical end of the network. Each resistor must be 120 ohms and placed across the CAN High and CAN Low wires. Too many or Too few resistors will cause CANbus faults in the Kaizen Relay Control Module. See below where the Termination Resistor is turned off, for this particular application:



```
//CAN Termination Resistor:
//EnableCANTermResistor();
DisableCANTermResistor();
```

End of Setup

All of the Setup Programming: Inputs, Outputs, and CANbus, has thus far been done in the ‘Setup’ function. In the next sections, we will take a look at the ‘Loop’ function and how we control each input, output, as well as send and receive CANbus messages.

Loop Function

The loop function is where the program effectively lives and is executed from top to bottom, on a continuous loop as long as the Kaizen Relay Control Module is powered. This type of scheduling is called ‘free-wheeling’ (the tasks are allowed to take as long as they need) and cycle times (the time it takes to do one loop) can drastically vary. Instead, we implemented a rigid timing structure, allowing users to schedule events. For example, send a CANbus message exactly every 10ms, or read a sensor every 100ms, or toggle an output every second. These timers are implemented using a method called interrupts.

These interrupts are specific triggers that pause the ‘loop’ loop, execute some code, and then return to the ‘loop’ loop where it left off. We have predefined timer interrupts at 10ms, 20ms, 50ms, 100ms, 500ms, and 1000ms, where code can be written and guaranteed to execute at these intervals.

NOTE: To give an idea of speed, it takes about 0.1ms (100us) to run one cycle of the microcontroller with very minimal code in free-wheeling mode.

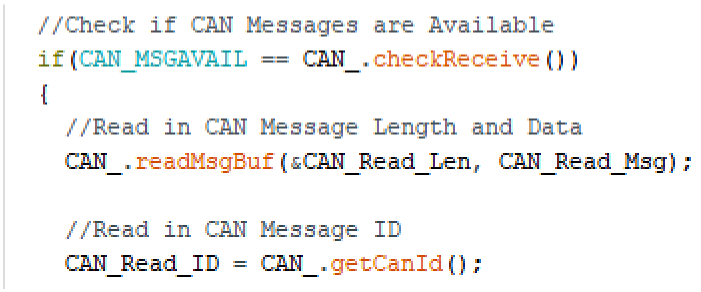
Reading CANbus Messages

The first thing we do every cycle is check for CANbus messages. This ensures that no CANbus messages are missed, and that they have priority in the freewheeling nature of the ‘loop’ function. If a message is received, it is saved into the following variables:

CAN_Read_ID stores the CANbus Message ID of the incoming message. It supports both 11-bit and 29-bit identifiers.

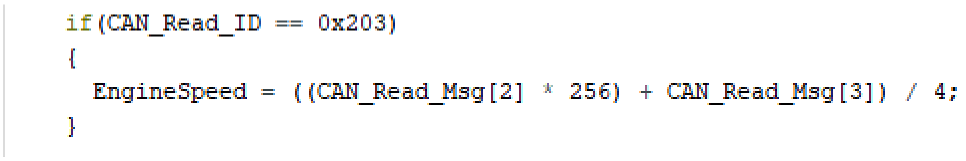
CAN_Read_Len stores the CANbus Message Size of the incoming message. This is how many bytes of data were received, between 0 and 8 bytes

CAN_Read_Msg stores the CANbus Message Data of the incoming message. This is an array of bytes, of size CAN_Read_Len. To access the 0th byte of CAN_Read_Msg you would type CAN_Read_Msg[0], and similarly to access the fifth byte of CAN_Read_Msg, you would type CAN_Read_Msg[5].



```
//Check if CAN Messages are Available
if(CAN_MSGAVAIL == CAN_.checkReceive())
{
  //Read in CAN Message Length and Data
  CAN_.readMsgBuf(&CAN_Read_Len, CAN_Read_Msg);

  //Read in CAN Message ID
  CAN_Read_ID = CAN_.getCanId();
```

Below is an example of how to check the ID of a CANbus Message and save the contents into a predefined variable.



```
if(CAN_Read_ID == 0x203)
{
  EngineSpeed = ((CAN_Read_Msg[2] * 256) + CAN_Read_Msg[3]) / 4;
}
```

If No CANbus Messages Are Present

When a CANbus message is received, a variable named CAN_MissedMsgs gets reset, as can be seen below. Alternatively, If no CANbus message is received that cycle, CAN_MissedMsgs gets 1 added to it, effectively counting how many cycles have gone by without a CANbus message being received. Because the average, used loosely, cycle time is 0.1ms, 10,000 missed msgs corresponds to 1 second without CANbus messages being received. The status of this CANbus timeout, whether messages are being received or not, is stored in the variable CAN_Timeout

```
  CAN_MissedMsgs = 0;
}
else CAN_MissedMsgs++;

CAN_Timeout = (CAN_MissedMsgs > 10000);

//Variable Overflow Handle
if(CAN_MissedMsgs > 60000) CAN_MissedMsgs = 11000;
```

Reading USB Messages

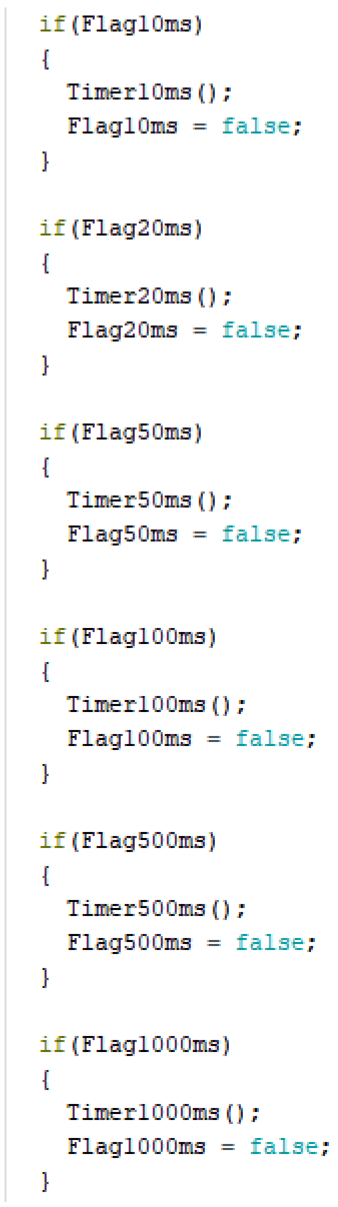
The following code is mostly reserved for future use and not used for the most part. It is there as a placeholder until the Kaizen Relay Manager software is setup to directly configure the Kaizen Relay Control Module. If receiving USB communication is important to your application, this is where your code would go.

```
//Check if Serial Messages are Available
if(Serial.available())
{
  while(Serial.available())
  {
    SerialInput += (char)Serial.read();
  }

  SerialIndex = SerialInput.substring(0, 3);
  SerialData = SerialInput.substring(4, SerialInput.length());

  ReadHWStatusUSB();

  SerialInput = "";
}
```

Setting up the Program Scheduler

The following lines contain the Program Scheduler Timer triggers. The code below should not be modified as there is a better location to place your application code. We wanted to bring attention to it, should there be questions about it.

NOTE: There are instances when more than one scheduler will execute it’s code. For example, when the application’s Timer triggers the 100ms scheduler, so too will it trigger the 10ms, 20ms, 50ms since they are all divisible into the 100ms. The same is true when the 1000ms timer triggers, all of the previous timers will also trigger, since 10ms, 20ms, 50ms, 100ms, and 500ms are all divisible into 1000ms. Remember this when prioritizing when each piece of code should get executed.



```
if(Flag10ms)
{
  Timer10ms();
  Flag10ms = false;
}

if(Flag20ms)
{
  Timer20ms();
  Flag20ms = false;
}

if(Flag50ms)
{
  Timer50ms();
  Flag50ms = false;
}

if(Flag100ms)
{
  Timer100ms();
  Flag100ms = false;
}

if(Flag500ms)
{
  Timer500ms();
  Flag500ms = false;
}

if(Flag1000ms)
{
  Timer1000ms();
  Flag1000ms = false;
}
```

Using the Program Schedulers to Write the Application Code

On the next page, we have examples of actual application code and how it can be used. There are specific functions written where you can place your code, and it will be executed when the scheduler is triggered. Code placed in the ‘Timer10ms’ function will execute every 10ms, code placed in the ‘Timer50ms’ function will execute every 50ms, and so forth. Generally speaking, code should not be run as quickly as possible, but be run when it makes sense.

As shown below, fuel quantity doesn’t change too quickly, so reading the fuel level every 10ms does not make sense. Instead, we can check it’s status every 500ms (half a second is still rather fast for fuel level as sloshing will create a varying response, but we do have a solution for that too!)

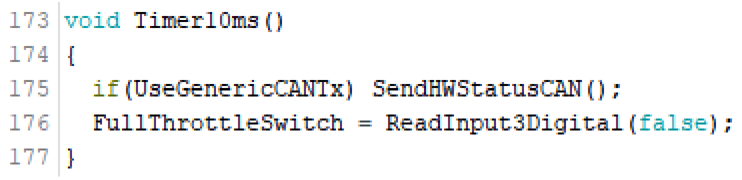
As a general rule of thumb for programming, suggest working backwards:

- What needs to powered (what devices are connected to the Kaizen Relays)
- What type of control does each device require (On/ Off or Variable Speed)
- When to activate/ How often should they be controlled (LEDs don’t need commands every 10ms, but a NOS solenoid might)
- If a ‘When’ depends on an input, What will be the input device (a CANbus message, a physical switch input)
- How often should this input be polled (a CANbus message can change every 10ms, while a human input device like a switch or a button is closer to a human reaction time of 200ms)
- Connect the inputs to the outputs in the Program Scheduler, examples below:

Application Code Examples

Explanation of Timer10ms Function Code below:
This code executes every 10ms and does two things in this example:

1. Send out its internal status via CANbus
2. Read the status of ‘FullThrottleSwitch’ on Input 3 using a digital read (On or Off).



```
173 void Timer10ms()
174 {
175   if(UseGenericCANTx) SendHWStatusCAN();
176   FullThrottleSwitch = ReadInput3Digital(false);
177 }
```

The first thing done (Line 175), the control module is capable of sending out what is happening internally through the CANbus, such as the status of the inputs and outputs, and whether the LEDs are on. To turn this feature on and off, we can toggle the CAN_Stream_Enabled variable, found in the KaizenRelayVariables.h file. If the variable is true (it is true by default), then the Kaizen Relay Control Module will send out its status over CANbus, cycling through 7 messages every 10ms.

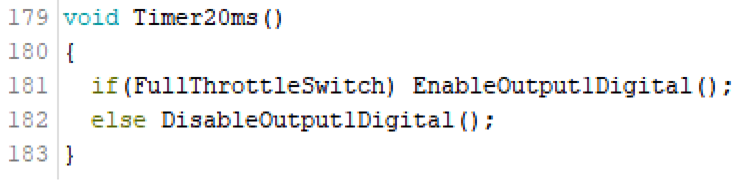
The second thing done is a digital read of Input 3 (Line 176).
ReadInputxDigital(false), ‘x’ can be 1 designating Input 1, 2 for Input 2, etc.
The ‘false’ designator is a choice of whether or not the status of the input should be inverted. For example:

If depressing the full throttle switch puts 5V on the input pin, then the function will return true when 5V is present, and the function will return false if less than 5V is present (simplifying here for the sake of the example)

If the full throttle switch was actually wired to connect to ground when fully pressed, then we would want to invert the return value so that the function will return true when less than 5V is present, and the function will return false when 5V is present (simplifying here for the sake of the example).

Explanation of Timer20ms Function Code below:
This code executes every 20ms and only does one thing in this example:

1. Toggle the state of Output 1



```
179 void Timer20ms()
180 {
181   if(FullThrottleSwitch) EnableOutput1Digital();
182   else DisableOutput1Digital();
183 }
```

In this function, we can see the program is evaluating the status of the FullThrottleSwitch variable (Line 181). If it's true, we turn on Output 1 in a digital fashion, using the following function:

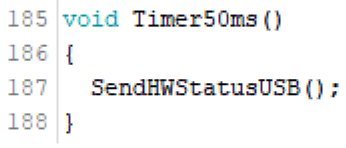
EnableOutputxDigital(), 'x' can be 1 designating Output 1, 2 for Output 2, etc.

If the FullThrottleSwitch is not true -- is false -- then we turn off Output 2:

DisableOutputxDigital(), 'x' can be 1 designating Output 1, 2 for Output 2, etc.

Explanation of Timer50ms Function Code below:
This code executes every 50ms and only does one thing in this example:

1. Send out the Kaizen Relay Control Module's internal status, through USB (Line 187):

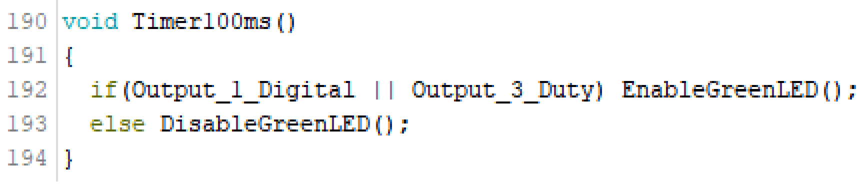


```
185 void Timer50ms()
186 {
187   SendHWStatusUSB();
188 }
```

This is how the Kaizen Relay Manager software and the Kaizen Relay Control Module communicate between each other.

Explanation of Timer100ms Function Code below:
This code executes every 100ms and only does one thing in this example:

1. Turns the Kaizen Relay Control Module's Green LED On and Off

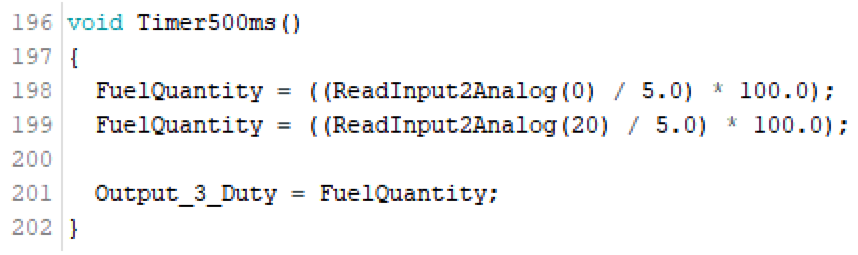


```
190 void Timer100ms()
191 {
192   if(Output_1_Digital || Output_3_Duty) EnableGreenLED();
193   else DisableGreenLED();
194 }
```

In this function, we evaluate the status of Output 1 and Output 3 (Line 192). If either output is active, the Green LED on the Kaizen Relay Control Module is turned on. Otherwise it is turned off (Line 193).

Explanation of Timer500ms Function Code below:
This code executes every 500ms and does two things in this example:

2. Reads the status of the ‘FuelQuantity’ on Input 2 using an analog read (voltage)
3. Sets the PWM Duty Cycle of Output 3 using the ‘FuelQuantity’

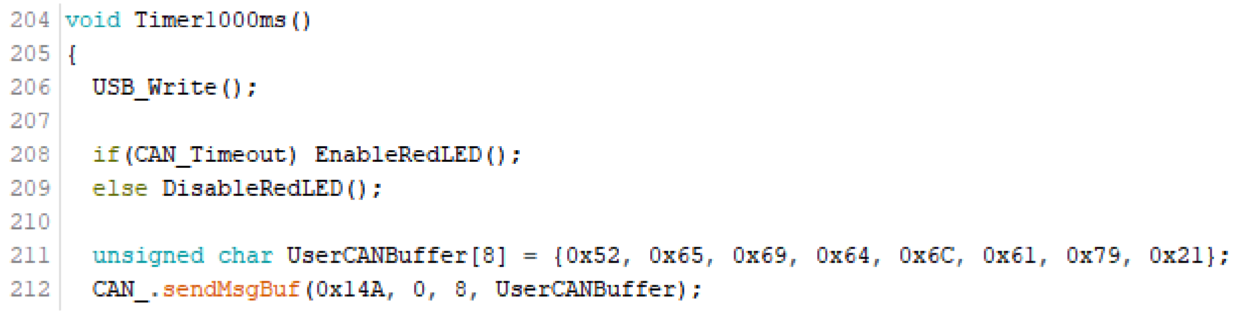


```
196 void Timer500ms()
197 {
198   FuelQuantity = ((ReadInput2Analog(0) / 5.0) * 100.0);
199   FuelQuantity = ((ReadInput2Analog(20) / 5.0) * 100.0);
200
201   Output_3_Duty = FuelQuantity;
202 }
```

The first thing done is an analog read of Input 2 (Line 198, Line 199). Shown above, are two versions: with 0 smoothing and 20 smoothing. These smoothing numbers are a choice of whether or not some software filtering is required on the analog input read. You do not have to call the function twice, this is just an example to show how one would enable the software filter. A value of 0 disables the software filter (Line 198), while a value above 0 enables the filter (Line 199). For more details on how the software filter works, please contact Kaizen Speed.

The second thing done is the setting of Output 3’s duty cycle. Output 3 was set into PWM mode previously, in the Setup function, so now, every 500ms the Duty Cycle is set for the output to operate at (Line 201).

Explanation of Timer1000ms Function Code below:



```
204 void Timer1000ms()
205 {
206   USB_Write();
207
208   if(CAN_Timeout) EnableRedLED();
209   else DisableRedLED();
210
211   unsigned char UserCANBuffer[8] = {0x52, 0x65, 0x69, 0x64, 0x6C, 0x61, 0x79, 0x21};
212   CAN_.sendMsgBuf(0x14A, 0, 8, UserCANBuffer);
```

This code executes every 1000ms and does three things in this example:

1. Runs the ‘USB_Write’ function which allows Users to communicate with the Kaizen Relay Manager Program, adding some debugging features to the code and additional diagnostics (Line 206). More on that below.
2. Reads the status of the ‘CAN_Timeout’ variable, and turns the Kaizen Relay Control Module’s Red LED On and Off, accordingly (Line 208, Line 209).
3. Sends a CANbus message from the Kaizen Relay Control Module (Line 212).

The first thing done is the call to ‘USB_Write’ function. This function was set up to give the user a place to communicate with the Kaizen Relay Manager software. More on this function below.

The second thing done is the evaluation of the ‘CAN_Timeout’ variable. If it is true, meaning no CAN messages have been read in recently, then the Kaizen Relay Control Module’s Red LED is turned on (Line 208). Otherwise, the Red LED is turned off (Line 209).

The third thing done is an example of how to send a CAN message from the Kaizen Relay Control Module. We first begin by assigning an array of unsigned chars to hold the CANbus message (Line 211). This should be setup in the “Program Variables” but for the purpose of this example, it was placed where the action is. The array set up in this example is called ‘UserCANBuffer’ and contains 8 bytes of data. The ‘0x’ designation in front of the numbers tells the Arduino program that these are coded in Hexadecimal notation. Next we send the CANbus message with the following line (Line 212). Explaining the format of the ‘CAN_.sendMsgBuf’ function:

0x14A is the CANbus Message ID - Edit as required
0 is whether the CANbus Message is a Remote Transmit Request (RTR) - Usually 0
8 is the CANbus Message Length - Edit as required
UserCANBuffer is the CANbus Message Buffer - Edit as required

Because the ‘CAN_.sendMsgBuf’ function is placed in the ‘Timer1000ms’ function, the CANbus message will be sent out every 1000ms. If a quicker transmit time is required, like 10ms or 20ms, etc., simply move this function to another ‘Timerxxms’ function. Call this function to send as many CANbus messages as required, For example:

CAN_.sendMsgBuf(0x2A, 0, 8, UserCANBuffer1) -- to send 0x2A message
CAN_.sendMsgBuf(0x167, 0, 4, UserCANBuffer2) -- to send 0x167 message
CAN_.sendMsgBuf(0x541, 0, 1, UserCANBuffer3) -- to send 0x541 message

Communicating with the Kaizen Relay Manager Software

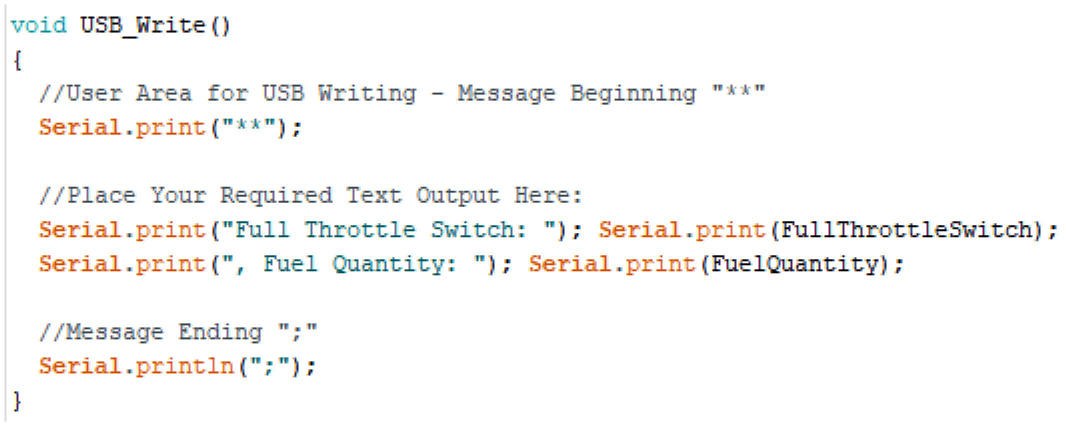
When developing the Kaizen Relay Control Module code, it was quite apparent early on that there needed to be a way for the Control Module to communicate with the outside world, without the use of CANbus as not everyone has those tools readily available. The answer to this was a function ‘USB_Write’, where the users can write their own diagnostic data, such as the printout of the variables or the status of some flags, and the output would be seen in the Kaizen Relay Manager software used for Input/ Output pin viewing and flashing of the Kaizen Relay Control Modules.

The format the Kaizen Relay Manager is looking for is pretty simple, and the beginning and ending of the message is pre-placed for the user. All that is required is the middle portion from the user.

The communication begins with a ‘Serial.print(“**”)’ -- this is what the Kaizen Relay Manager recognizes as the beginning of a User Transmission. Then you can add whatever you want, in between, referencing the Arduino Serial rules. The communication ends with a “Serial.println(“;”)’ which signals to the Kaizen Relay Manager software to print the line. Notice the ‘println’ on the last statement, instead of the ‘print’ on the preceding lines.

Depending on how frequently you would like your data printed to the screen, you can move the call to the USB_Write() function, currency in the ‘Timer1000ms’ function to a quicker function like ‘Timer100ms’ function.

NOTE: Data sent using the ‘USB_Write’ function between the Kaizen Relay Control Module and Kaizen Relay Manager software has a lower priority than the status messages already streaming from the Kaizen Relay Control Module. For this reason, it may appear that the user data is not displaying as frequently as suggested. If the user data is absolutely critical, move it to a faster ‘Timerxxms’ function.


```
void USB_Write()
{
  //User Area for USB Writing - Message Beginning "**"
  Serial.print("**");

  //Place Your Required Text Output Here:
  Serial.print("Full Throttle Switch: "); Serial.print(FullThrottleSwitch);
  Serial.print(", Fuel Quantity: "); Serial.print(FuelQuantity);

  //Message Ending ";"
  Serial.println(";");
}
```

The Leftovers

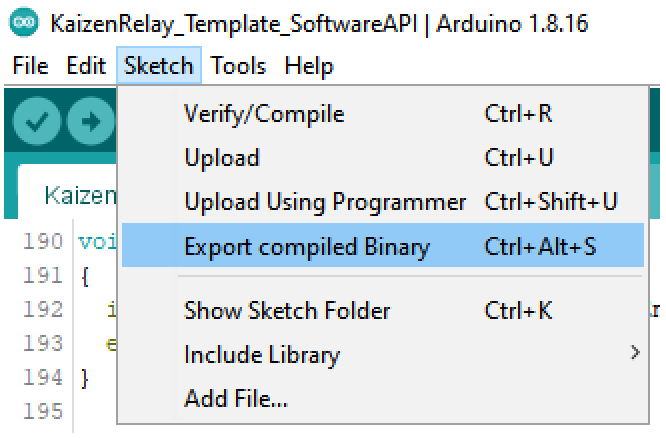
All of the code below the 'Timer1000ms' function are helper functions and should not be modified, unless extremely comfortable with coding. If there are any questions, please contact Kaizen Speed using the information in the Introduction section.

Distributing Programs

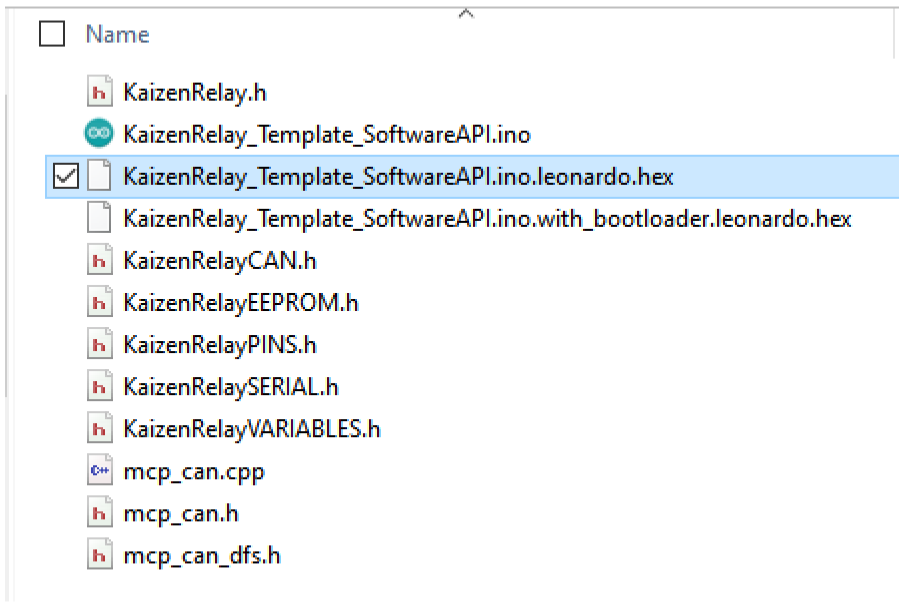
Once the code you have written is completed, there are two ways to distribute it:

1. Zip up the entire folder that contains the code and send it
    a. This gives the end user the ability to modify the code, but also the potential to use any intellectual property that may be had in the code
2. Use the Arduino IDE to export a hex file and send that
    a. This wraps up all of the code into one file that is not capable of being reverse engineered, and can be uploaded to the Control Module using the Kaizen Relay Manager.
    b. This is Kaizen Speed's preferred method of distribution

To export a hex file, select 'Export compiled Binary' from the Sketch menu:


KaizenRelay_Template_SoftwareAPI | Arduino 1.8.16
File Edit Sketch Tools Help
Verify/Compile Ctrl+R
Upload Ctrl+U
Upload Using Programmer Ctrl+Shift+U
Export compiled Binary Ctrl+Alt+S
Show Sketch Folder Ctrl+K
Include Library
Add File...

Once completed, navigate to the folder where the project lies and find the two new files created. One will be called ‘program.ino.lenoardo.hex’ which is the correct file to distribute:



```
Name
KaizenRelay.h
KaizenRelay_Template_SoftwareAPI.ino
KaizenRelay_Template_SoftwareAPI.ino.leonardo.hex
KaizenRelay_Template_SoftwareAPI.ino.with_bootloader.leonardo.hex
KaizenRelayCAN.h
KaizenRelayEEPROM.h
KaizenRelayPINS.h
KaizenRelaySERIAL.h
KaizenRelayVARIABLES.h
mcp_can.cpp
mcp_can.h
mcp_can_dfs.h
```

Follow the instructions in the Kaizen Relay Manager User Guide to upload the hex file to the customers’ Kaizen Relay Control Module

Index

Pin Mapping (Control Module to Arduino)

Input 1:	A0	Analog 0		
Input 1 Pullup Enable:	10	Digital 10		
Input 1 Pulldown Enable:	6	Digital 6		
Input 1 5V Sensor Supply Enable:	8	Digital 8		
Input 2:	A2	Analog 2		
Input 2 Pullup Enable:	A1	Analog 1		
Input 2 Pulldown Enable:	12	Digital 12		
Input 3:	A3	Analog 3		
Input 3 Pullup Enable:	A4	Analog 4		
Input 3 Pulldown Enable:	4	Digital 4		
Output 1:	9	Digital 9	Port B, Bit 5	PB5
Output 2:	13	Digital 13	Port C, Bit 7	PC7
Output 3:	5	Digital 5	Port C, Bit 6	PC6
Red LED:	3	Digital 3	Port D, Bit 0	PD0
Green LED:	2	Digital 2	Port D, Bit 1	PD1
CAN Chip Select:	0	Digital 0		
CAN Termination:	A5	Analog 5		

Program API Variables

Contained in KaizenRelayVariables.h

Inputs: Variable Types and Default Initialization

- bool Input_x_Pullup_Enabled = false;
  - Contains the status of the Pullup Resistor on that Input Pin
- bool Input_x_Pulldown_Enabled = false;
  - Contains the status of the Pulldown Resistor on that Input Pin
- bool Input_1_5V_Enabled = false;
  - Input 1 Only**
  - Contains the status of the 5V Supply on that Input Pin
- double Input_x_Voltage = 0.00;
  - Contains the voltage on that Input Pin
- bool Input_x_Smoothing_Enabled = false;
  - Contains the status of whether or not the software filter on the analog Input Pin
- byte Input_x_Smoothing = 0;
  - Contains the value of analog smoothing, 0 to 100, if enabled
- bool Input_x_Digital = false;
  - Contains the digital input status on that Input Pin
- bool Input_x_Invert = false;
  - Contains the status of whether or not the digital status is inverted on that Input Pin
- byte Input_x_Mode = 0;
  - Contains the mode in which the Input Pin is configured:
    - 0 = Off
    - 1 = Analog Input
    - 2 = Digital Input
    - 3 = 5V Supply Enabled (Input 1 Only)

Output Helper: Variable Type

- byte PWMpulseCounter = 0;
  - Contains a counter used for the PWM functionality, when enabled.
  - Counts from 0 to 99.

Outputs: Variable Types and Default Initializations

- byte Output_x_Mode = 0;
  - Contains the mode in which the Output Pin is configured
    - 0 = Off
    - 1 = Analog/ PWM
    - 2 = Digital

- byte Output_x_PWMFreq = 0;
  - Contains the PWM frequency at which each output operates at

- byte Output_x_Duty = 0;
    - Contains the Duty Cycle on that Output Pin
- bool Output_x_Digital = false;
    - Contains the digital input status on that Output Pin

CANbus: Variable Types and Default Initializations

- unsigned char CAN_Read_Msg[8];
    - Contains the received CAN message data
- unsigned char CAN_Read_Len = 0;
    - Contains the received CAN message length
- unsigned long CAN_Read_ID = 0;
    - Contains the received CAN message ID
- bool CAN_TermRes_Enabled = false;
    - Contains the status of the CANbus termination resistor
- bool CAN_Stream_Enabled = true;
    - Contains the status of the CANbus output stream
- unsigned char CAN_Status_Msg_x[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    - Contains the arrays of the CANbus output stream
    - There’s 7 messages sent cyclically
- unsigned int CAN_BaudRate = 500;
    - Contains the current CANbus baud rate
- unsigned int CAN_MissedMsgs = 0;
    - Contains a counter for the number of missed CANbus message cycles
- bool CAN_Timeout = false;
    - Contains the status of whether or not the CANbus has timed out
    - Timeout occurs when more than 10000 cycles have passed without receiving a CANbus message
- unsigned long CAN_Base_Address = 0x72A;
    - Contains the base ID of the CANbus output stream
- unsigned char CAN_Msg_Counter = 0;
    - Contains the number of the CAN_Status_Msg to send out, cycles from 0 to 6.

Misc: Variable Types and Default Initializations

- bool LED_Green_Status = false;
    - Contains the status of the Green LED on the Control Module
- bool LED_Red_Status = false;
    - Contains the status of the Red LED on the Control Module

Program API Functions

Inputs:

- void EnableInputxPullup()
  - Enables the pullup resistor on Input Pin x
- void DisableInputxPullup()
  - Disables the pullup resistor on Input Pin x
- void EnableInputxPulldown()
  - Enables the pulldown resistor on Input Pin x
- void DisableInputxPulldown()
  - Disables the pulldown resistor on Input Pin x
- void Enable5VSupply()
  - Input 1 Only**
  - Enables the 5V Supply on Input 1
- void Disable5VSupply()
  - Input 1 Only**
  - Disables the 5V Supply on Input 1
- void DisableInputxSwitches()
  - Disables all switches on Input Pin x, corresponding to pullup resistors, pulldown resistors, and 5V Supplies
- bool ReadInputxDigital(bool invert)
  - Returns the status of a Digital Read on Input Pin x
  - Invert argument allows you to flip the return value
    - This is useful for ground operated switches
- double ReadInputxAnalog(double smoothing)
  - Returns the voltage of an Analog Read on Input Pin x
  - Smoothing argument allows you to apply a software filter on the voltage so that it is more stable. See Running Average' function definition in KaizenRelay.h
- void DisableInputx()
  - Turns all input functionality off on Input Pin x

Outputs:

- void EnableOutputxDigital()
  - Turns the Output ON, on Output Pin x
  - ON = Ground Output
  - OFF = 12V Output
- void DisableOutputxDigital()
  - Turns the Output OFF, on Output Pin x
- void DisableOutputx()
  - Turns all output functionality off on Output Pin x
- void SetupPWM10Hz()
  - Sets up the Kaizen Relay Control Module to be able to output PWM at 10 Hz
  - All Output pins use the same PWM Frequency**

- void SetupPWM50Hz()
    - Sets up the Kaizen Relay Control Module to be able to output PWM at 50 Hz
    - All Output pins use the same PWM Frequency**
- void SetupPWM100Hz()
    - Sets up the Kaizen Relay Control Module to be able to output PWM at 100 Hz
    - All Output pins use the same PWM Frequency**

Misc.

- void EnableGreenLED()
    - Turns the Kaizen Relay Control Module Green LED ON
- void DisabeGreenLED()
    - Turns the Kaizen Relay Control Module Green LED OFF
- void EnableRedLED()
    - Turns the Kaizen Relay Control Module Red LED ON
- void DisableRedLED()
    - Turns the Kaizen Relay Control Module Green LED OFF
- double Interpolate(double voltage)
    - Helper function that performs a linear interpolation between points
    - Email Kaizen Speed for details
- double RunningAverage(double a, double instData, double buffData)
    - Helper function that performs a software filter for analog input readings
    - Email Kaizen Speed for details
- bool HysteresisDigital(bool currentStatus, double x,
  double ONthreshold, double OFFthreshold)
    - Helper function that adds some hysteresis functionality
    - Email Kaizen Speed for details
- bool HysteresisAnalog()
    - In development
    - Email Kaizen Speed for details