Industrial Smart Fuse

Justin Stark, Computer Engineering

Project Advisor: Dr. Anthony Richardson

April 26, 2019
Evansville, Indiana

Table of Contents

List of Figures

List of Tables

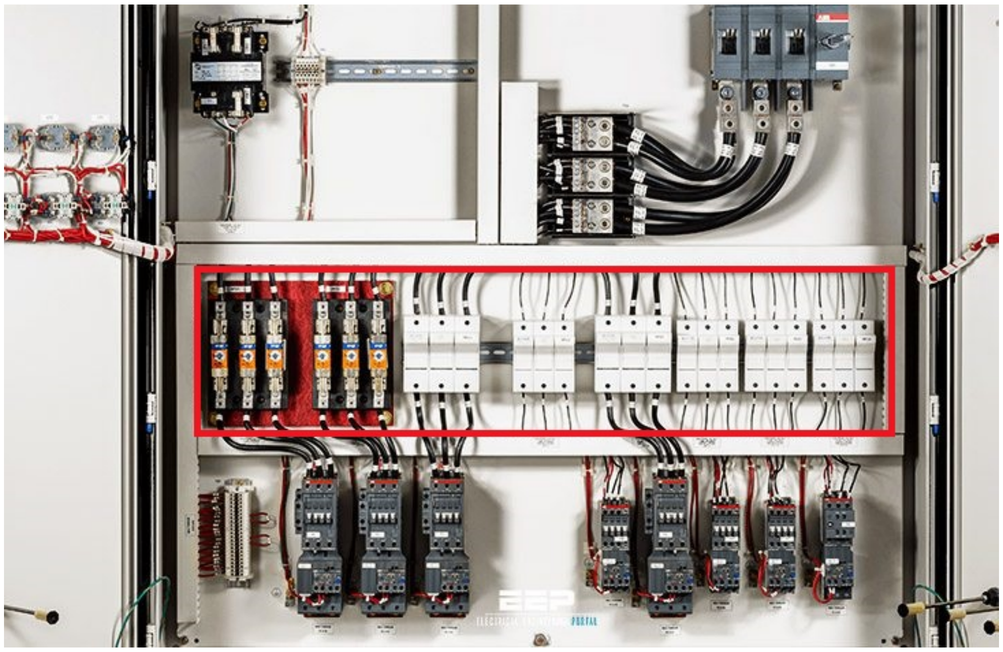
I. Introduction

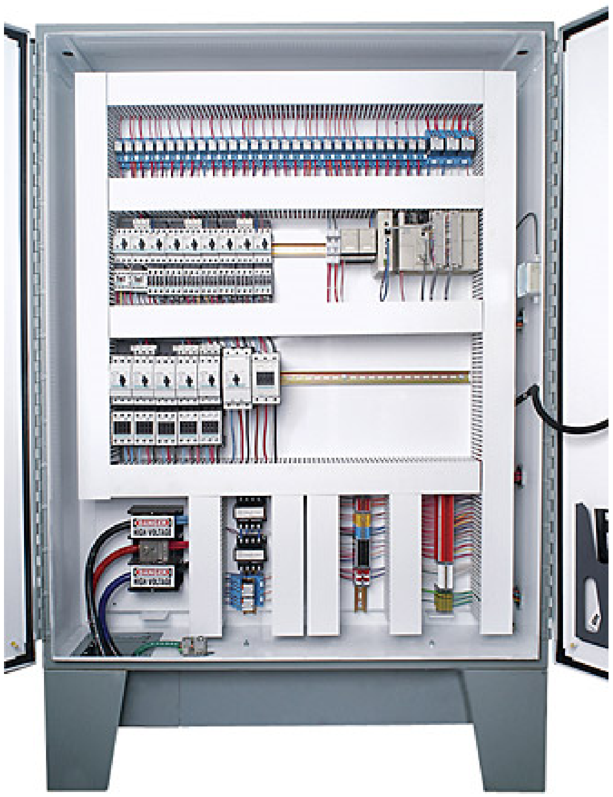
In a report released in 2006 [1], unplanned downtime costs automotive manufacturers an average of $22,000 a minute. The average amount of time lost due to unplanned downtime is somewhere between 1.5 to 4 hours. The cost of unplanned downtime in manufacturing is measured by accumulating the cost of man hours spent fixing the issue, making up for lost production by adding overtime man hours, and considering the amount of business lost due to lost production time and depleted inventory. The pressure to catch up with production also causes unnecessary strain on equipment and employees, which could lead to further problems. Production managers perform this analysis to gain some insight into the true costs of unplanned downtime, and technology is making analyzing the data easier than ever. Manufactures can now see how downtime directly affects their bottom line, which is why reducing downtime is more important now than ever before.

Not surprisingly, almost half of the unplanned downtime is directly related to hardware failures. Most manufactures currently do not take a predictive approach to maintaining and replacing equipment. Part of the problem is not having the information available to make educated decisions about equipment maintenance. Because the manufactures are not setup to predicatively replace equipment before failure, most resort to fixing the problem after equipment has already failed. Diagnostic speed is especially important in reactive maintenance, but the source of hardware failures is not always apparent. Diagnosing may be slow because a manufacturing control panel may contain many fuses, and the location of the control panel in a large manufacturing facility may not be known. [2, 3]

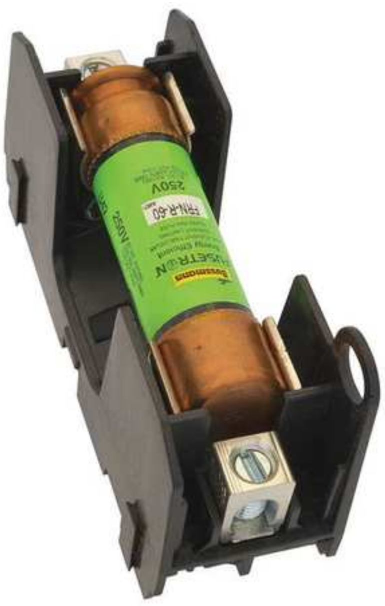
A system of smart fuses that could provide feedback on a fuse’s status, and other data collected by the fuse could provide the information needed to resolve hardware issues quickly. The main feature of the smart fuse would be its ability to relay its panel location and rack position to the control system when a fuse is blown via ethernet. The fuse would be flexible enough to work with many different existing control systems allowing for uniformity in the manufacturing environment and would allow faster adoption of this method of fuse monitoring.

II. Background

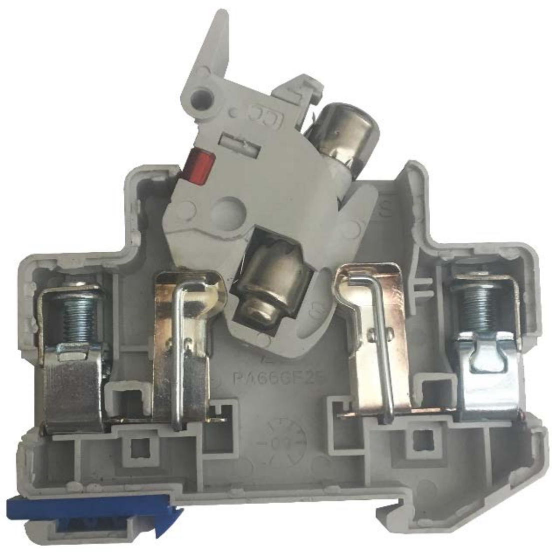

Fuse Holders Inside of a Panel
Figure 1 [5]



Fuse Holders Inside of a Panel
Figure 2 [6]



Closeup of a Fuse
Figure 3



A different kind of Fuse Holder
Figure 4

Because unplanned downtime is so costly, manufacturers are looking for new ways to identify problems. This project focuses on downtime particularly related to malfunctioning equipment. Control panels like those shown in figures 1 and 2 contain the software system, the fuses, and the other control circuitry for the equipment. Fuses and fuse holders like those shown in figures 3 and 4 are a part of just about every control circuit, and a common symptom of equipment failure is a blown fuse. The presence of a blown fuse could have directly caused the downtime problems or may help diagnose the problem from data collected from them. There has been some progress towards locating and diagnosing blown fuses, but it is still relatively time-consuming to locate the specific fuse that was blown and even harder to diagnose.

The first issue that needs solving will be finding blown fuses. Knowing when a fuse is blown and finding it quickly would save precious manufacturing time. Another problem would be diagnosing circuit problems and finding what caused the blown fuse. The device could monitor voltage and current at the fuse and have that data available for analysis.

There are currently devices on the market that can detect a blown fuse and send a discrete signal to a control system via wire. Relaying a discrete output from a fuse to the control system adds unneeded complexity to the system. This other system takes up scarce I/O ports. By using ethernet the modules can rely on existing infrastructure to relay the information.

Safety is also an issue with existing systems. If a technician is forced to troubleshoot a system while power is still turned on, there is the possibility of electrical shock. Technicians need a way of determining which fuse is blown without being in direct contact with the fuses.

III. Design Approach

A. Blown Fuse Detection

The industrial smart fuse would first need to be able to detect a blown fuse. Fuses do not always have current going through them so measuring current would not be able to determine if a fuse is blown. Other ways of determining if a fuse is blown are to either measure the resistance of the fuse, which is not possible because the fuse will have power being applied to it, or to measure the voltage on both sides of the fuse and verify that they are the same. If a fuse is blown, there should be a voltage differential. An analog to digital converter is used to measure the voltage on either side of the fuse to determine voltage. Analog to digital converters in STM32 allow for calibration to increase accuracy.

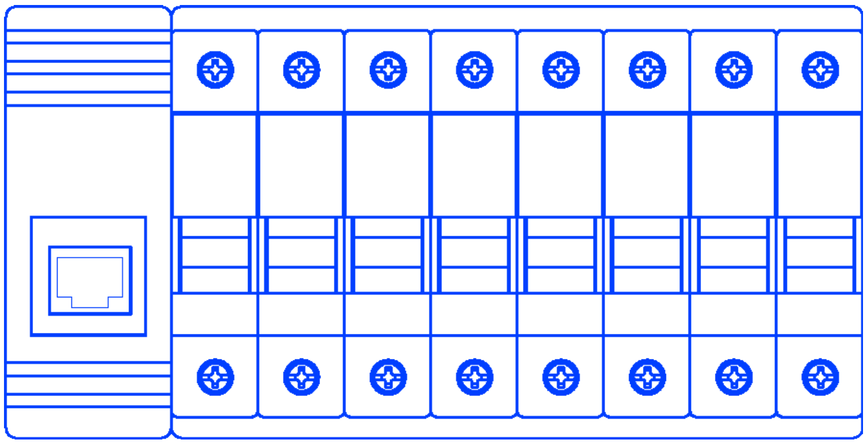
B. Electrical Isolation

Electrical isolation is the next priority for the device. An electrically isolated device would not only make sure the high voltages of the control circuit does not interfere with the low voltages of the fuse monitoring circuit, but the isolation would also make sure technicians are not in contact with these high voltages. An optical isolator is used to provide the isolation. Optical isolators can provide isolation up to 5000V. A linear optical isolator is used with a feedback control circuit to transfer the voltage from the fuse circuit to the measurement circuit without a physical connection.

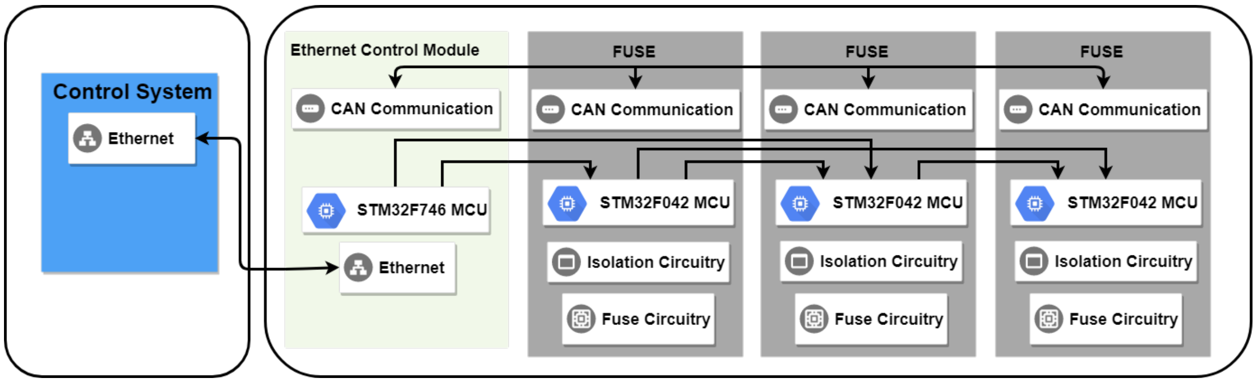
C. Control Circuit Protection

The fuse must also be able to withstand high voltages and currents. Electrical isolation solves the issue of being exposed to damaging high voltages. Even if there was a voltage spike, the spike could not be transferred to the measurement side of the circuit. The measurement circuitry would be in parallel to the control circuit and would have a very high input impedance which would not allow high currents through it.

D. Control System Communication



Packaged Industrial Smart Fuse
Figure 3



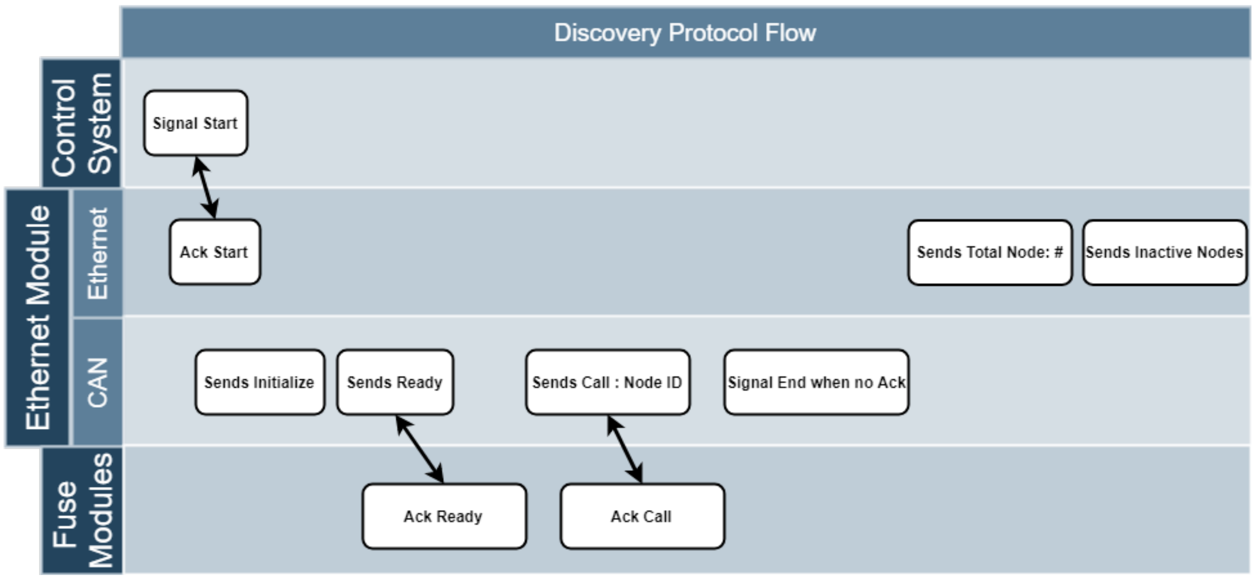
System Diagram of the Industrial Smart Fuse
Figure 4

The smart fuse system communicates over ethernet so that it will not use up the limited supply of I/O ports available on the control system and will also be able to communicate over long distances. Ethernet is ideal because it is fault tolerant and supports transfer rates starting at 10 megabits per second. Ethernet is becoming increasingly common in the industrial environment, so the fuses can use existing control infrastructure. Because there are many fuses in a panel, connecting ethernet to each fuse is impractical. A solution to this problem is to make the fuses modular as shown in figure 3 and 4. The fuse modules communicate along a common bus. In addition to the connected fuses, there is also an ethernet module which acts as a control hub for the connected fuses. The ethernet module does the communication with the control system.

E. Inter-module Communication

The fuses will need to be able to communicate with each other and with the ethernet module. A CAN bus provides the protocol necessary to communicate. Everything on the CAN bus will be connected in parallel and will be able to communicate to all other nodes. A CAN bus is more robust than other protocols because it uses differential signals to transfer packets. Differential signals help to reduce noise in the industrial environment. The CAN protocol also has a complete set of error checking features.

F. Node Indexing

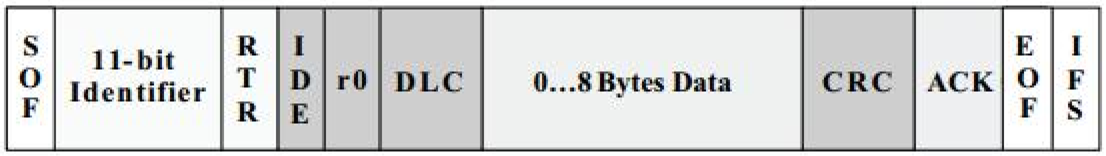


Discovery Protocol Message Flow
Figure 5

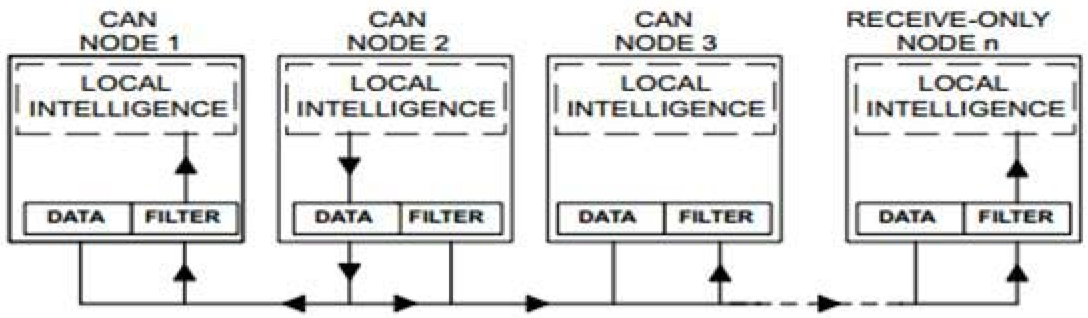
An issue with having all nodes on the bus in parallel is that the physical location and index of the nodes is not known. This issue can be circumvented by using a specialized discovery protocol. The protocol will use two additional wires connecting the microcontrollers together. Each microcontroller will have two input wires and two output wires as shown in figure 4. The one output wire will be connected to the next node in the series and the second wire will be connected to the node after the next one. The protocol uses two wires so that even if one node quits working the discovery protocol can skip it.

The discovery protocol is used index the fuse nodes and is depicted in figure 5. The discovery protocol is initiated by the control system by sending a message to the ethernet module. After receiving the signal, the ethernet module then sends a messaging initiating the protocol in the fuse modules. Next the ethernet module turns on its first discrete output. The ethernet module then sends out the node id call which includes the node id. The fuse which has its first input turned on receives the index as its own and sends an acknowledge message back to the ethernet module. If there is not a fuse at index one, or it is unresponsive, the ethernet module turns on its second output. The ethernet modules sends out the node id call again, and if there is a fuse at the second index, it will receive the index as its own and send back an acknowledge message. After the first or second index is identified, the ethernet module will now start sending out a ready one and ready two message. These two messages are used to command the most recently identified index to turn on its first and second output respectively because other fuses are checking for a high input signal to accept the node id from the node id call. Only one fuse module will have a high input at a time so only one fuse module will send the acknowledge back. After the ethernet module sends ready one and the most recently identified index returns a ready acknowledgement, the most recently identified index will turn on its first output. The next fuse modules in line will then respond to the node id call from the ethernet module. This method will continue going down the row until all fuses have been indexed and the ethernet module does not receive a call acknowledgement from both node id calls following consecutive ready one and ready two calls. The ethernet module will keep track of the total number of fuse modules indexed and which ones are inactive.

G. CAN Communication



CAN Packet
Figure 6

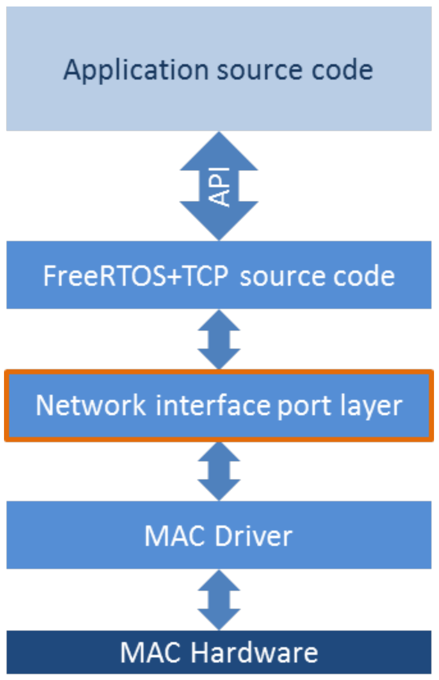


.

CAN Bus Depiction
Figure 7

The CAN communication protocol is a producer-consumer model. The CAN packet format is depicted in Figure 6. The CAN bus relies on CAN nodes to filer out different kinds of messages that it wants to receive as shown in figure 7 by filtering out different identifiers. The industrial smart fuse system has set identifiers to communicate between fuse modules and the ethernet module. The identifiers are heartbeat, fuse ok, fuse blown, fuse blown acknowledge, error 1, error 2, status update, discovery start, discovery ready 1, discovery ready 2, discovery call, discovery call response, and discovery end. These identifiers had ids of 1 through 15.

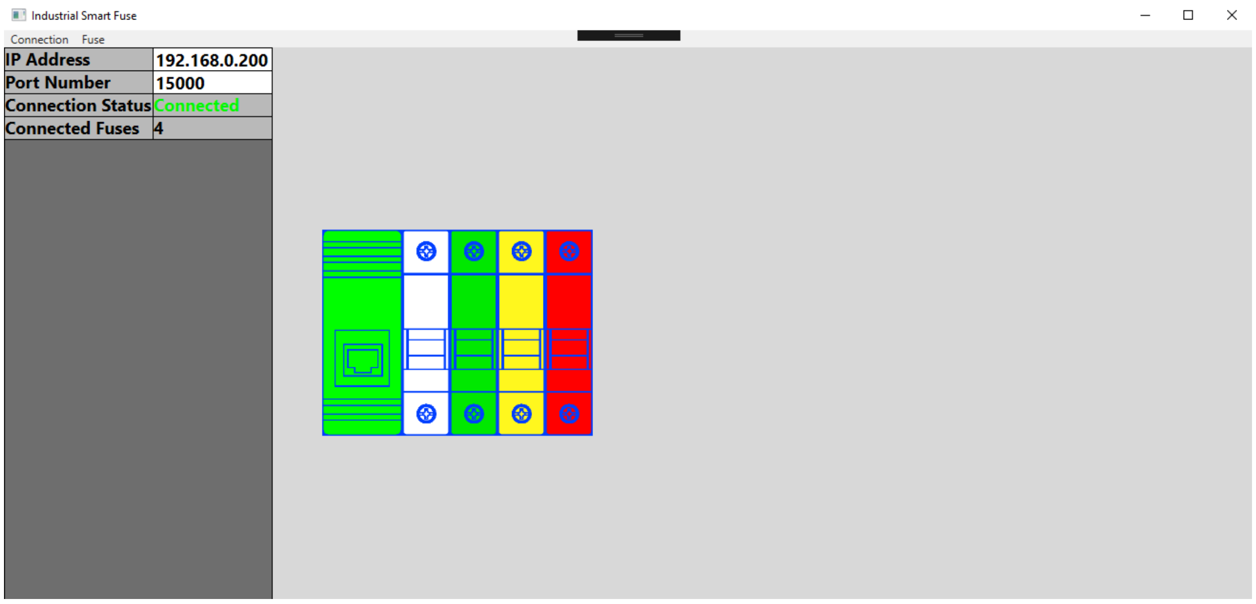
H. Ethernet Communication



FreeRTOS+TCP Stack Port Layers
Figure 8

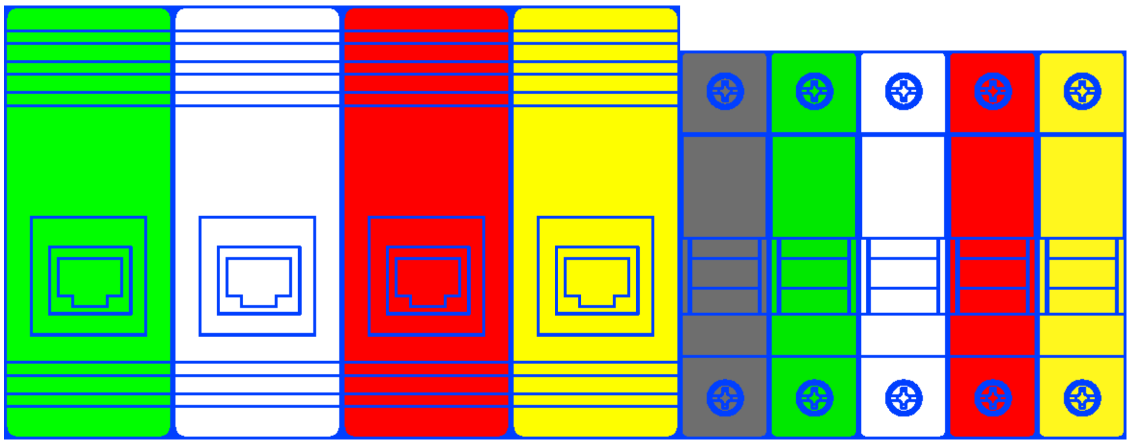
The system ethernet capability is provided by the FreeRTOS+TCP. The ethernet stack provides the Berkeley sockets API. This TCP stack is built on top of the FreeRTOS real time operating system. The TCP stack is made up of three parts: FreeRTOS+TCP source code, network interface port layer, and the MAC driver. The MAC driver was provided by a modified HAL library file. The network interface port layer was already ported for the STM32F746 MCU. The STM32F746 network interface port layer required some modification to work with the MCU and LAN8742A-CZ-TR ethernet transceiver.

I. Control System



IP Address	**192.168.0.200**
Port Number	**15000**
Connection Status	**Connected**
Connected Fuses	**4**

WPF Application (Example Control System)
Figure 9



Different Fuse and Ethernet Statuses
Figure 10

The control system communicates with the ethernet module. The ethernet module is a TCP server that the control systems subscribe to. Once a control system subscribes to the ethernet module, the ethernet module starts sending out status updates. The control system can also initiate the discovery protocol. The ethernet module then will send back updates from the discovery protocol. This feedback is used to update the number of fuses shown by the control system.

J. Future Features

There are many other features that can be added in the future. One of these features is in-application programming. As future software developments are made, existing fuse modules will need their firmware updated. These updates can be achieved through in-application programming available on the stm32f0 line of microcontrollers. If the existing firmware is still running, all the fuse modules will be able to receive the updates through packets received on the CAN bus. The Ethernet module will be able to download the updates and distribute them to all connected fuses.

Another feature is an embedded web server. The Ethernet module would be running the web server. The web interface will allow for configuration setting to be changed and will also allow for remapping of the fuses true index with a logical index through a lookup table. This table would be necessary in case a fuse is removed or added, but the rest of the fuses need to retain their existing index. The indexes might be used externally in the equipment control system, so they need to stay stable to make sure no other functionality is disrupted. The web interface will also have the added capability of displaying the status for all the attached fuses.

IV. Testing

The device will be tested by doing a voltage sweep and current sweep with different fuses. When the fuses blow, the time it takes to relay the data from the fuse to the control block and to the control system will be measured. If the system would be able to be used in a deterministic system, the time required to relay the information would need to be stable and absolute. In system monitoring for notification purposes, the fuses do not have to be deterministic.

Appendix A

References

Reports
[1] Downtime Costs Auto Industry $22k/Minute – Survey. *Thomas*. (2006, March) Available: https://news.thomasnet.com/companystory/downtime-costs-auto-industry-22k-minute-survey-481017

[2] Graham Immerman. (2018, May) The real cost of downtime in manufacturing. *Machine Metrics.* Available: https://www.machinemetrics.com/blog/the-real-cost-of-downtime-in-manufacturing

[3] Peter Daisyme. (2018, June). Understanding the financial cost of downtime in manufacturing. *Due.* Available: https://due.com/blog/understanding-the-financial-cost-of-downtime-in-manufacturing/

[4] Blown Fuse Monitoring Relays. *Automation Direct.* Available: https://www.automationdirect.com/adc/shopping/catalog/circuit_protection_-z-_fuses_-z-_disconnects/fuses_-a-_fuse_holders/blown_fuse_monitoring_relays

Images
[5] Edvard Csanyi. (2017, June) Wiring tips for connections and routing inside industrial control panel. *Electrical Engineering Portal.* [Online] Available: https://electrical-engineering-portal.com/tips-wiring-industrial-control-panel

[6]
Available: http://www.prestigecarwashequipment.com/custom/images/products/components/icp.jpg

Code

```
// Justin Stark
// 4/26/2019
// WPF Control System Application

using Microsoft.Windows.Controls;
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace Control_System_001
{

    public partial class MainWindow : Window
    {
////////////////////////////////////////////////////////////////////////////
        private delegate void noparamCallback();
        private delegate void fuseUpdateCallback(int index, FuseStatus fs);
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
        private enum EthernetStatus
        {
            Connected,
            Error,
            Unconnected
        }
        private enum FuseStatus
        {
            Unconnected,
            OK,
            Error,
            Blown
        }
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
        private enum MessageCodes
        {
            Heartbeat, // 0
            FuseOK, // 1
            FuseBlown, //2
            Error1, // 3
            Error2, // 4
            DiscoveryStart, //5
            DiscoveryAck, // 6
            DiscoveryNodes, // 7
            DiscoveryInactive, // 8
            DiscoveryFinished // 9
        }
        //////////////////////////////////////////////////////////////////////////

        //////////////////////////////////////////////////////////////////////////
        BitmapImage[] g_ethernetImages =
          { new BitmapImage(new Uri("/Resources/Individual_Ethernet_Green.png", UriKind.Relative)),
            new BitmapImage(new Uri("/Resources/Individual_Ethernet_Yellow.png", UriKind.Relative)),
            new BitmapImage(new Uri("/Resources/Individual_Ethernet_Original.png", UriKind.Relative)),
          };

        BitmapImage[] g_fuseImages =
          {
            new BitmapImage(new Uri("/Resources/Individual_Fuse_Original.png", UriKind.Relative)),
            new BitmapImage(new Uri("/Resources/Individual_Fuse_Green.png", UriKind.Relative)),
            new BitmapImage(new Uri("/Resources/Individual_Fuse_Yellow.png", UriKind.Relative)),
            new BitmapImage(new Uri("/Resources/Individual_Fuse_Red.png", UriKind.Relative)),
          };

        int g_NumFuses = 0;
        ArrayList g_FuseImagesArrayList;
        BackgroundWorker g_ethernetWorker, g_discoveryWorker;
        Socket g_ethernetSocket;
        BusyIndicator g_busyIndicator;
        string g_ipAddress;
        int g_portNumber;
        FuseStatus[] g_nodeStatuses;
        //////////////////////////////////////////////////////////////////////////

        //////////////////////////////////////////////////////////////////////////
        public MainWindow()
        {
            InitializeComponent();

            g_ethernetWorker = new BackgroundWorker();
            g_ethernetWorker.DoWork += new DoWorkEventHandler(g_ethernetWorker_DoWork);

            g_ethernetWorker.WorkerSupportsCancellation = true;

            g_discoveryWorker = new BackgroundWorker();
            g_discoveryWorker.DoWork += new DoWorkEventHandler(g_discoveryWorker_DoWork);
            g_discoveryWorker.RunWorkerCompleted += new RunWorkerCompletedEventHandler(g_discoveryWorker_RunWorkerCompleted);
            g_discoveryWorker.WorkerSupportsCancellation = true;

            g_FuseImagesArrayList = new ArrayList();
            g_busyIndicator = new BusyIndicator();
            g_nodeStatuses = new FuseStatus[256];
            g_Main_Grid.Children.Add(g_busyIndicator);

            EthernetDisconnectedIndicator();
        }
//////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////
        private void g_ethernetWorker_DoWork(object sender, DoWorkEventArgs e)
        {
            try
            {

                if (g_ethernetSocket == null || g_ethernetSocket.Connected == false)
                {
                    string ipString = g_ipAddress;
                    int port = g_portNumber;

                    IPAddress ipAddress = IPAddress.Parse(ipString);
                    IPEndPoint remoteEP = new IPEndPoint(ipAddress, port);

                    // Create a TCP/IP socket.
                    g_ethernetSocket = new Socket( ipAddress.AddressFamily,
                                                   SocketType.Stream,
                                                   ProtocolType.Tcp );

                    // Connect to the remote endpoint.
                    try
                    {
                        g_ethernetSocket.Connect(remoteEP);
                    }
                    catch (Exception)
                    {
                        BusyIndicatorOff();
                        g_ethernetSocket = null;
                        System.Windows.MessageBox.Show("Unable to connect", "Connect");
                        return;
                    }
                }

                EthernetConnectedIndicator();
                BusyIndicatorOff();

                byte[] recBytes = new Byte[1024];

                // 100 ms
                g_ethernetSocket.ReceiveTimeout = 100;

                while (true)
                {
                    // Receive Data
                    bool updated = false;
                    int numRecBytes = 0;
                    try
                    {
                      numRecBytes = g_ethernetSocket.Receive(recBytes);
                    }
                    catch (SocketException) { // TODO }


                    // Exit if exit requested
                    if (g_ethernetWorker.CancellationPending)
                    {
                        e.Cancel = true;
                        break;
                    }

                    // Process data
                    if (numRecBytes == 0)
                        continue;

                    switch (recBytes[0])
                    {
                        case (byte)MessageCodes.Heartbeat:
                            updated = Parse_Byte(numRecBytes, recBytes, false, FuseStatus.Error);
                            break;
                        case (byte)MessageCodes.FuseOK:
                            updated = Parse_Byte(numRecBytes, recBytes, true, FuseStatus.OK);
                            break;
                        case (byte)MessageCodes.FuseBlown:
                            updated = Parse_Byte(numRecBytes, recBytes, true, FuseStatus.Blown);
                            break;
                        case (byte)MessageCodes.Error1:
                            updated = Parse_Byte(numRecBytes, recBytes, true, FuseStatus.Error);
                            break;
                        case (byte)MessageCodes.Error2:
                            updated = Parse_Byte(numRecBytes, recBytes, true, FuseStatus.Error);
                            break;
                    }
                    if (updated)
                        Graphics_Update_All_Fuses();
                }
            }
            catch (Exception err)
            {
                // TODO
                System.Windows.MessageBox.Show("Weird Error", "Ethernet Worker");
            }
        }
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
        private bool Parse_Byte(int numRecBytes, byte[] recBytes, bool checkVal, FuseStatus fuseStatus, int offset = 0)
        {
            int num = 0;
            bool updated = false;

            if (checkVal)
                num = 1;

            for (int i = 1; i < numRecBytes; i++)
            {
                for (int j = 0; j < 8; j++)
                {
                    if ((recBytes[i + offset] & (1 << j)) == (num << j))
                    {
                        if (g_nodeStatuses[(i-1) * 8 + j] != fuseStatus)
                        {
                            g_nodeStatuses[(i-1) * 8 + j] = fuseStatus;
                            updated = true;
                        }
                    }
                }
            }
            return updated;
        }
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
        private void g_discoveryWorker_DoWork(object sender, DoWorkEventArgs e)
        {
            byte[] recBytes = new Byte[1024];
            byte startCode = (byte)MessageCodes.DiscoveryStart;
            byte[] startCallBytes = { startCode };
            int newNumFuses = 0;
            int numRecBytes = 0;

            for (int i = 0; i < 256; i++)
            {
                g_nodeStatuses[i] = FuseStatus.Unconnected;
            }

            // 15 s
            g_ethernetSocket.ReceiveTimeout = 15000;

            // Start Discovery Protocol
            g_ethernetSocket.Send(startCallBytes);

            bool finished = false;
            int offset = 0;
            try
            {
                while (finished == false)
                {
                    offset = 0;
                    numRecBytes = g_ethernetSocket.Receive(recBytes);

                    while (true)
                    {
                        // Wait for Ack
                        if (recBytes[offset] == (byte)MessageCodes.DiscoveryAck)
                        {
                            offset++;
                            if (offset >= numRecBytes)
                                break;
                        }

                        // Wait for number of nodes to be sent
                        if (recBytes[offset] == (byte)MessageCodes.DiscoveryNodes)
                        {
                            newNumFuses = recBytes[offset + 1];
                            offset += 2;
                            if (offset >= numRecBytes)
                                break;
                        }

                        // Wait for inactive nodes
                        if (recBytes[offset] == (byte)MessageCodes.DiscoveryInactive)
                        {
                            Parse_Byte(numRecBytes, recBytes, false, FuseStatus.OK, offset);
                            offset += 2;
                            if (offset >= numRecBytes)
                                break;
                        }

                        // Wait for Finished Message
                        if (recBytes[offset] == (byte)MessageCodes.DiscoveryFinished)
                        {
                            finished = true;
                            offset++;
                            if (finished || offset >= numRecBytes)
                                break;
                        }
                    }

                }
                e.Result = newNumFuses;
            }
            catch(Exception ex)
            {
                System.Windows.MessageBox.Show("Discovery Failed: " + ex.ToString(), "Discovery");
            }
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void g_discoveryWorker_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
        {
            if (e.Result != null)
            {
                int newNumFuses = (int)e.Result;

                // Process Results
                Graphics_Change_Fuse_Number(newNumFuses);


                Graphics_Update_All_Fuses();

                BusyIndicatorOff();
            }

            // Restart other background worker
            g_ethernetWorker.RunWorkerAsync();
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void Add_Fuse_Click(object sender, RoutedEventArgs e)
        {
            Graphics_Add_Fuse();
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void Remove_Fuse_Click(object sender, RoutedEventArgs e)
        {
            Graphics_Remove_Fuse();
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void Connect_Click(object sender, RoutedEventArgs e)
        {
            g_ipAddress = g_IPAddress_TextBox.Text;
            g_portNumber = Int32.Parse(g_PortNumber_TextBox.Text);


            // Try connecting if not already trying or connected
            BusyIndicatorOn("Connecting to Ethernet Module...");

            if (!g_ethernetWorker.IsBusy)
                g_ethernetWorker.RunWorkerAsync();
            else if (g_ethernetSocket != null && g_ethernetSocket.Connected)
                System.Windows.MessageBox.Show("Connection: Connected", "Connection Status");
            else
                System.Windows.MessageBox.Show("Connection: Pending", "Connection Status");

        }
//////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////
        private void Close_Click(object sender, RoutedEventArgs e)
        {
            if (g_ethernetSocket == null || !g_ethernetSocket.Connected)
            {
                System.Windows.MessageBox.Show("Connection not established", "Close");
                return;
            }

            g_ethernetWorker.CancelAsync();
            g_ethernetSocket.Close();
            EthernetDisconnectedIndicator();
        }
//////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////
        private void Discover_Click(object sender, RoutedEventArgs e)
        {
            if (g_ethernetSocket == null || !g_ethernetSocket.Connected)
            {
                System.Windows.MessageBox.Show("Connection not established", "Discover");
                return;
            }

            // Try connecting if not already trying or connected
            BusyIndicatorOn("Discovering new nodes...");

            g_ethernetWorker.CancelAsync();
            g_discoveryWorker.RunWorkerAsync();
```

```csharp
        }
        //////////////////////////////////////////////////////////////////////////

        //////////////////////////////////////////////////////////////////////////
        private void Busy_Click(object sender, RoutedEventArgs e)
        {
            BusyIndicatorOn("Discovering Nodes...");
        }
        //////////////////////////////////////////////////////////////////////////

        //////////////////////////////////////////////////////////////////////////
        private void Busy_Off_Click(object sender, RoutedEventArgs e)
        {
            BusyIndicatorOff();
        }
        //////////////////////////////////////////////////////////////////////////

        //////////////////////////////////////////////////////////////////////////
        private void EthernetConnectedIndicator()
        {
            if (this.g_Connection_Status_TextBox.Dispatcher.CheckAccess())
            {
                g_Connection_Status_TextBox.Text = "Connected";
                g_Connection_Status_TextBox.Foreground = new SolidColorBrush(Color.FromArgb(0xFF, 0, 0xF7, 0));
                g_Ethernet_Module_Image.Source = g_ethernetImages[(int)EthernetStatus.Connected];
            }
            else
                this.g_Connection_Status_TextBox.Dispatcher.BeginInvoke(new noparamCallback(EthernetConnectedIndicator));

        }
        //////////////////////////////////////////////////////////////////////////

        //////////////////////////////////////////////////////////////////////////
        private void EthernetDisconnectedIndicator()
        {
            if (this.g_Connection_Status_TextBox.Dispatcher.CheckAccess())
            {
                g_Connection_Status_TextBox.Text = "Disconnected";
                g_Connection_Status_TextBox.Foreground = new SolidColorBrush(Color.FromArgb(0xFF, 0xC3, 0, 0));
                g_Ethernet_Module_Image.Source = g_ethernetImages[(int)EthernetStatus.Unconnected];
            }

            else
                this.g_Connection_Status_TextBox.Dispatcher.BeginInvoke(new noparamCallback(EthernetDisconnectedIndicator));
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void EthernetErrorIndicator()
        {
            g_Connection_Status_TextBox.Text = "Error";
            g_Connection_Status_TextBox.Foreground = new SolidColorBrush(Color.FromArgb(0xFF, 0xE8, 0xFF, 0));
            g_Ethernet_Module_Image.Source = g_ethernetImages[(int)EthernetStatus.Error];
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        void BusyIndicatorOn(string message)
        {
            g_busyIndicator.BusyContent = message;

            g_busyIndicator.IsBusy = true;
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void BusyIndicatorOff()
        {
            if (this.g_busyIndicator.Dispatcher.CheckAccess())
                this.g_busyIndicator.IsBusy = false;
            else
                this.g_busyIndicator.Dispatcher.BeginInvoke(new noparamCallback(BusyIndicatorOff));
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void Graphics_Add_Fuse()
        {
            Image newFuseImage = new Image();
            newFuseImage.Source = g_fuseImages[(int)FuseStatus.Unconnected];
            newFuseImage.Width = 50;
            newFuseImage.Stretch = Stretch.Fill;

            g_ModuleHolderStackPanel.Width += 50;
            g_ModuleHolderStackPanel.Children.Add(newFuseImage);

            g_NumFuses++;
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void Graphics_Remove_Fuse()
        {
            g_ModuleHolderStackPanel.Width -= 50;
            g_ModuleHolderStackPanel.Children.RemoveAt(g_NumFuses);
            g_NumFuses--;
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void Graphics_Update_All_Fuses()
        {
            for (int i = 0; i < g_NumFuses; i++)
            {
                Graphics_Change_Fuse_Status(i, g_nodeStatuses[i]);
            }
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void Graphics_Change_Fuse_Status(int index, FuseStatus status)
        {
            if (this.g_ModuleHolderStackPanel.Dispatcher.CheckAccess())
                ((Image)g_ModuleHolderStackPanel.Children[index + 1]).Source = g_fuseImages[(int)status];
            else
                this.g_ModuleHolderStackPanel.Dispatcher.BeginInvoke(new fuseUpdateCallback(Graphics_Change_Fuse_Status), index, status);
        }
        ///////////////////////////////////////////////////////////////////////////

        ///////////////////////////////////////////////////////////////////////////
        private void Graphics_Change_Ethernet_Status(EthernetStatus status)
        {
            ((Image)g_ModuleHolderStackPanel.Children[0]).Source = g_ethernetImages[(int)status];
        }
```

```
//////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////
        private void Graphics_Change_Fuse_Number(int numFuses)
        {
            if (g_NumFuses < numFuses)
            {
                int numDifference = numFuses - g_NumFuses;
                for (int i = 0; i < numDifference; i++)
                    Graphics_Add_Fuse();
            }
            else if (g_NumFuses > numFuses)
            {
                int numDifference = g_NumFuses - numFuses;
                for (int i = 0; i < numDifference; i++)
                    Graphics_Remove_Fuse();
            }
            else // Fuse number isn't changed
            {

            }

            g_NumFuses = numFuses;
        }
//////////////////////////////////////////////////////////////////////////
    }
}
```

```
 // Justin Stark
// 4/26/2019
// STM32F042 Fuse Module Application

/**
******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
******************************************************************************
  * @attention
  *
  *
******************************************************************************
  */

///////////////////////////////////////////////////////////////////////////////
/* Includes ------------------------------------------------------------------*/
```

```c
#include "main.h"
#include "adc.h"
#include "can.h"
#include "gpio.h"
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
/* Public variables -----------------------------------------------------------*/
uint8_t Node_ID = 1;
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
void MX_FREERTOS_Init(void);
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
int main(void)
{
  /* MCU Configuration---------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_ADC_Init();
  MX_CAN_Init();

  /* Call init function for freertos objects (in freertos.c) */
  MX_FREERTOS_Init();

  /* Start scheduler */
  vTaskStartScheduler();

  /* We should never get here as control is now taken by the scheduler */

  /* Infinite loop */
  while (1) {  }
}
////////////////////////////////////////////////////////////////////////////////
```

```

////////////////////////////////////////////////////////////////////////////////
void SystemClock_Config(void)
{

  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI48;
  RCC_OscInitStruct.HSI48State = RCC_HSI48_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI48;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
  {
    Error_Handler();
  }
}
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
   if (htim->Instance == TIM1) {
     HAL_IncTick();
   }
}
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
void Error_Handler(void)
{
  /* User can add his own implementation to report the HAL error return state */
          while(1){}
}
////////////////////////////////////////////////////////////////////////////////
```

```c

////////////////////////////////////////////////////////////////////////////////
#ifdef  USE_FULL_ASSERT
void assert_failed(char *file, uint32_t line)
{
}
#endif
////////////////////////////////////////////////////////////////////////////////


/**
******************************************************************************
  * File Name          : freertos.c
  * Description        : Code for freertos applications
******************************************************************************
  * @attention
  *
  *
******************************************************************************
  */

////////////////////////////////////////////////////////////////////////////////
/* Includes ------------------------------------------------------------------*/
#include "FreeRTOS.h"
#include "task.h"
#include "main.h"
#include "cmsis_os.h"
#include "adc.h"
#include "can.h"
#include "stm32f0xx_hal_can.h"
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
/* Extern --------------------------------------------------------------------*/
extern uint8_t Node_ID;
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
/* Private typedef -----------------------------------------------------------*/
typedef enum
```

```
{
          Zero_Identifier = 0,
          Heartbeat_Identifier = 1, // send
          Fuse_OK_Identifier   = 2, // send
          Fuse_Blown_Identifier = 3, // send
          Fuse_Blown_Ack_Identifier = 4, // receive
          Error_1_Identifier = 5, // send
          Error_2_Identifier = 6, // send
          Status_Update_Identifier = 7, //receive
          Discovery_Start_Identifier = 8,
          Discovery_Ready_1_Identifier = 9, // receive
          Discovery_Ready_2_Identifier = 10, // receive
          Discovery_Ready_Response_Identifier = 11, // send
          Discovery_Call_Identifier = 12, // receive
          Disocvery_Call_Response_Identifier = 13, // send
          Discovery_End_Identifier = 14 // receive

} IdentifierCodes_Type;
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
/* Private variables ---------------------------------------------------------*/
TaskHandle_t ADC_Task_Handle;
EventGroupHandle_t ADC_MessageEventGroup;
#define BIT_ADC_BLOWN    ( 1 << 0 )
#define BIT_ADC_OK               ( 1 << 1 )
#define BIT_ERROR       ( 1 << 2 )
#define BIT_HEARTBEAT   ( 1 << 3 )
#define BIT_STATUS      ( 1 << 4 )
#define BIT_0     ( 1 << 0 )
QueueHandle_t Queue_ADC;
TaskHandle_t CAN_Task_Handle;
QueueHandle_t Queue_Fuse_Blown_Ack;
CanRxMsgTypeDef canMessage;
TaskHandle_t Discovery_Task_Handle;
QueueHandle_t Queue_Discovery;
CanRxMsgTypeDef discoveryMessage;
QueueHandle_t Queue_Status;
EventGroupHandle_t  g_ADC_Start_Event;
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
/* Private function prototypes -----------------------------------------------*/
static void ADC_Task( void *pvParameters );
static void CAN_Task( void *pvParameters );
static void Discovery_Task( void *pvParameters );
static QueueHandle_t * Switch_Queue(CanRxMsgTypeDef header);
void MX_FREERTOS_Init(void); /* (MISRA C 2004 rule 8.1) */
///////////////////////////////////////////////////////////////////////////////
```

```c

///////////////////////////////////////////////////////////////////////////////
/* Hook prototypes */
void configureTimerForRunTimeStats(void);
unsigned long getRunTimeCounterValue(void);
void vApplicationStackOverflowHook(xTaskHandle xTask, signed char *pcTaskName);
void vApplicationMallocFailedHook(void);
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
/* Functions needed when configGENERATE_RUN_TIME_STATS is on */
void configureTimerForRunTimeStats(void)
{

}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
unsigned long getRunTimeCounterValue(void)
{
          return 0;
}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void vApplicationStackOverflowHook(xTaskHandle xTask, signed char *pcTaskName)
{
   /* Run time stack overflow checking is performed if
   configCHECK_FOR_STACK_OVERFLOW is defined to 1 or 2. This hook function is
   called if a stack overflow is detected. */
          while(1) {     }
}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void vApplicationMallocFailedHook(void)
{
          while(1) {     }
}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
```

```c
void MX_FREERTOS_Init(void)
{
		BaseType_t xReturn;

		/* RTOS_QUEUES */
		Queue_ADC = xQueueCreate(2, sizeof(adc_set_t));
		if(Queue_ADC == NULL)
		{
			// TODO: error check  (Insufficient Heap)
		}
		Queue_Discovery  = xQueueCreate(1, sizeof(CanRxMsgTypeDef));
		if(Queue_Discovery == NULL)
		{
			// TODO: error check  (Insufficient Heap)
		}
		Queue_Fuse_Blown_Ack  = xQueueCreate(1, sizeof(CanRxMsgTypeDef));
		if(Queue_Fuse_Blown_Ack == NULL)
		{
			// TODO: error check  (Insufficient Heap)
		}
		Queue_Status = xQueueCreate(1, sizeof(CanRxMsgTypeDef));
		if(Queue_Status == NULL)
		{
			// TODO: error check  (Insufficient Heap)
		}

		/* Create the thread(s) */
		xReturn = xTaskCreate( ADC_Task, "ADC_Task", ( uint16_t )configMINIMAL_STACK_SIZE*1, NULL, ( UBaseType_t )configMAX_PRIORITIES - 2, &ADC_Task_Handle );
		if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
		{
			//TODO: LOG
		}

		xReturn = xTaskCreate( CAN_Task, "CAN_Task", ( uint16_t )configMINIMAL_STACK_SIZE*1, NULL, ( UBaseType_t )configMAX_PRIORITIES - 4, &CAN_Task_Handle );
		if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
		{
			//TODO: LOG
		}

		xReturn = xTaskCreate( Discovery_Task, "Discovery_Task", ( uint16_t )configMINIMAL_STACK_SIZE*1, NULL, ( UBaseType_t )configMAX_PRIORITIES - 3, &Discovery_Task_Handle );
		if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
		{
			//TODO: LOG
		}
}
///////////////////////////////////////////////////////////////////////////////

```

```c
///////////////////////////////////////////////////////////////////////////////
static void ADC_Task( void *pvParameters )
{
        EventBits_t uxBits;
        adc_set_t adcSet;

        /* Attempt to create the event group. */
        ADC_MessageEventGroup = xEventGroupCreate();
        if(ADC_MessageEventGroup == NULL)
        {
                // TODO: error check  (Insufficient Heap)
        }
        g_ADC_Start_Event = xEventGroupCreate();
        if(g_ADC_Start_Event == NULL)
        {
                // TODO: error check  (Insufficient Heap)
        }

        for(;;)
        {
                ADC_Start();

                const TickType_t xDelay = 2000 / portTICK_PERIOD_MS;
                vTaskDelay( xDelay );


                // Queue will block until conversion is complete
                if( xQueueReceive( Queue_ADC, &( adcSet ), ( TickType_t ) portMAX_DELAY ) )
                {
                        if( adcSet.valueOne <= 1.5 * adcSet.valueTwo && adcSet.valueOne >= .5 * adcSet.valueTwo)
                        {
                                // Good Fuse
                                xEventGroupSetBits( ADC_MessageEventGroup,
                                                    BIT_ADC_OK );
                        }
                        else
                        {
                                // Bad Fuse
                                xEventGroupSetBits( ADC_MessageEventGroup,
                                                    BIT_ADC_BLOWN );
                        }
                }
                else
                {
                        xEventGroupSetBits( ADC_MessageEventGroup,
                                                                        BIT_ERROR );
                }
        }
```

```
}
/////////////////////////////////////////////////////////////////////////////


/////////////////////////////////////////////////////////////////////////////
static void CAN_Task( void *pvParameters )
{
            EventBits_t uxBits;
            BaseType_t returnType;
            uint8_t dataNode[1];

            for(;;)
            {
                    /* Wait a maximum of 50 days for either bit 0-2 to be set within
                    the event group.  Clear the bits before exiting. */
                    uxBits = xEventGroupWaitBits(
                    ADC_MessageEventGroup,    /* The event group being tested. */
                    BIT_STATUS | BIT_ADC_BLOWN | BIT_ADC_OK | BIT_ERROR, /* The bits within the event group to wait for. */
                    pdTRUE,         /* BIT_ADC_BLOWN | BIT_ADC_OK | BIT_ADC_ERROR should be cleared before returning. */
                    pdFALSE,        /* Any bit will trigger */
                    portMAX_DELAY );/* Wait a maximum of 50 days */

                    if( (uxBits & BIT_ADC_BLOWN) == BIT_ADC_BLOWN)
                    {
                            xQueueReset(Queue_Fuse_Blown_Ack);

                            dataNode[0] = Node_ID;
                            CAN_Message_Send( Fuse_Blown_Identifier,
                                                          1, // Data length
                                                                 dataNode );

                            returnType = xQueueReceive( Queue_Fuse_Blown_Ack, &canMessage , ( TickType_t ) 250 );

                            if(returnType == pdFAIL)
                            {
                                    dataNode[0] = Node_ID;
                                    CAN_Message_Send( Fuse_Blown_Identifier,
                                                                                 1, // Data length
                                                                                 dataNode );

                                    returnType = xQueueReceive( Queue_Fuse_Blown_Ack, &canMessage , ( TickType_t ) 250 );
```

```c

                // TODO: send/log error
                if(returnType == pdFAIL)
                {
                }
            }
        }

        if( (uxBits & BIT_ADC_OK) == BIT_ADC_OK)
        {
            dataNode[0] = Node_ID;
            CAN_Message_Send( Fuse_OK_Identifier,
                              1, // Data length
                              dataNode );
        }

        if( (uxBits & BIT_ERROR) == BIT_ERROR)
        {
            dataNode[0] = Node_ID;
            CAN_Message_Send( Error_1_Identifier,
                          1, // Data length
                              dataNode );
        }

        if( (uxBits & BIT_STATUS) == BIT_STATUS)
        {
            // Update Status
        }
    }
}
/////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////
static void Discovery_Task( void *pvParameters )
{
    GPIO_PinState pinState;
    bool currentNode = false;
    bool readyOne = false;
    bool readyTwo = false;
    BaseType_t returnType;
    uint8_t dataNode[1];

    CAN_Receive_Setup();

    for(;;)
    {
        // Discovery Protocol

        // Wait for received messages
        returnType = xQueueReceive( Queue_Discovery, &discoveryMessage , ( TickType_t ) portMAX_DELAY );

        if(returnType == pdFAIL)
```

```
                            continue;

                    if( (discoveryMessage.StdId == Discovery_Ready_1_Identifier) && (currentNode == true) && (readyOne == false) )
                    {
                            HAL_GPIO_WritePin( DISC_OUTPUT_PORT, DISC_OUTPUT_1, GPIO_PIN_SET );

                            dataNode[0] = Node_ID;
                            CAN_Message_Send( Discovery_Ready_Response_Identifier,
                            1, // Data length
                            dataNode );
                            readyOne = true;

                    }
                    else if( (discoveryMessage.StdId == Discovery_Ready_2_Identifier) && (currentNode == true) && (readyTwo == false) )
                    {
                            HAL_GPIO_WritePin( DISC_OUTPUT_PORT, DISC_OUTPUT_2, GPIO_PIN_SET );

                            dataNode[0] = Node_ID;
                            CAN_Message_Send( Discovery_Ready_Response_Identifier,
                                                                  1, // Data length
                                                                  dataNode );
                            readyTwo = true;

                    }
                    else if( (discoveryMessage.StdId == Discovery_Call_Identifier)  && (currentNode == false) )
                    {
                            // Check both pins for input
                            pinState = HAL_GPIO_ReadPin( DISC_INPUT_PORT, DISC_INPUT_1 ) | HAL_GPIO_ReadPin( DISC_INPUT_PORT, DISC_INPUT_2 );

                            if( pinState == GPIO_PIN_SET )
                            {

                                    Node_ID = discoveryMessage.Data[0];

                                    dataNode[0] = Node_ID;
                                    CAN_Message_Send( Disocvery_Call_Response_Identifier,
                                                                          1, // Data length
                                                                          dataNode );

                                    currentNode = true;
```

```c
            }

        }
        else if( (discoveryMessage.StdId == Discovery_End_Identifier) )
        {
            currentNode = false;
            readyOne = false;
            readyTwo = false;

            HAL_GPIO_WritePin( DISC_OUTPUT_PORT, DISC_OUTPUT_1, GPIO_PIN_RESET);
            HAL_GPIO_WritePin( DISC_OUTPUT_PORT, DISC_OUTPUT_2, GPIO_PIN_RESET);

            xQueueReset(Queue_Discovery);

            // Start the ADC task
            xEventGroupSetBits(
                                g_ADC_Start_Event,    /* The event group being updated. */
                                BIT_0                 /* The bits being set. */
                              );
        }
        else if( (discoveryMessage.StdId == Discovery_Start_Identifier) )
        {
            currentNode = false;
            readyOne = false;
            readyTwo = false;

            /* Clear bit 0 in g_Start_Event. */
            xEventGroupClearBits(
                                g_ADC_Start_Event,  /* The event group being updated. */
                                BIT_0               /* The bits being cleared. */
                                );
        }
    }
}
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
void HAL_CAN_RxCpltCallback(CAN_HandleTypeDef* hcan)
{
    CanRxMsgTypeDef canMessage;
    BaseType_t xHigherPriorityTaskWoken1 = pdFALSE, xHigherPriorityTaskWoken2 = pdFALSE;
    uint8_t fifo = 0;


    if(hcan->pRxMsg->StdId == 0)
```

```
        {
                canMessage = *(hcan->pRx1Msg);
                fifo = 1;
        }
        else
                canMessage = *(hcan->pRxMsg);

        QueueHandle_t *pqueue = Switch_Queue(canMessage);

        if(pqueue == &Queue_Status)
        {

                /* xHigherPriorityTaskWoken must be initialised to pdFALSE. */
                xHigherPriorityTaskWoken1 = pdFALSE;

                /* Set bit 0 and bit 4 in xEventGroup. */
                xEventGroupSetBitsFromISR(
                        ADC_MessageEventGroup,   /* The event group being updated. */
                        BIT_STATUS, /* The bits being set. */
                        &xHigherPriorityTaskWoken1
                                                                  );
        }

        xQueueSendFromISR( *pqueue,
                                                &canMessage,
                                                &xHigherPriorityTaskWoken2 );


        if(fifo == 0 && HAL_CAN_Receive_IT(hcan, CAN_FIFO0) != HAL_OK)
        {
                Error_Handler();
        }
        else if(fifo == 1 && HAL_CAN_Receive_IT(hcan, CAN_FIFO1) != HAL_OK)
        {
                Error_Handler();
        }

        if( ( xHigherPriorityTaskWoken1 | xHigherPriorityTaskWoken2 ) != 0 )
        {
                vPortYield();
        }
}
/////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////
static QueueHandle_t * Switch_Queue(CanRxMsgTypeDef header)
{
```

```c
        QueueHandle_t *hqueue = &Queue_Status;

        switch(header.StdId)
        {
                case Fuse_Blown_Ack_Identifier:
                        hqueue = &Queue_Fuse_Blown_Ack;
                        break;
                case Status_Update_Identifier:
                        hqueue = &Queue_Status;
                        break;
                case Discovery_Ready_1_Identifier:
                        hqueue = &Queue_Discovery;
                        break;
                case Discovery_Ready_2_Identifier:
                        hqueue = &Queue_Discovery;
                        break;
                case Discovery_Call_Identifier:
                        hqueue = &Queue_Discovery;
                        break;
                case Discovery_End_Identifier:
                        hqueue = &Queue_Discovery;
                        break;
                case Discovery_Start_Identifier:
                        hqueue = &Queue_Discovery;
                        break;
                default:
                        break;
        }
        return hqueue;
}
////////////////////////////////////////////////////////////////////////////////

/**
  ******************************************************************************
  * @file    stm32f0xx_it.c
  * @brief   Interrupt Service Routines.
  ******************************************************************************
  * @attention
  *
  *
  ******************************************************************************
  */

////////////////////////////////////////////////////////////////////////////////
/* Includes -------------------------------------------------------------------*/
#include "main.h"
#include "stm32f0xx_it.h"
```

```c
#include "cmsis_os.h"
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
/* External variables --------------------------------------------------------*/
extern CAN_HandleTypeDef hcan;
extern WWDG_HandleTypeDef hwwdg;
extern TIM_HandleTypeDef htim1;
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
void NMI_Handler(void)
{
}
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
void HardFault_Handler(void)
{
  while (1)
  {
  }
}
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
void WWDG_IRQHandler(void)
{
  HAL_WWDG_IRQHandler(&hwwdg);
}
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
void TIM1_BRK_UP_TRG_COM_IRQHandler(void)
{
  HAL_TIM_IRQHandler(&htim1);
}
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
void CEC_CAN_IRQHandler(void)
{
  HAL_CAN_IRQHandler(&hcan);
}
///////////////////////////////////////////////////////////////////////////////

/**
  ******************************************************************************
  * File Name          : stm32f0xx_hal_msp.c
  * Description        : This file provides code for the MSP Initialization
  *                      and de-Initialization codes.
  ******************************************************************************
  * @attention
  *
  *
  ******************************************************************************
  */

///////////////////////////////////////////////////////////////////////////////
/* Includes ------------------------------------------------------------------*/
#include "main.h"
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void HAL_MspInit(void)
{
  __HAL_RCC_SYSCFG_CLK_ENABLE();
  __HAL_RCC_PWR_CLK_ENABLE();

  /* System interrupt init*/
  /* PendSV_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(PendSV_IRQn, 3, 0);
}
///////////////////////////////////////////////////////////////////////////////


/**
  ******************************************************************************
  * @file    stm32f0xx_hal_timebase_TIM.c
  * @brief   HAL time base based on the hardware TIM.
  ******************************************************************************
  * @attention
  *
  *
******************************************************************************
  */

/* Includes ------------------------------------------------------------------*/
#include "stm32f0xx_hal.h"
#include "stm32f0xx_hal_tim.h"


/* Private variables ---------------------------------------------------------*/
TIM_HandleTypeDef        htim1;

///////////////////////////////////////////////////////////////////////////////
HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
  RCC_ClkInitTypeDef    clkconfig;
  uint32_t              uwTimclock = 0;
  uint32_t              uwPrescalerValue = 0;
  uint32_t              pFLatency;

  /*Configure the TIM1 IRQ priority */
  HAL_NVIC_SetPriority(TIM1_BRK_UP_TRG_COM_IRQn, TickPriority ,0);

  /* Enable the TIM1 global Interrupt */
  HAL_NVIC_EnableIRQ(TIM1_BRK_UP_TRG_COM_IRQn);

  /* Enable TIM1 clock */
  __HAL_RCC_TIM1_CLK_ENABLE();

  /* Get clock configuration */
  HAL_RCC_GetClockConfig(&clkconfig, &pFLatency);

  /* Compute TIM1 clock */
  uwTimclock = HAL_RCC_GetPCLK1Freq();

  /* Compute the prescaler value to have TIM1 counter clock equal to 1MHz */
  uwPrescalerValue = (uint32_t) ((uwTimclock / 1000000) - 1);

  /* Initialize TIM1 */
  htim1.Instance = TIM1;

  /* Initialize TIMx peripheral as follow:
  + Period = [(TIM1CLK/1000) - 1]. to have a (1/1000) s time base.
  + Prescaler = (uwTimclock/1000000 - 1) to have a 1MHz counter clock.
  + ClockDivision = 0
  + Counter direction = Up
  */
  htim1.Init.Period = (1000000 / 1000) - 1;
  htim1.Init.Prescaler = uwPrescalerValue;
  htim1.Init.ClockDivision = 0;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  if(HAL_TIM_Base_Init(&htim1) == HAL_OK)
  {
    /* Start the TIM time Base generation in interrupt mode */
    return HAL_TIM_Base_Start_IT(&htim1);
  }

  /* Return function status */
  return HAL_ERROR;
}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void HAL_SuspendTick(void)
{
  /* Disable TIM1 update Interrupt */
  __HAL_TIM_DISABLE_IT(&htim1, TIM_IT_UPDATE);
}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void HAL_ResumeTick(void)
{
  /* Enable TIM1 Update interrupt */
  __HAL_TIM_ENABLE_IT(&htim1, TIM_IT_UPDATE);
}
///////////////////////////////////////////////////////////////////////////////

/**
******************************************************************************
  * File Name          : gpio.c
  * Description        : This file provides code for the configuration
  *                      of all used GPIO pins.
******************************************************************************
  * @attention
  *
  *
******************************************************************************
  */

///////////////////////////////////////////////////////////////////////////////
/* Includes ------------------------------------------------------------------*/
```

```
#include "gpio.h"
//////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////////////
void MX_GPIO_Init(void)
{

            GPIO_InitTypeDef GPIO_InitStruct = {0};

            /* GPIO Ports Clock Enable */
            __HAL_RCC_GPIOF_CLK_ENABLE();
            __HAL_RCC_GPIOA_CLK_ENABLE();
            __HAL_RCC_GPIOB_CLK_ENABLE();

            /*Configure GPIO pin Output Level */
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5|GPIO_PIN_6, GPIO_PIN_RESET);

            /*Configure GPIO pin Output Level */
            HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);

            /*Configure GPIO pins : PB4 PB5 */
            GPIO_InitStruct.Pin = DISC_INPUT_1|DISC_INPUT_2;
            GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
            GPIO_InitStruct.Pull = GPIO_NOPULL;
            HAL_GPIO_Init(DISC_INPUT_PORT, &GPIO_InitStruct);

            /*Configure GPIO pins : PA5 PA6 */
            GPIO_InitStruct.Pin = DISC_OUTPUT_1|DISC_OUTPUT_2;
            GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
            GPIO_InitStruct.Pull = GPIO_NOPULL;
            GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
            HAL_GPIO_Init(DISC_OUTPUT_PORT, &GPIO_InitStruct);

            /*Configure GPIO pin : PtPin */
            GPIO_InitStruct.Pin = LD3_Pin;
            GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
            GPIO_InitStruct.Pull = GPIO_NOPULL;
            GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
            HAL_GPIO_Init(LD3_GPIO_Port, &GPIO_InitStruct);

}
//////////////////////////////////////////////////////////////////////////////

/**
******************************************************************************
  * File Name          : CAN.c
  * Description        : This file provides code for the configuration
  *                      of the CAN instances.
```

```c
******************************************************************************
  * @attention
  *
  *
******************************************************************************
  */

////////////////////////////////////////////////////////////////////////////////
/* Includes ------------------------------------------------------------------*/
#include "can.h"
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
typedef enum
{
          Zero_Identifier = 0,
          Heartbeat_Identifier = 1,
          Fuse_OK_Identifier   = 2,
          Fuse_Blown_Identifier = 3,
          Fuse_Blown_Ack_Identifier = 4,
          Error_1_Identifier = 5,
          Error_2_Identifier = 6,
          Status_Update_Identifier = 7,
          Discovery_Start_Identifier = 8,
          Discovery_Ready_1_Identifier = 9,
          Discovery_Ready_2_Identifier = 10,
          Discovery_Ready_Response_Identifier = 11,
          Discovery_Call_Identifier = 12,
          Disocvery_Call_Response_Identifier = 13,
          Discovery_End_Identifier = 14

} IdentifierCodes_Type;
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
CAN_HandleTypeDef hcan;
CanRxMsgTypeDef   Rx0;
CanRxMsgTypeDef   Rx1;
CanTxMsgTypeDef   Tx;
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
/* CAN init function */
void MX_CAN_Init(void)
```

```
{
  hcan.Instance = CAN;
  hcan.Init.Prescaler = 24;
  hcan.Init.Mode = CAN_MODE_NORMAL;
  hcan.Init.SJW = CAN_SJW_1TQ;
  hcan.Init.BS1 = CAN_BS1_13TQ;
  hcan.Init.BS2 = CAN_BS2_2TQ;
  hcan.Init.TTCM = DISABLE;
  hcan.Init.ABOM = ENABLE; //DiSABLE;
  hcan.Init.AWUM = DISABLE;
  hcan.Init.NART = DISABLE;
  hcan.Init.RFLM = DISABLE;
  hcan.Init.TXFP = DISABLE;
  if (HAL_CAN_Init(&hcan) != HAL_OK)
  {
    Error_Handler();
  }

  CAN_Filter_Setup();
}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void HAL_CAN_MspInit(CAN_HandleTypeDef* canHandle)
{

  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(canHandle->Instance==CAN)
  {
    /* CAN clock enable */
    __HAL_RCC_CAN1_CLK_ENABLE();

    __HAL_RCC_GPIOA_CLK_ENABLE();
    /**CAN GPIO Configuration
    PA11     ------> CAN_RX
    PA12     ------> CAN_TX
    */
    GPIO_InitStruct.Pin = GPIO_PIN_11|GPIO_PIN_12;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF4_CAN;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    /* CAN interrupt Init */
    HAL_NVIC_SetPriority(CEC_CAN_IRQn, 3, 0);
    HAL_NVIC_EnableIRQ(CEC_CAN_IRQn);

  }
}
///////////////////////////////////////////////////////////////////////////////

```

```
///////////////////////////////////////////////////////////////////////////////
void HAL_CAN_MspDeInit(CAN_HandleTypeDef* canHandle)
{
  if(canHandle->Instance==CAN)
  {

    /* Peripheral clock disable */
    __HAL_RCC_CAN1_CLK_DISABLE();

    /**CAN GPIO Configuration
    PA11     ------> CAN_RX
    PA12     ------> CAN_TX
    */
    HAL_GPIO_DeInit(GPIOA, GPIO_PIN_11|GPIO_PIN_12);

    /* CAN interrupt Deinit */
    HAL_NVIC_DisableIRQ(CEC_CAN_IRQn);
  }
}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void CAN_Message_Send(uint32_t stdID, uint32_t length, uint8_t *data)
{
          Tx.StdId = stdID;
          Tx.RTR = CAN_RTR_DATA;
          Tx.IDE = CAN_ID_STD;
          Tx.DLC = length;

          for(int i = 0; i < length; i++)
          {
                  Tx.Data[i] = data[i];
          }

          /* Request transmission */
          hcan.pTxMsg = &Tx;

          if(HAL_CAN_Transmit(&hcan, 50) != HAL_OK)
          {
                  // TODO
          }
}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void CAN_Receive_Setup(void)
{
  hcan.pRxMsg = &Rx0;
  hcan.pRx1Msg = &Rx1;

          if(HAL_CAN_Receive_IT(&hcan, CAN_FIFO0) != HAL_OK)
```

```c
		{
			Error_Handler();
		}
		if(HAL_CAN_Receive_IT(&hcan, CAN_FIFO1) != HAL_OK)
		{
			Error_Handler();
		}
}
/////////////////////////////////////////////////////////////////////////////////


/////////////////////////////////////////////////////////////////////////////////
void CAN_Filter_Setup(void)
{
		CAN_Filter_Setup_LL( Fuse_Blown_Ack_Identifier, CAN_FILTER_FIFO0, 0);
		CAN_Filter_Setup_LL( Status_Update_Identifier, CAN_FILTER_FIFO0, 1);
		CAN_Filter_Setup_LL( Discovery_Start_Identifier, CAN_FILTER_FIFO1, 2);
		CAN_Filter_Setup_LL( Discovery_Ready_1_Identifier, CAN_FILTER_FIFO1, 3);
		CAN_Filter_Setup_LL( Discovery_Ready_2_Identifier, CAN_FILTER_FIFO1, 4);
		CAN_Filter_Setup_LL( Discovery_Call_Identifier, CAN_FILTER_FIFO1, 5);
		CAN_Filter_Setup_LL( Discovery_End_Identifier, CAN_FILTER_FIFO1, 6);

}
/////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////
void CAN_Filter_Setup_LL(uint32_t StdID, uint32_t fifoSelection, uint32_t filterBank)
{
		CAN_FilterConfTypeDef  sFilterConfig;

		sFilterConfig.FilterNumber = filterBank;
		sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
		sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
		sFilterConfig.FilterIdHigh = StdID << 5;
		sFilterConfig.FilterIdLow = 0x0000;
		sFilterConfig.FilterMaskIdHigh = 0xFFFF;
		sFilterConfig.FilterMaskIdLow = 0xFFFF;
		sFilterConfig.FilterFIFOAssignment = fifoSelection;
		sFilterConfig.FilterActivation = ENABLE;

		if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
		{
			/* Filter configuration Error */
			Error_Handler();
```

```c
          }
}
//////////////////////////////////////////////////////////////////////////////

/**
******************************************************************************
  * File Name          : ADC.c
  * Description        : This file provides code for the configuration
  *                      of the ADC instances.
******************************************************************************
  * @attention
  *
  *
******************************************************************************
  */
//////////////////////////////////////////////////////////////////////////////
/* Includes ------------------------------------------------------------------*/
#include "adc.h"
//////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////
ADC_HandleTypeDef hadc;
extern QueueHandle_t Queue_ADC;
//////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////
/* ADC init function */
void MX_ADC_Init(void)
{
  ADC_ChannelConfTypeDef sConfig = {0};

  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and number of conversion)
  */
  hadc.Instance = ADC1;
  hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
  hadc.Init.Resolution = ADC_RESOLUTION_12B;
  hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;
  hadc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc.Init.LowPowerAutoWait = DISABLE;
  hadc.Init.LowPowerAutoPowerOff = DISABLE;
  hadc.Init.ContinuousConvMode = DISABLE;
  hadc.Init.DiscontinuousConvMode = DISABLE;
  hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc.Init.DMAContinuousRequests = DISABLE;
  hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  if (HAL_ADC_Init(&hadc) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure for the selected ADC regular channel to be converted.
  */
  sConfig.Channel = ADC_CHANNEL_3;
  sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
  sConfig.SamplingTime = ADC_SAMPLETIME_239CYCLES_5;
  if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure for the selected ADC regular channel to be converted.
  */
  sConfig.Channel = ADC_CHANNEL_4;
  if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
}
/////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////
void HAL_ADC_MspInit(ADC_HandleTypeDef* adcHandle)
{

  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(adcHandle->Instance==ADC1)
  {
    /* ADC1 clock enable */
    __HAL_RCC_ADC1_CLK_ENABLE();

    __HAL_RCC_GPIOA_CLK_ENABLE();
    /**ADC GPIO Configuration
    PA0     ------> ADC_IN0
    PA1     ------> ADC_IN1
    */
    GPIO_InitStruct.Pin = GPIO_PIN_4|GPIO_PIN_3;
    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    /* ADC1 interrupt Init */
    HAL_NVIC_SetPriority(ADC1_IRQn, 3, 0);
    HAL_NVIC_EnableIRQ(ADC1_IRQn);

  }
}
```

```c
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void HAL_ADC_MspDeInit(ADC_HandleTypeDef* adcHandle)
{

  if(adcHandle->Instance==ADC1)
  {
    /* Peripheral clock disable */
    __HAL_RCC_ADC1_CLK_DISABLE();

    /**ADC GPIO Configuration
    PA0     ------> ADC_IN0
    PA1     ------> ADC_IN1
    */
    HAL_GPIO_DeInit(GPIOA, GPIO_PIN_4|GPIO_PIN_3);

    /* ADC interrupt Deinit */
          HAL_NVIC_DisableIRQ(ADC1_IRQn);
  }
}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void ADC_Start(void)
{
          HAL_ADC_Start_IT( &hadc );
}
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
          static adc_set_t adcSet;
          static int flag = 0;
          BaseType_t xHigherPriorityTaskWoken;
          static int finished = 0;

          /* We have not woken a task at the start of the ISR. */
          xHigherPriorityTaskWoken = pdFALSE;

          if( __HAL_ADC_GET_FLAG(hadc, ADC_FLAG_EOC) != 0)
          {
                  if(flag == 0)
                  {
                          adcSet.valueOne = HAL_ADC_GetValue(hadc);
                          flag = 1;
                  }
                  else if(flag == 1)
                  {
                          adcSet.valueTwo = HAL_ADC_GetValue(hadc);
                          flag = 0;
                          finished = 1;
                  }

          }
          if(finished == 1)
          {
                  finished = 0;
                   xQueueSendFromISR( Queue_ADC,
                                                      &adcSet,
                                                      &xHigherPriorityTaskWoken);


                  if( xHigherPriorityTaskWoken )
                  {
                          vPortYield();
                  }
          }
}
/////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////
void ADC1_IRQHandler ( void )
{
          HAL_ADC_IRQHandler(&hadc);
}
/////////////////////////////////////////////////////////////////////////////////
```

```
// Justin Stark
// 4/26/2019
// STM32F746 Ethernet Module

/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  *
  ******************************************************************************
```

```c
  */

////////////////////////////////////////////////////////////////
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "can.h"
#include "gpio.h"
#include "task.h"
#include "stm32f7xx_hal.h"
#include "stm32f7xx_hal_flash.h"
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
/* Private function prototypes -----------------------------*/
void SystemClock_Config(void);
void MX_FREERTOS_Init(void);
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
int main(void)
{
  /* MCU Configuration---------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  if(HAL_Init() != HAL_OK)
  {
    Error_Handler();
  }

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();

  /* Call init function for freertos objects (in freertos.c) */
  MX_FREERTOS_Init();

  /* Start scheduler */
  vTaskStartScheduler();

  /* We should never get here as control is now taken by the scheduler */
  /* Infinite loop */
  while (1){ }
}
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};

  /** Configure LSE Drive Capability
  */
  HAL_PWR_EnableBkUpAccess();
  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_BYPASS;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 4;
  RCC_OscInitStruct.PLL.PLLN = 216;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 9;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /** Activate the Over-Drive mode
  */
  if (HAL_PWREx_EnableOverDrive() != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_7) != HAL_OK)
  {
    Error_Handler();
  }
  PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_USART3|RCC_PERIPHCLK_CLK48;
  PeriphClkInitStruct.Usart3ClockSelection = RCC_USART3CLKSOURCE_PCLK1;
  PeriphClkInitStruct.Clk48ClockSelection = RCC_CLK48SOURCE_PLL;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Instance == TIM1) {
    HAL_IncTick();
  }
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void Error_Handler(void)
{
  /* User can add his own implementation to report the HAL error return state */
          while(1) {      }
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
#ifdef  USE_FULL_ASSERT
void assert_failed(uint8_t *file, uint32_t line)
{
}
#endif /* USE_FULL_ASSERT */
//////////////////////////////////////////////////////////////



/**
  ******************************************************************************
  * @file           : debug.c
  * @brief          : Debugging Task
  ******************************************************************************
  * @attention
  *
  *
  ******************************************************************************
  */

//////////////////////////////////////////////////////////////
#include "debug.h"
#include "main.h"
#include "stdio.h"
#include "stdint.h"
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
static void Debug_Task( void *pvParameters );
//////////////////////////////////////////////////////////////

```

```c
///////////////////////////////////////////////////////////////
TaskHandle_t Debug_Task_Handle;
QueueHandle_t queueDebug;
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void Setup_Debug(void)
{

          queueDebug = xQueueCreate(5, sizeof(Message));

          BaseType_t xReturn;
          xReturn = xTaskCreate( Debug_Task,
                                        "Debug_Task",
                   ( uint16_t )configMINIMAL_STACK_SIZE * 3,
                   NULL,
                   ( UBaseType_t )DEBUG_TASK_PRIORITY,
                   &Debug_Task_Handle );
          if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
          {
                  Error_Handler();
          }
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
static void Debug_Task( void *pvParameters )
{
          HAL_StatusTypeDef status = HAL_ERROR;

          PC_Setup_Serial();
          char test[] = "Connection Established\n";
          PC_Send_String((uint8_t *) test, strlen(test));


          if(queueDebug == NULL)
          {
                  char string[] = "Error creating queue Debug Task\n";

                  status = HAL_ERROR;

                  while(status != HAL_OK)
                  {
                          status = PC_Send_String((uint8_t *) string,
strlen(string));
                  }
          }
          Message recMsg;
          for(;;)
          {
                  xQueueReceive( queueDebug, (void *) &recMsg ,
(TickType_t) portMAX_DELAY  );
```

```c

                        status = HAL_ERROR;
                        while(status != HAL_OK)
                        {
                                status = PC_Send_String((uint8_t *) recMsg.message, strlen(recMsg.message));
                        }
                }
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void FreeRTOS_debug_printf(uint8_t* string, uint16_t size)
{
                const char *pcTaskName;
                static int index = 0;

                Message debugMsg;

                const char pcNoTask[] = "None";
                int xLength = 0;
                static long xMessageNumber = 0;

                /* Additional info to place at the start of the log. */
                if( xTaskGetSchedulerState() != taskSCHEDULER_NOT_STARTED )
                {
                        pcTaskName = pcTaskGetName( NULL );
                }
                else
                {
                        pcTaskName = pcNoTask;
                }

                xLength = snprintf(  (char *) debugMsg.message,
                                                dlMAX_PRINT_STRING_LENGTH,
                                                "Debug %lu %lu [%s] ",
                                                xMessageNumber++,
                                                ( unsigned long ) xTaskGetTickCount(),
                                                pcTaskName );

                snprintf( (char *) debugMsg.message + xLength,
                                  dlMAX_PRINT_STRING_LENGTH - xLength,
                                  "%s",
                                  string);

                xQueueSend( queueDebug ,
                                 (void*) &debugMsg,
                                 (TickType_t) 0 );

                if(index == 5)
                        index = 0;
                index++;

}
```

```
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void FreeRTOS_printf(uint8_t* string, uint16_t size)
{
          const char *pcTaskName;
          static int index = 0;

          Message Msg;

          const char pcNoTask[] = "None";
          int xLength = 0;
          static long xMessageNumber = 0;

          /* Additional info to place at the start of the log. */
          if( xTaskGetSchedulerState() != taskSCHEDULER_NOT_STARTED )
          {
                  pcTaskName = pcTaskGetName( NULL );
          }
          else
          {
                  pcTaskName = pcNoTask;
          }

          xLength = snprintf( (char *) Msg.message,
                                                dlMAX_PRINT_STRING_LENGTH,
                                                "%lu %lu [%s] ",
                                                xMessageNumber++,
                                                ( unsigned long ) xTaskGetTickCount(),
                                                pcTaskName );

          snprintf( (char *) Msg.message + xLength,
                           dlMAX_PRINT_STRING_LENGTH - xLength,
                           "%s",
                           string);


          xQueueSend( queueDebug ,
                                   (void *) &Msg,
                                   (TickType_t) 0 );

          if(index == 5)
                         index = 0;

          index++;
}
//////////////////////////////////////////////////////////////



/**
******************************************************************************
```

```c
  * @file           : Discovery.c
  * @brief          : Discovery Task
******************************************************************************
  * @attention
  *
  *
******************************************************************************
  */

//////////////////////////////////////////////////////////////
#include "main.h"
#include "stm32f7xx_hal.h"
#include "stm32f7xx_nucleo_144.h"
#include "can.h"
#include "FreeRTOS.h"
#include "queue.h"
#include "event_groups.h"
#include "network_application.h"
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
// Discovery Protocol Functions
void Discovery_Reset_Nodes(void);
bool Discovery_Fist_Node_First_Output(uint8_t Node_ID);
bool Discovery_Fist_Node_Second_Output(uint8_t Node_ID);
bool Discovery_First_Output(uint8_t Node_ID);
bool Discovery_Second_Output(uint8_t Node_ID);
void Discovery_Start(void);
void Discovery_Ready(IdentifierCodes_Type code, uint8_t Node_ID);
bool Discovery_Call(uint8_t Node_ID);
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
extern QueueHandle_t queueDiscovery;
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
EventGroupHandle_t g_Discovery_Event_Group;
//////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////
void CAN_Discovery_Task( void *pvParameters )
{

          bool firstFound = true;
          bool endRound = false;
          uint8_t Node_ID = 0;
          bool nodes[255] = {false};
          EventBits_t uxBits;
          uint8_t message[64] = {0};
          int numBytes = 0;
```

```

            /* Attempt to create the event group. */
            g_Discovery_Event_Group = xEventGroupCreate();

            /* Was the event group created successfully? */
            if( g_Discovery_Event_Group == NULL )
            {
                    /* The event group was not created because there was insufficient
                    FreeRTOS heap available. */
                    Error_Handler();
            }

            /* Block for 500ms. */
            const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

            /* Initialize all configured peripherals */
            BSP_PB_Init(BUTTON_USER, BUTTON_MODE_GPIO);
            BSP_LED_Init(LED1);
            BSP_LED_Init(LED2);

            for(;;)
            {
                    // Wait a maximum of 50 days for either bit 0 or bit 1 to be set within
                    // the event group.  Clear the bits before exiting.
                    uxBits = xEventGroupWaitBits(
                            g_Discovery_Event_Group,   // The event group being tested.
                            BIT_0,               // The bits within the event group to wait for.
                            pdTRUE,        // BIT_0 should be cleared before returning.
                            pdFALSE,       // Any bit will trigger
                            portMAX_DELAY );// Wait a maximum of 50 days

                    if( uxBits == 0 )
                            continue;

                    BSP_LED_On(LED2);

                    message[0] = DiscoveryAck;
                    Notify_Ethernet_Listeners(
                            message,
                            1
                            );

                    ///////////////////////////////////////////////////////////////////
                    // Start nodes discovery mode
                    Discovery_Start();

```

```
                ///////////////////////////////////////////////////////////////////

                ///////////////////////////////////////////////////////////////////
                vTaskDelay( xDelay );
                ///////////////////////////////////////////////////////////////////

                ///////////////////////////////////////////////////////////////////
                firstFound = Discovery_Fist_Node_First_Output(Node_ID);
                ///////////////////////////////////////////////////////////////////

                ///////////////////////////////////////////////////////////////////
                if(firstFound == true)
                {
                        nodes[0] = true;

                }
                else
                {
                        // First node was not found so increment node
                        Node_ID++;
                }
                ///////////////////////////////////////////////////////////////////

                ///////////////////////////////////////////////////////////////////
                if(firstFound == false)
                {
                        endRound = !Discovery_Fist_Node_Second_Output(Node_ID);
                        // Check to see if second node was found
                        if(endRound == false)
                                nodes[1] = true;
                }
                else
                        endRound = false;
                ///////////////////////////////////////////////////////////////////

                ///////////////////////////////////////////////////////////////////
                while(endRound != true)
                {
                        Node_ID++;

                        firstFound =  Discovery_First_Output(Node_ID);

                        if(firstFound == true)
                        {
```

```
                    nodes[Node_ID] = true;
                    endRound = false;
                }
                else
                {
                    // Node was not found so increment node
                    Node_ID++;
                    endRound = !Discovery_Second_Output(Node_ID);
                    // Check to see if second node was found
                    if(endRound == false)
                        nodes[Node_ID] = true;
                }
            }
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
            Node_ID--;
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
            // Tell nodes that discovery mode is over
            Discovery_Reset_Nodes();
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
            message[0] = DiscoveryNodes;
            message[1] = Node_ID;
            Notify_Ethernet_Listeners(
                                                              message,
                                                              2
                                                            );
//////////////////////////////////////////////////////////////////

            vTaskDelay( xDelay );

//////////////////////////////////////////////////////////////////
            message[0] = DiscoveryInactive;

            numBytes = (Node_ID / 8) + 1;
            for(int i = 0; i < numBytes; i++)
            {
                message[i + 1] = 0;
                for(int j = 0; j < 8; j++)
                {
```

```
                                 message[i + 1] |= (~(nodes[i*8+j] & true)) << j;
                         }
                  }
                  Notify_Ethernet_Listeners(
                                            message,
                                            numBytes + 1
                                                                 );
                  ///////////////////////////////////////////////////////////////////

                  vTaskDelay( xDelay );

                  ///////////////////////////////////////////////////////////////////
                  message[0] = DiscoveryFinished;

                  Notify_Ethernet_Listeners(
                                            message,
                                                                   1
                                                                 );
                  ///////////////////////////////////////////////////////////////////
                  BSP_LED_Off(LED2);
          }
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void Discovery_Start(void)
{
          CAN_Send( Discovery_Start_Identifier,
                                           0, // Data length
                                           NULL );
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
bool Discovery_Fist_Node_First_Output(uint8_t Node_ID)
{
          bool returnStatus;

          // Turn on the first output
          HAL_GPIO_WritePin( DISC_PORT, DISC_OUTPUT_1, GPIO_PIN_SET );

          returnStatus = Discovery_Call(Node_ID);

          HAL_GPIO_WritePin( DISC_PORT, DISC_OUTPUT_1, GPIO_PIN_RESET);

          return returnStatus;
}
///////////////////////////////////////////////////////////////

```

```
///////////////////////////////////////////////////////////////
bool Discovery_Fist_Node_Second_Output(uint8_t Node_ID)
{
          bool returnStatus;

          // Turn on the second output
          HAL_GPIO_WritePin( DISC_PORT, DISC_OUTPUT_2, GPIO_PIN_SET );

          returnStatus = Discovery_Call(Node_ID);

          HAL_GPIO_WritePin( DISC_PORT, DISC_OUTPUT_2, GPIO_PIN_RESET);

          return returnStatus;
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
bool Discovery_First_Output(uint8_t Node_ID)
{
          bool returnStatus;

          Discovery_Ready(Discovery_Ready_1_Identifier, Node_ID);

          returnStatus = Discovery_Call(Node_ID);

          return returnStatus;
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
bool Discovery_Second_Output(uint8_t Node_ID)
{
          bool returnStatus;

          Discovery_Ready(Discovery_Ready_2_Identifier, Node_ID);

          returnStatus = Discovery_Call(Node_ID);

          return returnStatus;
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void Discovery_Reset_Nodes(void)
{
          CAN_Send( Discovery_End_Identifier,
                                          0, // Data length
                                          NULL );
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
bool Discovery_Call(uint8_t Node_ID)
{
          bool successful = true;
          uint8_t dataNode[1];
```

```
        BaseType_t returnStatus = pdPASS;
        CAN_Message_Struct messageStruct;

        xQueueReset( queueDiscovery );

        dataNode[0] = Node_ID;
        CAN_Send( Discovery_Call_Identifier,
                                          1, // Data length
                                          dataNode );

        while( messageStruct.messageHeader.StdId != Disocvery_Call_Response_Identifier &&
                        returnStatus != errQUEUE_EMPTY )
        {
                returnStatus =  xQueueReceive(
                                        queueDiscovery,
                                        &messageStruct,
                                        250
                                        );
        }

        if(returnStatus == errQUEUE_EMPTY)
        {
                dataNode[0] = Node_ID;
                CAN_Send( Discovery_Call_Identifier,
                                                  1, // Data length
                                                  dataNode );

                while( messageStruct.messageHeader.StdId != Disocvery_Call_Response_Identifier && returnStatus != errQUEUE_EMPTY )
                {
                        returnStatus =  xQueueReceive(
                                                queueDiscovery,
                                                &messageStruct,
                                                250
                                                );
                }

                if(returnStatus == errQUEUE_EMPTY)
                {
                        successful = false;
                }
        }

        return successful;
}
/////////////////////////////////////////////////////////////////

```

```c
///////////////////////////////////////////////////////////////
void Discovery_Ready(IdentifierCodes_Type code, uint8_t Node_ID)
{
          BaseType_t returnStatus = pdPASS;
          //uint8_t dataNode[1];
          CAN_Message_Struct messageStruct;
          //bool firstFound = true;

          xQueueReset( queueDiscovery );

          CAN_Send( code,
                             0, // Data length
                             NULL );

          while( messageStruct.messageHeader.StdId != Discovery_Ready_Response_Identifier &&
                                    returnStatus != errQUEUE_EMPTY )
          {
                  returnStatus =  xQueueReceive(
                                    queueDiscovery,
                                    &messageStruct,
                                    250
                                    );
          }

          if(returnStatus == errQUEUE_EMPTY)
          {
                  CAN_Send( code,
                                                   0, // Data length
                                                   NULL );

                  while( messageStruct.messageHeader.StdId != Discovery_Ready_Response_Identifier &&
                                            returnStatus != errQUEUE_EMPTY )
                  {
                          returnStatus =  xQueueReceive(
                                            queueDiscovery,
                                            &messageStruct,
                                            250
                                            );
                  }

                  if(returnStatus == errQUEUE_EMPTY)
                  {
                          //TODO: Error Node isn't responding
                  }
                  else if(messageStruct.messageData[0] + 1 != Node_ID)
                  {
```

```
                            // TODO: Error wrong node responded
                    }
            }
            else if(messageStruct.messageData[0] + 1 != Node_ID)
            {
                    // TODO: Error wrong node responded
            }
}
//////////////////////////////////////////////////////////////



/**
*************************************************************************
****
   * @file           : error.c
   * @brief          : error handler
*************************************************************************
****
   * @attention
   *
   *
*************************************************************************
****
   */

//////////////////////////////////////////////////////////////
#include "FreeRTOS.h"
#include "task.h"
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void vAssertCalled( const char *pcFile, uint32_t ulLine )
{
 /* Inside this function, pcFile holds the name of the source file that
contains
 the line that detected the error, and ulLine holds the line number in
the source
 file. The pcFile and ulLine values can be printed out, or otherwise
recorded,
 before the following infinite loop is entered. */
            char string[255] = "";

            PC_Setup_Serial();

            snprintf(string,255, "Assert Failed - file: %s line:
%lu\n\n\n", pcFile, ulLine);

            HAL_StatusTypeDef status = HAL_ERROR;

            while(status != HAL_OK)
            {
                  status = PC_Send_String((uint8_t *) string, strlen(string));
          }
 /* Disable interrupts so the tick interrupt stops executing, then sit in a loop
 so execution does not move past the line that failed the assertion. */
 taskDISABLE_INTERRUPTS();
 for( ;; );
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void vApplicationMallocFailedHook(void)
{
          char string[100] = "";

          PC_Setup_Serial();


          size_t sizeLeft = xPortGetFreeHeapSize();

          sprintf(string, "Malloc Failed Hook - Heap Left: %u \n\n\n", sizeLeft);

          HAL_StatusTypeDef status = HAL_ERROR;

          while(status != HAL_OK)
          {
                  status = PC_Send_String((uint8_t *) string, strlen(string));
          }

          while(1);
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void vApplicationStackOverflowHook(xTaskHandle xTask, signed char *pcTaskName)
{
          char string[100] = "";

          PC_Setup_Serial();

          sprintf(string, "Stack Overflow: %s\n\n\n", pcTaskName);

          HAL_StatusTypeDef status = HAL_ERROR;

          while(status != HAL_OK)
          {
                  status = PC_Send_String((uint8_t *) string, strlen(string));
          }

          while(1);
}
```

```
//////////////////////////////////////////////////////////////




/**
***************************************************************************
  * File Name          : freertos.c
  * Description        : Code for freertos applications
***************************************************************************
  * @attention
  *
  *
***************************************************************************
  */

//////////////////////////////////////////////////////////////
// Includes
//////////////////////////////////////////////////////////////
#include "FreeRTOSConfig.h"
#include "FreeRTOSIPConfig.h"
#include "FreeRTOS.h"
#include "main.h"
#include "stm32f7xx_nucleo_144.h"
#include "task.h"
#include "queue.h"
#include "event_groups.h"

#include "discovery.h"
#include "can.h"
#include "network_application.h"

#include "debug.h"
#include "serialtopc.h"
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
#define NUM_IDENTIFIERS 20
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
// Extern variables
extern CAN_HandleTypeDef hcan1; // src/can.c
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
// Global Variables
TaskHandle_t CAN_LowLevel_Task_Handle,
                          CAN_Discovery_Task_Handle,
                            CAN_FuseOK_Task_Handle,
                            CAN_Error_Task_Handle,
                            CAN_Heartbeat_Task_Handle,
                            CAN_FuseBlown_Task_Handle,
                            Ethernet_Task_Handle;

QueueHandle_t queueDiscovery,
                         queueFuseBlown,
                         queueFuseOK,
                             queueError,
                             queueHeartbeat;

EventGroupHandle_t g_CAN_Message_Event_Group;
static char buffer[100];
uint8_t g_NumFuses = 5;
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
// Private Function Declarations
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
// Task Function Declarations
// CAN Functions
static void CAN_LowLevel_Task( void *pvParameters );
static void CAN_FuseOK_Task (void *pvParameters );
static void CAN_Error_Task(void *pvParameters );
static void CAN_Heartbeat_Task( void *pvParameters );
static void CAN_FuseBlown_Task( void *pvParameters );
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
// Helper Functions
static QueueHandle_t* Switch_Queue(CAN_RxHeaderTypeDef header);
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
// Public Function Declarations
///////////////////////////////////////////////////////////////
void MX_FREERTOS_Init(void);
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void MX_FREERTOS_Init(void) {
          /* RTOS_QUEUES */
          queueDiscovery = xQueueCreate( 64, sizeof( CAN_Message_Struct
) );
          if(queueDiscovery == NULL)
          {
                  Error_Handler();
          }
          queueFuseBlown = xQueueCreate( 64, sizeof( CAN_Message_Struct
) );
          if(queueFuseBlown == NULL)
          {
                  Error_Handler();
```

```c
	}
	queueFuseOK = xQueueCreate( 64, sizeof( CAN_Message_Struct ) );
	if(queueFuseOK == NULL)
	{
		Error_Handler();
	}
	queueError = xQueueCreate( 64, sizeof( CAN_Message_Struct ) );
	if(queueError == NULL)
	{
		Error_Handler();
	}
	queueHeartbeat = xQueueCreate( 64, sizeof( CAN_Message_Struct ) );
	if(queueHeartbeat == NULL)
	{
		Error_Handler();
	}

	/* RTOS_THREADS */
	BaseType_t xReturn;
	#if ipconfigHAS_DEBUG_PRINTF | ipconfigHAS_PRINTF
		Setup_Debug();
	#endif
	xReturn = xTaskCreate( CAN_LowLevel_Task, "CAN_LowLevel_Task", ( uint16_t )configMINIMAL_STACK_SIZE*3, NULL, ( UBaseType_t )CAN_LOWLEVEL_TASK_PRIORITY, &CAN_LowLevel_Task_Handle );
	if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
	{
		Error_Handler();
	}
	xReturn = xTaskCreate( Ethernet_Task, "Ethernet_Task", ( uint16_t )configMINIMAL_STACK_SIZE*3, NULL, ( UBaseType_t )ETHERNET_TASK_PRIORITY, &Ethernet_Task_Handle );
	if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
	{
		Error_Handler();
	}

	xReturn = xTaskCreate( CAN_Discovery_Task, "CAN_Discovery_Task", ( uint16_t )configMINIMAL_STACK_SIZE*3, NULL, ( UBaseType_t )CAN_DISCOVERY_TASK_PRIORITY, &CAN_Discovery_Task_Handle );
	if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
	{
		Error_Handler();
	}
	xReturn = xTaskCreate( CAN_FuseBlown_Task, "CAN_FuseBlown_Task", ( uint16_t )configMINIMAL_STACK_SIZE*3, NULL, ( UBaseType_t )CAN_FUSEBLOWN_TASK_PRIORITY, &CAN_FuseBlown_Task_Handle );
	if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
	{
		Error_Handler();
	}
	xReturn = xTaskCreate( CAN_FuseOK_Task, "CAN_FuseOK_Task", ( uint16_t )configMINIMAL_STACK_SIZE*3, NULL, ( UBaseType_t ) CAN_FUSEOK_TASK_PRIORITY, &CAN_FuseOK_Task_Handle );
```

```c
        if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
        {
                Error_Handler();
        }
        xReturn = xTaskCreate( CAN_Error_Task, "CAN_Error_Task", ( uint16_t )configMINIMAL_STACK_SIZE*3, NULL, ( UBaseType_t )CAN_ERROR_TASK_PRIORITY, &CAN_Error_Task_Handle );
        if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
        {
                Error_Handler();
        }
        xReturn = xTaskCreate( CAN_Heartbeat_Task, "CAN_Heartbeat_Task", ( uint16_t )configMINIMAL_STACK_SIZE*3, NULL, ( UBaseType_t )CAN_HEARTBEAT_TASK_PRIORITY, &CAN_Heartbeat_Task_Handle );
        if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
        {
                Error_Handler();
        }


}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
/**
  * @brief  Handles reading from CAN
  * @param  argument: Not used
  * @retval None
  */
static void CAN_LowLevel_Task( void *pvParameters )
{
        /* Attempt to create the event group. */
        g_CAN_Message_Event_Group = xEventGroupCreate();

        /* Was the event group created successfully? */
        if( g_CAN_Message_Event_Group == NULL )
        {
                /* The event group was not created because there was insufficient
                FreeRTOS heap available. */
                Error_Handler();
        }

        CAN_Setup();

        EventBits_t uxBits;

        for(;;)
        {
                CAN_Message_Struct mStruct;
                QueueHandle_t *pQueue;

                /* Wait a maximum of 50 days for either bit 0 or bit 1 to be set within
                the event group.  Clear the bits before exiting. */
                uxBits = xEventGroupWaitBits(
            g_CAN_Message_Event_Group,   /* The event group being tested. */
            BIT_FIFO_0 | BIT_FIFO_1, /* The bits within the event group to wait for. */
            pdTRUE,        /* BIT_0 | BIT_1 should be cleared before returning. */
            pdFALSE,       /* Any bit will trigger */
            portMAX_DELAY );/* Wait a maximum of 50 days */

        if( uxBits == 0 )
            continue;

        uint32_t fillLevel = 0;
        bool found = false;
        do
        {
            uint32_t fillLevel = HAL_CAN_GetRxFifoFillLevel(&hcan1, CAN_RX_FIFO0);
            if(fillLevel != 0)
            {
                CAN_Receive(CAN_RX_FIFO0, &mStruct.messageHeader, mStruct.messageData);

                pQueue = Switch_Queue(mStruct.messageHeader);

                xQueueSend( *pQueue,
                            &mStruct,
                            portMAX_DELAY );
                found = true;
            }

            if(found == true && fillLevel != 0)
            {
                HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING);
            }

        }while(fillLevel != 0);

        found = false;
        fillLevel = 0;
        do
        {
            uint32_t fillLevel = HAL_CAN_GetRxFifoFillLevel(&hcan1, CAN_RX_FIFO1);
            if(fillLevel != 0)
            {
                CAN_Receive(CAN_RX_FIFO1, &mStruct.messageHeader, mStruct.messageData);
                pQueue = Switch_Queue(mStruct.messageHeader);

```

```
                                xQueueSend( *pQueue,
                                                        &mStruct,
                                                        portMAX_DELAY );
                                found = true;
                        }


                        if(found == true && fillLevel != 0)
                        {
                                HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO1_MSG_PENDING);
                        }

                }while(fillLevel != 0);
        }
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
/**
  * @brief  Switches the queue based on the StdId code
  * @param  CAN header used to determine StdId
  * @retval None
  */
static QueueHandle_t* Switch_Queue(CAN_RxHeaderTypeDef header)
{
        uint32_t id = header.StdId;
        QueueHandle_t pQueue = &queueError;

        switch(id)
        {
                case Heartbeat_Identifier:
                        pQueue = &queueHeartbeat;
                        break;
                case Fuse_OK_Identifier:
                        pQueue = &queueFuseOK;
                        break;
                case Fuse_Blown_Identifier:
                        pQueue = &queueFuseBlown;
                        break;

                case Error_1_Identifier:
                        pQueue = &queueError;
                        break;
                case Error_2_Identifier:
                        pQueue = &queueError;
                        break;
                case Discovery_Ready_Response_Identifier:
                        pQueue = &queueDiscovery;
                        break;

                case Disocvery_Call_Response_Identifier:
                        pQueue = &queueDiscovery;
                        break;

                default:
```

```
                                //error
                                break;
            }


            return pQueue;
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
/**
  * @brief  Handle received blown fuse notification
  * @param  argument: Not used
  * @retval None
  */
static void CAN_FuseBlown_Task( void *pvParameters )
{
            BaseType_t returnStatus;
            CAN_Message_Struct messageStruct;
            uint8_t message[64] = {0};
            uint8_t canM[1] = {0};
            message[0] = FuseBlown;

            for(;;)
            {
                        returnStatus =  xQueueReceive(
    queueFuseBlown,
    &messageStruct,
    portMAX_DELAY
    );
                        if(returnStatus == pdFALSE)
                                continue;

                        canM[0] = messageStruct.messageData[0];

                        CAN_Send( Fuse_Blown_Ack_Identifier,
                                         1, // Data length
                                         canM );

                        // TODO: Log and send if turned on
                        message[1] = (1 << messageStruct.messageData[0]);
                        Notify_Ethernet_Listeners( message, (size_t) 2 );



                        sprintf(buffer,  "Fuse Blown: %x\n",
    messageStruct.messageData[0] );
                        FreeRTOS_debug_printf((uint8_t *)buffer,
    strlen(buffer));
            }
}
///////////////////////////////////////////////////////////////
```

```

///////////////////////////////////////////////////////////////
static void CAN_FuseOK_Task (void *pvParameters )
{
            BaseType_t returnStatus;
            CAN_Message_Struct messageStruct;
            uint8_t message[64] = {0};
            message[0] = FuseOK;

            for(;;)
            {
                    returnStatus =  xQueueReceive(
    queueFuseOK,
    &messageStruct,
    portMAX_DELAY
    );
                    if(returnStatus == pdFALSE)
                            continue;

                    // TODO: Log and send if turned on
                    message[1] = (1 << messageStruct.messageData[0]);
                    Notify_Ethernet_Listeners( message, (size_t) 2 );



                    sprintf(buffer,  "Fuse OK: %x\n",
    messageStruct.messageData[0] );
                    FreeRTOS_debug_printf((uint8_t *)buffer,
    strlen(buffer));

            }
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
static void CAN_Error_Task(void *pvParameters )
{
            BaseType_t returnStatus;
            CAN_Message_Struct messageStruct;
            uint8_t message[64] = {0};

            for(;;)
            {
                    returnStatus =  xQueueReceive(
    queueError,
    &messageStruct,
    portMAX_DELAY
    );
```

```
                        if(returnStatus == pdFALSE)
                                continue;

                        // TODO: Log and send if turned on
                        Notify_Ethernet_Listeners( message, (size_t) 1 );
                }
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
static void CAN_Heartbeat_Task( void *pvParameters )
{
                BaseType_t returnStatus;
                CAN_Message_Struct messageStruct;
                uint8_t message[64] = {0};

                for(;;)
                {
                        returnStatus =  xQueueReceive(
    queueHeartbeat,
    &messageStruct,
    portMAX_DELAY
    );
                        if(returnStatus == pdFALSE)
                                continue;

                        // TODO: Log and send if turned on
                        Notify_Ethernet_Listeners( message, (size_t) 1 );
                }
}
///////////////////////////////////////////////////////////////




/**
    ***************************************************************************
    ****
  * File Name          : gpio.c
  * Description        : This file provides code for the configuration
  *                      of all used GPIO pins.
    ***************************************************************************
    ****
  * @attention
  *
  *
    ***************************************************************************
    ****
  */
```

```c

////////////////////////////////////////////////////////////////
/* Includes --------------------------------------------------------------------*/
#include "gpio.h"
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
  __HAL_RCC_GPIOD_CLK_ENABLE();
  __HAL_RCC_GPIOG_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4|GPIO_PIN_5, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0|GPIO_PIN_14|GPIO_PIN_7, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOG, GPIO_PIN_6, GPIO_PIN_RESET);

  /*Configure GPIO pin : PC13 */
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  /*Configure GPIO pins : PC1 PC4 PC5 */
  GPIO_InitStruct.Pin = GPIO_PIN_1|GPIO_PIN_4|GPIO_PIN_5;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  /*Configure GPIO pins : PA1 PA2 PA7 */
  GPIO_InitStruct.Pin = GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_7;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /*Configure GPIO pins : PA4 PA5 */ /* Discovery Mode */
  GPIO_InitStruct.Pin = DISC_OUTPUT_1|DISC_OUTPUT_2;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(DISC_PORT, &GPIO_InitStruct);

  /*Configure GPIO pins : PB0 PB14 PB7 */
  GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_14|GPIO_PIN_7;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /*Configure GPIO pin : PB13 */
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /*Configure GPIO pin : PG6 */
  GPIO_InitStruct.Pin = GPIO_PIN_6;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);

  /*Configure GPIO pin : PG7 */
  GPIO_InitStruct.Pin = GPIO_PIN_7;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);

  /*Configure GPIO pins : PG11 PG13 */
  GPIO_InitStruct.Pin = GPIO_PIN_11|GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
  HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);

}
//////////////////////////////////////////////////////////////




/**
  ****************************************************************************
  * File Name          : network_application.c
  * Description        :
  ****************************************************************************
  * @attention
  *
  *
**************************************************************************
  */

/////////////////////////////////////////////////////////////
#include "network_application.h"
#include "main.h"
/////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////
#define BUFFER_SIZE 64
#define NUM_IDENTIFIERS 20
/////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////
extern EventGroupHandle_t g_Discovery_Event_Group;
/////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////
static void Listener_Task( void *pvParameters );
static void Sender_Task( void *pvParameters );
/////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////
/* Use by the pseudo random number generator. */
static UBaseType_t ulNextRand;
const uint8_t ucMACAddress[ 6 ] = { configMAC_ADDR0, configMAC_ADDR1, configMAC_ADDR2, configMAC_ADDR3, configMAC_ADDR4, configMAC_ADDR5 };

static const uint8_t ucIPAddress[ 4 ] = { configIP_ADDR0, configIP_ADDR1, configIP_ADDR2, configIP_ADDR3 };
static const uint8_t ucNetMask[ 4 ] = { configNET_MASK0, configNET_MASK1, configNET_MASK2, configNET_MASK3 };
static const uint8_t ucGatewayAddress[ 4 ] = { configGATEWAY_ADDR0, configGATEWAY_ADDR1, configGATEWAY_ADDR2, configGATEWAY_ADDR3 };
static const uint8_t ucDNSServerAddress[ 4 ] = { configDNS_SERVER_ADDR0, configDNS_SERVER_ADDR1, configDNS_SERVER_ADDR2, configDNS_SERVER_ADDR3 };

EventGroupHandle_t g_Ethernet_Start_Event;
TaskHandle_t Listener_Task_Handle, Sender_Task_Handle;
Message_Identifier_Struct g_Ethernet_Identifiers[20];
/////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////
void Notify_Ethernet_Listeners( uint8_t * pMessage, size_t length )
{
	// Notify Ethernet Queue Pointers
	for(int i = 0; i < NUM_IDENTIFIERS; i++)
	{
		if( g_Ethernet_Identifiers[i].valid == true )
		{
			MessageBufferHandle_t messageBuffer = g_Ethernet_Identifiers[i].messagePointer;
```

```

                                                    xMessageBufferSend(
        messageBuffer,
                                                                                                                                    pMessage,
                                                                                                                                    length,
        portMAX_DELAY
                                                                                                                         );
                                    }
                    }
}
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
void Ethernet_Task( void *pvParameters )
{
                EventBits_t uxBits;
                g_Ethernet_Start_Event = xEventGroupCreate();
                // Reset Ethernet Queue Pointers
                for(int i = 0; i < NUM_IDENTIFIERS; i++)
                {
                                g_Ethernet_Identifiers[i].valid = false;
                }

                NetworkConfig();


                /* Wait a maximum of 50 days for either bit 0 or bit 1 to be set within
                the event group.  Clear the bits before exiting. */
                uxBits = xEventGroupWaitBits(
        g_Ethernet_Start_Event,   /* The event group being tested. */
                                                                                                                                    BIT_0, /* The bits within the event group to wait for. */
                                                                                                                                    pdTRUE, /* BIT_0 should be cleared before returning. */
                                                                                                                                    pdFALSE, /* Any bit will trigger */
        portMAX_DELAY );/* Wait a maximum of 50 days */
                vTVPServerSock();

                // Shouldn't reach here but delete if it does
                vTaskDelete( NULL );
}
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
void vTVPServerSock(void)
{
                struct freertos_sockaddr xClient, xBindAddress;
                Socket_t xListeningSocket, xConnectedSocket;
                socklen_t xSize = sizeof( xClient );
                static const TickType_t xReceiveTimeOut = portMAX_DELAY;
                const BaseType_t xBacklog = 20;

                /* Attempt to open the socket. */
                xListeningSocket = FreeRTOS_socket( FREERTOS_AF_INET,
                                                    FREERTOS_SOCK_STREAM,  /* SOCK_STREAM for TCP. */
                                                    FREERTOS_IPPROTO_TCP );

                /* Check the socket was created. */
                configASSERT( xListeningSocket != FREERTOS_INVALID_SOCKET );

                /* If FREERTOS_SO_RCVBUF or FREERTOS_SO_SNDBUF are to be used with
                FreeRTOS_setsockopt() to change the buffer sizes from their default then do
                it here!.  (see the FreeRTOS_setsockopt() documentation. */

                /* If ipconfigUSE_TCP_WIN is set to 1 and FREERTOS_SO_WIN_PROPERTIES is to
                be used with FreeRTOS_setsockopt() to change the sliding window size from
                its default then do it here! (see the FreeRTOS_setsockopt()
                documentation. */

                /* Set a time out so accept() will just wait for a connection. */
                FreeRTOS_setsockopt( xListeningSocket,
                                                               0,
                                                               FREERTOS_SO_RCVTIMEO,
                                                               &xReceiveTimeOut,
                                                               sizeof( xReceiveTimeOut ) );

                /* Set the listening port to 10000. */
                xBindAddress.sin_port = ( uint16_t ) 10000;
                xBindAddress.sin_port = FreeRTOS_htons( xBindAddress.sin_port );

                /* Bind the socket to the port that the client RTOS task will send to. */
                FreeRTOS_bind( xListeningSocket, &xBindAddress, sizeof( xBindAddress ) );

                /* Set the socket into a listening state so it can accept connections.
                The maximum number of simultaneous connections is limited to 20. */
                FreeRTOS_listen( xListeningSocket, xBacklog );

                int i = 0;
                for( ;; )
                {
                        Task_Param_Struct msg;
                        /* Wait for incoming connections. */
                        xConnectedSocket = FreeRTOS_accept( xListeningSocket, &xClient, &xSize );
                        configASSERT( xConnectedSocket != FREERTOS_INVALID_SOCKET );

                        /* Spawn a RTOS task to handle the connection. */
                        msg.index = i;
                        msg.msg = (void *) xConnectedSocket;
                        xTaskCreate( Listener_Task,
                                    "Listener_Task",
                                    ( uint16_t ) configMINIMAL_STACK_SIZE * 3,
                                    ( void * ) &msg,
                                    LISTENER_TASK_PRIORITY - i,
                                    Listener_Task_Handle );

                        xTaskCreate( Sender_Task,
                                    "Sender_Task",
                                    ( uint16_t ) configMINIMAL_STACK_SIZE * 3,
                                    ( void * ) &msg,
                                    SENDER_TASK_PRIORITY - i,
                                    Sender_Task_Handle );

                        g_Ethernet_Identifiers[i].messagePointer = xMessageBufferCreate( 64 );
                        g_Ethernet_Identifiers[i].valid = true;
                        i++;
                }
            }
            ///////////////////////////////////////////////////////////////

            ///////////////////////////////////////////////////////////////
            static void Sender_Task( void *pvParameters )
            {
                int index = ((Task_Param_Struct *)pvParameters)->index;
                uint8_t ucRxData [64] = {0};
                size_t xReceivedBytes;
                MessageBufferHandle_t messageBuffer = g_Ethernet_Identifiers[index].messagePointer;

                Socket_t xSocket = (Socket_t) ((Task_Param_Struct *)pvParameters)->msg;
                BaseType_t xAlreadyTransmitted = 0, xBytesSent = 0;
                size_t xLenToSend;
                bool closeSocket = false;
                //BaseType_t returnStatus;


                while( closeSocket == false )
                {
                        xReceivedBytes = xMessageBufferReceive(
                                    messageBuffer,
                                    ucRxData,
                                    sizeof( ucRxData ),
                                    portMAX_DELAY
                                );

                        xAlreadyTransmitted = 0;
                        /* Keep sending until the entire buffer has been sent. */
                        while( xAlreadyTransmitted < xReceivedBytes && closeSocket == false)
                        {
                            /* How many bytes are left to send? */
                            xLenToSend = xReceivedBytes - xAlreadyTransmitted;
                            xBytesSent = FreeRTOS_send( /* The socket being sent to. */
                                                        xSocket,
                                                        /* The data being sent. */
                                                        &( ucRxData[ xAlreadyTransmitted ] ),
                                                        /* The remaining length of data to send. */
                                                        xLenToSend,
                                                        /* ulFlags. */
                                                        0 );

                            if( xBytesSent >= 0 )
                            {
                                /* Data was sent successfully. */
                                xAlreadyTransmitted += xBytesSent;
                            }
                            else
                            {
                                /* Error - break out of the loop for graceful socket close. */
                                break;
                            }
                        }
                }


                /* Initiate graceful shutdown. */
                FreeRTOS_shutdown( xSocket, FREERTOS_SHUT_RDWR );

                /* Wait for the socket to disconnect gracefully (indicated by FreeRTOS_recv()
                returning a FREERTOS_EINVAL error) before closing the socket. */
                while( FreeRTOS_recv( xSocket, ucRxData, sizeof( ucRxData ), 0 ) >= 0 )
                {
                        /* Wait for shutdown to complete.  If a receive block time is used then
                        this delay will not be necessary as FreeRTOS_recv() will place the RTOS task
                        into the Blocked state anyway. */
                        const TickType_t xDelay = 250 / portTICK_PERIOD_MS;
                        vTaskDelay( xDelay );

                        /* Note - real applications should implement a timeout here, not just
                        loop forever. */
                }

                /* The socket has shut down and is safe to close. */
                FreeRTOS_closesocket( xSocket );

                vTaskDelete( NULL );
        }
        ////////////////////////////////////////////////////////////////

        ////////////////////////////////////////////////////////////////
        static void Listener_Task( void *pvParameters )
        {

                Socket_t xSocket = (Socket_t) ((Task_Param_Struct *)pvParameters)->msg;
                static char cRxedData[ BUFFER_SIZE ];
                BaseType_t lBytesReceived;


                for( ;; )
                {
                        /* Receive another block of data into the cRxedData buffer. */
                        lBytesReceived = FreeRTOS_recv( xSocket, &cRxedData, BUFFER_SIZE, 0 );

                        if( lBytesReceived > 0 )
                        {
                            /* Data was received, process it here. */

                            if(cRxedData[0] == DiscoveryStart)
                            {
                                xEventGroupSetBits(
                                                        g_Discovery_Event_Group,
                                                        BIT_0
                                                        );
                            }
                            else
                            {
                                // TODO: Status Changed
```

```
            }
        }
        else if( lBytesReceived == 0 )
        {
            /* No data was received, but FreeRTOS_recv() did not return an error.
            Timeout? */
            continue;
        }
        else
        {
            /* Error (maybe the connected socket already shut down the socket?).
            Attempt graceful shutdown. */
            FreeRTOS_shutdown( xSocket, FREERTOS_SHUT_RDWR );
            break;
        }
    }

    /* The RTOS task will get here if an error is received on a read.  Ensure the
    socket has shut down (indicated by FreeRTOS_recv() returning a FREERTOS_EINVAL
    error before closing the socket). */

    while( FreeRTOS_recv( xSocket, cRxedData, sizeof(cRxedData), 0 ) >= 0 )
    {
        /* Wait for shutdown to complete.  If a receive block time is used then
        this delay will not be necessary as FreeRTOS_recv() will place the RTOS task
        into the Blocked state anyway. */
        const TickType_t xDelay = 250 / portTICK_PERIOD_MS;
        vTaskDelay( xDelay );

        /* Note - real applications should implement a timeout here, not just
        loop forever. */
    }

    /* Shutdown is complete and the socket can be safely closed. */
    FreeRTOS_closesocket( xSocket );

    /* Must not drop off the end of the RTOS task - delete the RTOS task. */
    vTaskDelete( NULL );
}
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
void NetworkConfig(void)
{
    // Init FreeRTOS IP
```

```
          FreeRTOS_IPInit( ucIPAddress, ucNetMask, ucGatewayAddress, ucDNSServerAddress, ucMACAddress );
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void vApplicationIPNetworkEventHook( eIPCallbackEvent_t eNetworkEvent )
{
          static BaseType_t xTasksAlreadyCreated = pdFALSE;

    /* Both eNetworkUp and eNetworkDown events can be processed here. */
    if( eNetworkEvent == eNetworkUp )
    {
        /* Create the tasks that use the TCP/IP stack if they have not already
        been created. */
        if( xTasksAlreadyCreated == pdFALSE )
        {

           xEventGroupSetBits( g_Ethernet_Start_Event,
                                                                BIT_0 );
            /*
             * For convenience, tasks that use FreeRTOS+TCP can be created here
             * to ensure they are not created before the network is usable.
             */
            xTasksAlreadyCreated = pdTRUE;
        }
    }
    else if( eNetworkEvent == eNetworkDown )
    {

    }
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
#if( ipconfigUSE_LLMNR != 0 ) || ( ipconfigUSE_NBNS != 0 ) || ( ipconfigDHCP_REGISTER_HOSTNAME == 1 )

          const char *pcApplicationHostnameHook( void )
          {
                  /* Assign the name "FreeRTOS" to this network node. This function will
                  be called during the DHCP: the machine will be registered with an IP
                  address plus this name. */
                  return mainHOST_NAME;
          }

#endif
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
#if( ipconfigUSE_LLMNR != 0 ) || ( ipconfigUSE_NBNS != 0 )
```

```

          BaseType_t xApplicationDNSQueryHook( const char *pcName )
          {
                  BaseType_t xReturn = 0 ;
                  return xReturn;
          }
#endif
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
uint32_t ulApplicationGetNextSequenceNumber( uint32_t ulSourceAddress,
                                         uint16_t usSourcePort,
                                         uint32_t ulDestinationAddress,
                                         uint16_t usDestinationPort )
{
          ( void ) ulSourceAddress;
          ( void ) usSourcePort;
          ( void ) ulDestinationAddress;
          ( void ) usDestinationPort;

          return uxRand();
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
UBaseType_t uxRand( void )
{
          const uint32_t ulMultiplier = 0x015a4e35UL, ulIncrement = 1UL;

          /* Utility function to generate a pseudo random number. */
          ulNextRand = ( ulMultiplier * ulNextRand ) + ulIncrement;
          return( ( int ) ( ulNextRand >> 16UL ) & 0x7fffUL );
}
//////////////////////////////////////////////////////////////




/**
******************************************************************************
  * File Name          : serialtopc.c
  * Description        :
******************************************************************************
  * @attention
  *
  *
******************************************************************************
```

```
  */

///////////////////////////////////////////////////////////////
#include "serialtopc.h"
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void USART3_Init(int baudRate);
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
UART_HandleTypeDef huart;
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void USART3_Init(int baudRate)
{
          huart.Instance = USART3;

          huart.Init.WordLength = UART_WORDLENGTH_8B;
          huart.Init.StopBits = UART_STOPBITS_1;
          huart.Init.Parity = UART_PARITY_NONE;
          huart.Init.Mode = UART_MODE_TX_RX;
          huart.Init.OverSampling = UART_OVERSAMPLING_16;
          huart.Init.BaudRate = baudRate;

          HAL_UART_Init(&huart);
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void PC_Setup_Serial(void)
{
          USART3_Init(115200);
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
HAL_StatusTypeDef PC_Send_String(uint8_t* string, uint16_t size)
{
          return HAL_UART_Transmit(&huart, string, size, 10);
}
///////////////////////////////////////////////////////////////




/**
  ******************************************************************************
  * File Name          : stm32f7xx_hal_msp.c
  * Description        : This file provides code for the MSP
Initialization
  *                      and de-Initialization codes.
```

```c
  ******************************************************************************
  * @attention
  *
  *
  ******************************************************************************
  */

//////////////////////////////////////////////////////////////
/* Includes ------------------------------------------------------------------*/
#include "main.h"
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void HAL_MspInit(void)
{

  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_RCC_SYSCFG_CLK_ENABLE();

  /* System interrupt init*/
  /* PendSV_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(PendSV_IRQn, 15, 0);
}
//////////////////////////////////////////////////////////////





/**
  ******************************************************************************
  * @file    stm32f7xx_hal_timebase_TIM.c
  * @brief   HAL time base based on the hardware TIM.
  ******************************************************************************
  * @attention
  *
  *
  ******************************************************************************
  */

//////////////////////////////////////////////////////////////
/* Includes ------------------------------------------------------------------*/
#include "stm32f7xx_hal.h"
#include "stm32f7xx_hal_tim.h"
```

```
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
TIM_HandleTypeDef        htim1;
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
  RCC_ClkInitTypeDef    clkconfig;
  uint32_t              uwTimclock = 0;
  uint32_t              uwPrescalerValue = 0;
  uint32_t              pFLatency;

  /*Configure the TIM1 IRQ priority */
  HAL_NVIC_SetPriority(TIM1_UP_TIM10_IRQn, TickPriority ,0);

  /* Enable the TIM1 global Interrupt */
  HAL_NVIC_EnableIRQ(TIM1_UP_TIM10_IRQn);

  /* Enable TIM1 clock */
  __HAL_RCC_TIM1_CLK_ENABLE();

  /* Get clock configuration */
  HAL_RCC_GetClockConfig(&clkconfig, &pFLatency);

  /* Compute TIM1 clock */
  uwTimclock = 2*HAL_RCC_GetPCLK2Freq();

  /* Compute the prescaler value to have TIM1 counter clock equal to 1MHz */
  uwPrescalerValue = (uint32_t) ((uwTimclock / 1000000) - 1);

  /* Initialize TIM1 */
  htim1.Instance = TIM1;

  /* Initialize TIMx peripheral as follow:
  + Period = [(TIM1CLK/1000) - 1]. to have a (1/1000) s time base.
  + Prescaler = (uwTimclock/1000000 - 1) to have a 1MHz counter clock.
  + ClockDivision = 0
  + Counter direction = Up
  */
  htim1.Init.Period = (1000000 / 1000) - 1;
  htim1.Init.Prescaler = uwPrescalerValue;
  htim1.Init.ClockDivision = 0;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  if(HAL_TIM_Base_Init(&htim1) == HAL_OK)
  {
    /* Start the TIM time Base generation in interrupt mode */
    return HAL_TIM_Base_Start_IT(&htim1);
  }

  /* Return function status */
  return HAL_ERROR;
}
//////////////////////////////////////////////////////////////
```

```c

////////////////////////////////////////////////////////////////
void HAL_SuspendTick(void)
{
  /* Disable TIM1 update Interrupt */
  __HAL_TIM_DISABLE_IT(&htim1, TIM_IT_UPDATE);
}
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
/**
  * @brief  Resume Tick increment.
  * @note   Enable the tick increment by Enabling TIM1 update interrupt.
  * @param  None
  * @retval None
  */
void HAL_ResumeTick(void)
{
  /* Enable TIM1 Update interrupt */
  __HAL_TIM_ENABLE_IT(&htim1, TIM_IT_UPDATE);
}
////////////////////////////////////////////////////////////////





/**
  ******************************************************************************
  * @file    stm32f7xx_it.c
  * @brief   Interrupt Service Routines.
  ******************************************************************************
  * @attention
  *
  *
  ******************************************************************************
  */

////////////////////////////////////////////////////////////////
#include "main.h"
#include "stm32f7xx_it.h"
#include "cmsis_os.h"
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
extern CAN_HandleTypeDef hcan1;
extern TIM_HandleTypeDef htim1;
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
```

```c
void NMI_Handler(void)
{
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void HardFault_Handler(void)
{
  while (1)
  {
  }
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void MemManage_Handler(void)
{
  while (1)
  {
  }
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void BusFault_Handler(void)
{
  while (1)
  {
  }
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void UsageFault_Handler(void)
{
  while (1)
  {
  }
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void DebugMon_Handler(void)
{

}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void CAN1_RX0_IRQHandler(void)
{
  HAL_CAN_IRQHandler(&hcan1);
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
```

```c
void CAN1_RX1_IRQHandler(void)
{
  HAL_CAN_IRQHandler(&hcan1);
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void TIM1_UP_TIM10_IRQHandler(void)
{
  HAL_TIM_IRQHandler(&htim1);
}
//////////////////////////////////////////////////////////////




/**
 ******************************************************************************
  * File Name          : CAN.c
  * Description        : This file provides code for the configuration
  *                      of the CAN instances.
 ******************************************************************************
  * @attention
  *
  *
 ******************************************************************************
  */

//////////////////////////////////////////////////////////////
#include "main.h"
#include "can.h"
#include "FreeRTOS.h"
#include "event_groups.h"
#include "timers.h"
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
CAN_HandleTypeDef hcan1;
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
extern EventGroupHandle_t g_CAN_Message_Event_Group;
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void HAL_CAN_MspInit(CAN_HandleTypeDef* canHandle)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(canHandle->Instance==CAN1)
  {
    /* CAN1 clock enable */
    __HAL_RCC_CAN1_CLK_ENABLE();

    __HAL_RCC_GPIOD_CLK_ENABLE();
    /**CAN1 GPIO Configuration
    PD0     ------> CAN1_RX
    PD1     ------> CAN1_TX
    */
    GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF9_CAN1;
    HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);

    /* CAN1 interrupt Init */
    HAL_NVIC_SetPriority(CAN1_RX0_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(CAN1_RX0_IRQn);
    HAL_NVIC_SetPriority(CAN1_RX1_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(CAN1_RX1_IRQn);\
  }
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void HAL_CAN_MspDeInit(CAN_HandleTypeDef* canHandle)
{

  if(canHandle->Instance==CAN1)
  {
    /* Peripheral clock disable */
    __HAL_RCC_CAN1_CLK_DISABLE();

    /**CAN1 GPIO Configuration
    PD0     ------> CAN1_RX
    PD1     ------> CAN1_TX
    */
    HAL_GPIO_DeInit(GPIOD, GPIO_PIN_0|GPIO_PIN_1);

    /* CAN1 interrupt Deinit */
    HAL_NVIC_DisableIRQ(CAN1_RX0_IRQn);
    HAL_NVIC_DisableIRQ(CAN1_RX1_IRQn);
  }
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
STATUS_StatusTypeDef CAN_Receive(uint32_t RxFifo, CAN_RxHeaderTypeDef
   *pHeader, uint8_t aData[])
{

          if(HAL_CAN_GetRxMessage(&hcan1, RxFifo, pHeader, aData) !=
   HAL_OK)
          {
                  // HAL_ERROR
          }
```

```c

		return STATUS_OK;
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
STATUS_StatusTypeDef CAN_Send(uint16_t stdid, uint8_t length, uint8_t *data)
{
		CAN_TxHeaderTypeDef   TxHeader;
		uint8_t               TxData[8];
		uint32_t              TxMailbox;

		/*##-- Start the Transmission process ##############*/
		TxHeader.StdId = stdid;
		TxHeader.RTR = CAN_RTR_DATA;
		TxHeader.IDE = CAN_ID_STD;
		TxHeader.DLC = length;
		TxHeader.TransmitGlobalTime = DISABLE;

		for(int i = 0; i < length; i++)
		{
				TxData[i] = data[i];
		}

		/* Request transmission */
		if(HAL_CAN_AddTxMessage(&hcan1, &TxHeader, TxData, &TxMailbox) != HAL_OK)
		{
				/* Transmission request Error */
		}

		//TODO This may not actually send the message

		return STATUS_OK;
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
// Configure the CAN peripheral
void CAN_Setup_Instance(void)
{
		// STM32F746 has two can controllers, but we are using the first one
		hcan1.Instance = CAN1; // First CAN instance

		// Configure CAN features
		hcan1.Init.TimeTriggeredMode     = DISABLE;
		hcan1.Init.AutoBusOff            = ENABLE;  // Hardware will automatically start recovering from Bus-Off state
		                                            // Recovery will happen once 128 occurences of 11 recessive bits occur
		hcan1.Init.AutoWakeUp            = DISABLE; // Sleep Mode isn't used for power saving mode so this isn't needed
```

```
        hcan1.Init.AutoRetransmission    = ENABLE;  // Keep sending the message until it is successful
        hcan1.Init.ReceiveFifoLocked     = DISABLE; // If the receive FIFO is full, overwrite old messages
        hcan1.Init.TransmitFifoPriority  = DISABLE; // Priority driver by the identifier of the message
        hcan1.Init.Mode                  = CAN_MODE_NORMAL;

        // APB1 Peripheral Clock: 54 MHz
        // Time Quantum 500.0 ns
        hcan1.Init.SyncJumpWidth         = CAN_SJW_1TQ;
        hcan1.Init.TimeSeg1              = CAN_BS1_13TQ;
        hcan1.Init.TimeSeg2              = CAN_BS2_2TQ;
        hcan1.Init.Prescaler             = 27;

        // Error Codes HAL_CAN_ERROR_TIMEOUT
        // Calls HAL_CAN_MspInit which initializes low level hardware
        //              Nothing should go wrong in this but need to check
        // Turns off sleep mode which could cause timeout
        //              Sleep mode for power saving is not currently used
        // Requests initialization
        //              The only thing that could go wrong with this is that
        //              the hardware doesn't monitor 11 consecutive recessive bits
        //                              Call HAL_CAN_Init agian if this happens
        //                              If it fails again reset device
        if(HAL_CAN_Init(&hcan1) != HAL_OK) // Successful call sets state HAL_CAN_STATE_READY
        {
                // Initialization timed out try again
                if(HAL_CAN_Init(&hcan1) != HAL_OK)
                {
                        // Initialization failed twice
                        //              Reset device
                        CAN_Error_Handler();
                }
        }
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
// Initialize CAN Filters
void CAN_Setup_Filters(uint16_t StdID, uint8_t fifoAssignment, uint32_t filterBank)
{
        CAN_FilterTypeDef  sFilterConfig;

  sFilterConfig.FilterBank = filterBank; // 28 Filter banks
  sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
  sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;

  sFilterConfig.FilterIdHigh = StdID << 5; // 16 bits
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0xFFFF;
sFilterConfig.FilterMaskIdLow = 0xFFFF;


        // CAN_RX_FIFO0 or CAN_RX_FIFO1
        sFilterConfig.FilterFIFOAssignment = fifoAssignment; // Two Receive FIFOs with three stages each
sFilterConfig.FilterActivation = ENABLE;            // Activate filter
sFilterConfig.SlaveStartFilterBank = 0;             // Assign all CAN Filter Banks to the activated CAN controller (CAN1)
                                                    //            This is used because this MCU has two CAN controllers and this parameter
                                                    //  configures which filter banks go to which controller


        // Error Codes HAL_CAN_ERROR_NOT_INITIALIZED
        // If HAL_CAN_ConfigFilter fails it is because the instance isn't setup
        //              or is in error state. The other option is sleeping but that isn't used
        //              in the application
if(HAL_CAN_ConfigFilter(&hcan1, &sFilterConfig) != HAL_OK)
{
                if(hcan1.State == HAL_CAN_STATE_RESET)
                {
                        // CAN wasn't initaitialized
                        CAN_Setup_Instance();

                        // If filter setup fails again reset device
                        if(HAL_CAN_ConfigFilter(&hcan1, &sFilterConfig) != HAL_OK)
                        {
                                // Filter configuration Error
                                CAN_Error_Handler();
                        }
                }
                else if (hcan1.State == HAL_CAN_STATE_ERROR)
                {

                        if( hcan1.ErrorCode == HAL_CAN_ERROR_TIMEOUT ||
                                  hcan1.ErrorCode == HAL_CAN_ERROR_NOT_INITIALIZED ||
                                        hcan1.ErrorCode == HAL_CAN_ERROR_NOT_READY ||
                                        hcan1.ErrorCode == HAL_CAN_ERROR_NOT_STARTED )
                        {
                                // Try reinitializing
                                CAN_Setup_Instance();

```

```c
                // If filter setup fails again reset device
                if(HAL_CAN_ConfigFilter(&hcan1, &sFilterConfig) != HAL_OK)
                {
                    // Filter configuration Error
                    CAN_Error_Handler();
                }
            }
            else
            {
                //TODO: Log (Should not get here)
                CAN_Error_Handler();
            }
        }
  }
}
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
void CAN_Setup(void)
{
    // 1) CAN_Setup_Instance
    //          Setup CAN instance
    CAN_Setup_Instance();

    // 2) CAN_Setup_Filters
    //          Setup filters
    CAN_Setup_Filters( Heartbeat_Identifier, CAN_RX_FIFO0, 0 );
    CAN_Setup_Filters( Fuse_OK_Identifier, CAN_RX_FIFO0, 1 );
    CAN_Setup_Filters( Fuse_Blown_Identifier, CAN_RX_FIFO1, 2 );
    CAN_Setup_Filters( Error_1_Identifier, CAN_RX_FIFO1, 3 );
    CAN_Setup_Filters( Error_2_Identifier, CAN_RX_FIFO1, 4 );
    CAN_Setup_Filters( Discovery_Ready_Response_Identifier, CAN_RX_FIFO1, 5 );
    CAN_Setup_Filters( Disocvery_Call_Response_Identifier, CAN_RX_FIFO1, 6 );

    HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING | CAN_IT_RX_FIFO1_MSG_PENDING);

    // Error Codes
    //          HAL_CAN_ERROR_TIMEOUT
    //          HAL_CAN_ERROR_NOT_READY
    if (HAL_CAN_Start(&hcan1) != HAL_OK)
    {
        // If return code is HAL_CAN_ERROR_NOT_READY then 1) and 2) were not successful
        CAN_Error_Handler();
    }
}
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
void CAN_Error_Handler(void)
{
        // TODO: Reset device
        while(1)
        {

        }
}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void CAN_State_Handler(void)
{
        switch (hcan1.State)
        {
                case HAL_CAN_STATE_RESET:
                        CAN_Setup();
                        break;
                case HAL_CAN_STATE_READY:

                        break;
                case HAL_CAN_STATE_LISTENING:

                        break;
                case HAL_CAN_STATE_SLEEP_PENDING:
                        // TODO: Log Error
                        // Should not get here
                        break;
                case HAL_CAN_STATE_SLEEP_ACTIVE:
                        // TODO: Log Error
                        // Should not get here
                        break;
                case HAL_CAN_STATE_ERROR:
                        switch(hcan1.ErrorCode)
                        {
                                case HAL_CAN_ERROR_EWG:
                                        // HAL_CAN_ERROR_EWG
                                        //             Protocol Error Warning
                                        //             This bit is set when Receive or Transmit Error Counter is greater than
                                        //              or equal to 96.
                                        break;
                                case HAL_CAN_ERROR_EPV:
                                        // HAL_CAN_ERROR_EPV
                                        //             Error Passive
                                        //             This bit is set when Receive or Transmit Error Counter is greater than
                                        //              127
                                        break;
                                case HAL_CAN_ERROR_BOF:
                                        // HAL_CAN_ERROR_BOF
                                        //             Bus-off error
                                        //             This bit is set when Receive or Transmit Error Counter is greater than
                                        //             255
                                        break;
                                case HAL_CAN_ERROR_STF:
                                        // HAL_CAN_ERROR_STF
                                        //             Stuff error
                                        break;
                                case HAL_CAN_ERROR_FOR:
                                        // HAL_CAN_ERROR_FOR
                                        //             Form error
                                        break;
                                case HAL_CAN_ERROR_ACK:
                                        // HAL_CAN_ERROR_ACK
                                        //             Acknowledgment error
                                        break;
                                case HAL_CAN_ERROR_BR:
                                        // HAL_CAN_ERROR_BR
                                        //             Bit recessive error
                                        break;
                                case HAL_CAN_ERROR_BD:
                                        // HAL_CAN_ERROR_BD
                                        //             Bit dominant error
                                        break;
                                case HAL_CAN_ERROR_CRC:
                                        // HAL_CAN_ERROR_CRC
                                        //             CRC error
                                        break;
                                case HAL_CAN_ERROR_RX_FOV0:
                                        // HAL_CAN_ERROR_RX_FOV0
                                        //             Rx FIFO0 overrun error
                                        break;
                                case HAL_CAN_ERROR_RX_FOV1:
                                        // HAL_CAN_ERROR_RX_FOV1
                                        //             Rx FIFO1 overrun error
                                        break;
                                case HAL_CAN_ERROR_TX_ALST0:
                                        // HAL_CAN_ERROR_TX_ALST0
                                        //             TxMailbox 0 transmit failure due to arbitration lost
                                        break;
                                case HAL_CAN_ERROR_TX_TERR0:
                                        // HAL_CAN_ERROR_TX_TERR0
                                        //             TxMailbox 1 transmit failure due to tranmit error
                                        break;
                                case HAL_CAN_ERROR_TX_ALST1:
                                        // HAL_CAN_ERROR_TX_ALST1
                                        //             TxMailbox 0 transmit failure due to arbitration lost
                                        break;
                                case HAL_CAN_ERROR_TX_TERR1:
```

```
                                         // HAL_CAN_ERROR_TX_TERR1
                                         //             TxMailbox 1 transmit failure due to tranmit error
                                         break;
                                 case HAL_CAN_ERROR_TX_ALST2:
                                         // HAL_CAN_ERROR_TX_ALST2
                                         //             TxMailbox 0 transmit failure due to arbitration lost
                                         break;
                                 case HAL_CAN_ERROR_TX_TERR2:
                                         // HAL_CAN_ERROR_TX_TERR2
                                         //             TxMailbox 1 transmit failure due to tranmit error
                                         break;
                                 case HAL_CAN_ERROR_TIMEOUT:
                                         // HAL_CAN_ERROR_TIMEOUT
                                         //             Timeout error
                                         break;
                                 case HAL_CAN_ERROR_NOT_INITIALIZED:
                                         // HAL_CAN_ERROR_NOT_INITIALIZED
                                         //             Peripheral not initialized
                                         break;
                                 case HAL_CAN_ERROR_NOT_READY:
                                         // HAL_CAN_ERROR_NOT_READY
                                         //             Peripheral not ready
                                         break;
                                 case HAL_CAN_ERROR_NOT_STARTED:
                                         // HAL_CAN_ERROR_NOT_STARTED
                                         //             Peripheral not started
                                         break;
                                 case HAL_CAN_ERROR_PARAM:
                                         // HAL_CAN_ERROR_PARAM
                                         //             Parameter error
                                         break;
                                 default:
                                         break;
                         }
                         break;

                 default:

                          break;
          }

}
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
          BaseType_t xHigherPriorityTaskWoken, xResult;

          /* xHigherPriorityTaskWoken must be initialized to pdFALSE. */
```

```c
        xHigherPriorityTaskWoken = pdFALSE;

        HAL_CAN_DeactivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING);


        /* Set bit 0 and bit 4 in xEventGroup. */
        xResult = xEventGroupSetBitsFromISR(
                                    g_CAN_Message_Event_Group,   /* The event group being updated. */
                                    BIT_FIFO_0, /* The bits being set. */
                                    &xHigherPriorityTaskWoken );

        /* Was the message posted successfully? */
        if( xResult != pdFAIL )
        {
          /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context
          switch should be requested.  The macro used is port specific and will
          be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
          the documentation page for the port being used. */
          portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
        }
        else
        {
                Error_Handler();
        }
}
///////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////
void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
        BaseType_t xHigherPriorityTaskWoken, xResult;

        /* xHigherPriorityTaskWoken must be initialized to pdFALSE. */
        xHigherPriorityTaskWoken = pdFALSE;

        HAL_CAN_DeactivateNotification(&hcan1, CAN_IT_RX_FIFO1_MSG_PENDING);

        /* Set bit 0 and bit 4 in xEventGroup. */
        xResult = xEventGroupSetBitsFromISR(
                                    g_CAN_Message_Event_Group,   /* The event group being updated. */
                                    BIT_FIFO_1, /* The bits being set. */
                                    &xHigherPriorityTaskWoken );

        /* Was the message posted successfully? */
        if( xResult != pdFAIL )
```

```
          {
            /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context
            switch should be requested.  The macro used is port specific and will
            be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
            the documentation page for the port being used. */
            portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
          }
          else
          {
                  Error_Handler();
          }
}
//////////////////////////////////////////////////////////////
```