Software Report for TEAM 13: Linear Actuator Controller for Rocket Thrust Vector Control

**Overview of Each Software Module**

***Gaia***

MasterControl.java

This module is responsible for the main Gaia window that displays the Olympus system status to the user. The status lights tell whether the board is streaming data, whether there exists a network connection, if the software is recording data, and which board is currently active. This module also acts as the central node that connects to other command windows, including the LAC Control panel. Every command window that connects to this module includes MasterControl in order to use its getCommandNetwork().sendMessage function and send commands to Olympus.

LACControl.java

This module contains the actual control panel for the LAC. It uses Java Swing to create the GUI elements such as the buttons and the slider, and sends each command via the getCommandNetwork().sendMessage() function. The code is generated as the user drags and drops various GUI elements onto the template, but manual coding had to be done in order to enable functionality.

***Olympus***

OlympusMain.Java

OlympusMain is the main class of the software. It is run when the program starts. OlympusMain is responsible for loading overarching system configs, starting the various modules, launching the GUI and logging system-level information. Additionally it serves as a bridge between various parts of the software, allowing the easy flow of data and commands from, for instance, the Communication Manager to the GUI.

Ethernet.java and EthernetManager.java

Ethernet.java is the “header” file for EthernetManager.java that OlympusMain uses to terminate or check if the EthenernetManager module is running. EthernetManager EthernetManager manages the Ethernet module, which loads the config file for the Ethernet that contains important information such as the packet protocol structure. The module also does a health check and lets Olympus know if the module is running or not.

DAQManager.java

This module is central to the functionality of the ethernet networking back end. As it loads the configuration files for the various boards, it determines if the board uses an ethernet or serial connection. This is one of the specifications given by the customer to create a modular addition that works with the preexisting hardware. In order to send a command to the board, it detects if there is an ethernet connection, whereby it calls the EthernetDataPacket module to construsct the packet and uses EthernetTransmitWorker to send the command over the specified port to the board’s IP address.

EthernetReceiver.java

Data is easily received by this module from the board through an ethernet connection and sent by the DAQManager to Gaia via the getTelem([Telemetry\_ID]).setData([Data], [time]) function. In order to send data through the setData function, it has to parse the bytearray structure (detailed below) to separate the telemetry id and data payload of the packet. This information will be used to plot the information along a time axis.

EthernetDataPacket.java and EthernetTransmitWorker.java

EthernetDataPacket constructs a data packet in the structure detailed below and EthernetTransmitWorker establishes a socket connection with the board, given the port and address of the specified board. The packets are sent as datagrams through a UDP connection.

Transmitter.py

This python program is used for testing the telemetry functionality via ethernet of Olympus by streaming data in the packet protocol format that it is expected to receive. For testing purposes, transmitter.py is connected to localhost on port 2005, and sends the packet as a bytearray to Olympus in the following format:

|  |  |  |
| --- | --- | --- |
| Byte Number | Function | Example |
| 0 | start byte | 0x85 |
| 1 | length | 0x10 |
| 2 | packet\_type | 13 (for telemetry) |
| 3 | telemetry id | 0x04 |
| 4..7 | data payload | 0x11223344 |
| 8..11 | timestamp | 0x34342584 |
| 12 | end byte | 0x80 |

The expected bytearray received by Olympus would thus be: [0x85, 0x10, 0x0D, 0x04, 0x11, 0x22, 0x33, 0x44, 0x34, 0x34, 0x25, 0x84, 0x80]

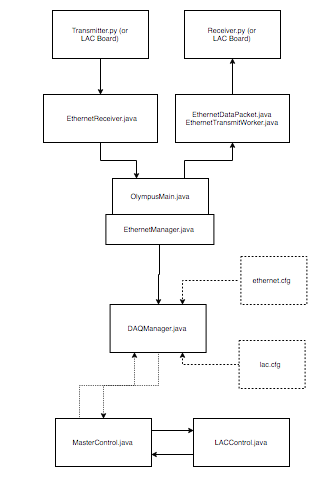
Receiver.py

This python program is used for testing the command sending functionality of the Olympus ethernet module. For testing purposes, receiver.py is connected to localhost on port 2004 and echos any command received from Olympus. Both transmitter.py and receiver.py are used to test the functionality of the ethernet module without the presence of a physical board.

***IRISConfig***

Ethernet.cfg and lac.cfg

This configuration file specifies the packet start, packet end, telemetry packet ID, and board configuration location. The LAC configuration file specifies that this board accepts an ethernet connection, as well as the IP address and the telemetry and command ports. The LAC configuration also contains a list of commands accepted by the board.

**Flow Chart for Ethernet and GUI**

**Dependencies, Installation, and Usage Instructions**

* The user must create an account with the BURPG Trac revision control system. This will allow the user to pull from the various repositories.
* The Netbeans IDE and SVN are required dependencies; they must be installed in order to pull from the repositories and run the programs. For Windows, Linux, and OSX based machines, the installation instructions may vary and it is up to the user to install the appropriate dependencies.
* Once the dependencies are resolved, the following repositories and their specified branches must be pulled from the BURPG Trac site: Gaia (Ethernet branch), Olympus (Ethernet branch), and IRISConfig (master).
* Once the repositories are pulled, the user must modify the **lac.cfg** file located in IRISConfig/Olympus/boardcfgs as it contains the target IP address of the LAC controller. This address must match the IP address of the LAC controller.
* With the configuration file appropriately modified, the program must be run in this order: Olympus, then Gaia -- as Gaia requires Olympus to be running in order to connect to it and will not reconnect to it if it is not already running.
* With Olympus and Gaia running, the user must enable both Network Stream and Board Stream on the main Gaia control panel window. Now, both data acquisition and sending commands are possible.
* On the main Gaia control panel window, under the “View” tab, there is an LAC Control option. This opens the LAC Control GUI that will allow the user to send commands.
* Also on the main Gaia control panel window, under the “View” tab, there is a Telemetry option. Searching for the correct Telemetry ID and double clicking on it will reveal the telemetry data received from the LAC board.

**Overview of Firmware**

**Firmware IDE**

Firmware dev was done with ChibiStudio: <https://sourceforge.net/projects/chibios/files/ChibiStudio/>. Preview 12 was used for this project, however, the most recent version will always have bug fixes to the OS and is the version that should be used.

ChibiStudio installation instructions (taken from the README):

There is no setup nor settings to do:

1) Unpack the archive under C:\ in order to have C:\ChibiStudio as

installation path.

2) Copy the shortcut "C:\ChibiStudio\Chibi Studio" on your desktop.

3) Use the shortcut to launch ChibiStudio.

The programmer we used is ST-Link/v2 and the IDE has native support for it, so there is no setup required beyond installing drivers. To import the project, just import the project as you would any other eclipse project. ChibiStudio will take care of the project links.

Documentation on ChibiOS:

* ChibiOS general - <http://www.chibios.org/dokuwiki/doku.php>
* AWESOME ChibiOS guide: <http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:start>
* Kernel - <http://chibios.sourceforge.net/docs3/rt/index.html>
* HAL - <http://chibios.sourceforge.net/docs/hal_stm32f4xx_rm/>

**Programming Tool**

The programming tool we used is the ST-Link/v2. It can be purchased for <= $20 from most electronics distributors. The only installing required is installing the drivers, which can be found at: <http://www2.st.com/content/st_com/en/products/development-tools/hardware-development-tools/development-tool-hardware-for-mcus/debug-hardware-for-mcus/debug-hardware-for-stm32-mcus/st-link-v2.html>. It must be plugged into the board using the harnessing supplied with the LAC.

Firmware folder:

* .dep/ - dependencies folder for gcc-arm
* Board/ - folder containing board config files
  + Board.mk - used by the ChibiOS makefile system
  + Board.c - generated by the ChibiOS makefile system
  + Board.h - generated by the ChibiOS makefile system
  + cfg/
    - Board.chcfg - XML board configuration file
* Build/ - this is where the generated build files by the compiler are placed
* Debug/ - this is the folder where the ChibiStudio debug configuration is placed
* Demos/ - Some python and RTOS modules to test ethernet connectivity
* Include/ - include directory
  + Adc\_lac.h - Header file for the ADC subsystem. It contains global declarations of the controllers and DAQ elements
  + Bldc.h - Contains declarations of motor control functions as well as global declarations of DAQ elements
  + Cmds.h - Header file containing structure information for the interactive shell and associated commands.
  + Comm.h - header file containing global declarations for the RTOS DAQ and Command server
* Drv8305.h - header file contains global declarations for functions to deal with the motor driver.
  + Packet\_codes.h - header file containing DAQ id’s for the different telemetry elements.
  + PI\_controller.h - header file declaring the PI controller functions
  + Utils.h - header file declaring various utility functions
* src/ - source code directy
  + Adc\_lac.c - This file contains functions that initialize ADC1 and ADC2. There is an interrupt for ADC1 that samples the ADC at 16 kHz, and every 16 samples triggers and interrupts. The samples are then filtered, and a control decision is made. ADC2 is the auxiliary ADC, and runs at 200 Hz, cycling through 5 different auxiliary channels.
  + Bldc.c - This is the file that contains the code to drive the BLDC motor. It has an input change interrupt on the hall effect sensor pins that trigger the commutation sequence and set the duty cycle appropriately. It also uses an input capture channel to capture the half period time of a hall effect sensor, so motor RPM can be calculated. It also has setter functions to change BLDC characteristics.
  + Cmds.c - File containing all the shell commands. The shell command documentation can be found in the user manual.
  + Drv8305.c - contains default initialization registers, as well as routines to initialize, read, and write to the DRV8305 chips.
  + Gse\_comm.c - Contains the source for the DAQ server. There are two threads this module creates, GSEListenerThread and GSESenderThread. GSEListenerThread opens a UDP port and listens for commands from either the flight computer or the ground software. It then parses the packet and acts accordingly. The GSESenderThread reads from a thread-safe mailbox telemetry elements to transmit, and then creates a packets and sends it to either the flight computer or the ground software over the telemetry port.
  + PI\_controller.c - Code that initializes and steps a PI controller. It takes a pointer to a struct, so multiple PI controllers can be run off of this.
  + Utils.c - contains the following functions:
    - float averagef(float \*buf,uint8\_t num,uint8\_t incr)
    - uint32\_t average32(uint32\_t \*buf,uint8\_t num,uint8\_t incr)
    - uint32\_t average16(uint16\_t \*buf,uint8\_t num,uint8\_t incr)
    - char UART\_CmdEquals(char \* UART\_Buffer,const char \* str,char len)
    - void get\_time(uint8\_t \* txBuf,int \* write\_ptr)
    - void write\_str(uint8\_t \* txBuf,int \* write\_ptr,const char \* str)
    - double atod(char \*s)
* web/ - contains source files for various web interfaces
  + Netstream.c - a wrapper for opening a ChibiOS command shell over a TCP connection
  + Netstream.h - header file for netstream.c
  + N.B. the netstream files were taken from: <http://forum.chibios.org/phpbb/viewtopic.php?f=14&t=1342>
  + Test\_ip.c - a \*DEPRECIATED\* file containing some code that tests ethernet connectivity and throughput.
  + Web.c - opens an HTML page at 192.168.1.20 and prints a simple message so one can verify connectivity with the board.
* Chconf.h - ChibiOS Kernel configuration
* Halconf.h - ChibiOS Hardware Abstraction Layer configuration
* Halconf\_community.h - ChibiOS HAL configuration for community provided drivers
* Lwipopts.h - Local options for the LWIP stack
* Mcuconf.h - MCU specific configuration options
* Mcuconf\_community.h - MCU specific configuration options