Firmware Design Document

*for*

ETHIIC (ETH to I2C)

*a derivative of*

IO Extender Module

COPYRIGHT 2014VCortex Ltd ©

This document contains valuable and confidential information proprietary to VCortex Ltd. The recipient of this document shall not disclose its contents, in whole or in part, to any third party without VCortex’ prior written approval.

|  |  |  |
| --- | --- | --- |
| *Version* | *Date* | *Description* |
| 1 | Nov 15, 2014 | Basic version |

Contents

[Firmware Design Document 1](#_Toc414358770)

[*for* 1](#_Toc414358771)

[ETHIIC (ETH to I2C) 1](#_Toc414358772)

[*a derivative of* 1](#_Toc414358773)

[IO Extender Module 1](#_Toc414358774)

[1. General 3](#_Toc414358775)

[1.1. Scope 3](#_Toc414358776)

[1.2. Overview 3](#_Toc414358777)

[2. Detailed Design 4](#_Toc414358778)

[2.1. Module Description 4](#_Toc414358779)

[2.2. LWIP – Light Weight TCP/IP stack 4](#_Toc414358780)

[2.3. HTTP Server module 5](#_Toc414358781)

[2.4. File System (fs) module 5](#_Toc414358782)

[2.5. UDP Handler 5](#_Toc414358783)

[2.6. I2C handler 6](#_Toc414358784)

# General

## Scope

This document describes the SW that was developed for the ETHIIC module, which runs on the TIVA microprocessor.

## Overview

The ETHIIC (Ethernet Integrated InterCom) is a bridge that converts Ethernet communication to I2C communication, to enable SW running on the main computer to control I2C controlled devices.

The ETHIIC is a partial implementation of the IOX (IO Extender module) that includes some more interfaces implementation. The GPIO and IIC are fully implemented in the ETHIIC board, as a derivative of the IOX module.

The ETHIIC can be connected as a single or multiple configuration, controlled by host computer through UDP commands. Therefore, the ETHIIC module bears a unique logical name, unique IP and unique MAC address.

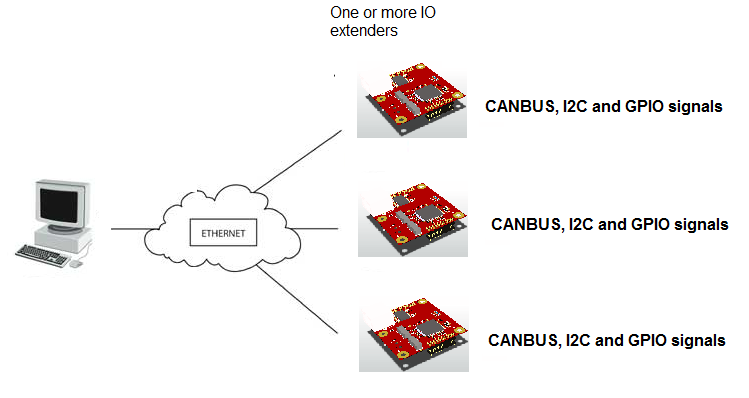


Figure 1: IOX multiple board configuration

# Detailed Design

## Module Description

The ETHIIC provides an interface between the main computer and the I2C bus.

The SW is combined from several infrastructure packages to handle the communication with the host PC, and a specific I2C handler to complete the host computer requests.

Following figure shows the block diagram of the ETHIIC firmware:

Main Computer

LWIP

UDP Handler

I2C Device

HTTP Server

I2C Handler

File Server

Figure 2: ETHIIC Firmware block diagram

## LWIP – Light Weight TCP/IP stack

lwIP is a small independent implementation of the TCP/IP protocol suite. The focus of the lwIP TCP/IP implementation is to reduce resource usage while still having a full scale TCP. This makes lwIP suitable for use in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code ROM.

Main features include:   
- Protocols: IP, ICMP, UDP, TCP, IGMP, ARP, PPPoS, PPPoE   
- DHCP client, DNS client, AutoIP/APIPA (Zeroconf), SNMP agent (private MIB support)   
- APIs: specialized APIs for enhanced performance, optional Berkeley-alike socket API   
- Extended features: IP forwarding over multiple network interfaces, TCP congestion control, RTT estimation and fast recovery/fast retransmit   
- Addon applications: HTTP server, SNTP client, SMTP client, ping, NetBIOS nameserver

lwIP is licenced under a BSD-style license:  <http://lwip.wikia.com/wiki/License>.

In the ETHIIC firmware we are using LWIP in UDP mode.

## HTTP Server module

The httpd module provides a rudimentary server-side-include facility which replaces tags of the form of <!--#tag--> in any file whose extension is .shtml, .shtm or .ssi with strings provided by an include handler whose pointer is provided to the module via function http\_set\_ssi\_handler().

Additionally, a simple common gateway interface (CGI) handling mechanism has been added to allow clients to hook functions to particular request URIs.

The HTTPD module uses File System module (fs) to read the .html, images and .ssi files.

## File System (fs) module

The file system data for the HTTPD application is included in enet\_fs.h

This file is generated by the makefsfile utility, using the following command:

makefsfile -i fs -o enet\_fs.h -r -h -q

If any changes are made to the static content of the web pages served by the application, this command must be used to regenerate enet\_fs.h in order for those changes to be picked up by the web server.

## UDP Handler

The UDP handler analyses the content of the host computer UDP transmissions and calls the I2C handler to complete them.

Each UDP message is checked for checksum validity, and updated in the log structure.

Following specific actions are taken for the various messages:

* For INIT message:
  + Check if the logical name is the same as the local name
  + If yes send the INIT message back to the host computer
* For INIT\_IICmessage:
  + Fetch the I2C speed from the message
  + Initialize the master I2C device of the TIVA MPU by calling I2C\_Handler->I2CMasterInitExpClk() function.
* For READ\_BYTES message:
  + Call I2C\_handler->i2c\_read() to perform the read command
  + Return the received bytes to the host computer, with the appropriate status as received from the I2C\_handler->i2c\_read() function.
* For WRITE\_BYTES message:
  + Call I2C\_handler->i2c\_write() to perform the write command
  + Return a message to the host computer, with the appropriate status as received from the I2C\_handler->i2c\_write() function.
* For WRITE\_AND\_READ\_BYTE message:
  + Call I2C\_handler->i2c\_write() function to write the byte to the slave
  + If received status is OK, Call I2C\_handler->i2c\_read() function to read a byte from the slave
  + Return a the received byte to the host computer, with the appropriate status as received from the I2C\_handler->i2c\_read() function.
* For WAKEUP\_IIC message:
  + Call I2C\_handler->wakeupI2cBus() function
  + Return an acknowledge message to the host computer

## I2C handler

I2C handler module manages the I2C device on the TIVA MPU.

Following functions are implemented:

**I2CMasterInitExpClk():**

This function calculates the clock dividers that will be right for the desired speed, and sets the I2C clock accordingly.

* Enable the device before doing anything else.
* Compute the clock divider that achieves the fastest speed less than or equal to the desired speed. The numerator is biased to favor a larger clock divider so that the resulting clock is always less than or equal to the desired clock, never greater.
* Check to see if this I2C peripheral is High-Speed enabled. If yes, also choose the fastest speed that is less than or equal to 3.4 Mbps.

**waitIICBus():**

This is a special wait function that is called after every setting of the I2C registers.

It waits for the ready signal to go up. If we don’t get a ready signal within certain time limit, an error is issued to the caller.

* Wait for the current byte to be transmitted, by checking the interrupt register. While waiting, check that wait is not greater than 20mSec. If 20mSec has passed, return I2CMCS\_TIMEOUT.
* Check that master busy is off. Again wait up to 20mSec. If 20mSec has passed, return I2CMCS\_TIMEOUT.
* Read raw interrupt status register for errors. If Nack bit is on, return I2CMCS\_NO\_ACK\_FROM\_SLAVE.
* Else, return *I2CMCS\_OK.*

**I2c\_write():**

This function writes N bytes to a designated slave.

The function does the following:

* Write slave address into the I2C master register. The slave address is not written to the bus until first transmit word is set in the register
* Put first byte in the I2C master data register
* Put I2C\_MASTER\_CMD\_BURST\_SEND\_START code in the I2C master control register
* Wait for I2C master – call waitIICBus()
* For the next bytes, less the last one, do
  + Put byte in the I2C master data register
  + Put I2C\_MASTER\_CMD\_BURST\_SEND\_CONT code in the I2C master control register
  + Wait for I2C master – call waitIICBus()
* Put the last byte in the I2C master data register
* Put I2C\_MASTER\_CMD\_BURST\_SEND\_FINISH code in the I2C master control register
* Wait for I2C master – call waitIICBus()

**I2c\_read():**

This function writes N bytes to a designated slave.

The function does the following:

* Write slave address into the I2C master register. The slave address is not written to the bus until first receive command is set in the control register
* Put I2C\_MASTER\_CMD\_BURST\_RECEIVE\_START code in the I2C master control register
* Wait for I2C master – call waitIICBus()
* Read first byte from I2C master data register
* For the next bytes, less the last one, do
  + Put I2C\_MASTER\_CMD\_BURST\_RECEIVE\_CONT code in the I2C master control register
  + Wait for I2C master – call waitIICBus()
  + Read byte from I2C master data register
* Put I2C\_MASTER\_CMD\_BURST\_RECEIVE\_FINISH code in the I2C master control register
* Wait for I2C master – call waitIICBus()
* Read last byte from I2C master data register

**wakeupI2CBus():**

This function performs special sequence to start a slave device on the I2C bus.

The function does the following:

* Disable I2C master device on the TIVA MPU
* Set I2C pins – SDA and SCL – as GPIO pins
* Set both pins to 1
* Wait 10 I2C clocks
* Set SCL to 0
* Wait 10 I2C clocks
* Set SDA to 0
* Wait 10 I2C clocks
* Do 10 times:
  + Set SCL to 1
  + Wait 10 I2C clocks
  + Set SCL to 0
  + Wait 10 I2C clocks
  + On the 6th time set SDA to 1
* Sleep 1 mSec
* Restore setting of pins as SCL and SDA (I2C definitions)
* Enable I2C Master device on the TIVA MPU