

Industrial Smart Fuse

Justin Stark, Computer Engineering

Project Advisor: Dr. Anthony Richardson

April 26, 2019
Evansville, Indiana

Table of Contents

- I. Introduction
- II. Background
- III. Design Approach
 - A. Blown Fuse Detection
 - B. Electrical Isolation
 - C. Control Circuit Protection
 - D. Control System Communication
 - E. Inter-module Communication
 - F. Node Indexing
 - G. CAN Communication
 - H. Ethernet Communication
 - I. Control System
 - J. Future Features
- IV. Testing

List of Figures

List of Tables

1. Introduction

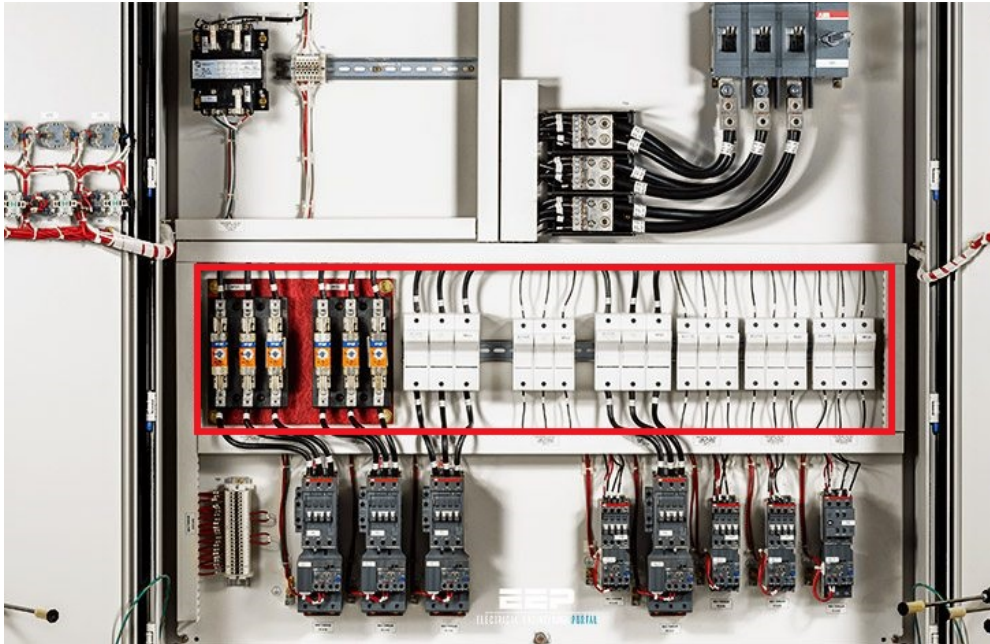
In a report released in 2006 [1], unplanned downtime costs automotive manufacturers an average of \$22,000 a minute. The average amount of time lost due to unplanned downtime is somewhere between 1.5 to 4 hours. The cost of unplanned downtime in manufacturing is measured by accumulating the cost of man 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]

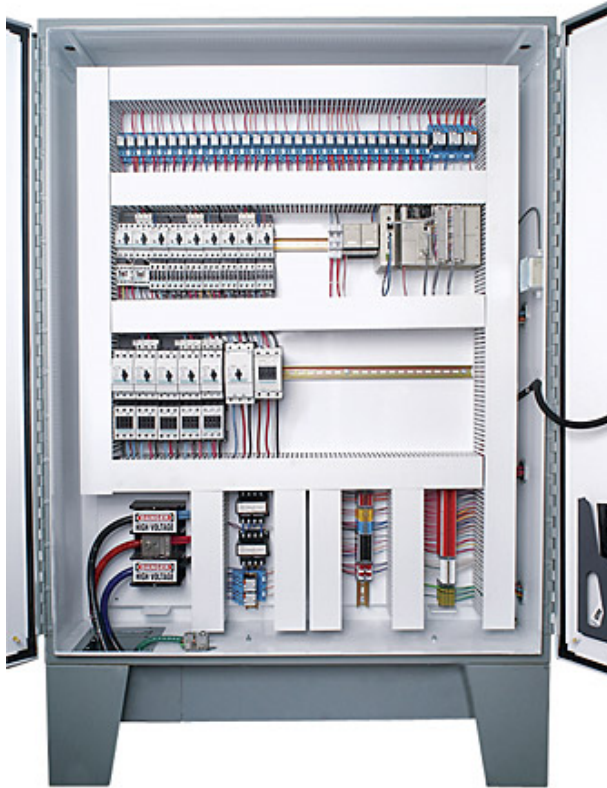
A system of smart fuses that could provide feedback on a fuse's status, and other data collected by the fuse could provide the information needed to resolve hardware issues quickly. The main feature of the smart fuse would be its ability to relay its panel location and rack

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



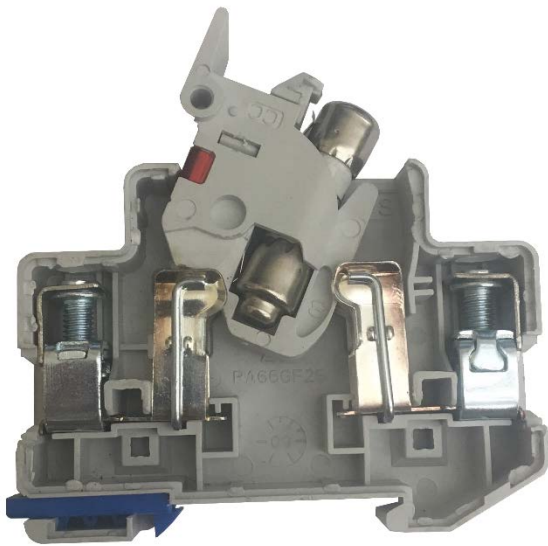
Fuse Holders Inside of a Panel
Figure 1 [5]



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

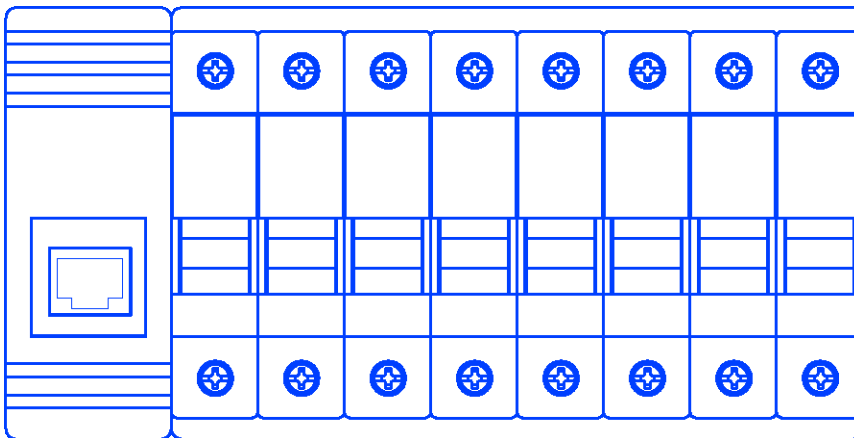
Electrical isolation is the next priority for the device. An electrically isolated device would not only make sure the high voltages of the control circuit does not interfere with the low voltages of the fuse monitoring circuit, but the isolation would also make sure technicians are not in contact with 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.

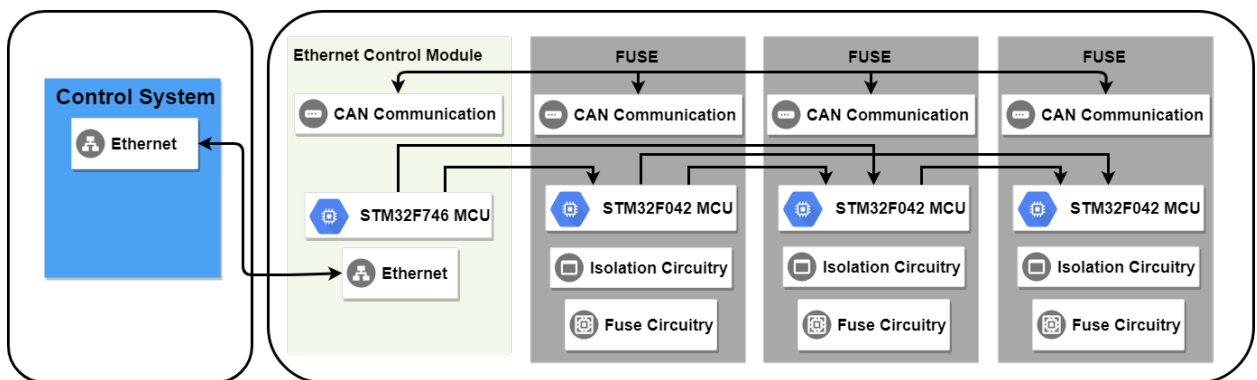
C. Control Circuit Protection

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



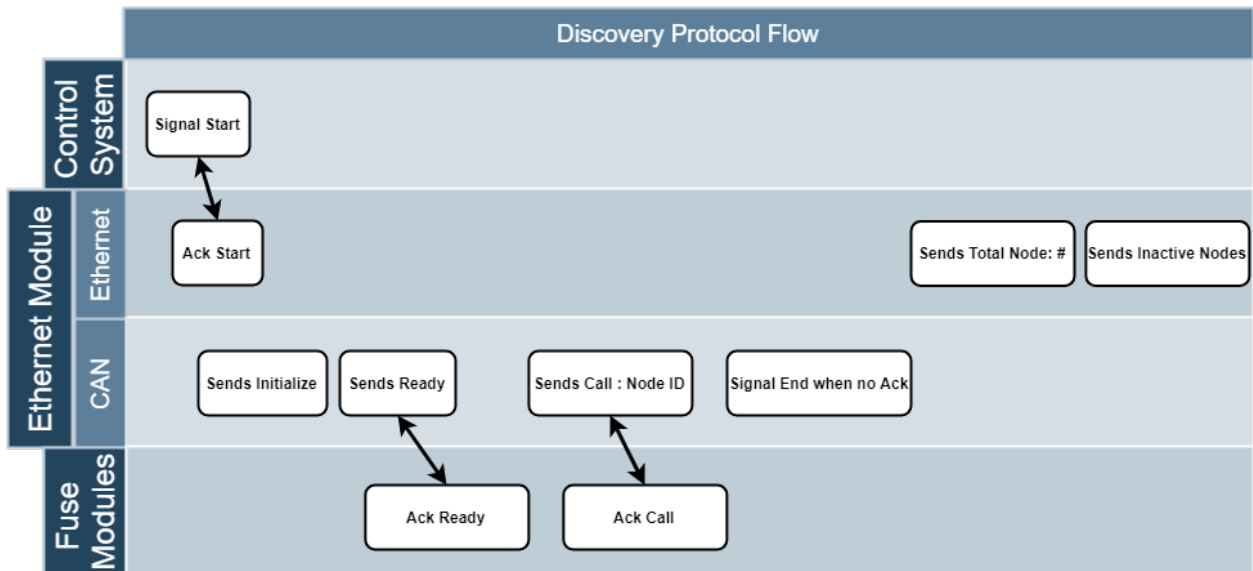
System Diagram of the Industrial Smart Fuse
Figure 4

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.

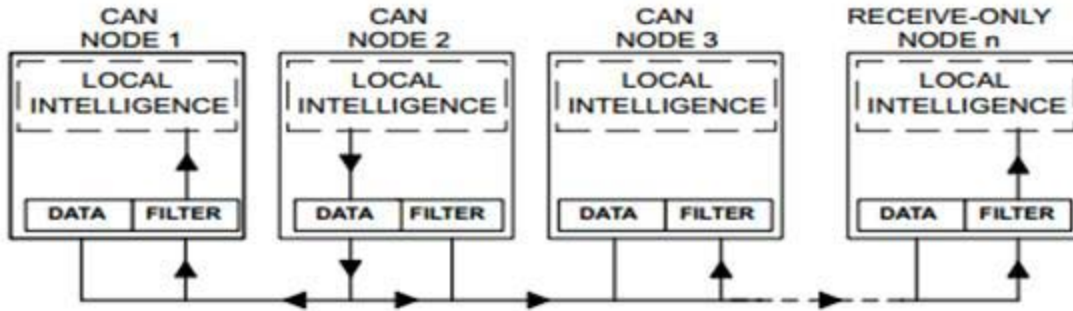
The discovery protocol is used index the fuse nodes and is depicted in figure 5. The discovery protocol is initiated by the control system by sending a message to the ethernet module. After receiving the signal, the ethernet module then sends a messaging initiating the protocol in the fuse modules. Next 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 module 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

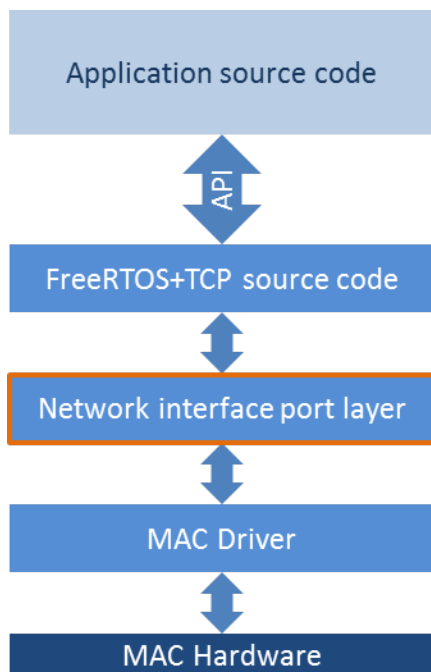


CAN Bus Depiction

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 filter 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

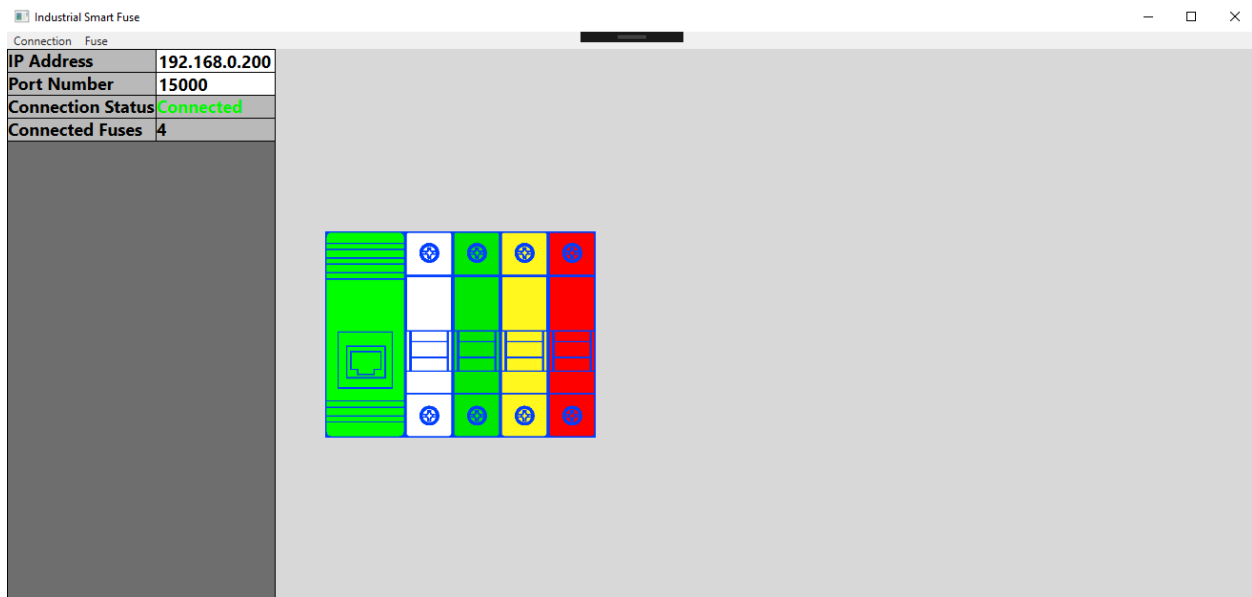


FreeRTOS+TCP Stack Port Layers

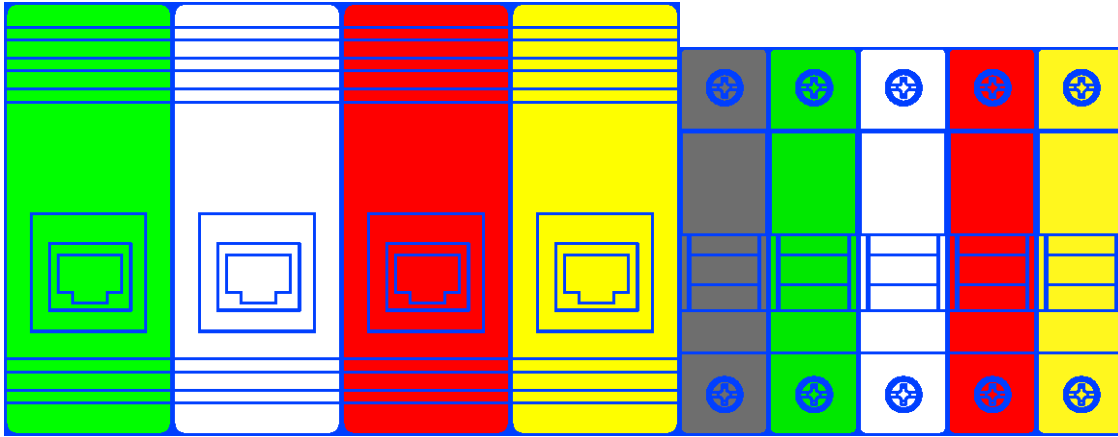
Figure 8

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



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

```
1. // Justin Stark
2. // 4/26/2019
3. // WPF Control System Application
4.
5. using Microsoft.Windows.Controls;
6. using System;
7. using System.Collections;
8. using System.Collections.Generic;
9. using System.ComponentModel;
10. using System.Linq;
11. using System.Net;
12. using System.Net.Sockets;
13. using System.Text;
```



```

14. using System.Threading;
15. using System.Threading.Tasks;
16. using System.Windows;
17. using System.Windows.Controls;
18. using System.Windows.Data;
19. using System.Windows.Documents;
20. using System.Windows.Input;
21. using System.Windows.Media;
22. using System.Windows.Media.Imaging;
23. using System.Windows.Navigation;
24. using System.Windows.Shapes;
25.
26. namespace Control_System_001
27. {
28.
29.     public partial class MainWindow : Window
30.     {
31.
32.         ///////////////////////////////////////////////////////////////////
33.         private delegate void noparamCallback();
34.         private delegate void fuseUpdateCallback(int index, FuseStatus
35.         fs);
36.         ///////////////////////////////////////////////////////////////////
37.
38.         private enum EthernetStatus
39.         {
40.             Connected,
41.             Error,
42.             Unconnected
43.         }
44.         private enum FuseStatus
45.         {
46.             Unconnected,
47.             OK,
48.             Error,
49.             Blown
50.         }
51.         ///////////////////////////////////////////////////////////////////
52.
53.         private enum MessageCodes
54.         {
55.             Heartbeat, // 0
56.             FuseOK, // 1
57.             FuseBlown, //2
58.             Error1, // 3
59.             Error2, // 4
60.             DiscoveryStart, //5
61.             DiscoveryAck, // 6
62.             DiscoveryNodes, // 7
63.             DiscoveryInactive, // 8

```

```

64.         DiscoveryFinished // 9
65.     }
66.
67.     ///////////////////////////////////////////////////////////////////
68.     ///////////////////////////////////////////////////////////////////
69.         BitmapImage[] g_ethernetImages =
70.         { new BitmapImage(new
71. Uri("/Resources/Individual_Ethernet_Green.png", UriKind.Relative)),
72.         new BitmapImage(new
73. Uri("/Resources/Individual_Ethernet_Yellow.png", UriKind.Relative)),
74.         new BitmapImage(new
75. Uri("/Resources/Individual_Ethernet_Original.png", UriKind.Relative)),
76.         };
77.
78.         BitmapImage[] g_fuseImages =
79.         {
80.         new BitmapImage(new
81. Uri("/Resources/Individual_Fuse_Original.png", UriKind.Relative)),
82.         new BitmapImage(new
83. Uri("/Resources/Individual_Fuse_Green.png", UriKind.Relative)),
84.         new BitmapImage(new
85. Uri("/Resources/Individual_Fuse_Yellow.png", UriKind.Relative)),
86.         new BitmapImage(new Uri("/Resources/Individual_Fuse_Red.png",
87. UriKind.Relative)),
88.         };
89.
90.         int g_NumFuses = 0;
91.         ArrayList g_FuseImagesArrayList;
92.         BackgroundWorker g_ethernetWorker, g_discoveryWorker;
93.         Socket g_ethernetSocket;
94.         BusyIndicator g_busyIndicator;
95.         string g_ipAddress;
96.         int g_portNumber;
97.         FuseStatus[] g_nodeStatuses;
98.
99.     ///////////////////////////////////////////////////////////////////
100.    ///////////////////////////////////////////////////////////////////
101.    public MainWindow()
102.    {
103.        InitializeComponent();
104.
105.        g_ethernetWorker = new BackgroundWorker();
106.        g_ethernetWorker.DoWork += new
107.        DoWorkEventHandler(g_ethernetWorker_DoWork);
108.
109.        g_ethernetWorker.WorkerSupportsCancellation = true;
110.
111.        g_discoveryWorker = new BackgroundWorker();
112.        g_discoveryWorker.DoWork += new
113.        DoWorkEventHandler(g_discoveryWorker_DoWork);
114.        g_discoveryWorker.RunWorkerCompleted += new
115.        RunWorkerCompletedEventHandler(g_discoveryWorker_RunWorkerCompleted);

```

```

106.         g_discoveryWorker.WorkerSupportsCancellation = true;
107.
108.         g_FuseImagesArrayList = new ArrayList();
109.         g_busyIndicator = new BusyIndicator();
110.         g_nodeStatuses = new FuseStatus[256];
111.         g_Main_Grid.Children.Add(g_busyIndicator);
112.
113.         EthernetDisconnectedIndicator();
114.     }
115.
116.     ///////////////////////////////////////////////////////////////////
117.     ///////////////////////////////////////////////////////////////////
118.     private void g_ethernetWorker_DoWork(object sender,
DoWorkEventArgs e)
119.     {
120.         try
121.         {
122.
123.             if (g_ethernetSocket == null ||
g_ethernetSocket.Connected == false)
124.             {
125.                 string ipString = g_ipAddress;
126.                 int port = g_portNumber;
127.
128.                 IPAddress ipAddress = IPAddress.Parse(ipString);
129.                 IPEndPoint remoteEP = new IPEndPoint(ipAddress,
port);
130.
131.                 // Create a TCP/IP socket.
132.                 g_ethernetSocket = new Socket(
ipAddress.AddressFamily,
133.                                     SocketType.Stream,
134.                                     ProtocolType.Tcp );
135.
136.                 // Connect to the remote endpoint.
137.                 try
138.                 {
139.                     g_ethernetSocket.Connect(remoteEP);
140.                 }
141.                 catch (Exception)
142.                 {
143.                     BusyIndicatorOff();
144.                     g_ethernetSocket = null;
145.                     System.Windows.MessageBox.Show("Unable to
connect", "Connect");
146.                     return;
147.                 }
148.             }
149.
150.             EthernetConnectedIndicator();
151.             BusyIndicatorOff();
152.
153.             byte[] recBytes = new Byte[1024];
154.

```

```

155.         // 100 ms
156.         g_ethernetSocket.ReceiveTimeout = 100;
157.
158.         while (true)
159.         {
160.             // Receive Data
161.             bool updated = false;
162.             int numRecBytes = 0;
163.             try
164.             {
165.                 numRecBytes = g_ethernetSocket.Receive(recBytes);
166.             }
167.             catch (SocketException) { // TODO }
168.
169.
170.             // Exit if exit requested
171.             if (g_ethernetWorker.CancellationPending)
172.             {
173.                 e.Cancel = true;
174.                 break;
175.             }
176.
177.             // Process data
178.             if (numRecBytes == 0)
179.                 continue;
180.
181.             switch (recBytes[0])
182.             {
183.                 case (byte)MessageCodes.Heartbeat:
184.                     updated = Parse_Byte(numRecBytes, recBytes,
false, FuseStatus.Error);
185.                     break;
186.                 case (byte)MessageCodes.FuseOK:
187.                     updated = Parse_Byte(numRecBytes, recBytes,
true, FuseStatus.OK);
188.                     break;
189.                 case (byte)MessageCodes.FuseBlown:
190.                     updated = Parse_Byte(numRecBytes, recBytes,
true, FuseStatus.Blown);
191.                     break;
192.                 case (byte)MessageCodes.Error1:
193.                     updated = Parse_Byte(numRecBytes, recBytes,
true, FuseStatus.Error);
194.                     break;
195.                 case (byte)MessageCodes.Error2:
196.                     updated = Parse_Byte(numRecBytes, recBytes,
true, FuseStatus.Error);
197.                     break;
198.             }
199.             if (updated)
200.                 Graphics_Update_All_Fuses();
201.         }
202.     }
203.     catch (Exception err)
204.     {
205.         // TODO

```

```

206.             System.Windows.MessageBox.Show("Weird Error", "Ethernet
Worker");
207.         }
208.     }
209.
210.     ///////////////////////////////////////////////////////////////////
211.     ///////////////////////////////////////////////////////////////////
212.     private bool Parse_Byte(int numRecBytes, byte[] recBytes, bool
checkVal, FuseStatus fuseStatus, int offset = 0)
213.     {
214.         int num = 0;
215.         bool updated = false;
216.
217.         if (checkVal)
218.             num = 1;
219.
220.         for (int i = 1; i < numRecBytes; i++)
221.         {
222.             for (int j = 0; j < 8; j++)
223.             {
224.                 if ((recBytes[i + offset] & (1 << j)) == (num << j))
225.                 {
226.                     if (g_nodeStatuses[(i-1) * 8 + j] != fuseStatus)
227.                     {
228.                         g_nodeStatuses[(i-1) * 8 + j] = fuseStatus;
229.                         updated = true;
230.                     }
231.                 }
232.             }
233.         }
234.         return updated;
235.     }
236.
237.     ///////////////////////////////////////////////////////////////////
238.     ///////////////////////////////////////////////////////////////////
239.     private void g_discoveryWorker_DoWork(object sender,
DoWorkEventArgs e)
240.     {
241.         byte[] recBytes = new Byte[1024];
242.         byte startCode = (byte)MessageCodes.DiscoveryStart;
243.         byte[] startCallBytes = { startCode };
244.         int newNumFuses = 0;
245.         int numRecBytes = 0;
246.
247.         for (int i = 0; i < 256; i++)
248.         {
249.             g_nodeStatuses[i] = FuseStatus.Unconnected;
250.         }
251.
252.         // 15 s
253.         g_ethernetSocket.ReceiveTimeout = 15000;
254.

```

```

255.         // Start Discovery Protocol
256.         g_ethernetSocket.Send(startCallBytes);
257.
258.         bool finished = false;
259.         int offset = 0;
260.         try
261.         {
262.             while (finished == false)
263.             {
264.                 offset = 0;
265.                 numRecBytes = g_ethernetSocket.Receive(recBytes);
266.
267.                 while (true)
268.                 {
269.                     // Wait for Ack
270.                     if (recBytes[offset] ==
271. (byte)MessageCodes.DiscoveryAck)
272.                     {
273.                         offset++;
274.                         if (offset >= numRecBytes)
275.                             break;
276.                     }
277.
278.                     // Wait for number of nodes to be sent
279.                     if (recBytes[offset] ==
280. (byte)MessageCodes.DiscoveryNodes)
281.                     {
282.                         newNumFuses = recBytes[offset + 1];
283.                         offset += 2;
284.                         if (offset >= numRecBytes)
285.                             break;
286.                     }
287.
288.                     // Wait for inactive nodes
289.                     if (recBytes[offset] ==
290. (byte)MessageCodes.DiscoveryInactive)
291.                     {
292.                         Parse_Byte(numRecBytes, recBytes, false,
293. FuseStatus.OK, offset);
294.                         offset += 2;
295.                         if (offset >= numRecBytes)
296.                             break;
297.                     }
298.
299.                     // Wait for Finished Message
300.                     if (recBytes[offset] ==
301. (byte)MessageCodes.DiscoveryFinished)
302.                     {
303.                         finished = true;
304.                         offset++;
305.                         if (finished || offset >= numRecBytes)
306.                             break;
307.                     }
308.                 }
309.             }
310.         }

```

```

306.         e.Result = newNumFuses;
307.     }
308.     catch(Exception ex)
309.     {
310.         System.Windows.MessageBox.Show("Discovery Failed: " +
ex.ToString(), "Discovery");
311.     }
312.     }
313.
////////////////////////////////////
314.
315.
////////////////////////////////////
316.     private void g_discoveryWorker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
317.     {
318.         if (e.Result != null)
319.         {
320.             int newNumFuses = (int)e.Result;
321.
322.             // Process Results
323.             Graphics_Change_Fuse_Number(newNumFuses);
324.
325.
326.             Graphics_Update_All_Fuses();
327.
328.             BusyIndicatorOff();
329.         }
330.
331.         // Restart other background worker
332.         g_ethernetWorker.RunWorkerAsync();
333.     }
334.
////////////////////////////////////
335.
336.
////////////////////////////////////
337.     private void Add_Fuse_Click(object sender, RoutedEventArgs e)
338.     {
339.         Graphics_Add_Fuse();
340.     }
341.
////////////////////////////////////
342.
343.
////////////////////////////////////
344.     private void Remove_Fuse_Click(object sender, RoutedEventArgs e)
345.     {
346.         Graphics_Remove_Fuse();
347.     }
348.
////////////////////////////////////
349.
350.
////////////////////////////////////
351.     private void Connect_Click(object sender, RoutedEventArgs e)

```

```

352.         {
353.             g_ipAddress = g_IPAddress_TextBox.Text;
354.             g_portNumber = Int32.Parse(g_PortNumber_TextBox.Text);
355.
356.
357.             // Try connecting if not already trying or connected
358.             BusyIndicatorOn("Connecting to Ethernet Module...");
359.
360.             if (!g_ethernetWorker.IsBusy)
361.                 g_ethernetWorker.RunWorkerAsync();
362.             else if (g_ethernetSocket != null &&
g_ethernetSocket.Connected)
363.                 System.Windows.MessageBox.Show("Connection: Connected",
"Connection Status");
364.             else
365.                 System.Windows.MessageBox.Show("Connection: Pending",
"Connection Status");
366.
367.         }
368.
369.         ///////////////////////////////////////////////////////////////////
370.         ///////////////////////////////////////////////////////////////////
371.         private void Close_Click(object sender, RoutedEventArgs e)
372.         {
373.             if (g_ethernetSocket == null || !g_ethernetSocket.Connected)
374.             {
375.                 System.Windows.MessageBox.Show("Connection not
established", "Close");
376.                 return;
377.             }
378.
379.             g_ethernetWorker.CancelAsync();
380.             g_ethernetSocket.Close();
381.             EthernetDisconnectedIndicator();
382.         }
383.
384.         ///////////////////////////////////////////////////////////////////
385.         ///////////////////////////////////////////////////////////////////
386.         private void Discover_Click(object sender, RoutedEventArgs e)
387.         {
388.             if (g_ethernetSocket == null || !g_ethernetSocket.Connected)
389.             {
390.                 System.Windows.MessageBox.Show("Connection not
established", "Discover");
391.                 return;
392.             }
393.
394.             // Try connecting if not already trying or connected
395.             BusyIndicatorOn("Discovering new nodes...");
396.
397.             g_ethernetWorker.CancelAsync();
398.             g_discoveryWorker.RunWorkerAsync();

```



```

399.         }
400.
401. ///////////////////////////////////////////////////////////////////
402. ///////////////////////////////////////////////////////////////////
403.         private void Busy_Click(object sender, RoutedEventArgs e)
404.         {
405.             BusyIndicatorOn("Discovering Nodes...");
406.         }
407.
408. ///////////////////////////////////////////////////////////////////
409. ///////////////////////////////////////////////////////////////////
410.         private void Busy_Off_Click(object sender, RoutedEventArgs e)
411.         {
412.             BusyIndicatorOff();
413.         }
414.
415. ///////////////////////////////////////////////////////////////////
416. ///////////////////////////////////////////////////////////////////
417.         private void EthernetConnectedIndicator()
418.         {
419.             if
420.             (this.g_Connection_Status_TextBox.Dispatcher.CheckAccess())
421.             {
422.                 g_Connection_Status_TextBox.Text = "Connected";
423.                 g_Connection_Status_TextBox.Foreground = new
SolidColorBrush(Color.FromArgb(0xFF, 0, 0xF7, 0));
424.                 g_Ethernet_Module_Image.Source =
g_ethernetImages[(int)EthernetStatus.Connected];
425.             }
426.             else
427.                 this.g_Connection_Status_TextBox.Dispatcher.BeginInvoke(new
noparamCallback(EthernetConnectedIndicator));
428.         }
429.
430. ///////////////////////////////////////////////////////////////////
431. ///////////////////////////////////////////////////////////////////
432.         private void EthernetDisconnectedIndicator()
433.         {
434.             if
435.             (this.g_Connection_Status_TextBox.Dispatcher.CheckAccess())
436.             {
437.                 g_Connection_Status_TextBox.Text = "Disconnected";
438.                 g_Connection_Status_TextBox.Foreground = new
SolidColorBrush(Color.FromArgb(0xFF, 0xC3, 0, 0));
439.                 g_Ethernet_Module_Image.Source =
g_ethernetImages[(int)EthernetStatus.Unconnected];

```

```

439.         }
440.
441.         else
442.         {
443.             this.g_Connection_Status_TextBox.Dispatcher.BeginInvoke(new
444.                 noparamCallback(EthernetDisconnectedIndicator));
445.         }
446.
447.         ///////////////////////////////////////////////////////////////////
448.         private void EthernetErrorIndicator()
449.         {
450.             g_Connection_Status_TextBox.Text = "Error";
451.             g_Connection_Status_TextBox.Foreground = new
452.                 SolidColorBrush(Color.FromArgb(0xFF, 0xE8, 0xFF, 0));
453.             g_Ethernet_Module_Image.Source =
454.                 g_ethernetImages[(int)EthernetStatus.Error];
455.         }
456.
457.         ///////////////////////////////////////////////////////////////////
458.         void BusyIndicatorOn(string message)
459.         {
460.             g_busyIndicator.BusyContent = message;
461.             g_busyIndicator.IsBusy = true;
462.         }
463.
464.         ///////////////////////////////////////////////////////////////////
465.         private void BusyIndicatorOff()
466.         {
467.             if (this.g_busyIndicator.Dispatcher.CheckAccess())
468.                 this.g_busyIndicator.IsBusy = false;
469.             else
470.                 this.g_busyIndicator.Dispatcher.BeginInvoke(new
471.                     noparamCallback(BusyIndicatorOff));
472.         }
473.
474.         ///////////////////////////////////////////////////////////////////
475.         private void Graphics_Add_Fuse()
476.         {
477.             Image newFuseImage = new Image();
478.             newFuseImage.Source =
479.                 g_fuseImages[(int)FuseStatus.Unconnected];
480.             newFuseImage.Width = 50;
481.             newFuseImage.Stretch = Stretch.Fill;

```

```

481.
482.         g_ModuleHolderStackPanel.Width += 50;
483.         g_ModuleHolderStackPanel.Children.Add(newFuseImage);
484.
485.         g_NumFuses++;
486.     }
487.
488.     ///////////////////////////////////////////////////////////////////
489.     ///////////////////////////////////////////////////////////////////
490.     private void Graphics_Remove_Fuse()
491.     {
492.         g_ModuleHolderStackPanel.Width -= 50;
493.         g_ModuleHolderStackPanel.Children.RemoveAt(g_NumFuses);
494.         g_NumFuses--;
495.     }
496.
497.     ///////////////////////////////////////////////////////////////////
498.     ///////////////////////////////////////////////////////////////////
499.     private void Graphics_Update_All_Fuses()
500.     {
501.         for (int i = 0; i < g_NumFuses; i++)
502.         {
503.             Graphics_Change_Fuse_Status(i, g_nodeStatuses[i]);
504.         }
505.     }
506.
507.     ///////////////////////////////////////////////////////////////////
508.     ///////////////////////////////////////////////////////////////////
509.     private void Graphics_Change_Fuse_Status(int index, FuseStatus
status)
510.     {
511.         if (this.g_ModuleHolderStackPanel.Dispatcher.CheckAccess())
512.             ((Image)g_ModuleHolderStackPanel.Children[index +
1]).Source = g_fuseImages[(int)status];
513.         else
514.             this.g_ModuleHolderStackPanel.Dispatcher.BeginInvoke(new
fuseUpdateCallback(Graphics_Change_Fuse_Status), index, status);
515.     }
516.
517.     ///////////////////////////////////////////////////////////////////
518.     ///////////////////////////////////////////////////////////////////
519.     private void Graphics_Change_Ethernet_Status(EthernetStatus
status)
520.     {
521.         ((Image)g_ModuleHolderStackPanel.Children[0]).Source =
g_ethernetImages[(int)status];
522.     }

```



```

63.
64. //////////////////////////////////////////////////
    //
65. void SystemClock_Config(void)
66. {
67.
68.     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
69.     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
70.
71.     /** Initializes the CPU, AHB and APB busses clocks
72.     */
73.     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI48;
74.     RCC_OscInitStruct.HSI48State = RCC_HSI48_ON;
75.     RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
76.     if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
77.     {
78.         Error_Handler();
79.     }
80.     /** Initializes the CPU, AHB and APB busses clocks
81.     */
82.     RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
83.
84.     |RCC_CLOCKTYPE_PCLK1;
85.     RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI48;
86.     RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
87.     RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
88.     if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
89.     {
90.         Error_Handler();
91.     }
92. }
93. //////////////////////////////////////////////////
    //
94.
95. //////////////////////////////////////////////////
    //
96. void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
97. {
98.     if (htim->Instance == TIM1) {
99.         HAL_IncTick();
100.     }
101. }
102. //////////////////////////////////////////////////
    //
103.
104. //////////////////////////////////////////////////
    //
105. void Error_Handler(void)
106. {
107.     /** User can add his own implementation to report the HAL error return
108.     state */
109.     while(1){}
110. }
    //
    //

```

```

111.
112. ////////////////////////////////////////////////////
113. #ifndef USE_FULL_ASSERT
114. void assert_failed(char *file, uint32_t line)
115. {
116. }
117. #endif
118. ////////////////////////////////////////////////////
119.
120.
121. /**
122.
123. * File Name      : freertos.c
124. * Description    : Code for freertos applications
125.
126. * @attention
127. *
128. *
129.
130. */
131.
132. ////////////////////////////////////////////////////
133. /* Includes -----
134. #include "FreeRTOS.h"
135. #include "task.h"
136. #include "main.h"
137. #include "cmsis_os.h"
138. #include "adc.h"
139. #include "can.h"
140. #include "stm32f0xx_hal_can.h"
141. ////////////////////////////////////////////////////
142.
143. ////////////////////////////////////////////////////
144. /* Extern -----
145. extern uint8_t Node_ID;
146. ////////////////////////////////////////////////////
147.
148. ////////////////////////////////////////////////////
149. /* Private typedef -----
150. typedef enum

```



```

200.
201. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
202. /* Hook prototypes */
203. void configureTimerForRunTimeStats(void);
204. unsigned long getRunTimeCounterValue(void);
205. void vApplicationStackOverflowHook(xTaskHandle xTask, signed char
    *pcTaskName);
206. void vApplicationMallocFailedHook(void);
207. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
208.
209. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
210. /* Functions needed when configGENERATE_RUN_TIME_STATS is on */
211. void configureTimerForRunTimeStats(void)
212. {
213.
214. }
215. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
216.
217. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
218. unsigned long getRunTimeCounterValue(void)
219. {
220.     return 0;
221. }
222. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
223.
224. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
225. void vApplicationStackOverflowHook(xTaskHandle xTask, signed char
    *pcTaskName)
226. {
227.     /* Run time stack overflow checking is performed if
228.     configCHECK_FOR_STACK_OVERFLOW is defined to 1 or 2. This hook
    function is
229.     called if a stack overflow is detected. */
230.     while(1) {
231.     }
232. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
233.
234. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
235. void vApplicationMallocFailedHook(void)
236. {
237.     while(1) {
238.     }
239. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
240.
241. ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////

```



```

289. ///////////////////////////////////////////////////////////////////
290. ///////////////////////////////////////////////////////////////////
291. static void ADC_Task( void *pvParameters )
292. {
293.     EventBits_t uxBits;
294.     adc_set_t adcSet;
295.
296.     /* Attempt to create the event group. */
297.     ADC_MessageEventGroup = xEventGroupCreate();
298.     if(ADC_MessageEventGroup == NULL)
299.     {
300.         // TODO: error check (Insufficient Heap)
301.     }
302.     g_ADC_Start_Event = xEventGroupCreate();
303.     if(g_ADC_Start_Event == NULL)
304.     {
305.         // TODO: error check (Insufficient Heap)
306.     }
307.     for(;;)
308.     {
309.         ADC_Start();
310.
311.         const TickType_t xDelay = 2000 / portTICK_PERIOD_MS;
312.         vTaskDelay( xDelay );
313.
314.
315.         // Queue will block until conversion is complete
316.         if( xQueueReceive( Queue_ADC, &( adcSet ), ( TickType_t
) portMAX_DELAY ) )
317.         {
318.             if( adcSet.valueOne <= 1.5 * adcSet.valueTwo &&
adcSet.valueOne >= .5 * adcSet.valueTwo)
319.             {
320.                 // Good Fuse
321.                 xEventGroupSetBits(
ADC_MessageEventGroup,
322.                 BIT_ADC_OK );
323.             }
324.             else
325.             {
326.                 // Bad Fuse
327.                 xEventGroupSetBits(
ADC_MessageEventGroup,
328.                 BIT_ADC_BLOWN );
329.             }
330.         }
331.         else
332.         {
333.             xEventGroupSetBits( ADC_MessageEventGroup,
334.                                 BIT_ERROR
) );
335.         }
336.     }

```

```

337. }
338. //////////////////////////////////////////////////
339. //////////////////////////////////////////////////
340.
341. //////////////////////////////////////////////////
342. static void CAN_Task( void *pvParameters )
343. {
344.     EventBits_t uxBits;
345.     BaseType_t returnType;
346.     uint8_t dataNode[1];
347.
348.     for(;;)
349.     {
350.         /* Wait a maximum of 50 days for either bit 0-2 to be
set within
351.         the event group. Clear the bits before exiting. */
352.         uxBits = xEventGroupWaitBits(
353.             ADC_MessageEventGroup, /* The event group being tested. */
354.             BIT_STATUS | BIT_ADC_BLOWN | BIT_ADC_OK | BIT_ERROR, /* The bits within
the event group to wait for. */
355.             pdTRUE, /* BIT_ADC_BLOWN | BIT_ADC_OK | BIT_ADC_ERROR should be
cleared before returning. */
356.             pdFALSE, /* Any bit will trigger */
357.             portMAX_DELAY ); /* Wait a maximum of 50 days */
358.
359.         if( (uxBits & BIT_ADC_BLOWN) == BIT_ADC_BLOWN)
360.         {
361.             xQueueReset( Queue_Fuse_Blown_Ack );
362.
363.             dataNode[0] = Node_ID;
364.             CAN_Message_Send( Fuse_Blown_Identifier,
365.                               1, // Data length
366.                               dataNode );
367.
368.             returnType = xQueueReceive(
Queue_Fuse_Blown_Ack, &canMessage , ( TickType_t ) 250 );
369.
370.             if(returnType == pdFAIL)
371.             {
372.                 dataNode[0] = Node_ID;
373.                 CAN_Message_Send(
Fuse_Blown_Identifier,
374.                                     1, //
Data length
375.                                     dataNode );
376.
377.                 returnType = xQueueReceive(
Queue_Fuse_Blown_Ack, &canMessage , ( TickType_t ) 250 );

```



```

430.             continue;
431.
432.             if( (discoveryMessage.StdId ==
Discovery_Ready_1_Identifier) && (currentNode == true) && (readyOne ==
false) )
433.                 {
434.                     HAL_GPIO_WritePin( DISC_OUTPUT_PORT,
DISC_OUTPUT_1, GPIO_PIN_SET );
435.
436.                     dataNode[0] = Node_ID;
437.                     CAN_Message_Send(
Discovery_Ready_Response_Identifier,
438.
439.                         1, // Data length
440.
441.                         dataNode );
442.                     readyOne = true;
443.                 }
444.             else if( (discoveryMessage.StdId ==
Discovery_Ready_2_Identifier) && (currentNode == true) && (readyTwo ==
false) )
445.                 {
446.                     HAL_GPIO_WritePin( DISC_OUTPUT_PORT,
DISC_OUTPUT_2, GPIO_PIN_SET );
447.
448.                     dataNode[0] = Node_ID;
449.                     CAN_Message_Send(
Discovery_Ready_Response_Identifier,
450.
451.                         1, // Data
452.                         length
453.                         dataNode );
454.                     readyTwo = true;
455.                 }
456.             else if( (discoveryMessage.StdId ==
Discovery_Call_Identifier) && (currentNode == false) )
457.                 {
458.                     // Check both pins for input
459.                     pinState = HAL_GPIO_ReadPin( DISC_INPUT_PORT,
DISC_INPUT_1 ) | HAL_GPIO_ReadPin( DISC_INPUT_PORT, DISC_INPUT_2 );
460.
461.                     if( pinState == GPIO_PIN_SET )
462.                     {
463.                         Node_ID = discoveryMessage.Data[0];
464.
465.                         dataNode[0] = Node_ID;
466.                         CAN_Message_Send(
Discovery_Call_Response_Identifier,
467.
468.                             1, //
469.                             Data length
470.                             dataNode );
471.
472.                         currentNode = true;

```

```

470.         }
471.
472.     }
473.     else if( (discoveryMessage.StdId ==
Discovery_End_Identifier) )
474.     {
475.         currentNode = false;
476.         readyOne = false;
477.         readyTwo = false;
478.
479.         HAL_GPIO_WritePin( DISC_OUTPUT_PORT,
DISC_OUTPUT_1, GPIO_PIN_RESET);
480.         HAL_GPIO_WritePin( DISC_OUTPUT_PORT,
DISC_OUTPUT_2, GPIO_PIN_RESET);
481.
482.         xQueueReset(Queue_Discovery);
483.
484.         // Start the ADC task
485.         xEventGroupSetBits(
486.             g_ADC_Start_Event, /* The event group being updated. */
487.             BIT_0
/* The bits being set. */
488.             );
489.     }
490.     else if( (discoveryMessage.StdId ==
Discovery_Start_Identifier) )
491.     {
492.         currentNode = false;
493.         readyOne = false;
494.         readyTwo = false;
495.
496.         /* Clear bit 0 in g_Start_Event. */
497.         xEventGroupClearBits(
498.             g_ADC_Start_Event, /* The event group being updated. */
499.             BIT_0
/* The bits being cleared. */
500.             );
501.     }
502. }
503. }
504. ///////////////////////////////////////////////////////////////////
505. ///////////////////////////////////////////////////////////////////
506. ///////////////////////////////////////////////////////////////////
507. void HAL_CAN_RxCpltCallback(CAN_HandleTypeDef* hcan)
508. {
509.     CanRxMsgTypeDef canMessage;
510.     BaseType_t xHigherPriorityTaskWoken1 = pdFALSE,
xHigherPriorityTaskWoken2 = pdFALSE;
511.     uint8_t fifo = 0;
512.
513.
514.     if(hcan->pRxMsg->StdId == 0)

```

```

515.     {
516.         canMessage = *(hcan->pRx1Msg);
517.         fifo = 1;
518.     }
519.     else
520.         canMessage = *(hcan->pRxMsg);
521.
522.     QueueHandle_t *pqueue = Switch_Queue(canMessage);
523.
524.     if(pqueue == &Queue_Status)
525.     {
526.
527.         /* xHigherPriorityTaskWoken must be initialised to
528.         pdFALSE. */
529.         xHigherPriorityTaskWoken1 = pdFALSE;
530.
531.         /* Set bit 0 and bit 4 in xEventGroup. */
532.         xEventGroupSetBitsFromISR(
533.             ADC_MessageEventGroup, /* The event group being updated. */
534.             BIT_STATUS, /* The bits being set. */
535.             &xHigherPriorityTaskWoken1
536.             );
537.
538.         xQueueSendFromISR( *pqueue,
539.                             &canMessage,
540.                             &xHigherPriorityTaskWoken2 );
541.
542.
543.         if(fifo == 0 && HAL_CAN_Receive_IT(hcan, CAN_FIFO0) != HAL_OK)
544.         {
545.             Error_Handler();
546.         }
547.         else if(fifo == 1 && HAL_CAN_Receive_IT(hcan, CAN_FIFO1) !=
548.             HAL_OK)
549.         {
550.             Error_Handler();
551.         }
552.         if( ( xHigherPriorityTaskWoken1 | xHigherPriorityTaskWoken2 )
553.             != 0 )
554.         {
555.             vPortYield();
556.         }
557.         ///////////////////////////////////////////////////
558.         ///////////////////////////////////////////////////
559.         ///////////////////////////////////////////////////
560.         static QueueHandle_t * Switch_Queue(CanRxMsgTypeDef header)
561.         {

```



```

562.         QueueHandle_t *hqueue = &Queue_Status;
563.
564.         switch(header.StdId)
565.         {
566.             case Fuse_Blown_Ack_Identifier:
567.                 hqueue = &Queue_Fuse_Blown_Ack;
568.                 break;
569.             case Status_Update_Identifier:
570.                 hqueue = &Queue_Status;
571.                 break;
572.             case Discovery_Ready_1_Identifier:
573.                 hqueue = &Queue_Discovery;
574.                 break;
575.             case Discovery_Ready_2_Identifier:
576.                 hqueue = &Queue_Discovery;
577.                 break;
578.             case Discovery_Call_Identifier:
579.                 hqueue = &Queue_Discovery;
580.                 break;
581.             case Discovery_End_Identifier:
582.                 hqueue = &Queue_Discovery;
583.                 break;
584.             case Discovery_Start_Identifier:
585.                 hqueue = &Queue_Discovery;
586.                 break;
587.             default:
588.                 break;
589.         }
590.         return hqueue;
591.     }
592.     //////////////////////////////////////
593.     //
594.     /**
595.     ****
596.     * @file    stm32f0xx_it.c
597.     * @brief   Interrupt Service Routines.
598.     ****
599.     * @attention
600.     *
601.     *
602.     ****
603.     */
604.
605.     //////////////////////////////////////
606.     /* Includes -----
607.     -----*/
607.     #include "main.h"
608.     #include "stm32f0xx_it.h"

```

```
609. #include "cmsis_os.h"
610. ///////////////////////////////////////////////////////////////////
611. ///////////////////////////////////////////////////////////////////
612. ///////////////////////////////////////////////////////////////////
613. /* External variables -----
-----*/
614. extern CAN_HandleTypeDef hcan;
615. extern WWDG_HandleTypeDef hwwdg;
616. extern TIM_HandleTypeDef htim1;
617. ///////////////////////////////////////////////////////////////////
618. ///////////////////////////////////////////////////////////////////
619. ///////////////////////////////////////////////////////////////////
620. void NMI_Handler(void)
621. {
622. }
623. ///////////////////////////////////////////////////////////////////
624. ///////////////////////////////////////////////////////////////////
625. ///////////////////////////////////////////////////////////////////
626. void HardFault_Handler(void)
627. {
628.     while (1)
629.     {
630.     }
631. }
632. ///////////////////////////////////////////////////////////////////
633. ///////////////////////////////////////////////////////////////////
634. ///////////////////////////////////////////////////////////////////
635. void WWDG_IRQHandler(void)
636. {
637.     HAL_WWDG_IRQHandler(&hwwdg);
638. }
639. ///////////////////////////////////////////////////////////////////
640. ///////////////////////////////////////////////////////////////////
641. ///////////////////////////////////////////////////////////////////
642. void TIM1_BRK_UP_TRG_COM_IRQHandler(void)
643. {
644.     HAL_TIM_IRQHandler(&htim1);
645. }
646. ///////////////////////////////////////////////////////////////////
647. ///////////////////////////////////////////////////////////////////
648. ///////////////////////////////////////////////////////////////////
649. void CEC_CAN_IRQHandler(void)
650. {
651.     HAL_CAN_IRQHandler(&hcan);
```



```

691.  *
692.  *
693.  *****
        *****
694.  */
695.
696.  /* Includes -----
        -----*/
697.  #include "stm32f0xx_hal.h"
698.  #include "stm32f0xx_hal_tim.h"
699.
700.
701.  /* Private variables -----
        -----*/
702.  TIM_HandleTypeDef          htim1;
703.
704.  //////////////////////////////////////
        //////////////////////////////////////
705.  HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
706.  {
707.      RCC_ClkInitTypeDef      clkconfig;
708.      uint32_t                 uwTimclock = 0;
709.      uint32_t                 uwPrescalerValue = 0;
710.      uint32_t                 pFLatency;
711.
712.      /*Configure the TIM1 IRQ priority */
713.      HAL_NVIC_SetPriority(TIM1_BRK_UP_TRG_COM_IRQn, TickPriority ,0);
714.
715.      /* Enable the TIM1 global Interrupt */
716.      HAL_NVIC_EnableIRQ(TIM1_BRK_UP_TRG_COM_IRQn);
717.
718.      /* Enable TIM1 clock */
719.      __HAL_RCC_TIM1_CLK_ENABLE();
720.
721.      /* Get clock configuration */
722.      HAL_RCC_GetClockConfig(&clkconfig, &pFLatency);
723.
724.      /* Compute TIM1 clock */
725.      uwTimclock = HAL_RCC_GetPCLK1Freq();
726.
727.      /* Compute the prescaler value to have TIM1 counter clock equal to
        1MHz */
728.      uwPrescalerValue = (uint32_t) ((uwTimclock / 1000000) - 1);
729.
730.      /* Initialize TIM1 */
731.      htim1.Instance = TIM1;
732.
733.      /* 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
734.      */
735.      htim1.Init.Period = (1000000 / 1000) - 1;
736.      htim1.Init.Prescaler = uwPrescalerValue;

```



```

784. #include "gpio.h"
785. ///////////////////////////////////////////////////////////////////
786. ///////////////////////////////////////////////////////////////////
787.
788. ///////////////////////////////////////////////////////////////////
789. void MX_GPIO_Init(void)
790. {
791.
792.     GPIO_InitTypeDef GPIO_InitStruct = {0};
793.
794.     /* GPIO Ports Clock Enable */
795.     __HAL_RCC_GPIOF_CLK_ENABLE();
796.     __HAL_RCC_GPIOA_CLK_ENABLE();
797.     __HAL_RCC_GPIOB_CLK_ENABLE();
798.
799.     /*Configure GPIO pin Output Level */
800.     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5|GPIO_PIN_6,
GPIO_PIN_RESET);
801.
802.     /*Configure GPIO pin Output Level */
803.     HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
804.
805.     /*Configure GPIO pins : PB4 PB5 */
806.     GPIO_InitStruct.Pin = DISC_INPUT_1|DISC_INPUT_2;
807.     GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
808.     GPIO_InitStruct.Pull = GPIO_NOPULL;
809.     HAL_GPIO_Init(DISC_INPUT_PORT, &GPIO_InitStruct);
810.
811.     /*Configure GPIO pins : PA5 PA6 */
812.     GPIO_InitStruct.Pin = DISC_OUTPUT_1|DISC_OUTPUT_2;
813.     GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
814.     GPIO_InitStruct.Pull = GPIO_NOPULL;
815.     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
816.     HAL_GPIO_Init(DISC_OUTPUT_PORT, &GPIO_InitStruct);
817.
818.     /*Configure GPIO pin : PtPin */
819.     GPIO_InitStruct.Pin = LD3_Pin;
820.     GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
821.     GPIO_InitStruct.Pull = GPIO_NOPULL;
822.     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
823.     HAL_GPIO_Init(LD3_GPIO_Port, &GPIO_InitStruct);
824.
825. }
826. ///////////////////////////////////////////////////////////////////
827. ///////////////////////////////////////////////////////////////////
828. /**
829.
830. *****
831. * File Name           : CAN.c
832. * Description        : This file provides code for the configuration
of the CAN instances.

```

```

833. *****
      ****
834.   * @attention
835.   *
836.   *
837. *****
      ****
838.   */
839.
840. //////////////////////////////////////
      //
841. /* Includes -----
      -----*/
842. #include "can.h"
843. //////////////////////////////////////
      //
844.
845. //////////////////////////////////////
      //
846. typedef enum
847. {
848.     Zero_Identifier = 0,
849.     Heartbeat_Identifier = 1,
850.     Fuse_OK_Identifier = 2,
851.     Fuse_Blown_Identifier = 3,
852.     Fuse_Blown_Ack_Identifier = 4,
853.     Error_1_Identifier = 5,
854.     Error_2_Identifier = 6,
855.     Status_Update_Identifier = 7,
856.     Discovery_Start_Identifier = 8,
857.     Discovery_Ready_1_Identifier = 9,
858.     Discovery_Ready_2_Identifier = 10,
859.     Discovery_Ready_Response_Identifier = 11,
860.     Discovery_Call_Identifier = 12,
861.     Discovery_Call_Response_Identifier = 13,
862.     Discovery_End_Identifier = 14
863. } IdentifierCodes_Type;
864. //////////////////////////////////////
      //
865.
866.
867. //////////////////////////////////////
      //
868. CAN_HandleTypeDef hcan;
869. CanRxMsgTypeDef Rx0;
870. CanRxMsgTypeDef Rx1;
871. CanTxMsgTypeDef Tx;
872. //////////////////////////////////////
      //
873.
874. //////////////////////////////////////
      //
875. /* CAN init function */
876. void MX_CAN_Init(void)

```



```

929. //////////////////////////////////////////////////
930. void HAL_CAN_MspDeInit(CAN_HandleTypeDef* canHandle)
931. {
932.     if(canHandle->Instance==CAN)
933.     {
934.
935.         /* Peripheral clock disable */
936.         __HAL_RCC_CAN1_CLK_DISABLE();
937.
938.         /**CAN GPIO Configuration
939.         PA11      -----> CAN_RX
940.         PA12      -----> CAN_TX
941.         */
942.         HAL_GPIO_DeInit(GPIOA, GPIO_PIN_11|GPIO_PIN_12);
943.
944.         /* CAN interrupt Deinit */
945.         HAL_NVIC_DisableIRQ(CEC_CAN_IRQn);
946.     }
947. }
948. //////////////////////////////////////////////////
949. //////////////////////////////////////////////////
950. //////////////////////////////////////////////////
951. void CAN_Message_Send(uint32_t stdID, uint32_t length, uint8_t *data)
952. {
953.     Tx.StdId = stdID;
954.     Tx.RTR = CAN_RTR_DATA;
955.     Tx.IDE = CAN_ID_STD;
956.     Tx.DLC = length;
957.
958.     for(int i = 0; i < length; i++)
959.     {
960.         Tx.Data[i] = data[i];
961.     }
962.
963.     /* Request transmission */
964.     hcan.pTxMsg = &Tx;
965.
966.     if(HAL_CAN_Transmit(&hcan, 50) != HAL_OK)
967.     {
968.         // TODO
969.     }
970. }
971. //////////////////////////////////////////////////
972. //////////////////////////////////////////////////
973. //////////////////////////////////////////////////
974. void CAN_Receive_Setup(void)
975. {
976.     hcan.pRxMsg = &Rx0;
977.     hcan.pRx1Msg = &Rx1;
978.
979.     if(HAL_CAN_Receive_IT(&hcan, CAN_FIFO0) != HAL_OK)

```

```

980.         {
981.             Error_Handler();
982.         }
983.         if(HAL_CAN_Receive_IT(&hcan, CAN_FIFO1) != HAL_OK)
984.         {
985.             Error_Handler();
986.         }
987.     }
988.     //////////////////////////////////////
989.     //////////////////////////////////////
990.
991.     //////////////////////////////////////
992.     void CAN_Filter_Setup(void)
993.     {
994.         CAN_Filter_Setup_LL( Fuse_Blown_Ack_Identifier,
995.             CAN_FILTER_FIFO0, 0);
996.         CAN_Filter_Setup_LL( Status_Update_Identifier,
997.             CAN_FILTER_FIFO0, 1);
998.         CAN_Filter_Setup_LL( Discovery_Start_Identifier,
999.             CAN_FILTER_FIFO1, 2);
1000.        CAN_Filter_Setup_LL( Discovery_Ready_1_Identifier,
1001.            CAN_FILTER_FIFO1, 3);
1002.        CAN_Filter_Setup_LL( Discovery_Ready_2_Identifier,
1003.            CAN_FILTER_FIFO1, 4);
1004.        CAN_Filter_Setup_LL( Discovery_Call_Identifier,
1005.            CAN_FILTER_FIFO1, 5);
1006.        CAN_Filter_Setup_LL( Discovery_End_Identifier,
1007.            CAN_FILTER_FIFO1, 6);
1008.    }
1009.    //////////////////////////////////////
1010.    //////////////////////////////////////
1011.
1012.    void CAN_Filter_Setup_LL(uint32_t StdID, uint32_t fifoSelection,
1013.        uint32_t filterBank)
1014.    {
1015.        CAN_FilterConfTypeDef  sFilterConfig;
1016.
1017.        sFilterConfig.FilterNumber = filterBank;
1018.        sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
1019.        sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
1020.        sFilterConfig.FilterIdHigh = StdID << 5;
1021.        sFilterConfig.FilterIdLow = 0x0000;
1022.        sFilterConfig.FilterMaskIdHigh = 0xFFFF;
1023.        sFilterConfig.FilterMaskIdLow = 0xFFFF;
1024.        sFilterConfig.FilterFIFOAssignment = fifoSelection;
1025.        sFilterConfig.FilterActivation = ENABLE;
1026.
1027.        if(HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)
1028.        {
1029.            /* Filter configuration Error */
1030.            Error_Handler();

```

```

1024.     }
1025. }
1026. //////////////////////////////////////////////////
1027. //////////////////////////////////////////////////
1028. /**
1029.
1030.     ****
1031.     * File Name           : ADC.c
1032.     * Description        : This file provides code for the configuration
1033.     *                     of the ADC instances.
1034.     ****
1035.     * @attention
1036.     *
1037.     ****
1038.     */
1039. //////////////////////////////////////////////////
1040. /* Includes -----
1041. -----*/
1042. #include "adc.h"
1043. //////////////////////////////////////////////////
1044. //////////////////////////////////////////////////
1045. ADC_HandleTypeDef hadc;
1046. extern QueueHandle_t Queue_ADC;
1047. //////////////////////////////////////////////////
1048. //////////////////////////////////////////////////
1049. //////////////////////////////////////////////////
1050. /* ADC init function */
1051. void MX_ADC_Init(void)
1052. {
1053.     ADC_ChannelConfTypeDef sConfig = {0};
1054.
1055.     /** Configure the global features of the ADC (Clock, Resolution, Data
1056.     Alignment and number of conversion)
1057.     */
1058.     hadc.Instance = ADC1;
1059.     hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
1060.     hadc.Init.Resolution = ADC_RESOLUTION_12B;
1061.     hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
1062.     hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;
1063.     hadc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
1064.     hadc.Init.LowPowerAutoWait = DISABLE;
1065.     hadc.Init.LowPowerAutoPowerOff = DISABLE;
1066.     hadc.Init.ContinuousConvMode = DISABLE;

```

```

1066. hadc.Init.DiscontinuousConvMode = DISABLE;
1067. hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START;
1068. hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
1069. hadc.Init.DMAContinuousRequests = DISABLE;
1070. hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
1071. if (HAL_ADC_Init(&hadc) != HAL_OK)
1072. {
1073.     Error_Handler();
1074. }
1075. /** Configure for the selected ADC regular channel to be converted.
1076. */
1077. sConfig.Channel = ADC_CHANNEL_3;
1078. sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
1079. sConfig.SamplingTime = ADC_SAMPLETIME_239CYCLES_5;
1080. if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
1081. {
1082.     Error_Handler();
1083. }
1084. /** Configure for the selected ADC regular channel to be converted.
1085. */
1086. sConfig.Channel = ADC_CHANNEL_4;
1087. if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
1088. {
1089.     Error_Handler();
1090. }
1091. }
1092. ////////////////////////////////////////////////////
1093. ////////////////////////////////////////////////////
1094. ////////////////////////////////////////////////////
1095. void HAL_ADC_MspInit(ADC_HandleTypeDef* adcHandle)
1096. {
1097.
1098.     GPIO_InitTypeDef GPIO_InitStructure = {0};
1099.     if(adcHandle->Instance==ADC1)
1100.     {
1101.         /* ADC1 clock enable */
1102.         __HAL_RCC_ADC1_CLK_ENABLE();
1103.
1104.         __HAL_RCC_GPIOA_CLK_ENABLE();
1105.         /**ADC GPIO Configuration
1106.         PA0      -----> ADC_IN0
1107.         PA1      -----> ADC_IN1
1108.         */
1109.         GPIO_InitStructure.Pin = GPIO_PIN_4|GPIO_PIN_3;
1110.         GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
1111.         GPIO_InitStructure.Pull = GPIO_NOPULL;
1112.         HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
1113.
1114.         /* ADC1 interrupt Init */
1115.         HAL_NVIC_SetPriority(ADC1_IRQn, 3, 0);
1116.         HAL_NVIC_EnableIRQ(ADC1_IRQn);
1117.
1118.     }
1119. }

```

```

1120. ////////////////////////////////////////////////////
1121. ////////////////////////////////////////////////////
1122. ////////////////////////////////////////////////////
1123. void HAL_ADC_MspDeInit(ADC_HandleTypeDef* adcHandle)
1124. {
1125.
1126.     if(adcHandle->Instance==ADC1)
1127.     {
1128.         /* Peripheral clock disable */
1129.         __HAL_RCC_ADC1_CLK_DISABLE();
1130.
1131.         /**ADC GPIO Configuration
1132.         PA0      -----> ADC_IN0
1133.         PA1      -----> ADC_IN1
1134.         */
1135.         HAL_GPIO_DeInit(GPIOA, GPIO_PIN_4|GPIO_PIN_3);
1136.
1137.         /* ADC interrupt Deinit */
1138.         HAL_NVIC_DisableIRQ(ADC1_IRQn);
1139.     }
1140. }
1141. ////////////////////////////////////////////////////
1142. ////////////////////////////////////////////////////
1143. ////////////////////////////////////////////////////
1144. void ADC_Start(void)
1145. {
1146.     HAL_ADC_Start_IT( &hadc );
1147. }
1148. ////////////////////////////////////////////////////
1149. ////////////////////////////////////////////////////
1150. ////////////////////////////////////////////////////
1151. void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
1152. {
1153.     static adc_set_t adcSet;
1154.     static int flag = 0;
1155.     BaseType_t xHigherPriorityTaskWoken;
1156.     static int finished = 0;
1157.
1158.     /* We have not woken a task at the start of the ISR. */
1159.     xHigherPriorityTaskWoken = pdFALSE;
1160.
1161.     if( __HAL_ADC_GET_FLAG(hadc, ADC_FLAG_EOC) != 0)
1162.     {
1163.         if(flag == 0)
1164.         {
1165.             adcSet.valueOne = HAL_ADC_GetValue(hadc);
1166.             flag = 1;
1167.         }
1168.         else if(flag == 1)
1169.         {

```

```

1170.             adcSet.valueTwo = HAL_ADC_GetValue(hadc);
1171.             flag = 0;
1172.             finished = 1;
1173.         }
1174.     }
1175. }
1176.     if(finished == 1)
1177.     {
1178.         finished = 0;
1179.         xQueueSendFromISR( Queue_ADC,
1180.                             &adcSet,
1181.                             &xHigherPriorityTaskWoken);
1182.
1183.
1184.         if( xHigherPriorityTaskWoken )
1185.         {
1186.             vPortYield();
1187.         }
1188.     }
1189. }
1190. ///////////////////////////////////////////////////
1191. ///////////////////////////////////////////////////
1192. ///////////////////////////////////////////////////
1193. void ADC1_IRQHandler ( void )
1194. {
1195.     HAL_ADC_IRQHandler(&hadc);
1196. }
1197. ///////////////////////////////////////////////////
1198. ///////////////////////////////////////////////////

```

```

1. // Justin Stark
2. // 4/26/2019
3. // STM32F746 Ethernet Module
4.
5. /**
6.
7. * @file           : main.c
8. * @brief          : Main program body
9.
10. * @attention
11. *
12. *
13.
14. *
15. *
16. *
17. *
18. *
19. *
20. *
21. *
22. *
23. *
24. *
25. *
26. *
27. *
28. *
29. *
30. *
31. *
32. *
33. *
34. *
35. *
36. *
37. *
38. *
39. *
40. *
41. *
42. *
43. *
44. *
45. *
46. *
47. *
48. *
49. *
50. *
51. *
52. *
53. *
54. *
55. *
56. *
57. *
58. *
59. *
60. *
61. *
62. *
63. *
64. *
65. *
66. *
67. *
68. *
69. *
70. *
71. *
72. *
73. *
74. *
75. *
76. *
77. *
78. *
79. *
80. *
81. *
82. *
83. *
84. *
85. *
86. *
87. *
88. *
89. *
90. *
91. *
92. *
93. *
94. *
95. *
96. *
97. *
98. *
99. *
100. *
101. *
102. *
103. *
104. *
105. *
106. *
107. *
108. *
109. *
110. *
111. *
112. *
113. *
114. *
115. *
116. *
117. *
118. *
119. *
120. *
121. *
122. *
123. *
124. *
125. *
126. *
127. *
128. *
129. *
130. *
131. *
132. *
133. *
134. *
135. *
136. *
137. *
138. *
139. *
140. *
141. *
142. *
143. *
144. *
145. *
146. *
147. *
148. *
149. *
150. *
151. *
152. *
153. *
154. *
155. *
156. *
157. *
158. *
159. *
160. *
161. *
162. *
163. *
164. *
165. *
166. *
167. *
168. *
169. *
170. *
171. *
172. *
173. *
174. *
175. *
176. *
177. *
178. *
179. *
180. *
181. *
182. *
183. *
184. *
185. *
186. *
187. *
188. *
189. *
190. *
191. *
192. *
193. *
194. *
195. *
196. *
197. *
198. *
199. *
200. *
201. *
202. *
203. *
204. *
205. *
206. *
207. *
208. *
209. *
210. *
211. *
212. *
213. *
214. *
215. *
216. *
217. *
218. *
219. *
220. *
221. *
222. *
223. *
224. *
225. *
226. *
227. *
228. *
229. *
230. *
231. *
232. *
233. *
234. *
235. *
236. *
237. *
238. *
239. *
240. *
241. *
242. *
243. *
244. *
245. *
246. *
247. *
248. *
249. *
250. *
251. *
252. *
253. *
254. *
255. *
256. *
257. *
258. *
259. *
260. *
261. *
262. *
263. *
264. *
265. *
266. *
267. *
268. *
269. *
270. *
271. *
272. *
273. *
274. *
275. *
276. *
277. *
278. *
279. *
280. *
281. *
282. *
283. *
284. *
285. *
286. *
287. *
288. *
289. *
290. *
291. *
292. *
293. *
294. *
295. *
296. *
297. *
298. *
299. *
300. *
301. *
302. *
303. *
304. *
305. *
306. *
307. *
308. *
309. *
310. *
311. *
312. *
313. *
314. *
315. *
316. *
317. *
318. *
319. *
320. *
321. *
322. *
323. *
324. *
325. *
326. *
327. *
328. *
329. *
330. *
331. *
332. *
333. *
334. *
335. *
336. *
337. *
338. *
339. *
340. *
341. *
342. *
343. *
344. *
345. *
346. *
347. *
348. *
349. *
350. *
351. *
352. *
353. *
354. *
355. *
356. *
357. *
358. *
359. *
360. *
361. *
362. *
363. *
364. *
365. *
366. *
367. *
368. *
369. *
370. *
371. *
372. *
373. *
374. *
375. *
376. *
377. *
378. *
379. *
380. *
381. *
382. *
383. *
384. *
385. *
386. *
387. *
388. *
389. *
390. *
391. *
392. *
393. *
394. *
395. *
396. *
397. *
398. *
399. *
400. *
401. *
402. *
403. *
404. *
405. *
406. *
407. *
408. *
409. *
410. *
411. *
412. *
413. *
414. *
415. *
416. *
417. *
418. *
419. *
420. *
421. *
422. *
423. *
424. *
425. *
426. *
427. *
428. *
429. *
430. *
431. *
432. *
433. *
434. *
435. *
436. *
437. *
438. *
439. *
440. *
441. *
442. *
443. *
444. *
445. *
446. *
447. *
448. *
449. *
450. *
451. *
452. *
453. *
454. *
455. *
456. *
457. *
458. *
459. *
460. *
461. *
462. *
463. *
464. *
465. *
466. *
467. *
468. *
469. *
470. *
471. *
472. *
473. *
474. *
475. *
476. *
477. *
478. *
479. *
480. *
481. *
482. *
483. *
484. *
485. *
486. *
487. *
488. *
489. *
490. *
491. *
492. *
493. *
494. *
495. *
496. *
497. *
498. *
499. *
500. *
501. *
502. *
503. *
504. *
505. *
506. *
507. *
508. *
509. *
510. *
511. *
512. *
513. *
514. *
515. *
516. *
517. *
518. *
519. *
520. *
521. *
522. *
523. *
524. *
525. *
526. *
527. *
528. *
529. *
530. *
531. *
532. *
533. *
534. *
535. *
536. *
537. *
538. *
539. *
540. *
541. *
542. *
543. *
544. *
545. *
546. *
547. *
548. *
549. *
550. *
551. *
552. *
553. *
554. *
555. *
556. *
557. *
558. *
559. *
560. *
561. *
562. *
563. *
564. *
565. *
566. *
567. *
568. *
569. *
570. *
571. *
572. *
573. *
574. *
575. *
576. *
577. *
578. *
579. *
580. *
581. *
582. *
583. *
584. *
585. *
586. *
587. *
588. *
589. *
590. *
591. *
592. *
593. *
594. *
595. *
596. *
597. *
598. *
599. *
600. *
601. *
602. *
603. *
604. *
605. *
606. *
607. *
608. *
609. *
610. *
611. *
612. *
613. *
614. *
615. *
616. *
617. *
618. *
619. *
620. *
621. *
622. *
623. *
624. *
625. *
626. *
627. *
628. *
629. *
630. *
631. *
632. *
633. *
634. *
635. *
636. *
637. *
638. *
639. *
640. *
641. *
642. *
643. *
644. *
645. *
646. *
647. *
648. *
649. *
650. *
651. *
652. *
653. *
654. *
655. *
656. *
657. *
658. *
659. *
660. *
661. *
662. *
663. *
664. *
665. *
666. *
667. *
668. *
669. *
670. *
671. *
672. *
673. *
674. *
675. *
676. *
677. *
678. *
679. *
680. *
681. *
682. *
683. *
684. *
685. *
686. *
687. *
688. *
689. *
690. *
691. *
692. *
693. *
694. *
695. *
696. *
697. *
698. *
699. *
700. *
701. *
702. *
703. *
704. *
705. *
706. *
707. *
708. *
709. *
710. *
711. *
712. *
713. *
714. *
715. *
716. *
717. *
718. *
719. *
720. *
721. *
722. *
723. *
724. *
725. *
726. *
727. *
728. *
729. *
730. *
731. *
732. *
733. *
734. *
735. *
736. *
737. *
738. *
739. *
740. *
741. *
742. *
743. *
744. *
745. *
746. *
747. *
748. *
749. *
750. *
751. *
752. *
753. *
754. *
755. *
756. *
757. *
758. *
759. *
760. *
761. *
762. *
763. *
764. *
765. *
766. *
767. *
768. *
769. *
770. *
771. *
772. *
773. *
774. *
775. *
776. *
777. *
778. *
779. *
780. *
781. *
782. *
783. *
784. *
785. *
786. *
787. *
788. *
789. *
790. *
791. *
792. *
793. *
794. *
795. *
796. *
797. *
798. *
799. *
800. *
801. *
802. *
803. *
804. *
805. *
806. *
807. *
808. *
809. *
810. *
811. *
812. *
813. *
814. *
815. *
816. *
817. *
818. *
819. *
820. *
821. *
822. *
823. *
824. *
825. *
826. *
827. *
828. *
829. *
830. *
831. *
832. *
833. *
834. *
835. *
836. *
837. *
838. *
839. *
840. *
841. *
842. *
843. *
844. *
845. *
846. *
847. *
848. *
849. *
850. *
851. *
852. *
853. *
854. *
855. *
856. *
857. *
858. *
859. *
860. *
861. *
862. *
863. *
864. *
865. *
866. *
867. *
868. *
869. *
870. *
871. *
872. *
873. *
874. *
875. *
876. *
877. *
878. *
879. *
880. *
881. *
882. *
883. *
884. *
885. *
886. *
887. *
888. *
889. *
890. *
891. *
892. *
893. *
894. *
895. *
896. *
897. *
898. *
899. *
900. *
901. *
902. *
903. *
904. *
905. *
906. *
907. *
908. *
909. *
910. *
911. *
912. *
913. *
914. *
915. *
916. *
917. *
918. *
919. *
920. *
921. *
922. *
923. *
924. *
925. *
926. *
927. *
928. *
929. *
930. *
931. *
932. *
933. *
934. *
935. *
936. *
937. *
938. *
939. *
940. *
941. *
942. *
943. *
944. *
945. *
946. *
947. *
948. *
949. *
950. *
951. *
952. *
953. *
954. *
955. *
956. *
957. *
958. *
959. *
960. *
961. *
962. *
963. *
964. *
965. *
966. *
967. *
968. *
969. *
970. *
971. *
972. *
973. *
974. *
975. *
976. *
977. *
978. *
979. *
980. *
981. *
982. *
983. *
984. *
985. *
986. *
987. *
988. *
989. *
990. *
991. *
992. *
993. *
994. *
995. *
996. *
997. *
998. *
999. *
1000. *

```

```

14.  */
15.
16.  //////////////////////////////////////
17.  /* Includes -----
    ----*/
18.  #include "main.h"
19.  #include "can.h"
20.  #include "gpio.h"
21.  #include "task.h"
22.  #include "stm32f7xx_hal.h"
23.  #include "stm32f7xx_hal_flash.h"
24.  //////////////////////////////////////
25.
26.  //////////////////////////////////////
27.  /* Private function prototypes -----*/
28.  void SystemClock_Config(void);
29.  void MX_FREERTOS_Init(void);
30.  //////////////////////////////////////
31.
32.  //////////////////////////////////////
33.  int main(void)
34.  {
35.      /* MCU Configuration-----*/
36.
37.      /* Reset of all peripherals, Initializes the Flash interface and the
    SysTick. */
38.      if(HAL_Init() != HAL_OK)
39.      {
40.          Error_Handler();
41.      }
42.
43.      /* Configure the system clock */
44.      SystemClock_Config();
45.
46.      /* Initialize all configured peripherals */
47.      MX_GPIO_Init();
48.
49.      /* Call init function for freertos objects (in freertos.c) */
50.      MX_FREERTOS_Init();
51.
52.      /* Start scheduler */
53.      vTaskStartScheduler();
54.
55.      /* We should never get here as control is now taken by the scheduler */
56.      /* Infinite loop */
57.      while (1){ }
58.  }
59.  //////////////////////////////////////
60.
61.  //////////////////////////////////////
62.  void SystemClock_Config(void)
63.  {
64.      RCC_OscInitTypeDef RCC_OscInitStruct = {0};
65.      RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
66.      RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};
67.

```

```

68.  /** Configure LSE Drive Capability
69.  */
70.  HAL_PWR_EnableBkUpAccess();
71.  /** Configure the main internal regulator output voltage
72.  */
73.  __HAL_RCC_PWR_CLK_ENABLE();
74.  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
75.  /** Initializes the CPU, AHB and APB busses clocks
76.  */
77.  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
78.  RCC_OscInitStruct.HSEState = RCC_HSE_BYPASS;
79.  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
80.  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
81.  RCC_OscInitStruct.PLL.PLLM = 4;
82.  RCC_OscInitStruct.PLL.PLLN = 216;
83.  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
84.  RCC_OscInitStruct.PLL.PLLQ = 9;
85.  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
86.  {
87.      Error_Handler();
88.  }
89.  /** Activate the Over-Drive mode
90.  */
91.  if (HAL_PWREx_EnableOverDrive() != HAL_OK)
92.  {
93.      Error_Handler();
94.  }
95.  /** Initializes the CPU, AHB and APB busses clocks
96.  */
97.  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
98.                               |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
99.  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
100.   RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
101.   RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
102.   RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
103.
104.   if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_7) !=
105.       HAL_OK)
106.   {
107.       Error_Handler();
108.   }
109.   PeriphClkInitStruct.PeriphClockSelection =
110.       RCC_PERIPHCLK_USART3|RCC_PERIPHCLK_CLK48;
111.   PeriphClkInitStruct.Usart3ClockSelection = RCC_USART3CLKSOURCE_PCLK1;
112.   PeriphClkInitStruct.Clk48ClockSelection = RCC_CLK48SOURCE_PLL;
113.   if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct) != HAL_OK)
114.   {
115.       Error_Handler();
116.   }
117.
118.
119. void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
120. {
121.     if (htim->Instance == TIM1) {

```



```

221.
222.         status = HAL_ERROR;
223.         while(status != HAL_OK)
224.         {
225.             status = PC_Send_String((uint8_t *)
recMsg.message, strlen(recMsg.message));
226.         }
227.     }
228. }
229. ///////////////////////////////////////////////////////////////////
230.
231. ///////////////////////////////////////////////////////////////////
232. void FreeRTOS_debug_printf(uint8_t* string, uint16_t size)
233. {
234.     const char *pcTaskName;
235.     static int index = 0;
236.
237.     Message debugMsg;
238.
239.     const char pcNoTask[] = "None";
240.     int xLength = 0;
241.     static long xMessageNumber = 0;
242.
243.     /* Additional info to place at the start of the log. */
244.     if( xTaskGetSchedulerState() != taskSCHEDULER_NOT_STARTED )
245.     {
246.         pcTaskName = pcTaskGetName( NULL );
247.     }
248.     else
249.     {
250.         pcTaskName = pcNoTask;
251.     }
252.
253.     xLength = snprintf( (char *) debugMsg.message,
254. dlMAX_PRINT_STRING_LENGTH,
255.                         "Debug %lu %lu [%s] ",
256.                         xMessageNumber++,
257.                         ( unsigned long )
xTaskGetTickCount(),
258.                         pcTaskName );
259.
260.     snprintf( (char *) debugMsg.message + xLength,
261. dlMAX_PRINT_STRING_LENGTH - xLength,
262.              "%s",
263.              string);
264.
265.     xQueueSend( queueDebug ,
266.                (void*) &debugMsg,
267.                (TickType_t) 0 );
268.
269.     if(index == 5)
270.         index = 0;
271.     index++;
272.
273. }

```

```

274. ///////////////////////////////////////////////////////////////////
275.
276. ///////////////////////////////////////////////////////////////////
277. void FreeRTOS_printf(uint8_t* string, uint16_t size)
278. {
279.     const char *pcTaskName;
280.     static int index = 0;
281.
282.     Message Msg;
283.
284.     const char pcNoTask[] = "None";
285.     int xLength = 0;
286.     static long xMessageNumber = 0;
287.
288.     /* Additional info to place at the start of the log. */
289.     if( xTaskGetSchedulerState() != taskSCHEDULER_NOT_STARTED )
290.     {
291.         pcTaskName = pcTaskGetName( NULL );
292.     }
293.     else
294.     {
295.         pcTaskName = pcNoTask;
296.     }
297.
298.     xLength = snprintf( (char *) Msg.message,
299.         dlMAX_PRINT_STRING_LENGTH,
300.                                     "%lu %lu [%s] ",
301.                                     xMessageNumber++,
302.                                     ( unsigned long )
xTaskGetTickCount(),
303.                                     pcTaskName );
304.
305.     snprintf( (char *) Msg.message + xLength,
306.                 dlMAX_PRINT_STRING_LENGTH - xLength,
307.                 "%s",
308.                 string);
309.
310.
311.     xQueueSend( queueDebug ,
312.                 (void *) &Msg,
313.                 (TickType_t) 0 );
314.
315.     if(index == 5)
316.         index = 0;
317.
318.     index++;
319. }
320. ///////////////////////////////////////////////////////////////////
321.
322.
323.
324. /**
325.
*****
****

```

```

326.  * @file          : Discovery.c
327.  * @brief         : Discovery Task
328.
    *****
    ****
329.  * @attention
330.  *
331.  *
332.
    *****
    ****
333.  */
334.
335.  //////////////////////////////////////
336.  #include "main.h"
337.  #include "stm32f7xx_hal.h"
338.  #include "stm32f7xx_nucleo_144.h"
339.  #include "can.h"
340.  #include "FreeRTOS.h"
341.  #include "queue.h"
342.  #include "event_groups.h"
343.  #include "network_application.h"
344.  //////////////////////////////////////
345.
346.  //////////////////////////////////////
347.  // Discovery Protocol Functions
348.  void Discovery_Reset_Nodes(void);
349.  bool Discovery_First_Node_First_Output(uint8_t Node_ID);
350.  bool Discovery_First_Node_Second_Output(uint8_t Node_ID);
351.  bool Discovery_First_Output(uint8_t Node_ID);
352.  bool Discovery_Second_Output(uint8_t Node_ID);
353.  void Discovery_Start(void);
354.  void Discovery_Ready(IdentifierCodes_Type code, uint8_t Node_ID);
355.  bool Discovery_Call(uint8_t Node_ID);
356.  //////////////////////////////////////
357.
358.  //////////////////////////////////////
359.  extern QueueHandle_t queueDiscovery;
360.  //////////////////////////////////////
361.
362.  //////////////////////////////////////
363.  EventGroupHandle_t g_Discovery_Event_Group;
364.  //////////////////////////////////////
365.
366.
367.  //////////////////////////////////////
368.  void CAN_Discovery_Task( void *pvParameters )
369.  {
370.
371.      bool firstFound = true;
372.      bool endRound = false;
373.      uint8_t Node_ID = 0;
374.      bool nodes[255] = {false};
375.      EventBits_t uxBits;
376.      uint8_t message[64] = {0};
377.      int numBytes = 0;

```

```

378.
379.     /* Attempt to create the event group. */
380.     g_Discovery_Event_Group = xEventGroupCreate();
381.
382.     /* Was the event group created successfully? */
383.     if( g_Discovery_Event_Group == NULL )
384.     {
385.         /* The event group was not created because there was
insufficient
386.         FreeRTOS heap available. */
387.         Error_Handler();
388.     }
389.
390.     /* Block for 500ms. */
391.     const TickType_t xDelay = 500 / portTICK_PERIOD_MS;
392.
393.     /* Initialize all configured peripherals */
394.     BSP_PB_Init(BUTTON_USER, BUTTON_MODE_GPIO);
395.     BSP_LED_Init(LED1);
396.     BSP_LED_Init(LED2);
397.
398.     for(;;)
399.     {
400.         /* Wait a maximum of 50 days for either bit 0 or bit 1
to be set within
401.         the event group. Clear the bits before exiting.
402.         uxBits = xEventGroupWaitBits(
403.         g_Discovery_Event_Group, // The event group being tested.
404.         BIT_0, // The bits within the event group to wait for.
405.         pdTRUE, // BIT_0 should be cleared before returning.
406.         pdFALSE, // Any bit will trigger
407.         portMAX_DELAY );// Wait a maximum of 50 days
408.
409.         if( uxBits == 0 )
410.             continue;
411.
412.         BSP_LED_On(LED2);
413.
414.         message[0] = DiscoveryAck;
415.         Notify_Ethernet_Listeners(
416.         message,
417.                                     1
418.                                     );
419.
420.
421.         //////////////////////////////////////
422.         // Start nodes discovery mode
423.         Discovery_Start();

```

```

424. ///////////////////////////////////////////////////////////////////
425.
426. ///////////////////////////////////////////////////////////////////
427.         vTaskDelay( xDelay );
428.
429. ///////////////////////////////////////////////////////////////////
430.
431.         firstFound =
Discovery_Fist_Node_First_Output(Node_ID);
432.
433. ///////////////////////////////////////////////////////////////////
434.
435.         if(firstFound == true)
436.         {
437.             nodes[0] = true;
438.         }
439.         else
440.         {
441.             // First node was not found so increment node
442.             Node_ID++;
443.         }
444.
445. ///////////////////////////////////////////////////////////////////
446.
447. ///////////////////////////////////////////////////////////////////
448.         if(firstFound == false)
449.         {
450.             endRound =
!Discovery_Fist_Node_Second_Output(Node_ID);
451.             // Check to see if second node was found
452.             if(endRound == false)
453.                 nodes[1] = true;
454.         }
455.         else
456.             endRound = false;
457.
458. ///////////////////////////////////////////////////////////////////
459. ///////////////////////////////////////////////////////////////////
460.         while(endRound != true)
461.         {
462.             Node_ID++;
463.
464.             firstFound = Discovery_First_Output(Node_ID);
465.
466.             if(firstFound == true)
467.             {

```

```

468.         nodes[Node_ID] = true;
469.         endRound = false;
470.     }
471.     else
472.     {
473.         // Node was not found so increment node
474.         Node_ID++;
475.         endRound =
!Discovery_Second_Output(Node_ID);
476.         // Check to see if second node was
found
477.         if(endRound == false)
478.             nodes[Node_ID] = true;
479.     }
480. }
481.
482. ///////////////////////////////////////////////////////////////////
483. ///////////////////////////////////////////////////////////////////
484.     Node_ID--;
485. ///////////////////////////////////////////////////////////////////
486. ///////////////////////////////////////////////////////////////////
487. ///////////////////////////////////////////////////////////////////
488.     // Tell nodes that discovery mode is over
489.     Discovery_Reset_Nodes();
490. ///////////////////////////////////////////////////////////////////
491. ///////////////////////////////////////////////////////////////////
492. ///////////////////////////////////////////////////////////////////
493.     message[0] = DiscoveryNodes;
494.     message[1] = Node_ID;
495.     Notify_Ethernet_Listeners(
496. message,
497.                                     2
498.                                     );
499. ///////////////////////////////////////////////////////////////////
500.
501.     vTaskDelay( xDelay );
502.
503. ///////////////////////////////////////////////////////////////////
504.     message[0] = DiscoveryInactive;
505.
506.     numBytes = (Node_ID / 8) + 1;
507.     for(int i = 0; i < numBytes; i++)
508.     {
509.         message[i + 1] = 0;
510.         for(int j = 0; j < 8; j++)
511.         {

```



```

512.                                     message[i + 1] |= (~(nodes[i*8+j] &
    true)) << j;
513.                                     }
514.                                     }
515.                                     Notify_Ethernet_Listeners(
516. message,
517. numBytes + 1
518.                                     );
519.
520. //////////////////////////////////////
521.                                     vTaskDelay( xDelay );
522.
523. //////////////////////////////////////
524.                                     message[0] = DiscoveryFinished;
525.
526.                                     Notify_Ethernet_Listeners(
527. message,
528.                                     1
529.                                     );
530.
531. //////////////////////////////////////
532.                                     BSP_LED_Off(LED2);
533.                                     }
534. //////////////////////////////////////
535. //////////////////////////////////////
536. //////////////////////////////////////
537. void Discovery_Start(void)
538. {
539.     CAN_Send( Discovery_Start_Identifier,
540.               0, // Data length
541.               NULL );
542. }
543. //////////////////////////////////////
544. //////////////////////////////////////
545. //////////////////////////////////////
546. bool Discovery_Fist_Node_First_Output(uint8_t Node_ID)
547. {
548.     bool returnStatus;
549.
550.     // Turn on the first output
551.     HAL_GPIO_WritePin( DISC_PORT, DISC_OUTPUT_1, GPIO_PIN_SET );
552.
553.     returnStatus = Discovery_Call(Node_ID);
554.
555.     HAL_GPIO_WritePin( DISC_PORT, DISC_OUTPUT_1, GPIO_PIN_RESET);
556.
557.     return returnStatus;
558. }
559. //////////////////////////////////////
560.

```

```

561. ///////////////////////////////////////////////////
562. bool Discovery_First_Node_Second_Output(uint8_t Node_ID)
563. {
564.     bool returnStatus;
565.
566.     // Turn on the second output
567.     HAL_GPIO_WritePin( DISC_PORT, DISC_OUTPUT_2, GPIO_PIN_SET );
568.
569.     returnStatus = Discovery_Call(Node_ID);
570.
571.     HAL_GPIO_WritePin( DISC_PORT, DISC_OUTPUT_2, GPIO_PIN_RESET);
572.
573.     return returnStatus;
574. }
575. ///////////////////////////////////////////////////
576.
577. ///////////////////////////////////////////////////
578. bool Discovery_First_Output(uint8_t Node_ID)
579. {
580.     bool returnStatus;
581.
582.     Discovery_Ready(Discovery_Ready_1_Identifier, Node_ID);
583.
584.     returnStatus = Discovery_Call(Node_ID);
585.
586.     return returnStatus;
587. }
588. ///////////////////////////////////////////////////
589.
590. ///////////////////////////////////////////////////
591. bool Discovery_Second_Output(uint8_t Node_ID)
592. {
593.     bool returnStatus;
594.
595.     Discovery_Ready(Discovery_Ready_2_Identifier, Node_ID);
596.
597.     returnStatus = Discovery_Call(Node_ID);
598.
599.     return returnStatus;
600. }
601. ///////////////////////////////////////////////////
602.
603. ///////////////////////////////////////////////////
604. void Discovery_Reset_Nodes(void)
605. {
606.     CAN_Send( Discovery_End_Identifier,
607.              0, // Data length
608.              NULL );
609. }
610. ///////////////////////////////////////////////////
611.
612. ///////////////////////////////////////////////////
613. bool Discovery_Call(uint8_t Node_ID)
614. {
615.     bool successful = true;
616.     uint8_t dataNode[1];

```

```

617.         BaseType_t returnStatus = pdPASS;
618.         CAN_Message_Struct messageStruct;
619.
620.         xQueueReset( queueDiscovery );
621.
622.         dataNode[0] = Node_ID;
623.         CAN_Send( Discovery_Call_Identifier,
624.                  1, // Data length
625.                  dataNode );
626.
627.         while( messageStruct.messageHeader.StdId !=
628.               Disocvery_Call_Response_Identifier &&
629.               returnStatus != errQUEUE_EMPTY )
630.             {
631.                 returnStatus = xQueueReceive(
632.                     queueDiscovery,
633.                     &messageStruct,
634.                     250
635.                 );
636.
637.                 if(returnStatus == errQUEUE_EMPTY)
638.                 {
639.                     dataNode[0] = Node_ID;
640.                     CAN_Send( Discovery_Call_Identifier,
641.                              1, // Data length
642.                              dataNode );
643.
644.                     while( messageStruct.messageHeader.StdId !=
645.                           Disocvery_Call_Response_Identifier && returnStatus != errQUEUE_EMPTY )
646.                         {
647.                             returnStatus = xQueueReceive(
648.                                 queueDiscovery,
649.                                 &messageStruct,
650.                                 250
651.                             );
652.
653.                             if(returnStatus == errQUEUE_EMPTY)
654.                             {
655.                                 successful = false;
656.                             }
657.                         }
658.
659.                     return successful;
660.                 }
661.                 //////////////////////////////////////
662.

```

```

663. ///////////////////////////////////////////////////////////////////
664. void Discovery_Ready(IdentifierCodes_Type code, uint8_t Node_ID)
665. {
666.     BaseType_t returnStatus = pdPASS;
667.     //uint8_t dataNode[1];
668.     CAN_Message_Struct messageStruct;
669.     //bool firstFound = true;
670.
671.     xQueueReset( queueDiscovery );
672.
673.     CAN_Send( code,
674.              0, // Data length
675.              NULL );
676.
677.     while( messageStruct.messageHeader.StdId !=
Discovery_Ready_Response_Identifier &&
678.           returnStatus != errQUEUE_EMPTY )
679.     {
680.         returnStatus = xQueueReceive(
681.             queueDiscovery,
682.             &messageStruct,
683.             250
684.         );
685.     }
686.
687.     if(returnStatus == errQUEUE_EMPTY)
688.     {
689.         CAN_Send( code,
690.                  0, // Data length
691.                  NULL );
692.
693.         while( messageStruct.messageHeader.StdId !=
Discovery_Ready_Response_Identifier &&
694.               returnStatus != errQUEUE_EMPTY )
695.         {
696.             returnStatus = xQueueReceive(
697.                 queueDiscovery,
698.                 &messageStruct,
699.                 250
700.             );
701.         }
702.
703.         if(returnStatus == errQUEUE_EMPTY)
704.         {
705.             //TODO: Error Node isn't responding
706.         }
707.         else if(messageStruct.messageData[0] + 1 != Node_ID)
708.         {

```

```

709.                                     // TODO: Error wrong node responded
710.                                     }
711.     }
712.     else if(messageStruct.messageData[0] + 1 != Node_ID)
713.     {
714.         // TODO: Error wrong node responded
715.     }
716. }
717. ///////////////////////////////////////////////////////////////////
718.
719.
720.
721. /**
722.
723. *****
724. * @file           : error.c
725. * @brief          : error handler
726. *****
727. * @attention
728. *
729. *
730. */
731.
732. ///////////////////////////////////////////////////////////////////
733. #include "FreeRTOS.h"
734. #include "task.h"
735. ///////////////////////////////////////////////////////////////////
736.
737. ///////////////////////////////////////////////////////////////////
738. void vAssertCalled( const char *pcFile, uint32_t ulLine )
739. {
740.     /* Inside this function, pcFile holds the name of the source file that
741.     contains
742.     the line that detected the error, and ulLine holds the line number in
743.     the source
744.     file. The pcFile and ulLine values can be printed out, or otherwise
745.     recorded,
746.     before the following infinite loop is entered. */
747.     char string[255] = "";
748.     PC_Setup_Serial();
749.     snprintf(string,255, "Assert Failed - file: %s line:
750. %lu\n\n\n", pcFile, ulLine);
751.     HAL_StatusTypeDef status = HAL_ERROR;
752.     while(status != HAL_OK)
753.     {

```

```

754.             status = PC_Send_String((uint8_t *) string,
      strlen(string));
755.         }
756.     /* Disable interrupts so the tick interrupt stops executing, then sit
      in a loop
757.     so execution does not move past the line that failed the assertion. */
758.     taskDISABLE_INTERRUPTS();
759.     for( ;; );
760. }
761. ///////////////////////////////////////////////////////////////////
762. ///////////////////////////////////////////////////////////////////
763. ///////////////////////////////////////////////////////////////////
764. void vApplicationMallocFailedHook(void)
765. {
766.     char string[100] = "";
767.
768.     PC_Setup_Serial();
769.
770.
771.     size_t sizeLeft = xPortGetFreeHeapSize();
772.
773.     sprintf(string, "Malloc Failed Hook - Heap Left: %u \n\n\n",
      sizeLeft);
774.
775.     HAL_StatusTypeDef status = HAL_ERROR;
776.
777.     while(status != HAL_OK)
778.     {
779.         status = PC_Send_String((uint8_t *) string,
      strlen(string));
780.     }
781.
782.     while(1);
783. }
784. ///////////////////////////////////////////////////////////////////
785. ///////////////////////////////////////////////////////////////////
786. ///////////////////////////////////////////////////////////////////
787. void vApplicationStackOverflowHook(xTaskHandle xTask, signed char
      *pcTaskName)
788. {
789.     char string[100] = "";
790.
791.     PC_Setup_Serial();
792.
793.     sprintf(string, "Stack Overflow: %s\n\n\n", pcTaskName);
794.
795.     HAL_StatusTypeDef status = HAL_ERROR;
796.
797.     while(status != HAL_OK)
798.     {
799.         status = PC_Send_String((uint8_t *) string,
      strlen(string));
800.     }
801.
802.     while(1);
803. }

```

```

804. ///////////////////////////////////////////////////
805.
806.
807.
808.
809.
810. /**
811.
      *****
      ****
812.  * File Name           : freertos.c
813.  * Description        : Code for freertos applications
814.
      *****
      ****
815.  * @attention
816.  *
817.  *
818.
      *****
      ****
819.  */
820.
821. ///////////////////////////////////////////////////
822. // Includes
823. ///////////////////////////////////////////////////
824. #include "FreeRTOSConfig.h"
825. #include "FreeRTOSIPConfig.h"
826. #include "FreeRTOS.h"
827. #include "main.h"
828. #include "stm32f7xx_nucleo_144.h"
829. #include "task.h"
830. #include "queue.h"
831. #include "event_groups.h"
832.
833. #include "discovery.h"
834. #include "can.h"
835. #include "network_application.h"
836.
837. #include "debug.h"
838. #include "serialtopc.h"
839. ///////////////////////////////////////////////////
840.
841. ///////////////////////////////////////////////////
842. #define NUM_IDENTIFIERS 20
843. ///////////////////////////////////////////////////
844.
845. ///////////////////////////////////////////////////
846. // Extern variables
847. extern CAN_HandleTypeDef hcan1; // src/can.c
848. ///////////////////////////////////////////////////
849.
850. ///////////////////////////////////////////////////
851. // Global Variables
852. TaskHandle_t CAN_LowLevel_Task_Handle,
853.              CAN_Discovery_Task_Handle,

```

```

854.         CAN_FuseOK_Task_Handle,
855.         CAN_Error_Task_Handle,
856.         CAN_Heartbeat_Task_Handle,
857.         CAN_FuseBlown_Task_Handle,
858.         Ethernet_Task_Handle;
859.
860. QueueHandle_t queueDiscovery,
861.             queueFuseBlown,
862.             queueFuseOK,
863.             queueError,
864.             queueHeartbeat;
865.
866. EventGroupHandle_t g_CAN_Message_Event_Group;
867. static char buffer[100];
868. uint8_t g_NumFuses = 5;
869. ////////////////////////////////////////////////////////////////////
870.
871. ////////////////////////////////////////////////////////////////////
872. // Private Function Declarations
873. ////////////////////////////////////////////////////////////////////
874.
875. ////////////////////////////////////////////////////////////////////
876. // Task Function Declarations
877. // CAN Functions
878. static void CAN_LowLevel_Task( void *pvParameters );
879. static void CAN_FuseOK_Task (void *pvParameters );
880. static void CAN_Error_Task(void *pvParameters );
881. static void CAN_Heartbeat_Task( void *pvParameters );
882. static void CAN_FuseBlown_Task( void *pvParameters );
883. ////////////////////////////////////////////////////////////////////
884.
885. ////////////////////////////////////////////////////////////////////
886. // Helper Functions
887. static QueueHandle_t* Switch_Queue(CAN_RxHeaderTypeDef header);
888. ////////////////////////////////////////////////////////////////////
889.
890. ////////////////////////////////////////////////////////////////////
891. // Public Function Declarations
892. ////////////////////////////////////////////////////////////////////
893. void MX_FREERTOS_Init(void);
894. ////////////////////////////////////////////////////////////////////
895.
896. ////////////////////////////////////////////////////////////////////
897. void MX_FREERTOS_Init(void) {
898.     /* RTOS_QUEUES */
899.     queueDiscovery = xQueueCreate( 64, sizeof( CAN_Message_Struct
900. ) );
901.     if(queueDiscovery == NULL)
902.     {
903.         Error_Handler();
904.     }
905.     queueFuseBlown = xQueueCreate( 64, sizeof( CAN_Message_Struct
906. ) );
907.     if(queueFuseBlown == NULL)
908.     {
909.         Error_Handler();

```



```

908.         }
909.         queueFuseOK = xQueueCreate( 64, sizeof( CAN_Message_Struct )
);
910.         if(queueFuseOK == NULL)
911.         {
912.             Error_Handler();
913.         }
914.         queueError = xQueueCreate( 64, sizeof( CAN_Message_Struct ) );
915.         if(queueError == NULL)
916.         {
917.             Error_Handler();
918.         }
919.         queueHeartbeat = xQueueCreate( 64, sizeof( CAN_Message_Struct
) );
920.         if(queueHeartbeat == NULL)
921.         {
922.             Error_Handler();
923.         }
924.
925.         /* RTOS_THREADS */
926.         BaseType_t xReturn;
927.         #if ipconfigHAS_DEBUG_PRINTF | ipconfigHAS_PRINTF
928.             Setup_Debug();
929.         #endif
930.         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 );
931.         if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
932.         {
933.             Error_Handler();
934.         }
935.         xReturn = xTaskCreate( Ethernet_Task, "Ethernet_Task", (
uint16_t )configMINIMAL_STACK_SIZE*3, NULL, ( UBaseType_t
)ETHERNET_TASK_PRIORITY, &Ethernet_Task_Handle );
936.         if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
937.         {
938.             Error_Handler();
939.         }
940.
941.         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 );
942.         if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
943.         {
944.             Error_Handler();
945.         }
946.         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 );
947.         if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
948.         {
949.             Error_Handler();
950.         }
951.         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 );

```

```

952.         if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
953.         {
954.             Error_Handler();
955.         }
956.         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 );
957.         if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
958.         {
959.             Error_Handler();
960.         }
961.         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 );
962.         if(xReturn == errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY)
963.         {
964.             Error_Handler();
965.         }
966.
967.
968.     }
969.     //////////////////////////////////////
970.
971.     //////////////////////////////////////
972.     /**
973.      * @brief  Handles reading from CAN
974.      * @param  argument: Not used
975.      * @retval None
976.      */
977.     static void CAN_LowLevel_Task( void *pvParameters )
978.     {
979.         /* Attempt to create the event group. */
980.         g_CAN_Message_Event_Group = xEventGroupCreate();
981.
982.         /* Was the event group created successfully? */
983.         if( g_CAN_Message_Event_Group == NULL )
984.         {
985.             /* The event group was not created because there was
insufficient
986.             FreeRTOS heap available. */
987.             Error_Handler();
988.         }
989.
990.         CAN_Setup();
991.
992.         EventBits_t uxBits;
993.
994.         for(;;)
995.         {
996.             CAN_Message_Struct mStruct;
997.             QueueHandle_t *pQueue;
998.
999.             /* Wait a maximum of 50 days for either bit 0 or bit 1
to be set within
1000.             the event group. Clear the bits before exiting. */
1001.             uxBits = xEventGroupWaitBits(

```

```

1002.     g_CAN_Message_Event_Group,    /* The event group being tested. */
1003.     BIT_FIFO_0 | BIT_FIFO_1, /* The bits within the event group to wait for.
    */
1004.     pdTRUE,          /* BIT_0 | BIT_1 should be cleared before returning. */
1005.     pdFALSE,        /* Any bit will trigger */
1006.     portMAX_DELAY ); /* Wait a maximum of 50 days */
1007.
1008.         if( uxBits == 0 )
1009.             continue;
1010.
1011.         uint32_t fillLevel = 0;
1012.         bool found = false;
1013.         do
1014.         {
1015.             uint32_t fillLevel =
1016.             HAL_CAN_GetRxFifoFillLevel(&hcan1, CAN_RX_FIFO0);
1017.             if(fillLevel != 0)
1018.             {
1019.                 CAN_Receive(CAN_RX_FIFO0,
1020.                 &mStruct.messageHeader, mStruct.messageData);
1021.
1022.                 pQueue =
1023.                 Switch_Queue(mStruct.messageHeader);
1024.
1025.                 xQueueSend( *pQueue,
1026.                 &mStruct,
1027.                 portMAX_DELAY );
1028.                 found = true;
1029.             }
1030.             if(found == true && fillLevel != 0)
1031.             {
1032.                 HAL_CAN_ActivateNotification(&hcan1,
1033.                 CAN_IT_RX_FIFO0_MSG_PENDING);
1034.             }
1035.         }while(fillLevel != 0);
1036.
1037.         found = false;
1038.         fillLevel = 0;
1039.         do
1040.         {
1041.             uint32_t fillLevel =
1042.             HAL_CAN_GetRxFifoFillLevel(&hcan1, CAN_RX_FIFO1);
1043.             if(fillLevel != 0)
1044.             {
1045.                 CAN_Receive(CAN_RX_FIFO1,
1046.                 &mStruct.messageHeader, mStruct.messageData);
1047.
1048.                 pQueue =
1049.                 Switch_Queue(mStruct.messageHeader);
1050.
1051.                 xQueueSend( *pQueue,
1052.                 &mStruct,
1053.                 portMAX_DELAY );
1054.                 found = true;
1055.             }
1056.             if(found == true && fillLevel != 0)
1057.             {
1058.                 HAL_CAN_ActivateNotification(&hcan1,
1059.                 CAN_IT_RX_FIFO1_MSG_PENDING);
1060.             }
1061.         }while(fillLevel != 0);
1062.
1063.         return pdTRUE;
1064.     }
1065. }

```

```

1045.         xQueueSend( *pQueue,
1046.                     &mStruct,
1047.                     portMAX_DELAY );
1048.         found = true;
1049.     }
1050.
1051.
1052.         if(found == true && fillLevel != 0)
1053.         {
1054.             HAL_CAN_ActivateNotification(&hcan1,
CAN_IT_RX_FIFO1_MSG_PENDING);
1055.         }
1056.
1057.         }while(fillLevel != 0);
1058.     }
1059. }
1060. ///////////////////////////////////////////////////////////////////
1061.
1062. ///////////////////////////////////////////////////////////////////
1063. /**
1064.  * @brief Switches the queue based on the StdId code
1065.  * @param CAN header used to determine StdId
1066.  * @retval None
1067.  */
1068. static QueueHandle_t* Switch_Queue(CAN_RxHeaderTypeDef header)
1069. {
1070.     uint32_t id = header.StdId;
1071.     QueueHandle_t pQueue = &queueError;
1072.
1073.     switch(id)
1074.     {
1075.         case Heartbeat_Identifier:
1076.             pQueue = &queueHeartbeat;
1077.             break;
1078.         case Fuse_OK_Identifier:
1079.             pQueue = &queueFuseOK;
1080.             break;
1081.         case Fuse_Blown_Identifier:
1082.             pQueue = &queueFuseBlown;
1083.             break;
1084.
1085.         case Error_1_Identifier:
1086.             pQueue = &queueError;
1087.             break;
1088.         case Error_2_Identifier:
1089.             pQueue = &queueError;
1090.             break;
1091.         case Discovery_Ready_Response_Identifier:
1092.             pQueue = &queueDiscovery;
1093.             break;
1094.
1095.         case Disocvery_Call_Response_Identifier:
1096.             pQueue = &queueDiscovery;
1097.             break;
1098.
1099.         default:

```



```

1150.
1151. //////////////////////////////////////////////////
1152. static void CAN_FuseOK_Task (void *pvParameters )
1153. {
1154.     BaseType_t returnStatus;
1155.     CAN_Message_Struct messageStruct;
1156.     uint8_t message[64] = {0};
1157.     message[0] = FuseOK;
1158.
1159.     for(;;)
1160.     {
1161.         returnStatus = xQueueReceive(
1162.             queueFuseOK,
1163.             &messageStruct,
1164.             portMAX_DELAY
1165.         );
1166.         if(returnStatus == pdFALSE)
1167.             continue;
1168.
1169.         // TODO: Log and send if turned on
1170.         message[1] = (1 << messageStruct.messageData[0]);
1171.         Notify_Ethernet_Listeners( message, (size_t) 2 );
1172.
1173.
1174.
1175.         sprintf(buffer, "Fuse OK: %x\n",
1176.             messageStruct.messageData[0] );
1177.         FreeRTOS_debug_printf((uint8_t *)buffer,
1178.             strlen(buffer));
1179.     }
1180. //////////////////////////////////////////////////
1181. //////////////////////////////////////////////////
1182. //////////////////////////////////////////////////
1183. static void CAN_Error_Task(void *pvParameters )
1184. {
1185.     BaseType_t returnStatus;
1186.     CAN_Message_Struct messageStruct;
1187.     uint8_t message[64] = {0};
1188.
1189.     for(;;)
1190.     {
1191.         returnStatus = xQueueReceive(
1192.             queueError,
1193.             &messageStruct,
1194.             portMAX_DELAY
1195.         );

```

```

1196.         if(returnStatus == pdFALSE)
1197.             continue;
1198.
1199.         // TODO: Log and send if turned on
1200.         Notify_Ethernet_Listeners( message, (size_t) 1 );
1201.     }
1202. }
1203. ///////////////////////////////////////////////////////////////////
1204.
1205. ///////////////////////////////////////////////////////////////////
1206. static void CAN_Heartbeat_Task( void *pvParameters )
1207. {
1208.     BaseType_t returnStatus;
1209.     CAN_Message_Struct messageStruct;
1210.     uint8_t message[64] = {0};
1211.
1212.     for(;;)
1213.     {
1214.         returnStatus = xQueueReceive(
1215.             queueHeartbeat,
1216.             &messageStruct,
1217.             portMAX_DELAY
1218.         );
1219.         if(returnStatus == pdFALSE)
1220.             continue;
1221.
1222.         // TODO: Log and send if turned on
1223.         Notify_Ethernet_Listeners( message, (size_t) 1 );
1224.     }
1225. }
1226. ///////////////////////////////////////////////////////////////////
1227.
1228.
1229.
1230.
1231. /**
1232.
1233.  * File Name           : gpio.c
1234.  * Description         : This file provides code for the configuration
1235.  *                     of all used GPIO pins.
1236.
1237.  * @attention
1238.  *
1239.  *
1240.
1241.  */

```

```

1242.
1243. ////////////////////////////////////////////////////
1244. /* Includes -----*
-----*/
1245. #include "gpio.h"
1246. ////////////////////////////////////////////////////
1247.
1248. ////////////////////////////////////////////////////
1249. void MX_GPIO_Init(void)
1250. {
1251.
1252.     GPIO_InitTypeDef GPIO_InitStruct = {0};
1253.
1254.     /* GPIO Ports Clock Enable */
1255.     __HAL_RCC_GPIOC_CLK_ENABLE();
1256.     __HAL_RCC_GPIOH_CLK_ENABLE();
1257.     __HAL_RCC_GPIOA_CLK_ENABLE();
1258.     __HAL_RCC_GPIOB_CLK_ENABLE();
1259.     __HAL_RCC_GPIOD_CLK_ENABLE();
1260.     __HAL_RCC_GPIOG_CLK_ENABLE();
1261.
1262.     /*Configure GPIO pin Output Level */
1263.     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4|GPIO_PIN_5, GPIO_PIN_RESET);
1264.
1265.     /*Configure GPIO pin Output Level */
1266.     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0|GPIO_PIN_14|GPIO_PIN_7,
GPIO_PIN_RESET);
1267.
1268.     /*Configure GPIO pin Output Level */
1269.     HAL_GPIO_WritePin(GPIOG, GPIO_PIN_6, GPIO_PIN_RESET);
1270.
1271.     /*Configure GPIO pin : PC13 */
1272.     GPIO_InitStruct.Pin = GPIO_PIN_13;
1273.     GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
1274.     GPIO_InitStruct.Pull = GPIO_NOPULL;
1275.     HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
1276.
1277.     /*Configure GPIO pins : PC1 PC4 PC5 */
1278.     GPIO_InitStruct.Pin = GPIO_PIN_1|GPIO_PIN_4|GPIO_PIN_5;
1279.     GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
1280.     GPIO_InitStruct.Pull = GPIO_NOPULL;
1281.     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
1282.     GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
1283.     HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
1284.
1285.     /*Configure GPIO pins : PA1 PA2 PA7 */
1286.     GPIO_InitStruct.Pin = GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_7;
1287.     GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
1288.     GPIO_InitStruct.Pull = GPIO_NOPULL;
1289.     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
1290.     GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
1291.     HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
1292.
1293.     /*Configure GPIO pins : PA4 PA5 */ /* Discovery Mode */
1294.     GPIO_InitStruct.Pin = DISC_OUTPUT_1|DISC_OUTPUT_2;
1295.     GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;

```



```

1296.  GPIO_InitStruct.Pull = GPIO_NOPULL;
1297.  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1298.  HAL_GPIO_Init(DISC_PORT, &GPIO_InitStruct);
1299.
1300.  /*Configure GPIO pins : PB0 PB14 PB7 */
1301.  GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_14|GPIO_PIN_7;
1302.  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
1303.  GPIO_InitStruct.Pull = GPIO_NOPULL;
1304.  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1305.  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
1306.
1307.  /*Configure GPIO pin : PB13 */
1308.  GPIO_InitStruct.Pin = GPIO_PIN_13;
1309.  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
1310.  GPIO_InitStruct.Pull = GPIO_NOPULL;
1311.  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
1312.  GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
1313.  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
1314.
1315.  /*Configure GPIO pin : PG6 */
1316.  GPIO_InitStruct.Pin = GPIO_PIN_6;
1317.  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
1318.  GPIO_InitStruct.Pull = GPIO_NOPULL;
1319.  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
1320.  HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);
1321.
1322.  /*Configure GPIO pin : PG7 */
1323.  GPIO_InitStruct.Pin = GPIO_PIN_7;
1324.  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
1325.  GPIO_InitStruct.Pull = GPIO_NOPULL;
1326.  HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);
1327.
1328.  /*Configure GPIO pins : PG11 PG13 */
1329.  GPIO_InitStruct.Pin = GPIO_PIN_11|GPIO_PIN_13;
1330.  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
1331.  GPIO_InitStruct.Pull = GPIO_NOPULL;
1332.  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
1333.  GPIO_InitStruct.Alternate = GPIO_AF11_ETH;
1334.  HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);
1335.
1336. }
1337. //////////////////////////////////////////////////
1338.
1339.
1340.
1341.
1342. /**
1343.
1344.  * File Name      : network_application.c
1345.  * Description    :
1346.
1347.  * @attention

```

```

1348.  *
1349.  *
1350.
    *****
    ****
1351.  */
1352.
1353. ///////////////////////////////////////////////////////////////////
1354. #include "network_application.h"
1355. #include "main.h"
1356. ///////////////////////////////////////////////////////////////////
1357.
1358. ///////////////////////////////////////////////////////////////////
1359. #define BUFFER_SIZE 64
1360. #define NUM_IDENTIFIERS 20
1361. ///////////////////////////////////////////////////////////////////
1362.
1363. ///////////////////////////////////////////////////////////////////
1364. extern EventGroupHandle_t g_Discovery_Event_Group;
1365. ///////////////////////////////////////////////////////////////////
1366.
1367. ///////////////////////////////////////////////////////////////////
1368. static void Listener_Task( void *pvParameters );
1369. static void Sender_Task( void *pvParameters );
1370. ///////////////////////////////////////////////////////////////////
1371.
1372. ///////////////////////////////////////////////////////////////////
1373. /* Use by the pseudo random number generator. */
1374. static UBaseType_t ulNextRand;
1375. const uint8_t ucMACAddress[ 6 ] = { configMAC_ADDR0, configMAC_ADDR1,
    configMAC_ADDR2, configMAC_ADDR3, configMAC_ADDR4, configMAC_ADDR5 };
1376.
1377. static const uint8_t ucIPAddress[ 4 ] = { configIP_ADDR0,
    configIP_ADDR1, configIP_ADDR2, configIP_ADDR3 };
1378. static const uint8_t ucNetMask[ 4 ] = { configNET_MASK0,
    configNET_MASK1, configNET_MASK2, configNET_MASK3 };
1379. static const uint8_t ucGatewayAddress[ 4 ] = { configGATEWAY_ADDR0,
    configGATEWAY_ADDR1, configGATEWAY_ADDR2, configGATEWAY_ADDR3 };
1380. static const uint8_t ucDNSServerAddress[ 4 ] = { configDNS_SERVER_ADDR0,
    configDNS_SERVER_ADDR1, configDNS_SERVER_ADDR2, configDNS_SERVER_ADDR3 };
1381.
1382. EventGroupHandle_t g_Ethernet_Start_Event;
1383. TaskHandle_t Listener_Task_Handle, Sender_Task_Handle;
1384. Message_Identifier_Struct g_Ethernet_Identifiers[20];
1385. ///////////////////////////////////////////////////////////////////
1386.
1387. ///////////////////////////////////////////////////////////////////
1388. void Notify_Ethernet_Listeners( uint8_t * pMessage, size_t length )
1389. {
1390.     // Notify Ethernet Queue Pointers
1391.     for(int i = 0; i < NUM_IDENTIFIERS; i++)
1392.     {
1393.         if( g_Ethernet_Identifiers[i].valid == true )
1394.         {
1395.             MessageBufferHandle_t messageBuffer =
                g_Ethernet_Identifiers[i].messagePointer;

```

```

1396.
1397.             xMessageBufferSend(
1398.
1399.             messageBuffer,
1400.
1401.             portMAX_DELAY
1402.
1403.             );
1404.         }
1405.     }
1406. ///////////////////////////////////////////////////////////////////
1407.
1408. ///////////////////////////////////////////////////////////////////
1409. void Ethernet_Task( void *pvParameters )
1410. {
1411.     EventBits_t uxBits;
1412.     g_Ethernet_Start_Event = xEventGroupCreate();
1413.     // Reset Ethernet Queue Pointers
1414.     for(int i = 0; i < NUM_IDENTIFIERS; i++)
1415.     {
1416.         g_Ethernet_Identifiers[i].valid = false;
1417.     }
1418.
1419.     NetworkConfig();
1420.
1421.
1422.     /* Wait a maximum of 50 days for either bit 0 or bit 1 to be
1423.        set within
1424.        the event group. Clear the bits before exiting. */
1425.     uxBits = xEventGroupWaitBits(
1426.         g_Ethernet_Start_Event, /* The event group being tested. */
1427.         BIT_0, /*
1428.            The bits within the event group to wait for. */
1429.         pdTRUE,
1430.         /* BIT_0 should be cleared before returning. */
1431.         pdFALSE,
1432.         /* Any bit will trigger */
1433.         portMAX_DELAY ); /* Wait a maximum of 50 days */
1434.     vTVPServerSock();
1435.
1436.     // Shouldn't reach here but delete if it does
1437.     vTaskDelete( NULL );
1438. }
1439. ///////////////////////////////////////////////////////////////////
1440.
1441. ///////////////////////////////////////////////////////////////////
1442. void vTVPServerSock(void)
1443. {
1444.     struct freertos_sockaddr xClient, xBindAddress;
1445.     Socket_t xListeningSocket, xConnectedSocket;
1446.     socklen_t xSize = sizeof( xClient );
1447.     static const TickType_t xReceiveTimeout = portMAX_DELAY;

```

```

1444.         const BaseType_t xBacklog = 20;
1445.
1446.         /* Attempt to open the socket. */
1447.         xListeningSocket = FreeRTOS_socket( FREERTOS_AF_INET,
1448.         FREERTOS SOCK_STREAM, /* SOCK_STREAM for TCP. */
1449.         FREERTOS_IPPROTO_TCP );
1450.
1451.         /* Check the socket was created. */
1452.         configASSERT( xListeningSocket != FREERTOS_INVALID_SOCKET );
1453.
1454.         /* If FREERTOS_SO_RCVBUF or FREERTOS_SO_SNDBUF are to be used
with
1455.         FreeRTOS_setsockopt() to change the buffer sizes from their
default then do
1456.         it here!. (see the FreeRTOS_setsockopt() documentation. */
1457.
1458.         /* If ipconfigUSE_TCP_WIN is set to 1 and
FREERTOS_SO_WIN_PROPERTIES is to
1459.         be used with FreeRTOS_setsockopt() to change the sliding
window size from
1460.         its default then do it here! (see the FreeRTOS_setsockopt()
1461.         documentation. */
1462.
1463.         /* Set a time out so accept() will just wait for a connection.
*/
1464.         FreeRTOS_setsockopt( xListeningSocket,
1465.                               0,
1466.                               FREERTOS_SO_RCVTIMEO,
1467.                               &xReceiveTimeOut,
1468.                               sizeof( xReceiveTimeOut
) );
1469.
1470.         /* Set the listening port to 10000. */
1471.         xBindAddress.sin_port = ( uint16_t ) 10000;
1472.         xBindAddress.sin_port = FreeRTOS_htons( xBindAddress.sin_port
);
1473.
1474.         /* Bind the socket to the port that the client RTOS task will
send to. */
1475.         FreeRTOS_bind( xListeningSocket, &xBindAddress, sizeof(
xBindAddress ) );
1476.
1477.         /* Set the socket into a listening state so it can accept
connections.
1478.         The maximum number of simultaneous connections is limited to
20. */
1479.         FreeRTOS_listen( xListeningSocket, xBacklog );
1480.
1481.         int i = 0;
1482.         for( ;; )
1483.         {
1484.             Task_Param_Struct msg;
1485.             /* Wait for incoming connections. */

```

```

1486.             xConnectedSocket = FreeRTOS_accept( xListeningSocket,
&xClient, &xSize );
1487.             configASSERT( xConnectedSocket !=
FREERTOS_INVALID_SOCKET );
1488.
1489.             /* Spawn a RTOS task to handle the connection. */
1490.             msg.index = i;
1491.             msg.msg = (void *) xConnectedSocket;
1492.             xTaskCreate( Listener_Task,
1493.                         "Listener_Task",
1494.                         ( uint16_t )
configMINIMAL_STACK_SIZE * 3,
1495.                         ( void * ) &msg,
1496.                         LISTENER_TASK_PRIORITY - i,
1497.                         Listener_Task_Handle );
1498.
1499.             xTaskCreate( Sender_Task,
1500.                         "Sender_Task",
1501.                         ( uint16_t )
configMINIMAL_STACK_SIZE * 3,
1502.                         ( void * ) &msg,
1503.                         SENDER_TASK_PRIORITY - i,
1504.                         Sender_Task_Handle );
1505.
1506.             g_Ethernet_Identifiers[i].messagePointer =
xMessageBufferCreate( 64 );
1507.             g_Ethernet_Identifiers[i].valid = true;
1508.             i++;
1509.         }
1510.     }
1511.     //////////////////////////////////////
1512.
1513.     //////////////////////////////////////
1514.     static void Sender_Task( void *pvParameters )
1515.     {
1516.         int index = ((Task_Param_Struct *)pvParameters)->index;
1517.         uint8_t ucRxData [64] = {0};
1518.         size_t xReceivedBytes;
1519.         MessageBufferHandle_t messageBuffer =
g_Ethernet_Identifiers[index].messagePointer;
1520.
1521.         Socket_t xSocket = (Socket_t) ((Task_Param_Struct
*)pvParameters)->msg;
1522.         BaseType_t xAlreadyTransmitted = 0, xBytesSent = 0;
1523.         size_t xLenToSend;
1524.         bool closeSocket = false;
1525.         //BaseType_t returnStatus;
1526.
1527.
1528.         while( closeSocket == false )
1529.         {
1530.             xReceivedBytes = xMessageBufferReceive(
1531.
messageBuffer,
1532.
ucRxData,

```

```

1533.         sizeof( ucRxData ),
1534.         portMAX_DELAY
1535.     );
1536.
1537.         xAlreadyTransmitted = 0;
1538.         /* Keep sending until the entire buffer has been sent.
1539.         */
1540.         while( xAlreadyTransmitted < xReceivedBytes &&
1541.             closeSocket == false)
1542.         {
1543.             /* How many bytes are left to send? */
1544.             xLenToSend = xReceivedBytes -
1545.                 xAlreadyTransmitted;
1546.             xBytesSent = FreeRTOS_send( /* The socket being
1547.                 sent to. */
1548.                 xSocket,
1549.                 /* The data being sent. */
1550.                 &( ucRxData[ xAlreadyTransmitted ] ),
1551.                 /* The remaining length of data to send. */
1552.                 xLenToSend,
1553.                 /* ulFlags. */
1554.                 0 );
1555.
1556.             if( xBytesSent >= 0 )
1557.             {
1558.                 /* Data was sent successfully. */
1559.                 xAlreadyTransmitted += xBytesSent;
1560.             }
1561.             else
1562.             {
1563.                 /* Error - break out of the loop for
1564.                 graceful socket close. */
1565.                 break;
1566.             }
1567.         }
1568.     }
1569.
1570.     /* Initiate graceful shutdown. */
1571.     FreeRTOS_shutdown( xSocket, FREERTOS_SHUT_RDWR );
1572.
1573.     /* Wait for the socket to disconnect gracefully (indicated by
1574.     FreeRTOS_recv()
1575.     returning a FREERTOS_EINVAL error) before closing the socket.
1576.     */

```

```

1571.         while( FreeRTOS_recv( xSocket, ucRxData, sizeof( ucRxData ), 0
1572.             ) >= 0 )
1573.             {
1574.                 /* Wait for shutdown to complete.  If a receive block
1575.                    time is used then
1576.                    this delay will not be necessary as FreeRTOS_recv()
1577.                    will place the RTOS task
1578.                    into the Blocked state anyway. */
1579.                 const TickType_t xDelay = 250 / portTICK_PERIOD_MS;
1580.                 vTaskDelay( xDelay );
1581.                 /* Note - real applications should implement a timeout
1582.                    here, not just
1583.                    loop forever. */
1584.             }
1585.             /* The socket has shut down and is safe to close. */
1586.             FreeRTOS_closesocket( xSocket );
1587.             vTaskDelete( NULL );
1588.         }
1589.         ///////////////////////////////////////////////////
1590.         ///////////////////////////////////////////////////
1591.         static void Listener_Task( void *pvParameters )
1592.         {
1593.
1594.             Socket_t xSocket = (Socket_t) ((Task_Param_Struct
1595.                 *)pvParameters)->msg;
1596.             static char cRxedData[ BUFFER_SIZE ];
1597.             BaseType_t lBytesReceived;
1598.
1599.             for( ;; )
1600.             {
1601.                 /* Receive another block of data into the cRxedData
1602.                    buffer. */
1603.                 lBytesReceived = FreeRTOS_recv( xSocket, &cRxedData,
1604.                     BUFFER_SIZE, 0 );
1605.                 if( lBytesReceived > 0 )
1606.                 {
1607.                     /* Data was received, process it here. */
1608.                     if(cRxedData[0] == DiscoveryStart)
1609.                     {
1610.                         xEventGroupSetBits(
1611.                             g_Discovery_Event_Group,
1612.                             BIT_0
1613.                                 );
1614.                     }
1615.                     else
1616.                     {
1617.                         // TODO: Status Changed

```

```

1618.         }
1619.     }
1620.     else if( lBytesReceived == 0 )
1621.     {
1622.         /* No data was received, but FreeRTOS_recv()
did not return an error.
1623.         Timeout? */
1624.         continue;
1625.     }
1626.     else
1627.     {
1628.         /* Error (maybe the connected socket already
shut down the socket?).
1629.         Attempt graceful shutdown. */
1630.         FreeRTOS_shutdown( xSocket, FREERTOS_SHUT_RDWR
);
1631.         break;
1632.     }
1633. }
1634.
1635.     /* The RTOS task will get here if an error is received on a
read. Ensure the
1636.     socket has shut down (indicated by FreeRTOS_recv() returning a
FREERTOS_EINVAL
1637.     error before closing the socket). */
1638.
1639.     while( FreeRTOS_recv( xSocket, cRxedData, sizeof(cRxedData), 0
) >= 0 )
1640.     {
1641.         /* Wait for shutdown to complete. If a receive block
time is used then
1642.         this delay will not be necessary as FreeRTOS_recv()
will place the RTOS task
1643.         into the Blocked state anyway. */
1644.         const TickType_t xDelay = 250 / portTICK_PERIOD_MS;
1645.         vTaskDelay( xDelay );
1646.
1647.         /* Note - real applications should implement a timeout
here, not just
1648.         loop forever. */
1649.     }
1650.
1651.     /* Shutdown is complete and the socket can be safely closed.
*/
1652.     FreeRTOS_closesocket( xSocket );
1653.
1654.     /* Must not drop off the end of the RTOS task - delete the
RTOS task. */
1655.     vTaskDelete( NULL );
1656. }
1657. //////////////////////////////////////////////////
1658.
1659. //////////////////////////////////////////////////
1660. void NetworkConfig(void)
1661. {
1662.     // Init FreeRTOS IP

```



```

1663.         FreeRTOS_IPInit( ucIPAddress, ucNetMask, ucGatewayAddress,
1664.             ucDNSServerAddress, ucMACAddress );
1665. //////////////////////////////////////////////////
1666. //////////////////////////////////////////////////
1667. //////////////////////////////////////////////////
1668. void vApplicationIPNetworkEventHook( eIPCallbackEvent_t eNetworkEvent )
1669. {
1670.     static BaseType_t xTasksAlreadyCreated = pdFALSE;
1671.
1672.     /* Both eNetworkUp and eNetworkDown events can be processed here. */
1673.     if( eNetworkEvent == eNetworkUp )
1674.     {
1675.         /* Create the tasks that use the TCP/IP stack if they have not
1676.         already
1677.         been created. */
1678.         if( xTasksAlreadyCreated == pdFALSE )
1679.         {
1680.             xEventGroupSetBits( g_Ethernet_Start_Event,
1681.                                 BIT_0 );
1682.             /*
1683.             * For convenience, tasks that use FreeRTOS+TCP can be
1684.             created here
1685.             * to ensure they are not created before the network is
1686.             usable.
1687.             */
1688.             xTasksAlreadyCreated = pdTRUE;
1689.         }
1690.     }
1691.     else if( eNetworkEvent == eNetworkDown )
1692.     {
1693.     }
1694. //////////////////////////////////////////////////
1695. //////////////////////////////////////////////////
1696. //////////////////////////////////////////////////
1697. #if( ipconfigUSE_LLMNR != 0 ) || ( ipconfigUSE_NBNS != 0 ) || (
1698.     ipconfigDHCP_REGISTER_HOSTNAME == 1 )
1699.     const char *pcApplicationHostnameHook( void )
1700.     {
1701.         /* Assign the name "FreeRTOS" to this network node.
1702.         This function will
1703.         be called during the DHCP: the machine will be
1704.         registered with an IP
1705.         address plus this name. */
1706.         return mainHOST_NAME;
1707.     }
1708. #endif
1709. //////////////////////////////////////////////////
1710. //////////////////////////////////////////////////
1711. #if( ipconfigUSE_LLMNR != 0 ) || ( ipconfigUSE_NBNS != 0 )

```

```

1712.
1713.         BaseType_t xApplicationDNSQueryHook( const char *pcName )
1714.         {
1715.             BaseType_t xReturn = 0 ;
1716.             return xReturn;
1717.         }
1718. #endif
1719. ///////////////////////////////////////////////////////////////////
1720.
1721. ///////////////////////////////////////////////////////////////////
1722. uint32_t ulApplicationGetNextSequenceNumber( uint32_t ulSourceAddress,
1723.
1724.                                             uint16_t usSourcePort,
1725.
1726.                                             uint32_t ulDestinationAddress,
1727.
1728.                                             uint16_t usDestinationPort )
1729. {
1730.     ( void ) ulSourceAddress;
1731.     ( void ) usSourcePort;
1732.     ( void ) ulDestinationAddress;
1733.     ( void ) usDestinationPort;
1734.     return uxRand();
1735. }
1736. ///////////////////////////////////////////////////////////////////
1737. UBaseType_t uxRand( void )
1738. {
1739.     const uint32_t ulMultiplier = 0x015a4e35UL, ulIncrement = 1UL;
1740.
1741.     /* Utility function to generate a pseudo random number. */
1742.     ulNextRand = ( ulMultiplier * ulNextRand ) + ulIncrement;
1743.     return( ( int ) ( ulNextRand >> 16UL ) & 0x7fffUL );
1744. }
1745. ///////////////////////////////////////////////////////////////////
1746.
1747.
1748.
1749.
1750. /**
1751.
1752. *****
1753. *****
1754. * File Name           : serialtopc.c
1755. * Description         :
1756. *****
1757. *****
1758. * @attention
1759. *
1760. *
1761. *****
1762. *****

```

```

1759.  */
1760.
1761. //////////////////////////////////////////////////
1762. #include "serialtopc.h"
1763. //////////////////////////////////////////////////
1764.
1765. //////////////////////////////////////////////////
1766. void USART3_Init(int baudRate);
1767. //////////////////////////////////////////////////
1768.
1769. //////////////////////////////////////////////////
1770. UART_HandleTypeDef huart;
1771. //////////////////////////////////////////////////
1772.
1773. //////////////////////////////////////////////////
1774. void USART3_Init(int baudRate)
1775. {
1776.     huart.Instance = USART3;
1777.
1778.     huart.Init.WordLength = UART_WORDLENGTH_8B;
1779.     huart.Init.StopBits = UART_STOPBITS_1;
1780.     huart.Init.Parity = UART_PARITY_NONE;
1781.     huart.Init.Mode = UART_MODE_TX_RX;
1782.     huart.Init.OverSampling = UART_OVERSAMPLING_16;
1783.     huart.Init.BaudRate = baudRate;
1784.
1785.     HAL_UART_Init(&huart);
1786. }
1787. //////////////////////////////////////////////////
1788.
1789. //////////////////////////////////////////////////
1790. void PC_Setup_Serial(void)
1791. {
1792.     USART3_Init(115200);
1793. }
1794. //////////////////////////////////////////////////
1795.
1796. //////////////////////////////////////////////////
1797. HAL_StatusTypeDef PC_Send_String(uint8_t* string, uint16_t size)
1798. {
1799.     return HAL_UART_Transmit(&huart, string, size, 10);
1800. }
1801. //////////////////////////////////////////////////
1802.
1803.
1804.
1805.
1806. /**
1807.
1808. *****
1809. *****
1810. * File Name           : stm32f7xx_hal_msp.c
1811. * Description         : This file provides code for the MSP
1812.   Initialization
1813. *                     and de-Initialization codes.

```

```

1811.
    *****
    ****
1812.  * @attention
1813.  *
1814.  *
1815.
    *****
    ****
1816.  */
1817.
1818. ///////////////////////////////////////////////////////////////////
1819. /* Includes -----
    -----*/
1820. #include "main.h"
1821. ///////////////////////////////////////////////////////////////////
1822.
1823. ///////////////////////////////////////////////////////////////////
1824. void HAL_MspInit(void)
1825. {
1826.
1827.  __HAL_RCC_PWR_CLK_ENABLE();
1828.  __HAL_RCC_SYSCFG_CLK_ENABLE();
1829.
1830.  /* System interrupt init*/
1831.  /* PendSV_IRQn interrupt configuration */
1832.  HAL_NVIC_SetPriority(PendSV_IRQn, 15, 0);
1833. }
1834. ///////////////////////////////////////////////////////////////////
1835.
1836.
1837.
1838.
1839.
1840. /**
1841.
    *****
    ****
1842.  * @file    stm32f7xx_hal_timebase_TIM.c
1843.  * @brief   HAL time base based on the hardware TIM.
1844.
    *****
    ****
1845.  * @attention
1846.  *
1847.  *
1848.
    *****
    ****
1849.  */
1850.
1851. ///////////////////////////////////////////////////////////////////
1852. /* Includes -----
    -----*/
1853. #include "stm32f7xx_hal.h"
1854. #include "stm32f7xx_hal_tim.h"

```

```

1855. ///////////////////////////////////////////////////
1856.
1857. ///////////////////////////////////////////////////
1858. TIM_HandleTypeDef          htim1;
1859. ///////////////////////////////////////////////////
1860.
1861. ///////////////////////////////////////////////////
1862. HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
1863. {
1864.     RCC_ClkInitTypeDef      clkconfig;
1865.     uint32_t                 uwTimclock = 0;
1866.     uint32_t                 uwPrescalerValue = 0;
1867.     uint32_t                 pFLatency;
1868.
1869.     /*Configure the TIM1 IRQ priority */
1870.     HAL_NVIC_SetPriority(TIM1_UP_TIM10_IRQn, TickPriority ,0);
1871.
1872.     /* Enable the TIM1 global Interrupt */
1873.     HAL_NVIC_EnableIRQ(TIM1_UP_TIM10_IRQn);
1874.
1875.     /* Enable TIM1 clock */
1876.     __HAL_RCC_TIM1_CLK_ENABLE();
1877.
1878.     /* Get clock configuration */
1879.     HAL_RCC_GetClockConfig(&clkconfig, &pFLatency);
1880.
1881.     /* Compute TIM1 clock */
1882.     uwTimclock = 2*HAL_RCC_GetPCLK2Freq();
1883.
1884.     /* Compute the prescaler value to have TIM1 counter clock equal to
1885.     1MHz */
1886.     uwPrescalerValue = (uint32_t) ((uwTimclock / 1000000) - 1);
1887.
1888.     /* Initialize TIM1 */
1889.     htim1.Instance = TIM1;
1890.
1891.     /* Initialize TIMx peripheral as follow:
1892.     + Period = [(TIM1CLK/1000) - 1]. to have a (1/1000) s time base.
1893.     + Prescaler = (uwTimclock/1000000 - 1) to have a 1MHz counter clock.
1894.     + ClockDivision = 0
1895.     + Counter direction = Up
1896.     */
1897.     htim1.Init.Period = (1000000 / 1000) - 1;
1898.     htim1.Init.Prescaler = uwPrescalerValue;
1899.     htim1.Init.ClockDivision = 0;
1900.     htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
1901.     if(HAL_TIM_Base_Init(&htim1) == HAL_OK)
1902.     {
1903.         /* Start the TIM time Base generation in interrupt mode */
1904.         return HAL_TIM_Base_Start_IT(&htim1);
1905.     }
1906.
1907.     /* Return function status */
1908.     return HAL_ERROR;
1909. }
1909. ///////////////////////////////////////////////////

```

```

1910.
1911. //////////////////////////////////////////////////
1912. void HAL_SuspendTick(void)
1913. {
1914.     /* Disable TIM1 update Interrupt */
1915.     __HAL_TIM_DISABLE_IT(&htim1, TIM_IT_UPDATE);
1916. }
1917. //////////////////////////////////////////////////
1918.
1919. //////////////////////////////////////////////////
1920. /**
1921.  * @brief Resume Tick increment.
1922.  * @note Enable the tick increment by Enabling TIM1 update interrupt.
1923.  * @param None
1924.  * @retval None
1925.  */
1926. void HAL_ResumeTick(void)
1927. {
1928.     /* Enable TIM1 Update interrupt */
1929.     __HAL_TIM_ENABLE_IT(&htim1, TIM_IT_UPDATE);
1930. }
1931. //////////////////////////////////////////////////
1932.
1933.
1934.
1935.
1936.
1937. /**
1938.
1939.     *****
1940.     *****
1941.     *****
1942.     *****
1943.     *****
1944.     *****
1945.     *****
1946.     *****
1947.     *****
1948.     *****
1949.     *****
1950.     *****
1951.     *****
1952.     *****
1953.     *****
1954.     *****
1955.     *****
1956.     *****
1957.     *****
1958.     *****
1959.     *****

```

```
1960. void NMI_Handler(void)
1961. {
1962. }
1963. ////////////////////////////////////////////////////
1964.
1965. ////////////////////////////////////////////////////
1966. void HardFault_Handler(void)
1967. {
1968.     while (1)
1969.     {
1970.     }
1971. }
1972. ////////////////////////////////////////////////////
1973.
1974. ////////////////////////////////////////////////////
1975. void MemManage_Handler(void)
1976. {
1977.     while (1)
1978.     {
1979.     }
1980. }
1981. ////////////////////////////////////////////////////
1982.
1983. ////////////////////////////////////////////////////
1984. void BusFault_Handler(void)
1985. {
1986.     while (1)
1987.     {
1988.     }
1989. }
1990. ////////////////////////////////////////////////////
1991.
1992. ////////////////////////////////////////////////////
1993. void UsageFault_Handler(void)
1994. {
1995.     while (1)
1996.     {
1997.     }
1998. }
1999. ////////////////////////////////////////////////////
2000.
2001. ////////////////////////////////////////////////////
2002. void DebugMon_Handler(void)
2003. {
2004.
2005. }
2006. ////////////////////////////////////////////////////
2007.
2008. ////////////////////////////////////////////////////
2009. void CAN1_RX0_IRQHandler(void)
2010. {
2011.     HAL_CAN_IRQHandler(&hcan1);
2012. }
2013. ////////////////////////////////////////////////////
2014.
2015. ////////////////////////////////////////////////////
```

```

2016. void CAN1_RX1_IRQHandler(void)
2017. {
2018.   HAL_CAN_IRQHandler(&hcan1);
2019. }
2020. //////////////////////////////////////////////////
2021.
2022. //////////////////////////////////////////////////
2023. void TIM1_UP_TIM10_IRQHandler(void)
2024. {
2025.   HAL_TIM_IRQHandler(&htim1);
2026. }
2027. //////////////////////////////////////////////////
2028.
2029.
2030.
2031.
2032. /**
2033.
2034.   ****
2035.   * File Name           : CAN.c
2036.   * Description        : This file provides code for the configuration
2037.   *                    : of the CAN instances.
2038.   ****
2039.   * @attention
2040.   *
2041.   ****
2042.   */
2043.
2044. //////////////////////////////////////////////////
2045. #include "main.h"
2046. #include "can.h"
2047. #include "FreeRTOS.h"
2048. #include "event_groups.h"
2049. #include "timers.h"
2050. //////////////////////////////////////////////////
2051.
2052. //////////////////////////////////////////////////
2053. CAN_HandleTypeDef hcan1;
2054. //////////////////////////////////////////////////
2055.
2056. //////////////////////////////////////////////////
2057. extern EventGroupHandle_t g_CAN_Message_Event_Group;
2058. //////////////////////////////////////////////////
2059.
2060. //////////////////////////////////////////////////
2061. void HAL_CAN_MspInit(CAN_HandleTypeDef* canHandle)
2062. {
2063.   GPIO_InitTypeDef GPIO_InitStructure = {0};
2064.   if(canHandle->Instance==CAN1)
2065.   {

```



```

2066.     /* CAN1 clock enable */
2067.     __HAL_RCC_CAN1_CLK_ENABLE();
2068.
2069.     __HAL_RCC_GPIOD_CLK_ENABLE();
2070.     /**CAN1 GPIO Configuration
2071.     PD0      -----> CAN1_RX
2072.     PD1      -----> CAN1_TX
2073.     */
2074.     GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1;
2075.     GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
2076.     GPIO_InitStruct.Pull = GPIO_NOPULL;
2077.     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
2078.     GPIO_InitStruct.Alternate = GPIO_AF9_CAN1;
2079.     HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
2080.
2081.     /* CAN1 interrupt Init */
2082.     HAL_NVIC_SetPriority(CAN1_RX0_IRQn, 5, 0);
2083.     HAL_NVIC_EnableIRQ(CAN1_RX0_IRQn);
2084.     HAL_NVIC_SetPriority(CAN1_RX1_IRQn, 5, 0);
2085.     HAL_NVIC_EnableIRQ(CAN1_RX1_IRQn);\
2086. }
2087. }
2088. ///////////////////////////////////////////////////
2089.
2090. ///////////////////////////////////////////////////
2091. void HAL_CAN_MspDeInit(CAN_HandleTypeDef* canHandle)
2092. {
2093.
2094.     if(canHandle->Instance==CAN1)
2095.     {
2096.         /* Peripheral clock disable */
2097.         __HAL_RCC_CAN1_CLK_DISABLE();
2098.
2099.         /**CAN1 GPIO Configuration
2100.         PD0      -----> CAN1_RX
2101.         PD1      -----> CAN1_TX
2102.         */
2103.         HAL_GPIO_DeInit(GPIOD, GPIO_PIN_0|GPIO_PIN_1);
2104.
2105.         /* CAN1 interrupt Deinit */
2106.         HAL_NVIC_DisableIRQ(CAN1_RX0_IRQn);
2107.         HAL_NVIC_DisableIRQ(CAN1_RX1_IRQn);
2108.     }
2109. }
2110. ///////////////////////////////////////////////////
2111.
2112. ///////////////////////////////////////////////////
2113. STATUS_StatusTypeDef CAN_Receive(uint32_t RxFifo, CAN_RxHeaderTypeDef
    *pHeader, uint8_t aData[])
2114. {
2115.
2116.         if(HAL_CAN_GetRxMessage(&hcan1, RxFifo, pHeader, aData) !=
    HAL_OK)
2117.         {
2118.             // HAL_ERROR
2119.         }

```

```

2120.
2121.         return STATUS_OK;
2122. }
2123. //////////////////////////////////////////////////
2124.
2125. //////////////////////////////////////////////////
2126. STATUS_StatusTypeDef CAN_Send(uint16_t stdid, uint8_t length, uint8_t
    *data)
2127. {
2128.     CAN_TxHeaderTypeDef TxHeader;
2129.     uint8_t TxData[8];
2130.     uint32_t TxMailbox;
2131.
2132.     /*##-- Start the Transmission process #####*/
2133.     TxHeader.StdId = stdid;
2134.     TxHeader.RTR = CAN_RTR_DATA;
2135.     TxHeader.IDE = CAN_ID_STD;
2136.     TxHeader.DLC = length;
2137.     TxHeader.TransmitGlobalTime = DISABLE;
2138.
2139.     for(int i = 0; i < length; i++)
2140.     {
2141.         TxData[i] = data[i];
2142.     }
2143.
2144.     /* Request transmission */
2145.     if(HAL_CAN_AddTxMessage(&hcan1, &TxHeader, TxData, &TxMailbox)
    != HAL_OK)
2146.     {
2147.         /* Transmission request Error */
2148.     }
2149.
2150.     //TODO This may not actually send the message
2151.
2152.     return STATUS_OK;
2153. }
2154. //////////////////////////////////////////////////
2155.
2156. //////////////////////////////////////////////////
2157. // Configure the CAN peripheral
2158. void CAN_Setup_Instance(void)
2159. {
2160.     // STM32F746 has two can controllers, but we are using the
    first one
2161.     hcan1.Instance = CAN1; // First CAN instance
2162.
2163.     // Configure CAN features
2164.     hcan1.Init.TimeTriggeredMode = DISABLE;
2165.     hcan1.Init.AutoBusOff = ENABLE; // Hardware will
    automatically start recovering from Bus-Off state
2166.
    // Recovery will
    happen once 128 occurrences of 11 recessive bits occur
2167.     hcan1.Init.AutoWakeUp = DISABLE; // Sleep Mode
    isn't used for power saving mode so this isn't needed

```

```

2168.         hcan1.Init.AutoRetransmission      = ENABLE; // Keep sending
              the message until it is successful
2169.         hcan1.Init.ReceiveFifoLocked       = DISABLE; // If the receive
              FIFO is full, overwrite old messages
2170.         hcan1.Init.TransmitFifoPriority     = DISABLE; // Priority driver
              by the identifier of the message
2171.         hcan1.Init.Mode                     = CAN_MODE_NORMAL;
2172.
2173.         // APB1 Peripheral Clock: 54 MHz
2174.         // Time Quantum 500.0 ns
2175.         hcan1.Init.SyncJumpWidth           = CAN_SJW_1TQ;
2176.         hcan1.Init.TimeSeg1                 = CAN_BS1_13TQ;
2177.         hcan1.Init.TimeSeg2                 = CAN_BS2_2TQ;
2178.         hcan1.Init.Prescaler                 = 27;
2179.
2180.         // Error Codes HAL_CAN_ERROR_TIMEOUT
2181.         // Calls HAL_CAN_MspInit which initializes low level hardware
2182.         //           Nothing should go wrong in this but need to
check
2183.         // Turns off sleep mode which could cause timeout
2184.         //           Sleep mode for power saving is not currently
used
2185.         // Requests initialization
2186.         //           The only thing that could go wrong with this is
that
2187.         //           the hardware doesn't monitor 11 consecutive
recessive bits
2188.         //           Call HAL_CAN_Init again if this
happens
2189.         //           If it fails again reset device
2190.         if(HAL_CAN_Init(&hcan1) != HAL_OK) // Successful call sets
state HAL_CAN_STATE_READY
2191.         {
2192.             // Initialization timed out try again
2193.             if(HAL_CAN_Init(&hcan1) != HAL_OK)
2194.             {
2195.                 // Initialization failed twice
2196.                 //           Reset device
2197.                 CAN_Error_Handler();
2198.             }
2199.         }
2200. }
2201. ///////////////////////////////////////////////////////////////////
2202.
2203. ///////////////////////////////////////////////////////////////////
2204. // Initialize CAN Filters
2205. void CAN_Setup_Filters(uint16_t StdID, uint8_t fifoAssignment, uint32_t
filterBank)
2206. {
2207.     CAN_FilterTypeDef  sFilterConfig;
2208.
2209.     sFilterConfig.FilterBank = filterBank; // 28 Filter banks
2210.     sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
2211.     sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
2212.
2213.     sFilterConfig.FilterIdHigh = StdID << 5; // 16 bits

```

```

2214.  sFilterConfig.FilterIdLow = 0x0000;
2215.  sFilterConfig.FilterMaskIdHigh = 0xFFFF;
2216.  sFilterConfig.FilterMaskIdLow = 0xFFFF;
2217.
2218.
2219.      // CAN_RX_FIFO0 or CAN_RX_FIFO1
2220.      sFilterConfig.FilterFIFOAssignment = fifoAssignment; // Two
        Receive FIFOs with three stages each
2221.  sFilterConfig.FilterActivation = ENABLE;           // Activate filter
2222.  sFilterConfig.SlaveStartFilterBank = 0;           // Assign all CAN
        Filter Banks to the activated CAN controller (CAN1)
2223.
                                                // This is
        used because this MCU has two CAN controllers and this parameter
2224.
                                                //
        configures which filter banks go to which controller
2225.
2226.
2227.      // Error Codes HAL_CAN_ERROR_NOT_INITIALIZED
2228.      // If HAL_CAN_ConfigFilter fails it is because the instance
        isn't setup
2229.      // or is in error state. The other option is
        sleeping but that isn't used
2230.      // in the application
2231.  if(HAL_CAN_ConfigFilter(&hcan1, &sFilterConfig) != HAL_OK)
2232.  {
2233.      if(hcan1.State == HAL_CAN_STATE_RESET)
2234.      {
2235.          // CAN wasn't initialized
2236.          CAN_Setup_Instance();
2237.
2238.          // If filter setup fails again reset device
2239.          if(HAL_CAN_ConfigFilter(&hcan1, &sFilterConfig)
        != HAL_OK)
2240.          {
2241.              // Filter configuration Error
2242.              CAN_Error_Handler();
2243.          }
2244.      }
2245.      else if (hcan1.State == HAL_CAN_STATE_ERROR)
2246.      {
2247.
2248.          if( hcan1.ErrorCode == HAL_CAN_ERROR_TIMEOUT ||
2249.             hcan1.ErrorCode ==
        HAL_CAN_ERROR_NOT_INITIALIZED ||
2250.             hcan1.ErrorCode ==
        HAL_CAN_ERROR_NOT_READY ||
2251.             hcan1.ErrorCode ==
        HAL_CAN_ERROR_NOT_STARTED )
2252.          {
2253.              // Try reinitializing
2254.              CAN_Setup_Instance();
2255.

```

```

2256.                                     // If filter setup fails again reset
      device
2257.                                     if(HAL_CAN_ConfigFilter(&hcan1,
      &sFilterConfig) != HAL_OK)
2258.                                     {
2259.                                         // Filter configuration Error
2260.                                         CAN_Error_Handler();
2261.                                     }
2262.                                     }
2263.                                     else
2264.                                     {
2265.                                         //TODO: Log (Should not get here)
2266.                                         CAN_Error_Handler();
2267.                                     }
2268.                                     }
2269.     }
2270. }
2271. ///////////////////////////////////////////////////////////////////
2272.
2273. ///////////////////////////////////////////////////////////////////
2274. void CAN_Setup(void)
2275. {
2276.     // 1) CAN_Setup_Instance
2277.     //         Setup CAN instance
2278.     CAN_Setup_Instance();
2279.
2280.     // 2) CAN_Setup_Filters
2281.     //         Setup filters
2282.     CAN_Setup_Filters( Heartbeat_Identifier, CAN_RX_FIFO0, 0 );
2283.     CAN_Setup_Filters( Fuse_OK_Identifier, CAN_RX_FIFO0, 1 );
2284.     CAN_Setup_Filters( Fuse_Blown_Identifier, CAN_RX_FIFO1, 2 );
2285.     CAN_Setup_Filters( Error_1_Identifier, CAN_RX_FIFO1, 3 );
2286.     CAN_Setup_Filters( Error_2_Identifier, CAN_RX_FIFO1, 4 );
2287.     CAN_Setup_Filters( Discovery_Ready_Response_Identifier,
      CAN_RX_FIFO1, 5 );
2288.     CAN_Setup_Filters( Disocvery_Call_Response_Identifier,
      CAN_RX_FIFO1, 6 );
2289.
2290.     HAL_CAN_ActivateNotification(&hcan1,
      CAN_IT_RX_FIFO0_MSG_PENDING | CAN_IT_RX_FIFO1_MSG_PENDING);
2291.
2292.     // Error Codes
2293.     //         HAL_CAN_ERROR_TIMEOUT
2294.     //         HAL_CAN_ERROR_NOT_READY
2295.     if (HAL_CAN_Start(&hcan1) != HAL_OK)
2296.     {
2297.         // If return code is HAL_CAN_ERROR_NOT_READY then 1)
      and 2) were not successful
2298.         CAN_Error_Handler();
2299.     }
2300. }
2301. ///////////////////////////////////////////////////////////////////
2302.
2303. ///////////////////////////////////////////////////////////////////
2304. void CAN_Error_Handler(void)
2305. {

```

```

2306.          // TODO: Reset device
2307.          while(1)
2308.          {
2309.
2310.          }
2311. }
2312. ///////////////////////////////////////////////////
2313.
2314. ///////////////////////////////////////////////////
2315. void CAN_State_Handler(void)
2316. {
2317.     switch (hcan1.State)
2318.     {
2319.         case HAL_CAN_STATE_RESET:
2320.             CAN_Setup();
2321.             break;
2322.         case HAL_CAN_STATE_READY:
2323.
2324.             break;
2325.         case HAL_CAN_STATE_LISTENING:
2326.
2327.             break;
2328.         case HAL_CAN_STATE_SLEEP_PENDING:
2329.             // TODO: Log Error
2330.             // Should not get here
2331.             break;
2332.         case HAL_CAN_STATE_SLEEP_ACTIVE:
2333.             // TODO: Log Error
2334.             // Should not get here
2335.             break;
2336.         case HAL_CAN_STATE_ERROR:
2337.             switch(hcan1.ErrorCode)
2338.             {
2339.                 case HAL_CAN_ERROR_EWG:
2340.                     // HAL_CAN_ERROR_EWG
2341.                     // Protocol Error
Warning
2342.                     // This bit is set
when Receive or Transmit Error Counter is greater than
2343.                     //
or equal to 96.
2344.                     break;
2345.                 case HAL_CAN_ERROR_EPV:
2346.                     // HAL_CAN_ERROR_EPV
2347.                     // Error Passive
2348.                     // This bit is set
when Receive or Transmit Error Counter is greater than
2349.                     //
127
2350.                     break;
2351.                 case HAL_CAN_ERROR_BOF:
2352.                     // HAL_CAN_ERROR_BOF
2353.                     // Bus-off error
2354.                     // This bit is set
when Receive or Transmit Error Counter is greater than

```

```

2355. //
2356.     255 break;
2357. case HAL_CAN_ERROR_STF:
2358.     // HAL_CAN_ERROR_STF
2359.     // Stuff error
2360.     break;
2361. case HAL_CAN_ERROR_FOR:
2362.     // HAL_CAN_ERROR_FOR
2363.     // Form error
2364.     break;
2365. case HAL_CAN_ERROR_ACK:
2366.     // HAL_CAN_ERROR_ACK
2367.     // Acknowledgment
2368.     error break;
2369. case HAL_CAN_ERROR_BR:
2370.     // HAL_CAN_ERROR_BR
2371.     // Bit recessive
2372.     error break;
2373. case HAL_CAN_ERROR_BD:
2374.     // HAL_CAN_ERROR_BD
2375.     // Bit dominant
2376.     error break;
2377. case HAL_CAN_ERROR_CRC:
2378.     // HAL_CAN_ERROR_CRC
2379.     // CRC error
2380.     break;
2381. case HAL_CAN_ERROR_RX_FOV0:
2382.     // HAL_CAN_ERROR_RX_FOV0
2383.     // Rx FIFO0 overrun
2384.     error break;
2385. case HAL_CAN_ERROR_RX_FOV1:
2386.     // HAL_CAN_ERROR_RX_FOV1
2387.     // Rx FIFO1 overrun
2388.     error break;
2389. case HAL_CAN_ERROR_TX_ALST0:
2390.     // HAL_CAN_ERROR_TX_ALST0
2391.     // TxMailbox 0
2392.     transmit failure due to arbitration lost
2393.     break;
2394. case HAL_CAN_ERROR_TX_TERR0:
2395.     // HAL_CAN_ERROR_TX_TERR0
2396.     // TxMailbox 1
2397.     transmit failure due to tranmit error
2398.     break;
2399. case HAL_CAN_ERROR_TX_ALST1:
2400.     // HAL_CAN_ERROR_TX_ALST1
2401.     // TxMailbox 0
2402.     transmit failure due to arbitration lost

```

```

2402.                                     // HAL_CAN_ERROR_TX_TERR1
2403.                                     //                               TxMailbox 1
    transmit failure due to tranmit error
2404.                                     break;
2405.                                     case HAL_CAN_ERROR_TX_ALST2:
2406.                                     // HAL_CAN_ERROR_TX_ALST2
2407.                                     //                               TxMailbox 0
    transmit failure due to arbitration lost
2408.                                     break;
2409.                                     case HAL_CAN_ERROR_TX_TERR2:
2410.                                     // HAL_CAN_ERROR_TX_TERR2
2411.                                     //                               TxMailbox 1
    transmit failure due to tranmit error
2412.                                     break;
2413.                                     case HAL_CAN_ERROR_TIMEOUT:
2414.                                     // HAL_CAN_ERROR_TIMEOUT
2415.                                     //                               Timeout error
2416.                                     break;
2417.                                     case HAL_CAN_ERROR_NOT_INITIALIZED:
2418.                                     // HAL_CAN_ERROR_NOT_INITIALIZED
2419.                                     //                               Peripheral not
    initialized
2420.                                     break;
2421.                                     case HAL_CAN_ERROR_NOT_READY:
2422.                                     // HAL_CAN_ERROR_NOT_READY
2423.                                     //                               Peripheral not
    ready
2424.                                     break;
2425.                                     case HAL_CAN_ERROR_NOT_STARTED:
2426.                                     // HAL_CAN_ERROR_NOT_STARTED
2427.                                     //                               Peripheral not
    started
2428.                                     break;
2429.                                     case HAL_CAN_ERROR_PARAM:
2430.                                     // HAL_CAN_ERROR_PARAM
2431.                                     //                               Parameter error
2432.                                     break;
2433.                                     default:
2434.                                     break;
2435.                                     }
2436.                                     break;
2437.
2438.                                     default:
2439.
2440.                                     break;
2441.                                     }
2442.
2443. }
2444. ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2445.
2446. ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2447. void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
2448. {
2449.     BaseType_t xHigherPriorityTaskWoken, xResult;
2450.
2451.     /* xHigherPriorityTaskWoken must be initialized to pdFALSE. */

```



```

2452.         xHigherPriorityTaskWoken = pdFALSE;
2453.
2454.         HAL_CAN_DeactivateNotification(&hcan1,
CAN_IT_RX_FIFO0_MSG_PENDING);
2455.
2456.
2457.         /* Set bit 0 and bit 4 in xEventGroup. */
2458.         xResult = xEventGroupSetBitsFromISR(
2459.         g_CAN_Message_Event_Group, /* The event group being updated. */
2460.         BIT_FIFO_0, /* The bits being set. */
2461.         &xHigherPriorityTaskWoken );
2462.
2463.         /* Was the message posted successfully? */
2464.         if( xResult != pdFAIL )
2465.         {
2466.             /* If xHigherPriorityTaskWoken is now set to pdTRUE then a
context
2467.             switch should be requested. The macro used is port specific
and will
2468.             be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() -
refer to
2469.             the documentation page for the port being used. */
2470.             portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
2471.         }
2472.         else
2473.         {
2474.             Error_Handler();
2475.         }
2476.     }
2477. //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2478.
2479. //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2480. void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan)
2481. {
2482.     BaseType_t xHigherPriorityTaskWoken, xResult;
2483.
2484.     /* xHigherPriorityTaskWoken must be initialized to pdFALSE. */
2485.     xHigherPriorityTaskWoken = pdFALSE;
2486.
2487.     HAL_CAN_DeactivateNotification(&hcan1,
CAN_IT_RX_FIFO1_MSG_PENDING);
2488.
2489.     /* Set bit 0 and bit 4 in xEventGroup. */
2490.     xResult = xEventGroupSetBitsFromISR(
2491.     g_CAN_Message_Event_Group, /* The event group being updated. */
2492.     BIT_FIFO_1, /* The bits being set. */
2493.     &xHigherPriorityTaskWoken );
2494.
2495.     /* Was the message posted successfully? */
2496.     if( xResult != pdFAIL )

```

```
2497.      {
2498.          /* If xHigherPriorityTaskWoken is now set to pdTRUE then a
context
2499.          switch should be requested.  The macro used is port specific
and will
2500.          be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() -
refer to
2501.          the documentation page for the port being used. */
2502.          portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
2503.      }
2504.      else
2505.      {
2506.          Error_Handler();
2507.      }
2508.  }
2509.  //////////////////////////////////////
```