ON-BOARD PROTECTION DEVICE FOR BATTERY CHARGER

SHEIDA POURSHAKOUR
ASHWINI UMARIKAR
ABSTRACT

The goal of the thesis is to document the implementation of an On-board Protection Device that is to be used for On-board charging system in Electrical Vehicles.

On-board Protection Device is a protection part of the On-board charging system which communicates via CAN communication with the main charger unit. It includes basic fault detection hardware such as Ground Fault Monitoring, Residual Current Device and Main microcontroller for interface to and from peripheral fault monitoring devices and external communication networks. The Control logic is implemented inside a microcontroller, which also communicates with input channels and control output signals to circuit breaker which connects charger unit with supply. The microcontroller also send periodically the necessary data and status information about the peripheral protection devices to battery charger and also initiates close or open commands to circuit breaker, if there is any fault.

The thesis outlines the details of the protection device system design that can be helpful in utilizing or upgrading this system. The thesis is presented in two parts- The first part documents the design of Residual Current Device with 6mA leakage current and Ground Fault Monitoring device. The suggested solution is simulated by Pspice and is tested in the laboratory. The principle of protection devices is demonstrated through the experimental results. The second part documents the main logic part, implemented using TI Stellaris LM3S9B92 microcontroller, which has the required peripheral components to interface with different protection hardware and also has inbuilt CAN bus peripheral to communicate different CAN bus messages over CAN network. The control logic is first simulated in MATLAB using SIMULINK and STATEFLOW toolbox and then the actual implementation on the Evaluation Kit EKK-LM3S9B92 is described.
ACKNOWLEDGEMENT

We wish to express our sincere gratitude to our project supervisor from ABB, Dr. Hector Zelaya De La Parra for his exemplary guidance, support, monitoring and constant encouragement throughout the course of this thesis. We would also like to thank our supervisor and examiner Prof. Lars Asplund from Mälardalen University, for his helpful suggestions. And last but not the least we would like to thank our family for their constant encouragement and support without which this thesis would not be possible.
TABLE OF CONTENTS

1 INTRODUCTION .................................................................................................................. 1
  1.1 Purpose ........................................................................................................................... 1
  1.2 Scope .............................................................................................................................. 1
  1.3 Definitions ...................................................................................................................... 2
  1.4 Structure .......................................................................................................................... 3

2 PROBLEM DESCRIPTION .................................................................................................... 4

3 REQUIREMENTS .................................................................................................................... 6
  3.1 Hardware Requirements ................................................................................................. 6
    3.1.1 Power Supply ........................................................................................................... 6
    3.1.2 Ground Fault Monitoring ....................................................................................... 6
    3.1.3 Residual Current Monitoring .................................................................................. 6

4 SOLUTION ............................................................................................................................. 7
  4.1 Hardware Design Overview ............................................................................................ 7
  4.2 Technical Specification .................................................................................................... 8

5 HARDWARE RESULTS ........................................................................................................ 10
  5.1 Simulation Result .......................................................................................................... 10
  5.2 Implementation Results .................................................................................................. 10
    5.2.1 Test Result without Fault current ............................................................................. 10
    5.2.2 Test Result with 1mA Fault current ......................................................................... 11
    5.2.3 Test Result with 2mA Fault current ......................................................................... 11
    5.2.4 Test Result with 3mA Fault current ......................................................................... 12
    5.2.5 Test Result with 4mA Fault current: ....................................................................... 12
    5.2.6 Test Result with 5mA Fault current: ....................................................................... 13
    5.2.7 Test Result with 6mA Fault current: ....................................................................... 14
    5.2.8 Test Result with -6mA Fault current: ...................................................................... 14
    5.2.9 Tripping time with different waveforms: ................................................................. 15
    5.2.10 Threshold current with different waveforms: ......................................................... 17
6 SOFTWARE REQUIREMENT SPECIFICATIONS ................................................. 19

7 MICRO-CONTROLLER AND INTERFACE HARDWARE DESCRIPTION .......... 21

7.1 Stellaris family-LM3S9000 Series ................................................................. 21
7.2 Evaluation Kit EKK-LM3S9B92 ................................................................. 22
7.3 µVision IDE Keil ......................................................................................... 23
7.4 Microcontroller Peripheral ........................................................................ 23
  7.4.1 ADC ........................................................................................................ 23
  7.4.2 CAN ......................................................................................................... 24
  7.4.3 GPIO ........................................................................................................ 24

7.5 CAN module operation ................................................................................. 24
  7.5.1 CAN Specification .................................................................................. 24
  7.5.2 CAN Communication ............................................................................ 25
  7.5.3 CAN bus operation ................................................................................ 26

7.6 Flowchart of the protection device control logic ............................................. 27

8 MICROCONTROLLER CONTROL LOGIC ....................................................... 30

8.1 Input output signal description .................................................................... 32
8.2 CAN implementation and CAN MSG specifications ...................................... 33
8.3 Simulation of Protection device control logic ................................................ 35
  8.3.1 Simulink Representation ....................................................................... 35
  8.3.2 Protection device State machine implemented in Stateflow .................... 39
  8.3.3 MATLAB function “ChargerSim” ............................................................ 40
  8.3.4 MATLAB function “Can decode” ............................................................ 43
  8.3.5 MATLAB function “Can1 send decode” ................................................ 45

9 HARDWARE IMPLEMENTATION .................................................................... 47

9.1 Implementation Results ................................................................................ 49
  9.1.1 Wait state ............................................................................................... 50
  9.1.2 Precharging state ................................................................................... 50
  9.1.3 Charging state ....................................................................................... 50
  9.1.4 Charging complete state ....................................................................... 51
9.1.5 Charging fail state ................................................................................................. 51
9.1.6 RCD trip limit set point change state .................................................................. 51

10 CONCLUSIONS ........................................................................................................... 52
11 FUTURE WORK .......................................................................................................... 53
12 REFERENCES .............................................................................................................. 54
13 ENCLOSURES ............................................................................................................ 55
  13.1 Source code of Microcontroller logic implementation ......................................... 55
1 INTRODUCTION

1.1 Purpose

The purpose of this report is to gather all information necessary to develop simulations and experimental implementations of an On-board protection device for electric vehicle battery charger.

Developing of the EV is beneficial both for saving energy and environment purposes. As to increase the use of EV, there is a need to consider a high degree of safety and reliability of electrical system compliant with the standards "CEN-CENELEC" that can protect the human life and reduce the danger of fire-risk.

On board battery chargers are commonly available in both EV and plug-in hybrid EV. The chargers are supplied from a single phase AC or a three – phase supply. Due to the relatively high power levels that are involved during charging, safety and protection of this operation is very important.

1.2 Scope

This report describes the Protection device for the battery charger. The protection functions of the device, described as shown in Figure 1-1, are RCD and GFM device. The commands from protection functions are interfaced a microcontroller. Microcontroller incorporates main control logic part which is responsible for processing on data coming from RCD or GFM and sending necessary information through the CAN bus to the battery charger. At the same time, if it detects any fault, the microcontroller also initiates necessary actions to disconnect the Circuit Breaker (CB) to disconnect charger from supply, thus preventing any further damages.

Figure 1-1: Electrical schematic of the electric vehicle

Figure 1-1 describes the protection device and its main components. The block diagram of overall system, as shown in Figure 1-2, Figure 1-2 also describes the communication requirements which are to be fulfilled by the protection device.
1.3 Definitions

Table 1: Definitions of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV</td>
<td>Electrical Vehicle</td>
</tr>
<tr>
<td>CEN</td>
<td>European Committee for Standardization</td>
</tr>
<tr>
<td>CENELEC</td>
<td>European Committee for Electro technical Standardization</td>
</tr>
<tr>
<td>RCD</td>
<td>Residual Current Device</td>
</tr>
<tr>
<td>GFM</td>
<td>Ground Fault Monitoring</td>
</tr>
<tr>
<td>CB</td>
<td>Circuit Breaker</td>
</tr>
<tr>
<td>RCCB</td>
<td>Residual-Current Circuit Breaker</td>
</tr>
<tr>
<td>CT</td>
<td>Current Transformer</td>
</tr>
<tr>
<td>PD</td>
<td>Protection Device</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog-to-Digital Converter</td>
</tr>
</tbody>
</table>
1.4 **Structure**

This report has the following structure:

1. Introduction describes the purpose and scope for this report as well as terms, abbreviations and acronyms used.
2. Problem Description of the project.
3. The Project report is divided into two different parts:
   - Part A: Hardware part
     - a. Hardware requirement
     - b. Hardware design and simulation
     - c. Implementation and tests
     - d. Result from hardware part
   - Part B: Software part
     - a. Software requirement
     - b. Simulation and implementation
     - c. Hardware implementation
     - d. Result
4. Conclusion section
5. Future work
6. References (specify source material and further reading.)
7. Enclosures are supplementing information
2 PROBLEM DESCRIPTION

The main feature of the on-board battery charger is charging without using galvanic isolation so it reduces weight and price with higher efficiency. So in the event of any earth leaks, due to cables becoming damaged or water in the plugs etc. protection device is needed in order to trip the system or, in the events of short circuit, protection device will trip the unit to increase safety level of electrical equipment operation [1].

A protection device is working similar to a Residual-Current Circuit Breaker (RCCB). RCCB is an electrical wiring device that disconnects a circuit whenever it detects that the electric current is not balanced between the energized conductor and the return neutral conductor [2].

The protection device has to be design which includes basic fault detection hardware such as Ground fault monitoring, Residual current device and main microcontroller for interface to and from peripheral fault monitoring devices and external communication networks.

The design of this RCD is similar to Type B (for every kind of earth current) [13] with ultra-sensitive tripping level current of 6mA leakage [14] [15]. The Printed Circuit Board (PCB) should be designed and tested. The Pspice simulation has to carry out to check the experimental result.

The Control logic has to design with a microcontroller, which also communicates with input channels and control output signals to circuit breaker which connects charger unit with supply. The microcontroller also send periodically the necessary data and status information about the peripheral protection devices to battery charger and also initiates close or open commands to circuit breaker, if there is any fault. The control logic has to be simulated first in MATLAB using SIMULINK and STATEFLOW toolbox and then the actual implementation will be done on the Evaluation Kit EKK-LM3S9B92.
PART A
3 REQUIREMENTS

On board protection device is a protection system which is divided into two major parts, hardware and software. Hardware part includes the design of protection device that can detect the leakage current and ground fault.

3.1 Hardware Requirements

In order to fulfill the requirement of the protection device, hardware design includes the following parts:

3.1.1 Power Supply

Two different levels of voltages, +14V and -6V, are supposed power to feed to the protection device. For a microcontroller, in control logic part, the required power is 3.3V and all other comparators and amplifiers need ±6. A DC-DC converter is needed to convert 14V to 3.3V and +6V.

3.1.2 Ground Fault Monitoring

In order to increase safety and reliability for On-board charging process of an EV, a control circuit is needed to take into consideration. The Ground Fault detecting circuit is responsible for checking continuously proper connection between car chassis and ground.

3.1.3 Residual Current Monitoring

The main functionality of protection device is detecting the leakage current, 6mA<sub>dc</sub> and 30 mA<sub>ac</sub> with different waveforms. A residual current device is needed to detect the leakage current in different level of DC and AC.
4 SOLUTION

4.1 Hardware Design Overview

In order to meet the requirements of a protection device, the main features of the device should include measuring of the leakage current, check the ground connection, control the output signals and sending them through the CAN bus Figure 4-1. The schematic of suggested design over each part is attached in section 13.

![Figure 4-1 Block diagram of Hardware Structure](image-url)
### 4.2 Technical Specification

Technical specification of the final prototype is gathered in the following table.

**Table 2: Technical Specification**

<table>
<thead>
<tr>
<th>Category</th>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microprocessor</td>
<td>STM32F101C4 [3]</td>
<td>- ARM 32-bit Cortex™-M3 CPU with 16 or 32 KB Flash</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 36 MHz maximum Frequency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Low power</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 1 x 12-bit, 1 μs A/D converter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Up to 4 Communication interfaces</td>
</tr>
<tr>
<td>Microprocessor Supervisory Circuit</td>
<td>ADM 803 [4]</td>
<td>- Specified over Temperature</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Low Power Consumption</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Precision Voltage Monitor: 2.5 V, 3 V, 3.3 V, 5 V Options</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Reset Assertion Down to 1V VCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 140 ms Min Power-On Reset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Open-Drain RESET Output</td>
</tr>
<tr>
<td>3.3-V CAN TRANSCEIVERS</td>
<td>SN65HVD232 [5]</td>
<td>- Operates With a 3.3-V Supply</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Thermal shutdown Protection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Bus/Pin ESD Protection Exceeds 16 kV HBM</td>
</tr>
<tr>
<td>Dual Step-Down Regulator</td>
<td>LT3509EMSE[6]</td>
<td>- Two 700mA Switching Regulators with Internal Power Switches</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Wide 3.6V to 36V Operating Range</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Over-Voltage Lockout Protects Circuit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Short Circuit Robust</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Low Dropout Voltage (95% Maximum Duty Cycle)</td>
</tr>
<tr>
<td>Low power J-FET dual operational</td>
<td>TL 062 [7]</td>
<td>- Very low power consumption: 200uA</td>
</tr>
<tr>
<td>amplifiers</td>
<td></td>
<td>- Wide common mode and differential voltage ranges</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Low input bias and offset currents</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Output short-circuit protection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- High input impedance J-FET input stage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Internal frequency compensation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Latch up free operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- High slew rate: 3.5v/ms</td>
</tr>
<tr>
<td>Low power quad operational</td>
<td>LM324[8]</td>
<td>- Wide gain bandwidth: 1.3MHz</td>
</tr>
<tr>
<td>amplifiers</td>
<td></td>
<td>- Input common-mode voltage range includes ground</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Large voltage gain: 100dB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Very low supply current/ampli: 375mA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Low input bias current: 20 nA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Low input offset voltage: 5 mV max.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Low input offset current: 2nA</td>
</tr>
</tbody>
</table>
| **Precision Rail-to-Rail Comparator** | **LT1716 [9]** | **Wide power supply range:**
| | | Single supply: +3V to +30V
| | | Dual supplies: ±1.5V to ±15V
| **Current Transformer** | **CT** | **Operates from 2.7V to 44V**
| | | Micro power: 35μA IQ
| | | Offset Voltage: 1.5mV Max
| | | Valid Output with Either Input 5V Below V– Rail-to-Rail Output Swing
| | | Output Can Drive Loads Above V+
| | | Internal Pull-Up Current
| | | –40°C to 125°C Operating Temperature Range
| | | N1=3 and N2=250 Turns
5 HARDWARE RESULTS

The results of simulation and test the designed circuit are shown in following figures.

5.1 Simulation Result

The simulation of circuit, with different fault current 1, 2, 3, 4, 5 and 6 mA dc, is achieved by Pspice and is shown in Figure 5-1.

![Simulation Result of Different Fault current](image)

**Figure 5-1: Simulation Result of Different Fault current**

As is shown in Figure 5-1 when there is no fault offset voltage is about 1.4V. With increasing the leakage current output voltage increases. The maximum value reached by 6mA leakage current is almost 3.7V.

5.2 Implementation Results

In the laboratory, designed circuit is tested and following result is recorded by oscilloscope with different fault currents.

5.2.1 Test Result without Fault current

As is shown in the figures, channel 1 shows the voltage after first amplifying, channel 2 shows the second inverted amplified voltage and channel 3 is the output voltage. When there is not any fault current output voltage is 1.12V.
5.2.2 Test Result with 1mA Fault current

With 1mA fault current output voltage is 1.28V.

5.2.3 Test Result with 2mA Fault current

Output voltage is 1.76V when the fault current increases to 2mA.
5.2.4 Test Result with 3mA Fault current

The output voltage reaches 2.40V when the fault current is 3mA.

5.2.5 Test Result with 4mA Fault current:

With 4mA fault current, output voltage is 3.12V.
5.2.6 Test Result with 5mA Fault current:
The output voltage increases to 3.76V when the current reaches 5mA.
5.2.7 **Test Result with 6mA Fault current:**

The tripping voltage that is equal to 6mA is 4.40V.

![Figure 5-8: Test Result with 6 mA dc Fault Current](image1)

5.2.8 **Test Result with -6mA Fault current:**

All the test situations are repeated with minus current value. The tripping voltage values show almost the same as the plus fault current values. Figure 5-9 shows the tripping value with -6mA.

![Figure 5-9: Test Result with -6 mA dc Fault Current](image2)
5.2.9 Tripping time with different waveforms:

Tripping time of the On-board protection device in different frequencies is measured and recorded. All recorded results are shown in the following figures.

A. Tripping time with pulse waveform in low frequency (f=6.329Hz):

![Tripping time with pulse waveform in low frequency](image)

Figure 5-10: Tripping time with pulse waveform in low frequency

B. Tripping time with pulse waveform in middle frequency (f=19.23Hz):
C. Tripping time with pulse waveform in higher frequency ($f=38.46\,Hz$):

Figure 5-12: Tripping time with pulse waveform in higher frequency

D. Tripping time with AC waveform at $30\,ma_{ac}$:
The result of tripping time with AC waveform is shown in the next figure.

Figure 5-13: Tripping time with AC waveform at 30ma_{ac}

5.2.10 Threshold current with different waveforms:
All threshold current results with different waveforms are gathered in table which is shown below.

Table 3: Triggering Threshold with different waveforms

<table>
<thead>
<tr>
<th>Waveform</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>5.8 mA</td>
</tr>
<tr>
<td>AC</td>
<td>15.7 mA</td>
</tr>
<tr>
<td>0°</td>
<td>7.1 mA</td>
</tr>
<tr>
<td>180°</td>
<td>8.1 mA</td>
</tr>
<tr>
<td>90°</td>
<td>8.6 mA</td>
</tr>
<tr>
<td>270°</td>
<td>9.6 mA</td>
</tr>
<tr>
<td>135°</td>
<td>9.3 mA</td>
</tr>
<tr>
<td>315°</td>
<td>11.3 mA</td>
</tr>
</tbody>
</table>
PART B
6 SOFTWARE REQUIREMENT SPECIFICATIONS

On board protection device is a protection system which is divided into two major parts, hardware and software. Details about different fault sensors like RCD, Ground monitor device etc are described in Part I. This part or the report is deal with the microcontroller hardware, different functions associated with it and the implementation of the control logic for protection device..

The functional requirement for controller are mainly the design and implementation of the control logic related to communications of status signals from peripheral protection functions and CAN bus communications to the charger unit. Figure 6-1 describes the overall Protection device and Charger system along with different signal flow paths. The status signals from Ground monitor device and Residual current device are input to the coming to the microcontroller unit of the Protection device. The Protection device will check these inputs, if there is any fault detected by the peripheral devices, microcontroller will not send "close CB" command to circuit breaker, preventing the charger unit to start charging process, otherwise it will close the circuit breaker and send corresponding message to Charger about no fault detected by peripheral protection devices and starting the charging process. The Protection device communicates continuously with peripheral devices to check for faults based on the input from the devices GMD and RCD, and send corresponding message to Charger about the status (Fault or No fault). The microcontroller unit also receives messages from charger through the CAN bus. The different messages being received through CAN bus are operation commands like charging process start, stop commands, leakage current set points, etc, as shown in Figure 6-1. All these message communication between Protection device and Charger is done using CAN, the details of CAN bus communications are described in details in section 7.4.2.

![Figure 6-1: System block diagram](image)

The microcontroller functions are thus summarized as below:

- Configuration of ADC channels to interface with analog input signals from RCD hardware like residual leakage current detector- both AC and DC currents.
- Interface GPIO ports- CB close/open command output and CB status monitoring, Status monitoring from Ground fault monitor device.
- CAN bus communications with Charger unit. Configuration of CAN bus to communicate various messages with charger for communication.
- control logic including Intelligent state machine to control CB based on status of peripheral fault sensing devices as well as commands received from charger

**Figure 6-2: Software process flow**

Part II of report is based on the above aspects of control functions, the different chapters in Part II are organized in following fashion.

- Chapter 7 deals with selection of microcontroller with required functionalities and brief description of selected microcontroller.
- Chapter 8 presents the overall control logic, various CAN message specifications. The simulation of Protection device logic using Simulink MATLAB is also presented in this chapter.
- Chapter 9 describes the actual implementation of the control logic on the prototype microcontroller board and presents the sample results.
7 MICRO-CONTROLLER AND INTERFACE HARDWARE DESCRIPTION

The basic requirements for the selected microcontroller are listed as:
- It should have low supply voltage range (1.8V to 3.6V).
- It should be low cost.
- It should have internal CAN peripheral.
- It can be used for serial onboard programming, no external power supplies needed for programming.
- It should be able to use for an automotive application.

As described above, the microcontroller should have certain functionality within. The selection of the microcontroller is based on considering all the requirements and design. The requirements and some advanced features of microcontroller lead to selection of the Stellaris family from TI, i.e. Stellaris ARM cortex-M3 microcontrollers. The Stellaris family combines sophisticated, flexible, mixed signal system on chip integration. It provides unparalleled real time multitasking capabilities. Also it is cost effective and simple to program.

As considering the requirements, the Microcontroller should have CAN peripheral within, and Stellaris microcontroller from family series 9000 provides this feature with some more advanced features. The microcontroller LM3S9B92 has some advanced features which are not required for our project. The main requirement from microcontroller is CAN and ADC functionalities.

7.1 Stellaris family-LM3S9000 Series

![Stellaris family block diagram]  
Figure 7-1: Stellaris family block diagram
Texas Instruments’ LM3S9000 Series features an on-chip combination of 10/100 Ethernet MAC/PHY, USB On-The-Go/Host/Device, and Controller Area Network (CAN). In addition to several product enhancements, the LM3S9000 Series adds new features, such as a versatile External Peripheral Interface (EPI) with modes to support SDRAM, SRAM/Flash, Host-Bus, and M2M, an Integrated Interchip Sound (I2S) interface, simultaneous dual ADC capability, a second watchdog timer with independent clock for safety critical applications.

7.2 Evaluation Kit EKK-LM3S9B92

![Evaluation board LM3S9B92 and In-circuit debug interface board](image)

The EKK-LM3S9B92 evaluation kit is used for first prototype. This kit has two boards Evaluation board and In-Circuit debug interface (BD-ICDI) board.

Features of evaluation board:

- It has LM3S9B92 high performance Stellaris microcontroller and large memory
  - 32 bit ARM Cortex-M3 core
  - 256 KB single cycle Flash memory
  - 96 KB single cycle RAM
- It also has a 10/100 Mbit Ethernet port.
- A full speed USB-OTG port
- Virtual serial communications port capability
- Detachable ICDI board
- Easy to customize
7.3 µVision IDE Keil

µVision IDE from Keil is used for the source code editing and program debugging. This IDE is provided with the Evaluation kit. This µVision IDE by Keil is easy to use for creating embedded programs that works. The µVision editor and debugger are integrated in a single application that provides a seamless embedded project development environment.

7.4 Microcontroller Peripheral

The main peripherals of Stellaris LM3S9B92 microcontroller are as shown in Figure 7-1. Only the peripherals on interest to the project specifications are described in this section, these are ADC channels and related interface, CAN bus, General purpose I/O (GPIO)

7.4.1 ADC

An analog-to-digital converter (ADC) is a peripheral that converts a continuous analog voltage to a discrete digital number. The Stellaris LM3S9B92 microcontroller has two identical converter modules. These ADC modules support 10-bit conversion resolution, 16 input channels and an internal temperature sensor. Each module contains 4 programmable sequencers allowing the sampling of multiple analog input sources without controller intervention [10]. Four sampling sequencers, has configurable trigger events which can be captured. The first sequencer captures up to eight samples, second and third captures up to four samples and the fourth sequencer captures single sample. Each sample can be the same channel, different channels or any combinations in any order. (For additional detail refer [10]&[11])
7.4.2 CAN

Controller Area Network (CAN) is multicast, shared serial bus for connecting electronic control unit (ECU’s). Stellaris LM3S9B92 microcontroller includes two CAN units with following features,

- CAN protocol version 2.0 part A/B
- Bit rates up to 1 Mbps
- 32 message objects with individual identifier mask
- Programmable FIFO mode enables storage of multiple message objects
- Gluelessly attaches to an external CAN transceiver through the CANnTX and CANnRX signals

The Stellaris CAN module provides hardware processing of the CAN data link layer. It can be configured with message filters and preloaded message data so that it can autonomously send and receive messages on the bus and notify the application accordingly. It automatically handles generation and checking CRCs, error processing and retransmission of CAN messages [11].

These CAN modules can be configured with message filters and preloaded message data so that they can autonomously send and receive messages on the CAN bus and notify the application accordingly. The message objects are stored in the CAN controller and provide the main interface for the CAN module on the CAN bus. The Message object can be programmed to handle a separate message ID or can be chained together for a sequence of frames with the same ID. The message identifier filters provide masking that can be programmed to match any or all of the message ID bits and frame types. (For additional detail refer [10]& [11]).

7.4.3 GPIO

The GPIO module of Stellaris LM3S9B92 microcontroller is consist of nine physical GPIO blocks, each corresponding to an individual GPIO port (Port A, Port B, Port C, Port D, Port E, Port F, Port G, Port H, Port J). The GPIO supports up 65 programmable input/output pins, depending on the peripheral being used [10] & [11].

The GPIO pin can configure to be GPIO or peripheral pin. Not all pins on all part have peripheral function; in this case pin can be only used as a GPIO [10]& [11]. The GPIO pins are specified using bit packed byte, where each bit that is set identifies the pin to be accessed. Bit 0 represents GPIO port pin 0, bit1 represents GPIO port pin 1, bit 2 represents GPIO port pin 2 and so on[11].(For additional detail refer [10]&[11]).

7.5 CAN module operation

7.5.1 CAN Specification

The CAN messages, from Charger or from Protection device are the message with some specific data as well as Identification (Message ID). For some specific message these CAN message ID’s are assigned. Message transfers that includes data, remote, error and overload frames with an 11 bit identifier (standard) or 29-bit identifier (extended) are supported. The CAN data frame contains data for transmission. The remote frame does not contain any data; it is used for requesting the transmission of specific message object. The data frame is constructed as shown in following Figure 7-4 : CAN data/remote Figure 7-4 [10].
### 7.5.2 CAN Communication

CAN is a multicast, shared serial bus standard for connecting electronic control units (ECU's). CAN was specifically designed to be robust in electromagnetically-noisy environment. It's created originally for automotive purposes, but also used in many embedded control applications. For the network lengths of less than 40 it can provide bit rate up to 1Mbps.

The Stellaris CAN controller conforms to the CAN protocol version 2.0 (parts A and B). It has bit rate up to 1Mbps. It has 32 message objects with individual identifier mask. And it gluelessly attaches to the an external CAN transceiver through the CANnRX and CANnTX.

The Stellaris LM3S9B92 has two CAN modules, each CAN module consist of three major parts:

1. CAN protocol controller and message handler
2. Message memory
3. CAN register interface

The Protocol controller transfers and receives the serial data from CAN bus and passes the data onto the message handler. The message handler loads this data into appropriate message object. The message handler is also responsible for generating interrupts based on events on the CAN bus.

The message memory is set of 32 identical memory blocks that hold the current configuration, status, and actual data for each message object. These memory blocks are accessed via either of the CAN message object register interfaces.

The Stellaris CAN controller provides an interface to communicate with the message memory via two CAN interface register sets for communicating with the message objects, as the message memory is not directly accessible. These two interfaces must be used to read or write to each message object [10]. The two message object interface allows parallel access to the CAN controller message objects when multiple objects may have new information that must be processed. Here one interface is used for transmit data and one for receive data.

---

**Figure 7-4: CAN data/remote frame [10]**

<table>
<thead>
<tr>
<th>Bus Idle</th>
<th>Message Delimiter</th>
<th>Remote Transmission Request</th>
<th>Control Field</th>
<th>Data Field</th>
<th>CRC Sequence</th>
<th>ACK</th>
<th>BOP</th>
<th>IFS</th>
<th>Bus idle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>11 or 20</td>
<td>1</td>
<td>6</td>
<td>0…64</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit Stuffing</th>
<th>CRC Sequence</th>
<th>CRC Field</th>
<th>End of Frame Field</th>
<th>Interframe Field</th>
</tr>
</thead>
</table>

**CAN Data Frame**

---
### 7.5.3 CAN bus operation

Mainly CAN bus operation includes the transmitting and receiving the DATA to and from the CAN bus.

**Sequence of transmitting data on CAN bus [12]:**

- If the CAN bus is idle, any node can start sending CAN frame.
- All other node will receive it.
- There is priority process, which will decide the priority of the node. The node with highest priority will transmit first. The nodes with less priority will stop transmitting and start receiving.
- After transmitting the frame the node will wait for one bit time for the acknowledgement (ACK) bit to be set, which is the indication of that some node has received it.
- If it the acknowledgment (ACK) bit is not set then, this node will resend the fame as soon as possible.
- Now all nodes again start to send their CAN far over the CAN bus.

**Sequence of receiving data on CAN bus [12]:**

- All nodes except those currently transmitting frames are in listening mode.
- The CAN frame is transmitted as described above.
- The frame is received by all other nodes, if deemed to be a valid CAN message an acknowledgment (ACK) bit is set.
- The frame is then sent through controller’s acceptance filter mechanism. If this frame is rejected it will get discarded otherwise it is sent to controller FIFO memory. If the FIFO is full the old frame is lost.
- The host processor gets alerted to the fact that a valid frame is ready to be read from FIFO. This frame must be read as soon as possible.
- The host processor takes decision about what to do with message as determined by software.

Following are the steps for CAN operation from defining till sending the message [10].

1. Define the CAN message object.
2. Configure GPIO port to utilize its CAN function.
3. Initialize CAN controller.
4. Set up bit rate for the CAN bus.
5. Enable interrupts on the CAN peripheral.
6. Enable CAN interrupt on the Processor.
7. Enable CAN for Operation.
8. Initialize the CAN message object for sending and receiving messages.
9. Load the message object into the CAN peripheral, once loaded the CAN will receive any message on the bus and an interrupt will occur.
10. Check for the logical condition and set the CAN message ID.
11. Send the CAN message over the CAN bus.

Steps 1-7 are executed during the initialization phase of the microcontroller logic, whereas steps 9-11 are repeated in a loop of the main control logic.
7.6 **Flowchart of the protection device control logic**

Based on the software requirement specification as described in chapter 6 and the microcontroller and its peripherals described in section 7.1 and 7.4, the flowchart of the different decision making process in microcontroller is prepared which is describes in this section.

The overall control logic will have following sections

1. **Initialization section**

   The initialization section includes the declaration and initialization of different variables, enabling the peripherals followed by the setting the GPIO port and pins for the required operation like ADC, CAN or just as GPIO input or output.

2. **Main Control Logic section**

   The main control logic includes, reading the input values and depending on these input values design the logic. In detail it follows some steps as,

   a. Receiving and decoding of CAN messages.
   b. Check for the faults from the peripheral devices- Residual current device and Ground monitor device.
   c. Setting control logic flags accordingly.
   d. Take actions specific to closing/opening of CB
   e. Encoding CAN messages and sending messages over CAN

The initialization part is executed at the start of the program only, afterwards the main program executed in the loop.
Figure 7-5: Main program structure
Control Logic

Receive & Decoding CAN bus MSG’s

Check Ground fault, Earth leakager current fault,

If Fault?

Yes

Setting Fault flags

No

GPIO actions

CB, Status display

Send CAN bus MSG’s

PD status, Fault flags

Figure 7-6: Flowchart of main control logic
8 MICROCONTROLLER CONTROL LOGIC

The software task of the project is to implement the software as per the system requirement specifications described in chapter 6, which are also elaborated by the flow chart presented in section 7.6. The performance testing and verification of the control logic needs that all peripherals are present and communicate with the microcontroller. Another requirement is that the CAN network is available and the CAN MSGs from charger are available. Thus to design the control logic for protection device, a dedicated charger hardware (CAN communications part) is required for the signal (messages) communication. As during the project stage, it is practically not possible to get the charger for checking the control logic of protection device, the following alternate system, as shown in Figure 8-1, is proposed, which is different from the original system requirement specifications.

Figure 8-1: System proposed using two CAN modules

1. The Stellaris LM3S9B92 microcontroller has two CAN modules, using these two CAN modules, the system with Protection device and Charger is simulated. The same microcontroller now also performs operation of CAN signal generation of charger.

2. The chargers states, which are transmitted over CAN network, are generated using two digital input signals S1 and S2. The microcontroller thus incorporates a simple decode logic which generates the charger states based on digital I/O status.

3. The 0 or 5 V is fed to the GMD status input to simulate the GMD status.

4. A resistor-switch (two pole switch) pair is used to generate 0 or 5 V depending upon position of the switch. A number of these networks are used to generate the different status signal needed for logic simulation. A LED is connected to the input pins to indicate status of input signals as shown in Figure 8-2.
5. An analog voltage reference with variation between 0-3V used as earth leakage current input. This is done since the actual RCD hardware was not ready at the time of microcontroller implementation. But the RCD input can be directly connected to the same port with proper scaling as shown in Figure 8-3. The example values shown in Figure 8-3 are with the setting of 7mA RCD current input, which is converted equivalent voltage of 2.94 V, the ADC input voltage range is between 0-3V.

6. The output from the control logic - PD status and charging status is interfaced with two digital i/Os configured as output- S5 and S8 as shown in Figure 8-2. LEDs are connected at the digital input and outputs for visual information of various digital inputs and output.

**Figure 8-3: ADC interface for actual leakage current signal from RCD**

The later part of this chapter is arranged as following,

- First, the various digital input and output signals are described in section 8.1.
- Then the CAN specifications are described in section 8.2 with actual CAN message IDs and DATA for various conditions, the specifications include the CAN MSGs from both charger and protection device.
Section 8.3 presents the simulation of the complete logic along with the simulation results. The complete logic is simulated on MATLAB-SIMULINK software platform whereas the state machine is implemented using STATEFLOW toolbox.

Finally the actual implementation of the logic on the Stellaris EKK-LM3S9B92 evaluation board is described in section 0.

8.1 Input output signal description

The table below lists the various digital input and output signals used in the implementation. As described in above section, the digital status signals from various peripherals are simulated by 2 pole switches which can connect to either 5V or GND, thus generating either 1 or 0 state (Figure 8-2) Similarly, analog voltage input equivalent to earth leakage current is simulated by using a potentiometer with variable voltage from 0 to 3 V (maximum allowable input voltage for ADC channels is 3V). The output signals from the microcontroller are connected to LEDs for status indication. Table 4 describes the details about these signals along with the port numbers on evaluation board where these signals are connected.

**Table 4: Input signal table**

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>From</th>
<th>To</th>
<th>Type</th>
<th>Port-pin number</th>
<th>Switch/ADC input</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Earth_leakage current_val</td>
<td>Residual current device</td>
<td>Protection device</td>
<td>Analog</td>
<td>E2</td>
<td>ADC1</td>
<td>0-3 V</td>
<td>Earth_leakage_current_val&lt;6mA or 30mA: No Fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Earth_leakage_current_val&gt;6mA or 30mA: Fault</td>
</tr>
<tr>
<td>GMD_status</td>
<td>Ground Monitor Device</td>
<td>Protection device</td>
<td>Digital</td>
<td>F4</td>
<td>S6</td>
<td>0 to 1</td>
<td>GMD_status = 0 : No Fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GMD_status=1: Ground Fault</td>
</tr>
<tr>
<td>CB_status</td>
<td>Circuit Breaker</td>
<td>Protection device</td>
<td>Digital</td>
<td>B6</td>
<td>S7</td>
<td>0 to 1</td>
<td>CB_Status=0 : Open Circuit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CB_Status=1 : Close Circuit</td>
</tr>
<tr>
<td>Charger status 1</td>
<td>Charger</td>
<td>Protection device</td>
<td>Digital</td>
<td>E0</td>
<td>S1</td>
<td>0 to 1</td>
<td>Combination of two status signals “S2S1” are used to generate 4 charger</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>status states- “00”-charging wait, “01”-charging start, “10”- charging,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>“11”-charging complete</td>
</tr>
<tr>
<td>Charger status 2</td>
<td>Charger</td>
<td>Protection device</td>
<td>Digital</td>
<td>E1</td>
<td>S2</td>
<td>0 to 1</td>
<td></td>
</tr>
<tr>
<td>Set point enable</td>
<td>Charger</td>
<td>Protection device</td>
<td>Digital</td>
<td>E6</td>
<td>S3</td>
<td>0 to 1</td>
<td>0: No Change</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1: Change the current trip level Set</td>
</tr>
</tbody>
</table>
Table 5: Output signal table

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>From</th>
<th>To</th>
<th>Type</th>
<th>Port</th>
<th>LED</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
</table>
| PD_status         | Protection device | Charger          | Digital| E4   | LED1 | 0 to 1 | 0: No fault continue charging  
|                   |                 |                  |        |      |      | 1: Fault detected by PD, stop charging          |
| Command_to_CB     | Protection device | Circuit breaker  | Digital| E5   | LED2 | 0 to 1 | 0: Open Circuit  
|                   |                 |                  |        |      |      | 1: Close Circuit                                |

8.2 CAN implementation and CAN MSG specifications

As described above and shown in Figure 8-1, the protection device control logic as well as charger CAN bus messages are implemented using the two CAN modules in Stellaris LM3S9B92 evaluation board.

CAN bus module CAN0 is configured to send Charger messages, these messages are generated using combination of the digital inputs as described in Table 4. Similarly, the generation of CAN bus messages to be send from Charger are also generated first at the beginning of main control logic and are sent using CAN0 module. CAN1 module is considered as Protection device. Protection device part of the logic receives the information from charger by decoding the CAN messages which are received by CAN1. CAN1 also transmits the information sent from protection device using the CAN1 module.

CAN bus messages have unique IDs and data for sending different signal information between charger and protection device. These signal information are listed in Table 6.

The Process will start with the CAN message for closing the circuit breaker from Charger to Protection device. The Protection device will then check for occurrence of fault in any of the devices (GMD and RCD). If GMD fault is there it will send message of immediate shutdown to the Charger. If RCD fault is arise, it will send the immediate shutdown message as well as it will send the current value of “fault current” to the Charger. If there is no fault then it will send the message of status of Protection device to the Charger. In between the Charger will send the message about its status. When the process of charging is over, The Charger will send the message as charging complete to the Protection device and the Protection device will open the circuit breaker. If there is change in the trip level value of current then the Charger will send the message with new set point values of fault current for both AC and DC. Following table summarizes these CAN messages referring to

Table 6: CAN messages

<table>
<thead>
<tr>
<th>CAN messages from Charger</th>
<th>CAN messages from Protection device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close the circuit breaker</td>
<td>Protection device status</td>
</tr>
<tr>
<td>Charger status</td>
<td>Immediate shutdown</td>
</tr>
<tr>
<td>Charging complete</td>
<td>Current value of fault current</td>
</tr>
<tr>
<td>Fault current set point value</td>
<td></td>
</tr>
</tbody>
</table>
Following Table 7 describes the Message ID and the data it contains with interpretation as well as description. The message contains 8 bytes of data. The first table summarizes the CAN messages send from protection device to the Charger and Table 8 lists the collective messages coming from Charger to the Protection device with MSG IDs and its data content.

**Table 7: Messages from protection device to Charger**

<table>
<thead>
<tr>
<th>MSGID</th>
<th>Interpretation</th>
<th>Data byte0</th>
<th>Data byte1</th>
<th>Data byte2</th>
<th>Data byte3</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0FF</td>
<td>Immediate shutdown</td>
<td>0x5C</td>
<td></td>
<td></td>
<td></td>
<td>Current below threshold</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0X01</td>
<td></td>
<td></td>
<td></td>
<td>AC fault current detected</td>
</tr>
<tr>
<td></td>
<td>any other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DC fault current detected</td>
</tr>
<tr>
<td>0x120</td>
<td>Current value of fault current</td>
<td>AC</td>
<td>DC</td>
<td></td>
<td></td>
<td>As read by ADC input channels</td>
</tr>
<tr>
<td>0x131</td>
<td>status of PD for charger</td>
<td>state</td>
<td></td>
<td></td>
<td></td>
<td>PD State machine status</td>
</tr>
</tbody>
</table>

In the present implementation, the bytes 1 to 3 are left default for MSG id 0x0FF. Similarly since only one ADC input is used in present implementation which is treated as AC current input, the DC current input value in byte 2 and 3 for MSG id 0x120 is left default. Similarly byte 1 to 3 for MSG id 0x131 is unused. The priority of different messages is as below

- The MSG 0x131 is transmitted continuously in normal operation.
- In the event of fault, the highest priority MSG 0x0FF is transmitted once (this message has information about trip action because of fault detected by protection device).
- This MSG is followed by 0x120, which contains the information about fault current values.
- After this, again message 0x0131 is transmitted continuously as before.

**Table 8 : Messages from Charger to protection device**

<table>
<thead>
<tr>
<th>MSG ID</th>
<th>Interpretation</th>
<th>Data byte 0</th>
<th>Data byte 1</th>
<th>Data byte 2,3</th>
<th>Data byte 4,5</th>
<th>Data byte 6,7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x130</td>
<td>charger status for PD</td>
<td>0x00</td>
<td>disconnected from power grid</td>
<td>value of compensated fault current</td>
<td>value of RMS current drawn from</td>
<td>value of RMS voltage of power</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x01</td>
<td>connected to grid - inrush</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x02</td>
<td>connected to</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In the present implementation, the bytes 1 to 7 are left default for MSG id 0xFF; this is done since actual charger is not present. Instead, as explained earlier in Table 4, combination of S1 and S2 switches is used to generate one of four charger states being transmitted in byte 0 of MSG id 0xFF. Similarly since only one ADC input is used in present implementation, only one Set point value (AC in byte 0 and 1) is used in MSG id 0x320. The priority of different messages is as below:

- The MSG 0x130 is transmitted continuously in normal operation.
- In the event of fault current value setting, MSG ox320 is transmitted. (this message has information about AC fault current trip level)
- After this again message 0x0130 is transmitted continuously as before.

8.3 Simulation of Protection device control logic

The control logic shown in described in Figure 8-1 is simulated using the MATLAB. The main logic is simulated in Simulink environment whereas the main control state machine is simulated using the Stateflow toolbox.

8.3.1 Simulink Representation

Figure 8-4 shows the main block diagram of the overall control logic in Simulink. The Protection device module is consist of Stateflow module as control logic, an embedded MATLAB function block is used as a decoder block, which will decode the CAN messages coming from the Charger and Similarly other embedded MATLAB function block is used for encoding the CAN messages from messages from Protection device which are transmitted to the Charger.

The various inputs to the control logic are simulated as equivalent low or high inputs. These include status signals from Ground fault monitoring device, circuit breaker, etc. The analog input signal like RCD current and set point value are generated using slider gain blocks in Simulink. A slider gain is used to provide input through this signal as it is representing threshold value which is varying from 0mA to 6mA. The digital output signals are connected to scope and displays for monitoring. Stateflow used to show the control logic part in the Simulink. Similar to the actual implementation, the simulation also include encode and decode logic blocks associated with CAN bus. The CAN bus messages can be viewed using scopes connected to CAN bus. The encoder and decoder logic for CAN 0 and CAN1 module is implemented using simple MATLAB function blocks.

The protection device gets commands from Charger through CAN bus to start the charging process as well as some other commands as described in Table 8. The protection device will also receive input signals from the Ground monitor device and Earth
leakage current device and will send output commands to circuit breaker. These outputs from control logic are encoded and send to charger through CAN 1 module, these specifications are described in Table 7.

Figure 8-4 Simulink representation of Protection device and Charger

The Charger part of simulation is again shown in Figure 8-5; this block has following input signals,

- Charger status
- Set point change
- Earth leakage current trip level set point value

Figure 8-5: Simulink blocks related to charger section

An embedded MATLAB function "Charger Sim" is used for Charger CAN message encoding, with three input signals- two digital and one analog. First input signal is a four state signal emulated by multiport switch is used for sending the “Charger status”. The multiport switch contains following switch state (refer to Table 8)
Table 9: Charger signal description

<table>
<thead>
<tr>
<th>Charger Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Disconnected from grid</td>
</tr>
<tr>
<td>2</td>
<td>Connected to grid (inrush)-interpreted as charging start</td>
</tr>
<tr>
<td>3</td>
<td>Connected to grid (charging)</td>
</tr>
<tr>
<td>4</td>
<td>Connected to grid (not charging) - interpreted as charging complete</td>
</tr>
</tbody>
</table>

Second digital input signal in the Charger module is to command Earth leakage current trip level set point value. The set point value is input from input 3 to the charger encode logic. The output from this module will be CAN message ID and data as per CAN specification (refer to Table 8).

The Protection device module has following input signals as shown in Figure 8-6,

- Power supply status
- Earth leakage current val (RCD)
- Ground monitor device (GMD)
- Circuit breaker status
- Reset
- Circuit breaker close request
- Charging complete
- Setpoint change command
- New current trip level input
Figure 8-6: Simulink representation of protection device part

All inputs are digital inputs except two are analog input signals, one is from RCD and other is new current trip level input. There are two output signals from Protection device:

- Command to circuit breaker
- Protection device status

All these signals are summarize in the section 8.1 (Table 4 and Table 5),

Table 10: Default States of input switches in protection device

<table>
<thead>
<tr>
<th>Input switch</th>
<th>Status [Off(0) or On(1)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>power supply status</td>
<td>On</td>
</tr>
<tr>
<td>Ground monitor</td>
<td>Off</td>
</tr>
<tr>
<td>Circuit breaker</td>
<td>Off</td>
</tr>
<tr>
<td>Reset</td>
<td>Off</td>
</tr>
</tbody>
</table>

Before starting the simulation the position for every input switch to the protection device is as mentioned in Table 10. The Earth_leakage_current_val input is an analog input; slider gain is used for this input. The value of slider gain is ranges from 0 to 7 considering the requirement from the RCD. Reading greater the threshold value will generate a fault. In the beginning of the simulation process the position of this slider gain should be below 5.

The simulation process follows following sequence:

- The charger reads the input signals and generate the CAN messages to be transmitted over the CAN bus from CAN0 module.
- On the arrival of the CAN message from the charger, the "CAN decode" logic decodes the CAN message and determines the status information sent by charger,
this include the status of signals like CB close request, Charging complete, Setpoint, New current trip level set point value.

- The control logic implemented in Stateflow evaluates the current state based on the status of different input signals. The Stateflow logic is described in next section
- The output from Stateflow is also input to "CAN1 send encode" generates the CAN messages to be sent from protection device to charger as mentioned in the Table 7 of section 8.2.

**8.3.2 Protection device State machine implemented in Stateflow**

The protection device interprets all input signals coming from peripheral devices as well as CAN bus using the state machine. The state machine includes following states

1. Wait
2. Precharging
3. Charging
4. Charging complete
5. Charging Failure
6. Current set point

The Figure 8-7 below shows different states as well as different actions for transition from one state to another.

**Figure 8-7: signal flow of protection device**

1. **Wait state**: The protection device will be in the wait state at the start of simulation. On arrival of CB close request from the charger, it will transit to next state.
2. **Precharging state:** In this state the control logic will check the inputs from power supply status (power supply is on) and circuit breaker (open). Also it performs checks on the input devices connected to it, like residual current device, ground monitor device if there is “no fault” then it will go to next state “charging state” otherwise it will go to the “charging fail” state.

3. **Charging state:** In this state, command to close circuit breaker is initiated, and the charging process will start. The status of protection device will be send to the charger. While being in this state the fault checks are also performed continuously. If any of the devices indicates occurrence of fault, then immediate state transition from this state to the “charging fail” state is initiated. Otherwise the state machine remains in this state until “charging complete” signal arrives from charger.

4. **Charging complete:** In this state, command to open the circuit breaker will be send. And the control will return to the wait state again.

5. **Charging fail:** If the fault is detected in “precharging” or “charging” state, the control will transit to this state. In this state, first command to open the circuit breaker will be send as well as fault status message will also be send to the charger to indicate the fault occurrences in the devices. After removal of the faults (through “reset” input) the control will set back to the wait state.

6. **Current set point:** When the charger will send message to the protection device about the change in the fault current set point value from the wait state the control will transit to this state. After setting the new value for the fault current. The control will transit back to the wait state. The transition to this state is only possible from “wait” state, thus it is not possible to change the current trip set point during charging process in ongoing.

Following Table 11 summarizes the states and their transitions,

**Table 11: Different states and transitions in the main control logic**

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Transition state(no fault)</th>
<th>Transition state(fault)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait</td>
<td>Arrival of “CB_close_request” from charger</td>
<td>Precharging</td>
<td></td>
</tr>
<tr>
<td>Precharging</td>
<td>Checking for faults(Ground fault or leakage current fault)</td>
<td>Charging</td>
<td>Failure</td>
</tr>
<tr>
<td>Charging</td>
<td>Checking for faults(Ground fault or leakage current fault)</td>
<td>Completion</td>
<td>Failure</td>
</tr>
<tr>
<td>Completion</td>
<td>Wait for sometime</td>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td>Failure</td>
<td>Checking for faults(Ground fault or leakage current fault)</td>
<td>Wait</td>
<td>Failure</td>
</tr>
</tbody>
</table>

The MATLAB functions used in "charger sim", "Can decode" and "CAN1 send decode" are described in Appendix.

**8.3.3 MATLAB function "ChargerSim"**

%******************************************************************************
******************************************************************************

**********
% Function "ChargerSim" - Charger simulation is to simulate the charger
% Author- Ashwini Umarikar
% Date: - 2012-06-16
% Details-
%
% It has given three input signals in1, in2, in3

% "in1" is a 4 state signal emulated by a multiswitch for sending the "charger status".
% The multiswitch contains following switch state(AS PER REQUIREMENTS IN CAN MESSAGES)
% 1 for "Disconnected from power grid"(Wait)
% 2 for "Connected to grid-inrush"(Precharging)
% 3 for "Connected to grid-charging"(Charging)
% 4 for "Connected to grid-Not charging"(Charging Complete)
% these 4 states are generated by using to GPIO bits E0 & E1 in
% implementation

% "in2" for "Fault current setpoint".- E6 pin on GPIO
% "in3" for "Current_trip_level" : the next setpoint for the fault current,
% In implementation, a constant setpoint value is send but can also be
% input by using third ADC channel
function [MSG,data]= ChargerSim(in1,in2,in3)

%#eml

data=zeros(1,8);

if (in2==1) % " Fault Current setpoint"

    MSG=320;
    data(1)=in3; %"Current_trip_level"

else

    MSG=130;
    switch in1 %"charger status"

    case 1
        data(1)=1;
    case 2
        data(1)=3;
    case 3
        data(1)=3;
    case 4
        data(1)=4;
    otherwise
        data(1)=1;
    end

    data(3)=5; %default data
    data(4)=5;%default data
8.3.4 MATLAB function "Can decode"

%*******************************************************************
%**********
% Function - "CAN Decode"
% Author - Ashwini Umarikar
% Date: - 2012-06-16
% Details -
% function [CB_close_request,Charging_complete,
Setpoint,Setpoint_current]= CAN_Decode(MSG,data)
%#eml

    CB_close_request=0;
    Charging_complete=0;
    Setpoint=0;
    Setpoint_current=0;

    if (MSG==320)
        Setpoint=1;
        Setpoint_current=data(1);
elseif (MSG==130)
Setpoint=0;
Setpoint_current=0;

switch data(1)

case 1
is open and Charging complete is zero
CB_close_request=0;
% In Wait state
Charging_complete=0;

case 2
to grid_Inrush
CB_close_request=1;
% In Precharging state
Charging_complete=0;

case 3
to grid_Charging
CB_close_request=1;
% In Charging state
Charging_complete=0;

case 4
not charging
CB_close_request=0;
% In Complete state
Charging_complete=1;

otherwise

CB_close_request=0;
Charging_complete=0;
end
8.3.5 MATLAB function "Can1 send decode"

%*******************************************************************
**********
% Function  - "CAN1 send Decode"
% Author - Ashwini Umarikar
% Date: - 2012-06-16
% Details -

%This function is used to decode the data which is sent by the
"Protection device"
%The output of the function is MSGID and DATA
% Following MSGID's are used to send message from "Protection
device" to "Charger"
% "131" for sending the protection device status: this msg will be
 send by "Protection device" continuously
% which contains the state of "Protection device"
%(Wait-Precharging-Charging-Complete-Failure)
% "255" if there is some fault arise in "Protection device" like
"Ground fault" or
% "Earth leakage current fault" which cause the immediate shutdown.
This
% message will send which fault is get detected.
% Different databytes are used for it. Here we used first
databyte...which
% will send the message "01" for AC fault current get detected.
% "120" will send this current value of leakage current to the
charger
% through this message
function [MSG, data] = CAN1_Send_Decode(PD_fault, Fault_Current, PD_State, count)

data = zeros(1, 8);

if (PD_fault == 0)
    MSG = 131;
    data(1) = PD_State;
else
    if (count < 100000)
        MSG = 255;
        data(1) = 01;
    else
        MSG = 120;
        data(1) = Fault_Current;
    end
end

end
9 HARDWARE IMPLEMENTATION

This chapter describes the hardware implementation of the control logic as shown in Figure 8-1. The complete control logic is implemented using EKK-LM3S9B92 evaluation kit which incorporates LM3S9B92 microcontroller. The µVision IDE from Keil is used for the source code programming and program debugging. The details of the board are already described in section 7.1 and 7.2. As shown in Figure 8-1, the CAN bus is made using two CAN module CAN0 and CAN1 available in LM3S9B92. CAN transceivers SN65HVD232 are used in between CAN bus and microcontroller board as isolation and buffer stage. Refer to chapter 8 in particular to section 8.1 and 8.2 for other implementation details.

The implementation of the control logic can explained through below flowchart, as shown in Figure 9-1. The implemented logic can be divided in two main sections.

1. Initialization section
2. Main program section

The initialization section includes,

- Parameter declaration: The parameters which are used in the implementation codes are declared in this part of section.
- Define CAN interrupt handler: The interrupt handling code for interrupts occurs during CAN message transmission is written here. CAN0 interrupt handler is followed by the CAN1 interrupt handler.
- Define CAN message object: CAN message objects which will hold separate CAN messages is defined and also the CAN message buffers that hold the contents of different messages that are being transmitted and received.
- Enable & configure GPIO port: The GPIO port need to enable before use. Enable the ports which are used for different operations. Also the GPIO pin should be properly configured in order to function correctly as GPIO input or output.
- Configuration of ADC and CAN: The configuration of GPIO port to use as special peripheral like ADC and CAN. Here The ADC and CAN function will selected for the port and the pins.
- Initialize CAN message object: The CAN message object will be initialize, which is used for sending CAN messages. The message is 8 bytes that contains an incrementing value. It is initializing to 0. If the CAN message object is not shared it only needs to initialized one time, and can be used repeatedly sending the same message ID.
Figure 9-1: Flowchart of the implemented control logic

The main program section include the following process,

- Read input from GPIO port: The input values from GPIO port will be read from pins which are configured as input. (To describes this in detail, then each GPIO port bit has “weights” bit 0 = 1, bit1 = 2, bit 2 =4, bit 3 = 8 and so on. While reading the GPIO pin value these value will be return. To have the value in 0 or
Identify the charger status and CAN0 message transmission: Depending on the charger status input encode the CAN message with specific message ID and data as described in section 8.2. Finally the message will transmit to CAN1.

CAN1 message receive and decode: CAN 1 will receive the messages coming from CAN0 with specific message ID and data. The message will be decoded for the information it contained.

Control logic and state machine: Depending on the data (information) control logic will performs checks using state machine logic.

PD status and Command to CB: After processing the data from the CAN messages as well as from the gpio input ports the result will send to CAN1 with specified CAN message ID and data as mentioned in tables Table 7 and Table 8 of section 8.2.

9.1 Implementation Results

The values of various parameters during the program execution are displayed on HyperTerminal on the computer which is interfaced to the evaluation board through USB connection. This section presents the sample tests results during the program execution, which are basically of various parameters in different states. The following messages are sent to HyperTerminal during program execution

- Display the RCD current value read on ADC2 input channel in uA.
- Display text at the time of CAN0 module sending MSG- this includes MSG ID and MSG data
- Display text at the time of CAN1 modules receives any message- this includes MSG ID, MSG length and MSG data
- Display the status of charger states namely- CB close, charging complete and set point change request
- Display the current state of the state machine
- Display the status of output signals from state machine- CB close command and PD status
- Display text at the time of CAN1 modules sends any message- this includes MSG ID and MSG data
- Display text "loop end" at the end of the main logic after which program execution returns to start of main logic

The only exception to above is during RCD trip limit set point change state. In this state the sequence of the text displayed on HyperTerminal is as below

- Display the RCD current value read on ADC2 input channel in uA.
- **New value of the set point trip level in uA, its hex value and its lower and upper bytes which are transmitted through CAN0 module**
- Display text at the time of CAN0 module sending MSG- this includes MSG ID and MSG data
- Display text at the time of CAN1 modules receives any message- this includes MSG ID, MSG length and MSG data
- Display the status of charger states namely- CB close, charging complete and set point change request
- Display the current state of the state machine
- Display the status of output signals from state machine- CB close command and PD status
- Display the lower and upper byte received through CAN message corresponding to new RCD current trip level
- Display text at the time of CAN1 modules sends any message- this includes MSG ID and MSG data
- Display text "loop end" at the end of the main logic after which program execution returns to start of main logic

The below presented are the sample results as displayed on HyperTerminal during program execution for different states

### 9.1.1 Wait state
ival5_ADC2 4107 uA
CAN_0 Sending Msg ID=0x000000130 msg: 0x00 00 11 22 33 44 55 66
CAN_1 Receive Msg ID=0x000000130 len=8 data=0x00 00 11 22 33 44 55 66
CAN_1 Receive= 0 0 0
Wait state
CB close command 0, PD status 0
CAN_1 Sending Msg ID=0x000000131 msg: 0x01 00 00 00 00 00 00 00
loop end

### 9.1.2 Precharging state
ival5_ADC2 4107 uA
CAN_0 Sending Msg ID=0x000000130 msg: 0x02 00 11 22 33 44 55 66
CAN_1 Receive Msg ID=0x000000130 len=8 data=0x02 00 11 22 33 44 55 66
CAN_1 Receive= 0 1 0
Precharging state
CB close command 0, PD status 0
CAN_1 Sending Msg ID=0x000000131 msg: 0x01 00 00 00 00 00 00 00
loop end

### 9.1.3 Charging state
ival5_ADC2 4107 uA
CAN_0 Sending Msg ID=0x000000130 msg: 0x02 00 11 22 33 44 55 66
CAN_1 Receive Msg ID=0x000000130 len=8 data=0x02 00 11 22 33 44 55 66
CAN_1 Receive= 0 1 0
Charging state
CB close command 0, PD status 0
CAN_1 Sending Msg ID=0x000000131 msg: 0x01 00 00 00 00 00 00 00

50
loop end

9.1.4 Charging complete state

ival5_ADC2 4107 uA
CAN_0 Sending Msg ID=0x00000130 msg:0x02 00 11 22 33 44 55 66
CAN_1 Receive Msg ID=0x00000130 len=8 data=0x02 00 11 22 33 44 55 66
CAN_1 Receive= 0 1 0
Complete state
CB close command 0, PD status 0
CAN_1 Sending Msg ID=0x00000131 msg: 0x01 00 00 00 00 00 00 00
loop end

9.1.5 Charging fail state

ival5_ADC2 6835 uA
CAN_0 Sending Msg ID=0x00000130 msg:0x02 00 11 22 33 44 55 66
CAN_1 Receive Msg ID=0x00000130 len=8 data=0x02 00 11 22 33 44 55 66
CAN_1 Receive=0 1 0
Fail state- charging failure
CB close command 0, PD status 1
CAN_1 Sending Msg ID=0x000000ff msg: 0x01 00 00 00 00 00 00 00
CAN_1 Sending Msg ID=0x00000120 msg: 0xdf 1b 00 00 00 00 00 00
loop end

9.1.6 RCD trip limit set point change state

ival5_ADC2 3771 uA
New Limit 5800 uA= 0x16a8, LowByte= 0xa8, HighByte= 0x16
CAN_0 Sending Msg ID=0x00000320 msg: 0xa8 16 00 00 00 00 00 00
CAN_1 Receive Msg ID=0x00000320 len=8 data=0xa8 16 00 00 00 00 00 00
CAN_1 Receive= 0 0 1
RCD trip limit set point state
I_DC_Set= 0x16a8, LowByte= 0xa8, HighByte= 0x16
CB close command 0, PD status 0
CAN_1 Sending Msg ID=0x00000131 msg: 0x01 00 00 00 00 00 00 00
loop end
10 CONCLUSIONS

According to requirements from hardware-software parts:

The given Power supply has to convert to the required voltages for the peripherals. It is achieved by using a dual step-down regulator.

The Ground fault monitoring is designed and tested.

The Residual current monitoring for detecting the desired leakage current is designed and simulated in Pspice. The implementation of RCD is tested. The test is succeed with tripping at 6mA leakage current which is ultra sensitive compare to the previous work[15].

The control logic is designed considering the requirement specifications for control, peripheral interface functions for input and output of initiated commands. A suitable microcontroller kit- Stellaris EKK-LM3S9B92 evaluation kit is selected which has the peripheral functions required for communication tasks like CAN modules to communicate over CAN network, ADC interface to receive analog inputs from peripheral RCD hardware like residual leakage current detector- both AC and DC currents as well as digital input status commands from different peripherals and digital output commands to initiate trip command to circuit breaker.

The whole protection device logic is simulated in MATLAB-Simulink using Stateflow toolbox.

Since the separate charger module was not available, the CAN part of charger is also simulated using the same microcontroller board; the selected microcontroller board has two CAN ports which are not back-to-back connected. The real CAN messages as sent by actual charger are generated with additional logic implemented in the microcontroller and are transmitted over CAN network, these are them received over other CAN port used for control logic. This increased the complexity of the implemented control logic.

The control logic is implemented as a state machine. The control logic changes the states depending upon the status signals and fault status as well as the messages received over CAN network. The transmission of CAN messages from charger, the receipt of the same by CAN port assigned to control logic and the interpretation of messages, accordingly the change of states of control logic is printed on HyperTerminal for testing the operation of the implemented logic. These same results are also reported in this report.

All the results are satisfiable and tested several times but not consecutively. The results can be more accurate with consecutive test.
11 FUTURE WORK

For further extension to the existing work, another measurement technique can be implemented. High frequency response can be considered.

The implemented control logic on EKK-LM3S9B92 evaluation kit also includes additional part for generating the CAN messages to be sent from charger. This will effect on the execution timing of the main control loop. The real timing for various critical operations should be determined by disabling this additional logic and connecting a separate CAN module for charger CAN message generation. The different delays are intentionally added in the program to allow sufficient time for transfer of messages between CAN0 and CAN1 modules. These delays can be removed in the real implementation with the actual CAN module of "charger" is available of CAN network. This way the realistic execution time of the algorithm can be determined.

The implemented program also prints messages on the HyperTerminal for debugging purpose, which is a processor time consuming task; this should be removed after the logic verification is complete.
REFERENCES

[4] ADM803
[6] LT3509EMSE.
[7] TL062
[8] LM324:
   http://www.ehu.es/instru_virtualdaq/Planoak/LM324.pdf
[9] Comparator1716
[10] Microcontroller LM3S9B92 datasheet:
    http://www.ti.com/lit/ds/spms180m/spms180m.pdf
[11] Stellaris peripheral driver library user's guide:
    http://www.ti.com/mcu/docs/mcuorphantoolsw.tsp?sectionId=632&orphantabId=8
13 ENCLOSURES

13.1 Source code of Microcontroller logic implementation

/*********************************************
// Main program with CAN and ADC implementation
*********************************************

// Author- Ashwini Umarikar
// Date:- 2012-06-10

/*********************************************

// Details-
// This program is written for TI's microcontroller LM3S9B92.
// This program delivers the Protection device logic in Battery charger.
// This logic includes reading the input devices like Recidual current device and
// Ground monitor device as well as CAN messages from the Charger. Depending on these
// inputs, the logic should process the input data and should send the output as CAN
// messages to Charger and also command to circuit breaker for open/close the circuit.
// The Microcontroller LM3S9B92 has two CANs. They are used to simulate both protection
// device and Charger.CAN0 used as a Charger and CAN1 is used as Protection device.
// CAN0 is only sending messages as a "Charger requests". CAN1 will receives those messages
// from Charger, these messages will get processed and the result will be send as CAN messages
// to the Charger. The Charger will also receive the status the Protection device.
// CAN Messages are as per Bosch specifications.(Complete program with CAN logic emulating the
// battery Charger signals)
// CAN_0 is used to send the Charger status, Charger fault and CB_close request
// signals to Protection device
// CAN_1 receives the signals from CAN BUS also transmits Protection device status and fault
// signals to battery charger
// ADC inputs added to CAN logic (The program does not include ADC implemetation to read
// analog value of leakage currents)
// Two channels of ADC are configured for two analog input signals
// Following are inputs and outputs
// (Switch mentioned here are the switches which are used on bredboard for giving input)
// ival2 -- Charger status1 -- Switch S1 - GPIO port E0- input
// ival3 -- Charger status2 -- Switch S2 - GPIO port E1- input

/*****************************************************************************/
//CHARGER COMMANDS

<table>
<thead>
<tr>
<th>data</th>
<th>status1</th>
<th>status2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x02</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x03</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

// disconnected from grid

// connected to grid-inrush

// connected to grid-charging

// connected to grid-not charging

// ival4 -- setpoint for fault current from Charger
-- Switch S3 - GPIO port E6-input

// ival8-- Protection Device fault status (0- no fault, 1- fault)
-- Switch S5 - GPIO port E4- output

// ival6-- Ground Monitor Device fault
-- Switch S6 - GPIO port F4- input

// ival1-- Circuit Breaker status
-- Switch S7 - GPIO port B6- input

// ival7-- Command to Circuit Breaker
-- Switch S8 - GPIO port E5- output

// ADC2-E3- analog input-- ival5-- Earth leakage fault

// ADC1-E2- analog input-- not used

//********************************************************************************************
//*******************************************************************
**********
//Header files
//*****************************************************************************
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "inc/hw_can.h"
#include "driverlib/can.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "utils/uartstdio.h"
//****************************************************************************
// Variable Declaration
//****************************************************************************
int i,ival1,ival2,ival3,ival4,ival5,ival6,ival7,ival8,value,uldx, ival2_CAN,ival3_CAN,ival4_CAN;
unsigned long ulADC0_Value [2],ulADC_data[2];
int R_ADC[2]={100,420};
int I_ADC[2]={0,0},ival5_ADC;
int I_DC_Set=6000,I_AC_Set=30000;
// Fixed values has been taken for ADC, the value should not exceed than these values.
char x,y; int val;
unsigned int state=0; // Variable for Switch operation
//****************************************************************************
// A counter that keeps track of the number of times the TX interrupt has
// occurred, which should match the number of TX messages that were sent.
//****************************************************************************
volatile unsigned long gt_ulMsgCount1 = 0; // Message count for transmission from CAN0
volatile unsigned long gt_ulMsgCount2 = 0; // Message count for transmission from CAN1
//****************************************************************************
// A flag to indicate that some transmission error occurred.
//****************************************************************************
volatile unsigned long gt_bErrFlag1 = 0;
volatile unsigned long gt_bErrFlag2 = 0;
//****************************************************************************
// A counter that keeps track of the number of times the RX interrupt has
// occurred, which should match the number of messages that were received.
//****************************************************************************
volatile unsigned long g_ulMsgCount = 0; // Message count for of receiving messages
//****************************************************************************
// A flag for the interrupt handler to indicate that a message was received.
//****************************************************************************
volatile unsigned long g_bRXFlag = 0;
//****************************************************************************
// A flag to indicate that some reception error occurred.
//****************************************************************************

volatile unsigned long g_bErrFlag = 0;

//*****************************************************************************/

// A flag to indicate that CAN0 controller message object 1 has sent a message.
// This is need when sending multiple MSG's in sequence
// This situation is occurring only for CAN 0 since it send two consecutive MSG's
//*****************************************************************************/

volatile unsigned long g_bMsgObj1Sent = 0;

//*****************************************************************************/

// The error routine that is called if the driver library encounters an error.
//*****************************************************************************/

#ifdef DEBUG
Void __error__(char *pcFilename, unsigned long ulLine)
{
} #endif

//*****************************************************************************/

// This function sets up UART0 to be used for a console to display information
// as the example is running.
//*****************************************************************************/

Void InitConsole(void)
{
    // Enable GPIO port A which is used for UART0 pins.
    // TODO: change this to whichever GPIO port you are using.
    // SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    // configure the pin muxing for UART0 functions on port A0 and A1.
    // This step is not necessary if your part does not support pin muxing.
    // TODO: change this to select the port/pin you are using.
    //
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    //
    // Select the alternate (UART) function for these pins.
    // TODO: change this to select the port/pin you are using.
    //
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
}
// Initialize the UART for console I/O.

UARTStdioInit(0);

//****************************************************************************
// This function provides a 1 second delay using a simple polling method.
//****************************************************************************

void
SimpleDelay(void)
{

  // Delay cycles for 1 second

  SysCtlDelay(16000000 / 3);
}

//****************************************************************************
// CAN 0 interrupt handler
// This function is the interrupt handler for the CAN peripheral. It checks
// for the cause of the interrupt, and maintains a count of all messages that
// have been transmitted.
//****************************************************************************

void
CAN0_IntHandler(void)
{

  unsigned long ulStatus_CAN0;

  // Read the CAN interrupt status to find the cause of the interrupt

  ulStatus_CAN0 = CANIntStatus(CAN0_BASE, CAN_INT_STS_CAUSE);

  // If the cause is a controller status interrupt, then get the status

  if(ulStatus_CAN0 == CAN_INT_INTID_STATUS)
{
    //
    // Read the controller status. This will return a field of status
    // error bits that can indicate various errors. Error processing
    // is not done in this example for simplicity. Refer to the
    // API documentation for details about the error status bits.
    // The act of reading this status will clear the interrupt. If the
    // CAN peripheral is not connected to a CAN bus with other CAN devices
    // present, then errors will occur and will be indicated in the
    // controller status.
    //
    ulStatus_CAN0 = CANStatusGet(CAN0_BASE, CAN_STS_CONTROL);
    //
    // Set a flag to indicate some errors may have occurred.
    //
    gt_bErrFlag1 = 1;
}

//
// Check if the cause is message object 1, which what we are using for
// sending messages.
//
else if(ulStatus_CAN0 == 1)
{
    //
    // Getting to this point means that the TX interrupt occurred on
    // message object 1, and the message TX is complete. Clear the
    // message object interrupt.
    //
    CANIntClear(CAN0_BASE, 1);
    //
    // Increment a counter to keep track of how many messages have been
    // sent. In a real application this could be used to set flags to
    // indicate when a message is sent.
    //
    gt_ulMsgCount1++;
// Since the message was sent, clear any error flags.

//
gt_bErrFlag1 = 0;

//
// Otherwise, something unexpected caused the interrupt. This should
// never happen.

//
else
{
    //
    // Spurious interrupt handling can go here.
    //
}

//******************************************************

//******************************************************************

// CAN 1 interrupt handler

// This function is the interrupt handler for the CAN peripheral. It checks
// for the cause of the interrupt, and maintains a count of all messages that
// have been received.

//******************************************************************

//******************************************************

void CAN1_IntHandler(void)
{
    unsigned long ulStatus_CAN1;

    //
    // Read the CAN interrupt status to find the cause of the interrupt
    //
    ulStatus_CAN1 = CANIntStatus(CAN1_BASE, CAN_INT_STS_CAUSE);

    //
    // If the cause is a controller status interrupt, then get the status
    //
    if(ulStatus_CAN1 == CAN_INT_INTID_STATUS)
{  
    //  
    // Read the controller status. This will return a field of status  
    // error bits that can indicate various errors. Error processing  
    // is not done in this example for simplicity. Refer to the  
    // API documentation for details about the error status bits.  
    // The act of reading this status will clear the interrupt.  
    //
    ulStatus_CAN1 = CANStatusGet(CAN1_BASE, CAN_STS_CONTROL);
    
    //
    // Set a flag to indicate some errors may have occurred.  
    //
    g_bErrFlag = 1; // error flag for transmission

    gt_bErrFlag2 = 1; // error flag for receiving

    }
    //
    // Check if the cause is message object 2, which what we are using for  
    // receiving messages.
    //
    else if(ulStatus_CAN1 == 2)
    {
        CANIntClear(CAN1_BASE, 2);

        gt_ulMsgCount2++; // counting for transmission

        gt_bErrFlag2 = 0; // error flag for transmission

        //
        // Set the flag indicating that a message was sent using message  
        // object 2. The program main loop uses this to know when to send  
        // another message using message object 2.

        g_bMsgObj1Sent = 1

    }
    else if(ulStatus_CAN1 == 1)
    {
        //
        // Getting to this point means that the RX interrupt occurred on

// message object 1, and the message reception is complete. Clear the
// message object interrupt.

//
CANIntClear(CAN1_BASE, 1);
//
// Increment a counter to keep track of how many messages have been
// received. In a real application this could be used to set flags to
// indicate when a message is received.

//
g_ulMsgCount++;  // counting for receiving
//
// Set flag to indicate received message is pending.

//
g_bRXFlag = 1;
//
// Since a message was received, clear any error flags.

//

    g_bErrFlag = 0;  // error flag for receiving
}
//
// Otherwise, something unexpected caused the interrupt. This should
// never happen.

//
else
{

    //
    // Spurious interrupt handling can go here.

/

}
//********************************************************************************
// Function for extracting bit
// These functions are used for shifting bits for using the "desired bits" for
// the operations
/*******************************************************************************/
int extractBit(char byte, int pos)
{
    return (byte >> pos) & 0x01;
}
/*******************************************************************************/

unsigned int ExtractLowByte (unsigned int val)
{
    int Bytes=0xFF & val; //extract the lower byte
    return Bytes;
}

unsigned int ExtractHighByte (unsigned int val)
{
    int Bytes=0xFF & (val>>8); //extract the upper byte
    return Bytes;
}

unsigned int CombineByte (unsigned char lowByte, unsigned char highByte)
{
    int val = (highByte << 8) + lowByte;
    //Combining both lower and upper bits
    return val;
}

int main(void)
{
    // Define CAN Message object
    // CAN message objects that will hold the separate CAN messages.
    // CAN Message buffers that hold the contents of the different messages that
    // are being transmitted and received. Each one is of length 8 of data bytes.
    //*********************************************************************************
    //CAN Message object1 for sending
    tCANMsgObject sCANMessage1;
    unsigned char ucMsgData1[8]={0,0,0,0,0,0,0,0};
    //CAN Message object2 for sending
    tCANMsgObject sCANMessage2;
unsigned char ucMsgData2[8]={0,0,0,0,0,0,0,0};

// CAN message object for receiving
    tCANMsgObject rCANMessage;
    unsigned char rucMsgData[8]={0,0,0,0,0,0,0,0};

/****************
************************************************************
// Set the clocking to run at 16 MHz using the PLL. When
// using the ADC, you must either use the PLL or supply a 16 MHz clock source.
// The SYSCTL_XTAL_ value must be changed to match the value of the
// crystal on your board.
="/***************************************************/
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
|SYSCTL_XTAL_16MHZ);

/*******************************************************************************/
// Set up the serial console to use for displaying messages. This is
// just for this example program and is not needed for ADC operation.
/*******************************************************************************/
initConsole();

/*******************************************************************************/
// The ADC0 peripheral must be enabled for use.
/*******************************************************************************/
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

/*******************************************************************************/
// GPIO Pin Setting
/*******************************************************************************/
// Enable the peripherals used. GPIO ports need to enabled before use.
// Enable GPIO port B,D,E F which are used for different operations
/*******************************************************************************/
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
//*****************************************************************************
// After enabling the peripheral delay operation is used.
//*****************************************************************************

SimpleDelay();

//*****************************************************************************
// The GPIO pins must be properly configured in order to function correctly as
// GPIO input/output.
//*****************************************************************************

GPIOPinTypeGPIOInput(GPIO_PORTB_BASE, GPIO_PIN_4|GPIO_PIN_6);
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, GPIO_PIN_4);
GPIOPinTypeGPIOInput(GPIO_PORTE_BASE, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_6);
GPIOPinTypeGPIOOutput(GPIO_PORTE_BASE, GPIO_PIN_4|GPIO_PIN_5);
GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_0);

//*****************************************************************************
// Configuration of ADC 0
//*****************************************************************************

// Select the analog ADC function for these pins.
// Consult the data sheet to see which functions are allocated per pin.
// The port/pin can be selected as per the need.
//
// GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_2);
// GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_3);
//
// Enable sample sequence 0 with a processor signal trigger. Sequence 0
// will do a single sample when the processor sends a signal to start the
// conversion. Each ADC module has 4 programmable sequences, sequence 0
// to sequence 3.
//
// ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
//
// Configure step 0 on sequence 0. Sample channel 0 (ADC_CTL_CH0) in
// single-ended mode (default) and configure the interrupt flag
// (ADC_CTL_IE) to be set when the sample is done. Tell the ADC logic
// that this is the last conversion on sequence 0 (ADC_CTL_END).
//
// Sequence 0 has 8 programmable steps. Sequence 1 and 2 have 4 steps,
// sequence 3 has only one programmable step.

// For more information on the ADC sequences and steps, reference the datasheet.

ADCSequenceStepConfigure(ADC0_BASE, 0, 0, ADC_CTL_CH8);
ADCSequenceStepConfigure(ADC0_BASE, 0, 1, ADC_CTL_CH9 | ADC_CTL_IE | ADC_CTL_END); // 3, 0

// Since sample sequence 0 is now configured, it must be enabled.

ADCSequenceEnable(ADC0_BASE, 0);

// Clear the interrupt status flag. This is done to make sure the
// interrupt flag is cleared before we sample.

ADCIntClear(ADC0_BASE, 0);

//*****************************************************************************/

// Configuration of CAN 0

//*****************************************************************************/

// For this example CAN0 is used with RX and TX pins on port A6 and A7

// GPIO port A needs to be enabled so these pins can be used.

// you can change this to whichever GPIO port you are using.

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

// Configure the GPIO pin muxing to select CAN0 functions for these pins.

// This step selects which alternate function is available for these pins.

GPIOPinConfigure(GPIO_PA6_CAN0RX);

GPIOPinConfigure(GPIO_PA7_CAN0TX);

// Enable the alternate function on the GPIO pins. The above step selects
// which alternate function is available. This step actually enables the
// alternate function instead of GPIO for these pins.

GPIOPinTypeCAN(GPIO_PORTA_BASE, GPIO_PIN_6 | GPIO_PIN_7);
//
// The GPIO port and pins have been set up for CAN. The CAN peripheral
// must be enabled.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_CAN0);
//
// Initialize the CAN controller
//
CANInit(CAN0_BASE);

//
// Set up the bit rate for the CAN bus. This function sets up the CAN
// bus timing for a nominal configuration. You can achieve more control
// over the CAN bus timing by using the function CANBitTimingSet() instead
// of this one, if needed.
// In this example, the CAN bus is set to 500 kHz. In the function below,
// the call to SysCtlClockGet() is used to determine the clock rate that
// is used for clocking the CAN peripheral. This can be replaced with a
// fixed value if you know the value of the system clock, saving the extra
// function call. For some parts, the CAN peripheral is clocked by a fixed
// 8 MHz regardless of the system clock in which case the call to
// SysCtlClockGet() should be replaced with 8000000. Consult the data
// sheet for more information about CAN peripheral clocking.
//
CANBitRateSet(CAN0_BASE, SysCtlClockGet(), 500000);

//
// Enable interrupts on the CAN peripheral. This example uses static
// allocation of interrupt handlers which means the name of the handler
// is in the vector table of startup code. If you want to use dynamic
// allocation of the vector table, then you must also call CANIntRegister()
// here.
//
// CANIntRegister(CAN0_BASE, CANIntHandler); // if using dynamic vectors
//
CANIntEnable(CAN0_BASE, CAN_INT_MASTER | CAN_INT_ERROR | CAN_INT_STATUS);
// Enable the CAN interrupt on the processor (NVIC).

IntEnable(INT_CAN0);

// Enable the CAN for operation.

CANEnable(CAN0_BASE);

//****************************************************************************
// Configuration of CAN 1
//****************************************************************************

// For this example CAN1 is used with RX and TX pins on port F0 and F1.
// GPIO port F needs to be enabled so these pins can be used.

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

// Configure the GPIO pin muxing to select CAN1 functions for these pins.
// This step selects which alternate function is available for these pins.

GPIOPinConfigure(GPIO_PF0_CAN1RX);
GPIOPinConfigure(GPIO_PF1_CAN1TX);

// Enable the alternate function on the GPIO pins. The above step selects
// which alternate function is available. This step actually enables the
// alternate function instead of GPIO for these pins.

GPIOPinTypeCAN(GPIO_PORTF_BASE, GPIO_PIN_0 | GPIO_PIN_1);

// The GPIO port and pins have been set up for CAN. The CAN peripheral
// must be enabled.

SysCtlPeripheralEnable(SYSCTL_PERIPH_CAN1);
//
// Initialize the CAN controller
//
CANInit(CAN1_BASE);

//
// Set up the bit rate for the CAN bus. This function sets up the CAN
// bus timing for a nominal configuration. You can achieve more control
// over the CAN bus timing by using the function CANBitTimingSet() instead
// of this one, if needed.
// In this example, the CAN bus is set to 500 kHz. In the function below,
// the call to SysCtlClockGet() is used to determine the clock rate that
// is used for clocking the CAN peripheral. This can be replaced with a
// fixed value if you know the value of the system clock, saving the extra
// function call. For some parts, the CAN peripheral is clocked by a fixed
// 8 MHz regardless of the system clock in which case the call to
// SysCtlClockGet() should be replaced with 8000000. Consult the data
// sheet for more information about CAN peripheral clocking.
//
CANBitRateSet(CAN1_BASE, SysCtlClockGet(), 500000);

//
// Enable interrupts on the CAN peripheral. This example uses static
// allocation of interrupt handlers which means the name of the handler
// is in the vector table of startup code. If you want to use dynamic
// allocation of the vector table, then you must also call CANIntRegister()
// here.
//
// CANIntRegister(CAN0_BASE, CANIntHandler); // if using dynamic vectors
//
CANIntEnable(CAN1_BASE, CAN_INT_MASTER | CAN_INT_ERROR | CAN_INT_STATUS);
//
IntEnable(INT_CAN1);

//
// Enable the CAN for operation.
//
CANEnable(CAN1_BASE);

//******************************************************************************
// Initialize the message object that will be used for sending CAN
// messages. The message will be 8 bytes that will contain an incrementing
// value. Initially it will be set to 0.
//******************************************************************************

//********************************************************************************
// CAN0 message object
//********************************************************************************
// Initialize message object 1 to be able to send CAN message 1. This
// message object is not shared so it only needs to be initialized one
// time, and can be used for repeatedly sending the same message ID.
//-------------------------------------------------------------------------------
*(unsigned long *)ucMsgData1 = 0;
    sCANMessage1.ulMsgID = 0x1001;                        // CAN message ID - use 1
    sCANMessage1.ulMsgIDMask = 0;                    // no mask needed for TX
    sCANMessage1.ulFlags = MSG_OBJ_TX_INT_ENABLE;    // enable interrupt on TX
    sCANMessage1.ulMsgLen = sizeof(ucMsgData1);       // size of message is 4
    sCANMessage1.pucMsgData = ucMsgData1;             // ptr to message content

//********************************************************************************
// CAN1 message object2
//********************************************************************************
// Initialize message object 2 to be able to send CAN message 2. This
// message object is not shared so it only needs to be initialized one
// time, and can be used for repeatedly sending the same message ID.
//-------------------------------------------------------------------------------
*(unsigned long *)ucMsgData2 = 0;
sCANMessage2.ulMsgID = 0x2001;                        // CAN message ID - use 1
sCANMessage2.ulMsgIDMask = 0;                        // no mask needed for TX
sCANMessage2.ulFlags = MSG_OBJ_TX_INT_ENABLE;        // enable interrupt on TX
sCANMessage2.ulMsgLen = sizeof(ucMsgData2);         // size of message is 4
sCANMessage2.pucMsgData = ucMsgData2;              // ptr to message content

polator message object to be used for receiving CAN messages with
any CAN ID. In order to receive any CAN ID, the ID and mask must both
be set to 0, and the ID filter enabled.

rCANMessage.ulMsgID = 0x1001;                        // CAN msg ID - 0 for any
rCANMessage.ulMsgIDMask = 0xfffff;                  // mask is 0 for any ID
rCANMessage.ulFlags = MSG_OBJ_RX_INT_ENABLE |       // MSG_OBJ_USE_ID_FILTER | MSG_OBJ_EXTENDED_ID;
                 rCANMessage.ulMsgLen = 8;                        // allow up to 8 bytes

Now load the message object into the CAN peripheral. Once loaded the
CAN will receive any message on the bus, and an interrupt will occur.
Use message object 1 for receiving messages (this is not the same as
the CAN ID which can be any value in this example).

CANMessageSet(CAN1_BASE, 1, &rCANMessage, MSG_OBJ_TYPE_RX);
while(1)
{
    //Read ADC 0 Values
    //
    // Trigger the ADC conversion.
    //
    ADCProcessorTrigger(ADC0_BASE, 0);
    //
    // Wait for conversion to be completed.
    //
    while(!ADCIntStatus(ADC0_BASE, 0, false))
    {
    }
    //
    // Clear the ADC interrupt flag.
    //
    ADCIntClear(ADC0_BASE, 0);
    //
    // Read ADC Value.
    //

    ADCSequenceDataGet(ADC0_BASE, 0, ulADC0_Value);

    //
    // Display the AIN0 (PE7) digital value on the console.
    //

    //Valid values are for 2 and 3.
    for(i=0;i<2;i++)
    {
        // Converted to millivolts.
ulADC_data[i] = (ulADC0_Value[i] * 1.0 / (3.0 / 1024) * 1000); // in mV Range 0-3000nV
I_ADC[i] = 1000 * ulADC_data[i] / R_ADC[i]; // if resistance value is known, value in uA

ival5_ADC = I_ADC[1];

//
// This function provides a means of generating a constant length
delay. The function delay (in cycles) = 3 * parameter. Delay
// 250ms arbitrarily.
//
SysCtlDelay(SysCtlClockGet() / 12);

// Read Input values from GPIO ports
// ival1 -- Circuit Breaker status -- GPIO port B6 - input
ival1 = GPIOPinRead(GPIO_PORTB_BASE, GPIO_PIN_6);
ival1 = extractBit(ival1, 6);

// ival2 -- Charger status1 -- GPIO port E0 - input
ival2 = GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_0);

// ival3 -- Charger status2 -- GPIO port E1 - input
ival3 = GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_1);
ival3 = extractBit(ival3, 1);

// ival4 -- setpoint for fault current from Charger -- GPIO port E6 - input
ival4 = GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_6);
ival4 = extractBit(ival4, 6);

// ival6 -- Ground Monitor Device fault -- GPIO port F4 - input
ival6 = GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_4);
ival6 = extractBit(ival6, 4);

if(0) // change to 1 to check switch connections
{
    UARTprintf(" CB_status (S7) \%d ", ival1);
    UARTprintf(" Charger-stat_0(S1) \%d ", ival2);
UARTprintf(" Charger-stat_0(S2) \%d ", ival3);
UARTprintf(" Earth_Leakage_Setpoint(S3) \%d ", ival4);
UARTprintf(" Earth_leakage_Fault(ADC2) \%d ", ival5);
UARTprintf(" GMD_fault(S6) \%d\n\n ", ival6);
}

// Charger send the CAN message for changing the Earth leakage setpoint
//
// CAN messages from Charger to Protection device (CAN0 to CAN1)
//---------------------------------------------------------------------------------------------
// CAN MSG ID | description
Data
//---------------------------------------------------------------------------------------------
// 0x130       |   Disconnected from grid   0x00
//   Connected to grid-inrush        0x01
//   Connected to grid-charging      0x02
//   Connected to grid-not charging  0x03
//---------------------------------------------------------------------------------------------
// 0x320       |   Fault current setpoint
//---------------------------------------------------------------------------------------------
//
// ival4 represents that earth leakage setpoint
if(ival4==1) //Earth leakage setpoint request came from Charger
{
    for(i=1;i<9;i++) ucMsgData1[i]=0x00; //reset the CAN0 data

    // Trip current level 5.8 mA is assumed
    x=ExtractLowByte ( 5800);
    y=ExtractHighByte( 5800);

    UARTprintf(" NewLimit 5800 \uA= 0x%X, LowByte= 0x%02X, HighByte= 0x%02X \n",5800, x,y);
    ucMsgData1[0]=x;
    ucMsgData1[1]=y;
}
sCANMessage1.ulMsgID = 0x0320;  //CAN MSG ID for fault current setpoint is assigned here
else
{
    for(i=1;i<9;i++) ucMsgData1[i]=0x00;//reset the CAN1 data
    //CAN encoding based on Status1(ival2) and Status2(ival3)
    if (ival2==0 & ival3==0)
        ucMsgData1[0]=0x00;  // Disconnected from power grid
    if (ival2==0 & ival3==1)
        ucMsgData1[0]=0x01; // Connected to grid
                     - inrush
    if (ival2==1 & ival3==0)
        ucMsgData1[0]=0x02; // Connected to grid
                     - charging
    if (ival2==1 & ival3==1)
        ucMsgData1[0]=0x03; // Connected to grid
                     - not charging

    ucMsgData1[1]=0x00;//compensation data byte--- currently not implemented
    ucMsgData1[2]=0x11;//Value of compensated fault current
    ucMsgData1[3]=0x22;
    ucMsgData1[4]=0x33;//Value of rms current drawn from power grid
    ucMsgData1[5]=0x44;
    ucMsgData1[6]=0x55;//RMS voltage of power grid
    ucMsgData1[7]=0x66;

sCANMessage1.ulMsgID = 0x130; //CAN MSG ID for Charger status for Protection device is assigned here.

}//end else loop

// Send the CAN message using object number 1 (not the same thing as
// CAN ID, which is also 1 in this example). This function will cause
// the message to be transmitted right away.

    CANMessageSet(CAN0_BASE, 1, &sCANMessage1, MSG_OBJ_TYPE_TX);
    SimpleDelay();
if(gt_bErrFlag1) //If some error occured error flag will be set.
{
    UARTprintf(" error - cable connected?\n");
}
else
{
    //
    // If no errors then print the count of message sent
    //
}

//********************************************************************************
// Receive Messages
//*********************************************************************************

// Enter loop to process received messages. This loop just checks a flag
// that is set by the interrupt handler, and if set it reads out the
// message and displays the contents. This is not a robust method for
// processing incoming CAN data and can only handle one messages at a time.
// If many messages are being received close together, then some messages
// may be dropped. In a real application, some other method should be used
// for queuing received messages in a way to ensure they are not lost. You
// can also make use of CAN FIFO mode which will allow messages to be
// buffered before they are processed.
//********************************************************************************

// If the flag is set, that means that the RX interrupt occurred and
// there is a message ready to be read from the CAN
//
if(g_bRXFlag)
{
    //
    // Reuse the same message object that was used earlier to configure
    // the CAN for receiving messages. A buffer for storing the
    // received data must also be provided, so set the buffer pointer
    // within the message object.
CANMessageGet(CAN1_BASE, 1, &rCANMessage, 0);

// Clear the pending message flag so that the interrupt handler can
// set it again when the next message arrives.
//
g_bRXFlag = 0;

// Check to see if there is an indication that some messages were
// lost.
//
if(rCANMessage.ulFlags & MSG_OBJ_DATA_LOST)
{
    UARTprintf("CAN_1 message loss detected\n");
}

// Print out the contents of the message that was received.
//
UARTprintf("CAN_1 Receive Msg ID=0x%08X len=%u data=0x",
    rCANMessage.ulMsgID, rCANMessage.ulMsgLen);
for(uIdx = 0; uIdx < rCANMessage.ulMsgLen; uIdx++)
{
    UARTprintf("%02X ", rucMsgData[uIdx]);
}
UARTprintf("\n");
if(rCANMessage.ulMsgID==0x320)
{
    ival4_CAN=1;//Setpoint request
    state=6;
    // Check for other CAN MSG ID 0x0130. If true then Decode the CAN messages coming from Charger.
    else if(rCANMessage.ulMsgID==0x0130)
    {
        ival2_CAN=0;//initialize Charging complete
        ival3_CAN=0;//initialize CB close request
        ival4_CAN=0;//Reset setpoint
        // Decoding CAN message
        // Depending on the data in CAN MSG set the value for the variables
        if(rucMsgData[0]==0x00) {ival2_CAN=0;ival3_CAN=0;}// Disconnected from power grid
        if(rucMsgData[0]==0x01) {ival2_CAN=0;ival3_CAN=1;}// Connected to grid-inrush
        if(rucMsgData[0]==0x02) {ival2_CAN=0;ival3_CAN=1;}// Connected to grid- charging
        if(rucMsgData[0]==0x03) {ival2_CAN=1;ival3_CAN=0;}// Connected to grid-not charging---assumed as charging complete
    }

    //Print the values of variables charging complete, CB_close request and reset setpoint
    UARTprintf("CAN_1 Receive= %d  %d  %d
", ival2_CAN,ival3_CAN,ival4_CAN);
}

//Earth leakage current logic-
//if input voltage greater than 2.5V, Earth leakage fault generated
//suitable value of input resistor to be selected so that current trip limit is converter to 2.5V

if(ival5_ADC<I_DC_Set)  ival5=0;//logic to detect if the current value is above setpoint
else ival5=1;

// The Communication between Protection device and Charger can be described in following states.
// "Wait state" : when there are no command from Charger to Protection device.
// "Precharging state" : when the Charger send command for close circuit breaker and start charging

// The Protection device will check fault from other device RCD and Ground monitor device.

// "Charging state" : when there is no fault in any other device the process of "charging" starts
// and while charging it will also check for faults continuously.

// "Complete state" : when the process of charging will get complete.

// "Failure state" : when some fault arise during the charging.

// "RCD trip limit setpoint state" : when Charger will send the new fault current setpoint.

// Switch statement is used for showing all these different state.

switch (state)
{
  case 1: UARTprintf(" Wait state\n");
  if (ival4_CAN==1)
    state=6; //current trip setpoint action state
  else if(ival3_CAN==1)
    state=2; // "CB_close_request" is received, move to state 2- precharging
  else state=1; // else remain in wait state
  break;
  case 2: UARTprintf(" precharging state\n"); // Precharging state
    // Go to fail state if tripIf there is any of charger fault,GMD fault and Earth leakage current fault
    if(ival5|ival6|ival1==1)
      {
        state=5; // Control will goto "fail" state
      }
    else
      // Else it will goto "charging" state
      {
        state=3;
        GPIOPinWrite(GPIO_PORTA_BASE,GPIO_PIN_4,0); //"PD status" to reset to "low" indicating " no fault"
      }
      break;
  case 3: UARTprintf(" charging state\n"); // Charging state
    GPIOPinWrite(GPIO_PORTA_BASE,GPIO_PIN_5,32); //enable "CB_close cmd", high- Close CB
    if(ival5|ival6|ival1==1)
      "
// During charging if there is charger fault, GMD fault and Earth leakage current fault
{
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_5, 0);  // immediate Open CB
    state = 5;  // go to fail state
}
else if (ival2_CAN == 1)  
    state = 4;  // else if (there is no fault and) charging complete signal is high
    goto "complete" state;
break;
case 4:  UARTprintf(" complete state\n");
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_5, 0);  // Open CB
    if (ival2_CAN == 1 & ival5 == 0 & ival6 == 0 & ival1 == 0)  // Complete state
        state = 1;
        else state = 5;
    break;
case 5:  UARTprintf(" fail state- charging failure\n");
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_4, 16);  // enable "PD status" to high indicating "fault"
    if (ival5 == 0 & ival6 == 0 & ival1 == 0)  
        
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_4, 0);
        state = 1;
    }
break;
case 6:  UARTprintf(" RCD trip limit setpoint state\n");
    val = CombineByte(rucMsgData[0], rucMsgData[1]);
    UARTprintf(" I_DC_Set= 0x%X, LowByte= 0x%02X, HighByte= 0x%02X \n", val, rucMsgData[0], rucMsgData[1]);
    I_DC_Set = val;
    state = 1;
    break;
default: state = 1;
}

// Protection device send the CAN messages to Charger (CAN1 to CAN0)
//
// CAN messages from Charger to Protection device
//-------------------------------------------------------------

// CAN MSG ID | description
//-------------------------------------------------------------
// 0x0FF | Immediate shutdown
//-------------------------------------------------------------
// 0x120 | Current value of fault current
//-------------------------------------------------------------
// 0x131 | Status of Protection device
//-------------------------------------------------------------

// CAN1 MSG generation and transmission
// CAN output ports are read to determine actual status of "CB close command" and "PD fault status"

//ival7-- Command to Circuit Breaker-- GPIO port E5- output
ival7 = GPIOPinRead(GPIO_PORTE_BASE,GPIO_PIN_5);  // Circuit breaker close
ival7=extractBit( ival7, 5);                      // "CB close command"

//ival8-- Protection Device fault status (0- no fault, 1- fault) GPIO port E4- output
ival8 = GPIOPinRead(GPIO_PORTE_BASE,GPIO_PIN_4);  //Protection device status
ival8=extractBit( ival8, 4);                      //"PD fault status"

if(ival8==1) // If there is fault in Protection device
{
  for(i=1;i<9;i++) ucMsgData2[i]=0x00;   // Reset the CAN1 data
  sCANMessage2.ulMsgID=0x00FF&0xFF;    // MSG id is changing,so used masking
  ucMsgData2[i]=0x02;

  // Print a message to the console showing the message count and the
  // contents of the message being sent.

  UARTprintf("CAN_1 Sending Msg ID=0x%08X ", sCANMessage2.ulMsgID);
  UARTprintf("msg: 0x%02X %02X %02X %02X %02X %02X %02X %02X\n", ucMsgData2[0], ucMsgData2[1], ucMsgData2[2], ucMsgData2[3], ucMsgData2[4], ucMsgData2[5], ucMsgData2[6], ucMsgData2[7]);
  //
}
// Clear the flag that indicates that message 3 has been sent. This flag will be set in the interrupt handler when a message has been sent using message object 3.
//
g_bMsgObj1Sent = 0;

// Send the CAN message using object number 2, This function will cause the message to be transmitted right away.
//
CANIMessageSet(CAN1_BASE, 2, &sCANMessage2, MSG_OBJ_TYPE_TX);

// Wait for the indication from the interrupt handler that message object 1 is done, because we are re-using it for another message.
//
while(!g_bMsgObj1Sent)
{
}

// Now wait 1 second before continuing
//
SimpleDelay();
x=ExtractLowByte ( ival5_ADC);
y=ExtractHighByte( ival5_ADC);
//Assign the data in CAN MSG
for(i=1;i<9;i++) ucMsgData2[i]=0x00; //reset the CAN1 data
ucMsgData2[0]=x;
ucMsgData2[1]=y;
ucMsgData2[2]=0x11; //Change to lower byte of DC fault current
ucMsgData2[3]=0x11; //Change to upper byte of DC fault current
sCANMessage2.ulMsgID = 0x0120; // Set the CAN message ID for sending current value of fault current

// Print a message to the console showing the message count and the contents of the message being sent.
//
UARTprintf("CAN_1 Sending Msg ID=0x%08X ", sCANMessage2.ulMsgID);
UARTprintf(" msg: 0x%02X %02X %02X %02X %02X %02X %02X %02X",
ucMsgData2[0], ucMsgData2[1], ucMsgData2[2], ucMsgData2[3],
ucMsgData2[4], ucMsgData2[5], ucMsgData2[6], ucMsgData2[7]);

//
// Send the CAN message using object number 2. This function will cause
// the message to be transmitted right away.
//
CANMessageSet(CAN1_BASE, 2, &sCANMessage2, MSG_OBJ_TYPE_TX);

else // If there is no fault in Protection device
{
for(i=1;i<9;i++) ucMsgData2[i]=0x00;  //reset the CAN1 data
sCANMessage2.ulMsgID=0x0131;            //Set the CAN message ID for sending status of
Protection device
ucMsgData2[0]=state;
// Print a message to the console showing the message count and the
// contents of the message being sent.
//
UARTprintf("CAN_1 Sending Msg ID=0x%08X ", sCANMessage2.ulMsgID);
UARTprintf(" msg: 0x%02X %02X %02X %02X %02X %02X %02X %02X",
ucMsgData2[0], ucMsgData2[1], ucMsgData2[2], ucMsgData2[3],
ucMsgData2[4], ucMsgData2[5], ucMsgData2[6], ucMsgData2[7]);

// Send the CAN message using object number 2. This function will cause
// the message to be transmitted right away.
//
CANMessageSet(CAN1_BASE, 2, &sCANMessage2, MSG_OBJ_TYPE_TX);
}

//
// Wait for a second before continuing
//
SimpleDelay();

// Check the error flag to see if errors occurred
//
if(gt_bErrFlag2)
{
    UARTprintf(" error - cable connected? \n");
}
else
{

    // If no errors then print the count of message sent
    //
    UARTprintf(" CAN_1 count = %u \n", gt_ulMsgCount2);
}

// Increment the value in the message data.
//
// CAN1 MSG sent
UARTprintf("\n loop end \n\n");
} //while loop
} //end main