MobilEye Implementation

Jason Hwang (jyh37)

Autonomous Systems Lab – Skynet

Professor Mark Campbell

Table of Contents

I. Introduction

The goal of using the MobilEye vision sensor is to detect information about vehicles, pedestrians, lanes, and signs in front of Skynet to help understand the car's environment. For the MobilEye sensor to be implemented into Skynet, the incoming data packets must be processed quickly and completely and time stamped so that the data is synchronized with the rest of the Skynet system.

MobilEye 560

The MobilEye 560 is a commercial product that uses computer vision to identify the vehicle's surroundings. The sensor uses a single camera to detect obstacles (vehicles, pedestrians, cyclists), lanes, and signs. The camera unit is mounted on the windshield and is connected to an EyeWatch display that is placed on the dashboard.

The unlocked version of MobilEye used by Skynet is capable of outputting vast amounts of data related to the conditions on the road. The data outputs information such as the number of obstacles on the road and their positions, the coefficients of lane parameters, speed limits, etc.

Figure 1 – MobilEye 560 and EyeWatch

CAN Messages

MobilEye outputs data to Skynet through the CAN protocol. The CAN protocol is an industry standard used by many automobile companies to relay messages between various systems within a vehicle. For an example, the car's headlights, turn signals, and door locks are all controlled by CAN messages. Similarly, MobilEye uses the CAN protocol for its own data output. MobilEye uses a CAN header of 11 bits and a baud rate of 500 kbps. The CAN header contains the 'message ID', which specifies what type of CAN message was just received. Knowing the message ID allows the user to understand how to dissect the data by following the message protocols stated in the MobilEye manual (example shown in figure 2). A 'length' variable tells the number of bytes of data in the packet and each packet can have up to eight bytes of data. Upon the Skynet servers receiving the data packets, the servers will then parse the data according to the message protocol to extract the information.

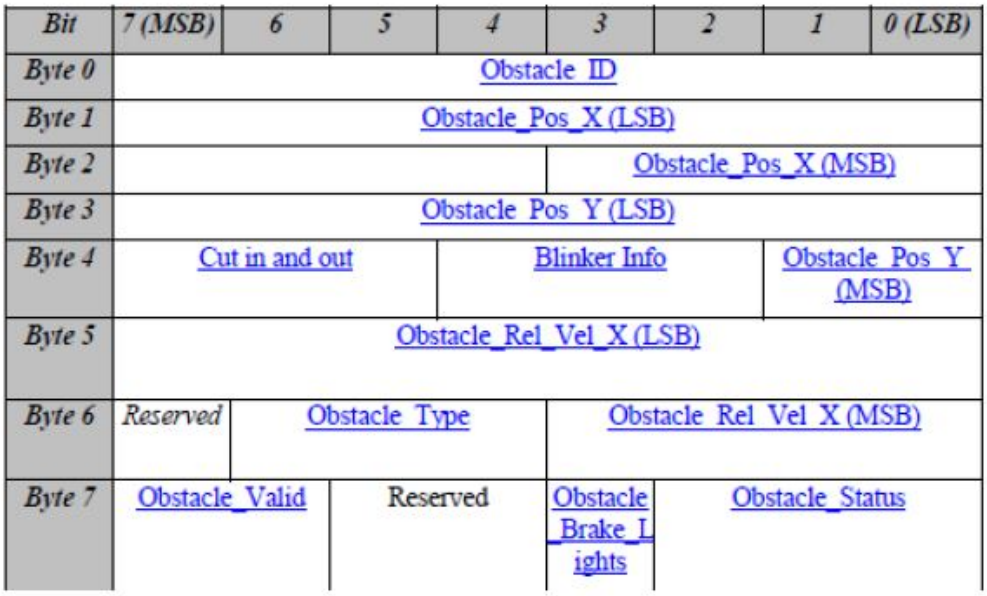

Figure 2 – Message Protocol for Message ID 0x739 (Obstacle Data A)

For MobilEye to be implemented into Skynet, a method for processing CAN packets must be used. Therefore, all hardware and software chosen are compatible with the CAN protocol.

Implementation Overview

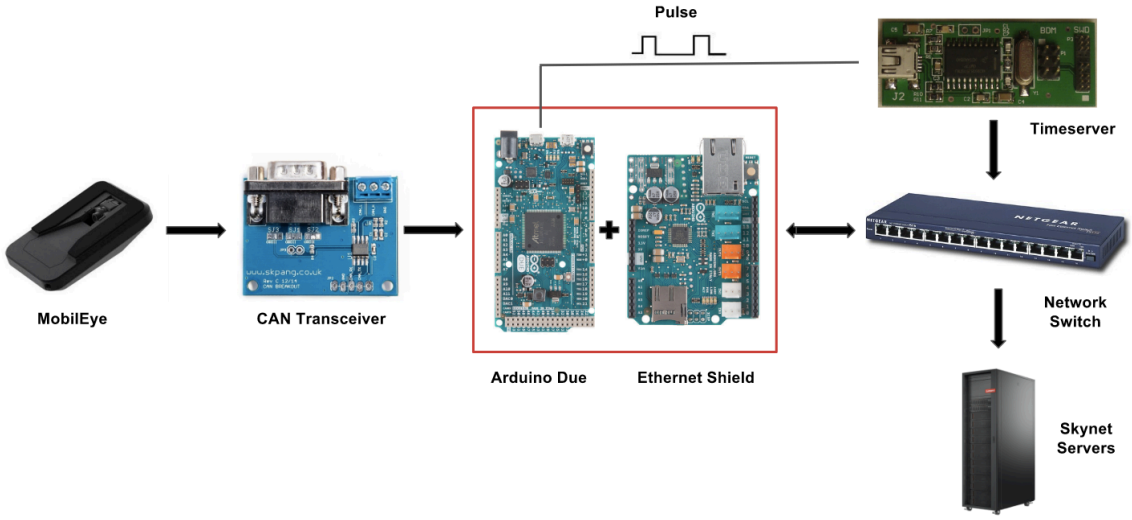

Figure 3 – Implementation Schematic

The implementation for MobilEye consists of:

MobilEye – MobilEye 560 vision sensor outputs data packets using the CAN protocol

CAN Transceiver – converts CAN packets so they are compatible with Arduino Due

Arduino Due – microcontroller unit (MCU) (attached to Ethernet Shield)

Ethernet Shield – provides MCU with ethernet capabilities (attached to Arduino Due)

Timeserver – Skynet's central timing system, sends out time packets

Pulse – timing pulse to notify MCU's of incoming time packet, used for synchronization

Network Switch – manages network traffic

Skynet Servers – parses sensor data

The implementation scheme starts at the MobilEye sensor which outputs data packets using the CAN protocol. The data packets are then received by the CAN transceiver, which converts the CAN data packets into a compatible form to be used by the Arduino Due microcontroller. Within the Arduino Due is a built in CAN controller which processes the data packets to extract the message ID, length, and data. The Arduino Due appends the timestamp to each data packet to synchronize each packet with Skynet and then sends the data packet to the Ethernet Shield. The Ethernet Shield then

multicasts the data packets to the network switch. The network switch finally sends the data packets to the proper Skynet servers.

At the same time, the timeserver sends the current time of the Skynet system to the network switch once every second. The network switch sends the time packets to all devices connected to the network switch, including the Arduino Due. Before the timeserver sends out the time, the timeserver firsts sends a pulse to each of the MCU's. The pulse is a physical connection that goes directly from the timeserver to each MCU. The pulse notifies the MCU's of an incoming time packet for device synchronization.

II. MCU and Hardware

Arduino Due

The Arduino Due was chosen as the microcontroller (MCU) to implement MobilEye because of the Due's fast processing speed (84 MHz clock), relatively low cost (~$30), ease of use, open source libraries/code, expandable functions (ie. Ethernet Shield), CAN capabilities, and numerous input/output ports. The Due contains 54 digital I/O pins and 12 analog inputs, of which all digital pins have interrupt capabilities (interrupts covered in more depth in Section III: Timeserver). The Due operates on 3.3V so all inputs into the Due should not be higher than 3.3V or it may damage the board. To power the Due, it is recommended to supply the board with 7-12VDC. The Due used for MobilEye receives 12V from the RJ45 cable containing the timeserver pulse. Since the timeserver supplies 24V, voltage dividers were used to obtain 12V from the 24V.

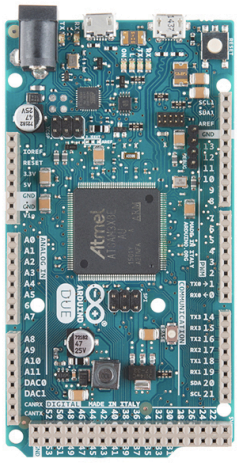

Figure 4 – Arduino Due

CAN Transceiver/Controller

A CAN transceiver is a chip which converts the raw CAN_H (CAN High) and CAN_L (CAN Low) signals from the CAN source into a form that is compatible to be read with the CAN controller. Specifically, the CAN transceiver converts the differential CAN_H and CAN_L signals into a TX signal. A CAN controller receives the TX signal, sends a RX signal back to the transceiver, and then analyzes the TX/RX signals. The CAN controller's main purpose is to extract the message ID, length, and data from the TX/RX signals. Often times, the TX of the transceiver goes to the RX of the controller and the RX of the transceiver goes to the TX of the controller. A general hardware overview for implementing CAN is seen in figure 5.

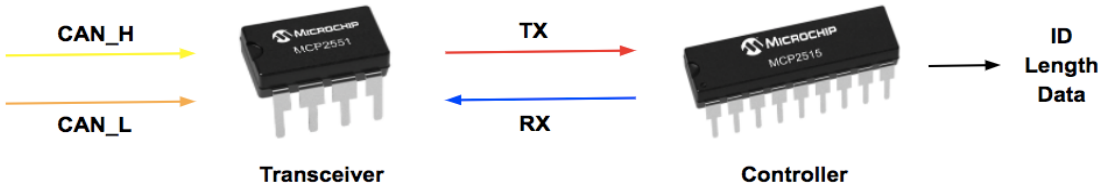

Figure 5 – General CAN Implementation

For the MobilEye's implementation, the MobilEye outputs the CAN_H and CAN_L signals via a D-Sub connector. The D-Sub is connected to a 3.3V CAN Bus Breakout Board which has an on-board MCP2551 transceiver. The breakout board then outputs the TX and RX to the Arduino Due. The Arduino Due has a built in CAN controller which provides the message ID, length, and data directly to the Due's processor. Note that the TX/RX signals are not crossed in this implementation. The TX from the breakout board goes to the TX on the Due and the RX from the breakout board goes to the RX on the Due. The MobilEye's CAN implementation is seen in figure 6.

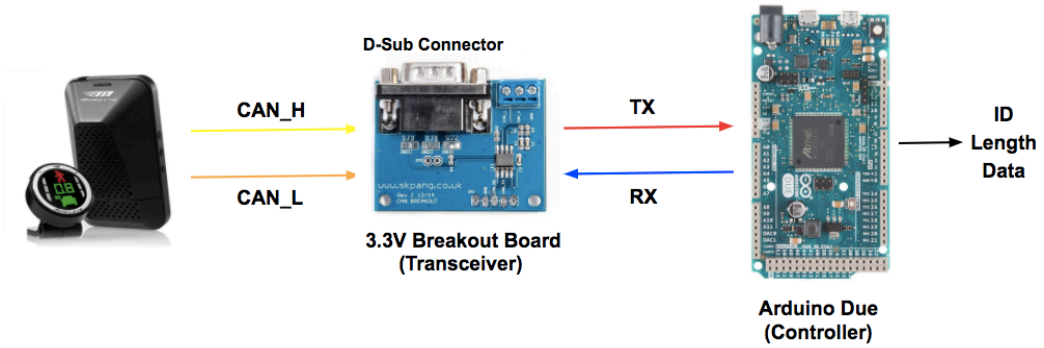

Figure 6 – MobilEye CAN Implementation

***Note that the default Arduino library does not support the protocols for the built in CAN controller. The CAN controller library for the Due needs to be downloaded from:** https://github.com/collin80/due_can

Performance

Upon the Arduino Due receiving a data packet from the transceiver, it takes 10-18 μs for the data packet to be ready to be sent to the network switch. Within this 10-18 μs, the controller processes the data and extracts the message ID, length, and data bytes, the timestamp is appended to the data packet, and the data packet is packaged into a 15 byte array. Since MobilEye data packets come in at a rate between 170-250 μs, there is sufficient time to process the data since it only takes 10-18 μs to process the packets and thus no packets are lost. ***Performance was slightly improved by changing the SPI clock settings in Ethernet3-master>src>utility>w5500.cpp to 84000000**

Prior to using the Arduino Due, an Arduino Ethernet was used to implement the MobilEye. The Arduino Ethernet lost half the data packets since it took over 400 μs to process a data packet. This is because the Arduino Ethernet has a lower clock speed (16 MHz) and doesn't have a built in CAN controller. The Arduino Ethernet required an external controller, which used Serial Peripheral Interface (SPI) to transmit the message ID, length, and data to the Arduino Ethernet. Since SPI required a few hundred μs to transmit the information, half the incoming MobilEye packets were lost.

III. Time Server

One of the main criteria for implementing MobilEye is to timestamp the data packets so they are synchronized with the rest of the Skynet system. This way, the Skynet servers know exactly when an event occurred once the data packet has been processed. For example, if MobilEye detects a pedestrian in front of the car, by time-stamping the data packet, Skynet knows the exact time when there was a pedestrian present. To synchronize MobilEye's data packets with Skynet's time, the Arduino Due must be able to obtain time packets from Skynet's timeserver.

The timeserver is located in the back of Skynet and the time is kept with an HC12 microcontroller.

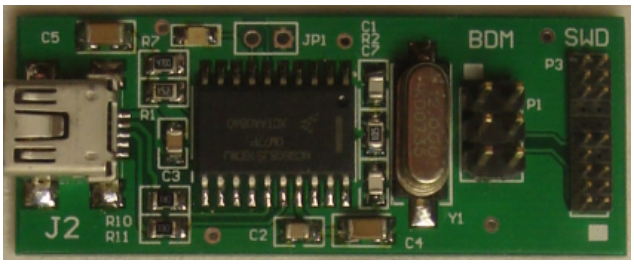

Figure 7 – Timeserver (HC12 MCU)

The timeserver sends Skynet's time to each device by sending time packets to the network switch. Therefore, it is essential for every MCU to have ethernet capabilities. In the case of the Arduino Due, the Ethernet Shield provides ethernet functionality and allows for both UDP and TCP protocols (timeserver uses UDP). The timeserver has a source IP address of 192.168.1.2 and sends time packets from port 30 on the timeserver to port 30 on the corresponding MCU. The time packets are broadcasted once every second, meaning that the time packets are sent to all devices on the network switch. The destination IP for broadcasting is 255.255.255.255 and is further explained in section IV.

Time Packet

Each time packet is composed of three fields: type, seconds and ticks. The type is one byte and represents the type of time packet that is sent. If the time packet was broadcasted automatically every second, the type is 0. If a time packet was requested by an MCU, the type is 1 (please refer to Adam Shapiro's documentation of the time server for more information).

The seconds keeps track of how many seconds have elapsed since Skynet has been operating. The ticks keep track of time more precisely where one tick represents 100 μs and is the smallest unit of time. Since the timeserver sends a time packet on the second, each time packet has a tick value of 0. Therefore, each MCU keeps track of the number of ticks that have elapsed using the MCU's own timer.

Figure 8 – Time Packet

Timing Pulse

To notify each MCU of when a time packet is sent so that the MCU can quickly update it's time to Skynet's time and minimize time delay, a timing pulse is used. The timing pulse is a physical signal that is connected to each device and the pulse is sent out once every second (1 Hz). The pulse begins 2 ms (20 ticks) prior to each elapsed second (ie. pulse sent at second = #, ticks = 9,980). Once the pulse ends 2 ms later, the time packet is broadcasted (ie. time packet broadcasted at second = #+1, ticks = 0).

The pulse is a differential pulse that is active low (ie. normally high and goes low once every second). However, the Arduino Due analyzes only one end of the differential pulse and is seen in figure 9.

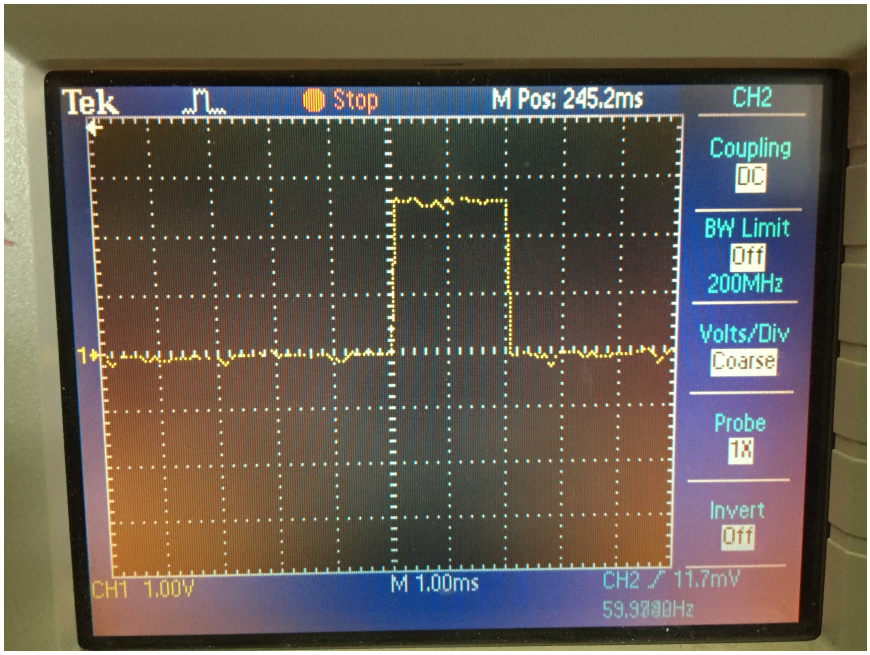

Figure 9 – Timing Pulse

Since the pulse the Arduino Due sees is active high, the Due waits for the pulse to drop low before checking to see if a time packet had been received. Since the pulse is done in hardware, interrupts are used to tell the MCU when the pulse has gone low. The pulse is connected to digital pin 2 on the Due and an interrupt is attached to the pin. The interrupt service routine (ISR) connected to pin 2 is called whenever the pulse changes states. Within the ISR, a variable is incremented each time the pulse changes states. Every two state changes means that the pulse has gone low and a new time packet is

ready (the first state change means the pulse has gone from low to high, the second state change means the pulse has gone from high to low).

Ideally, the ISR should be triggered whenever the state goes from high to low. However, this event on the Due was triggered incorrectly so the more complicated procedure mentioned above using a state change as a trigger was used.

The pulse is obtained directly from the timeserver by splicing the existing timeserver's output. The timeserver output includes +24V, GND, Pulse+, and Pulse-. The spliced connection uses an RJ45 cable to carry the signals to the front of the car where the Arduino Dues are located. The wire provides both the timing pulse as well as power to the Arduino Dues. The Arduino Due uses Pulse- (figure 9) only and ignores Pulse+.

IV. Multicasting

Multicasting is an essential part in implementing the MobilEye since all sensors on the Skynet network utilize multicasting to send their data to the proper servers. To understand multicasting, it is helpful to also understand the two other main routing schemes: unicasting and broadcasting.

Routing Schemes

Unicast - data packets are sent to only one IP address destination and is a one-to-one routing scheme

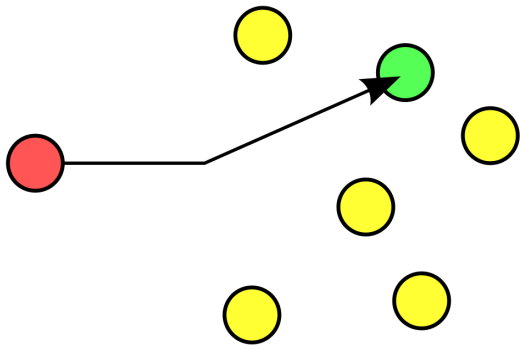

Figure 10 - Unicast

Broadcast – data packets are sent to all devices on the network (ie. all devices connected to the network switch), broadcasting uses a special IP address 255.255.255.255

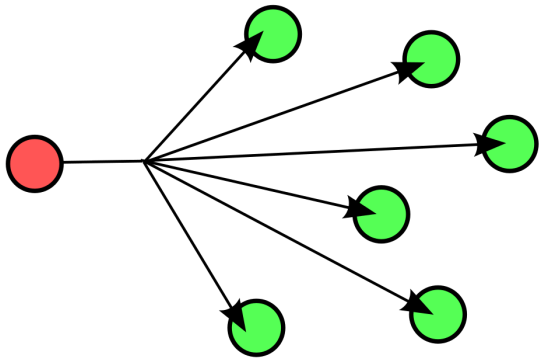

Figure 11 – Broadcast

Multicast – data packets are sent to a multicast group where multiple clients can be subscribed to the multicast group. Only clients subscribed to the multicast group will receive the data packets. Data packets are sent out only once from the source. Multicasting IP addresses range between 224.0.0.0 through 239.255.255.255

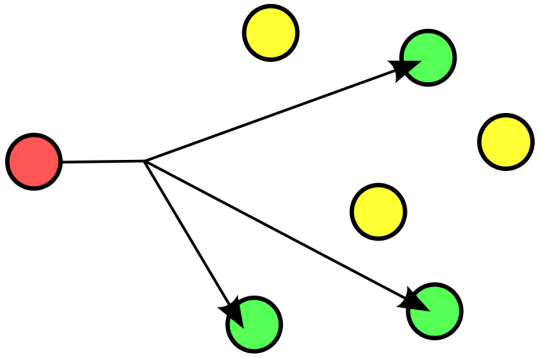

Figure 12 – Multicast

The multicast group IP address assigned to MobilEye is 239.132.1.45 and the data packets are sent to port 30045 on the Skynet servers. Therefore, any Skynet server subscribed to 239.132.1.45 will receive MobilEye data packets on port 30045.

***Using multicasting requires the library from**: https://github.com/sstaub/Ethernet3

Ethernet Shield

To provide the Arduino Ethernet with ethernet capabilities to receive time packets and to multicast MobilEye data packets, an Ethernet Shield was used. The Ethernet Shield is capable of up to 8 simultaneous connections where each connection is called a 'socket' (IP address + port number). For the MobilEye implementation, two sockets are used, one for the broadcasted time packets and one for multicasting MobilEye packets. The Ethernet Shield is powered with 5V and uses a Wiznet W5500 ethernet controller chip. The Ethernet Shield communicates with the Arduino Due through SPI.

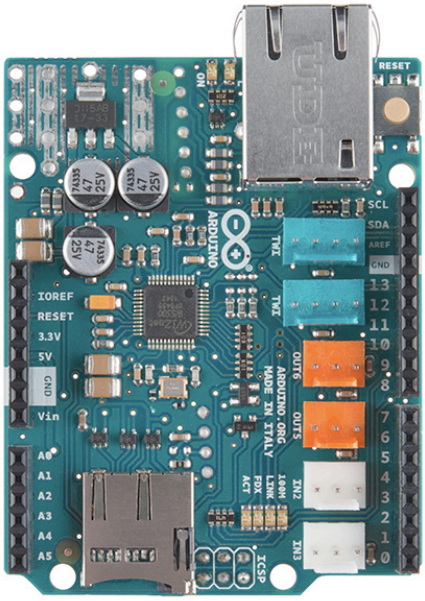

Figure 13 – Ethernet Shield v2

Ethernet Configurations

The software required to implement multicasting requires configuring all ethernet fields properly. The unique IP address assigned to the MobilEye's Ethernet Shield is 192.168.1.85 and should not be used by any other device on the subnet. The MAC address of the Ethernet Shield is {0x90, 0xA2, 0xDA, 0x10, 0xE9, 0xF6} and is written on a sticker attached to the shield. The local port on the Arduino Due is port 30 (incoming time packets) and the destination port is 30045 (outgoing MobilEye packets). The IP address of the timeserver is 192.168.1.2 and the IP address for multicasting MobilEye packets is 239.132.1.45.

V. Results

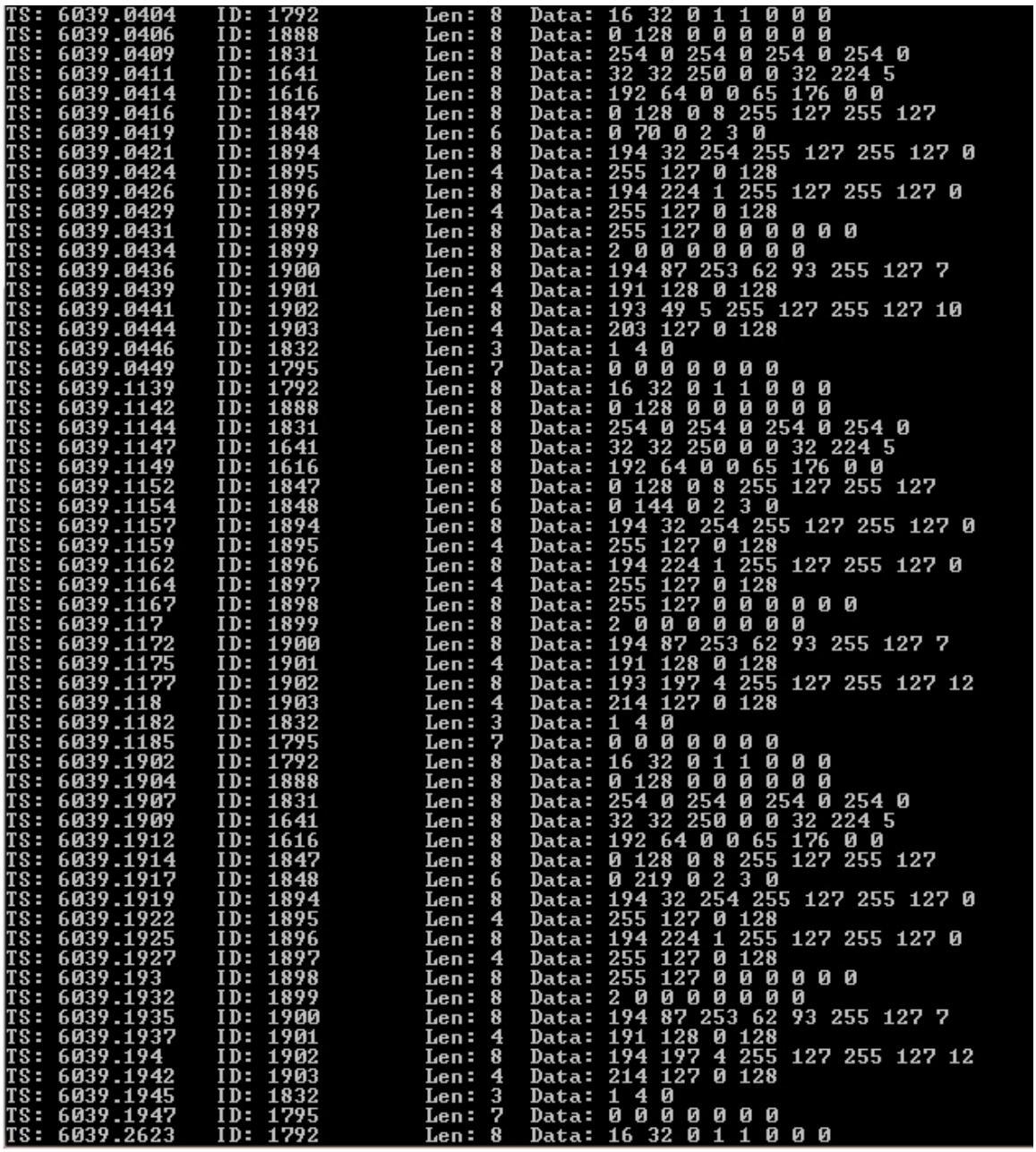

Figure 14 – Results

Figure 14 shows the data packets received from the Skynet servers using the implementation method described in this report. The results show that the timestamp work properly and that the CAN transceiver/controller correctly process the data from the MobilEye into a form that can be deciphered using the CAN message protocols located in

the MobilEye manual. Since the results were captured with Skynet parked in Ward under a controlled setting, the message ID's in figure 14 agree with the expected message ID's for when Skynet is parked in Ward. Since all of the expected message ID's were displayed, the packets were captured completely and no packets were lost.

SensorView Testing

Using Skynet's testing software SensorView, the parsed MobilEye data was able to display objects around Skynet on SensorView. SensorView displayed the object's type (ie. vehicle/pedestrian), relative position to the car, the relative speed of the object, the object's status (ie. moving/standing), and the age of the object. As Skynet drives on the road, SensorView is able to display the MobilEye's data live.

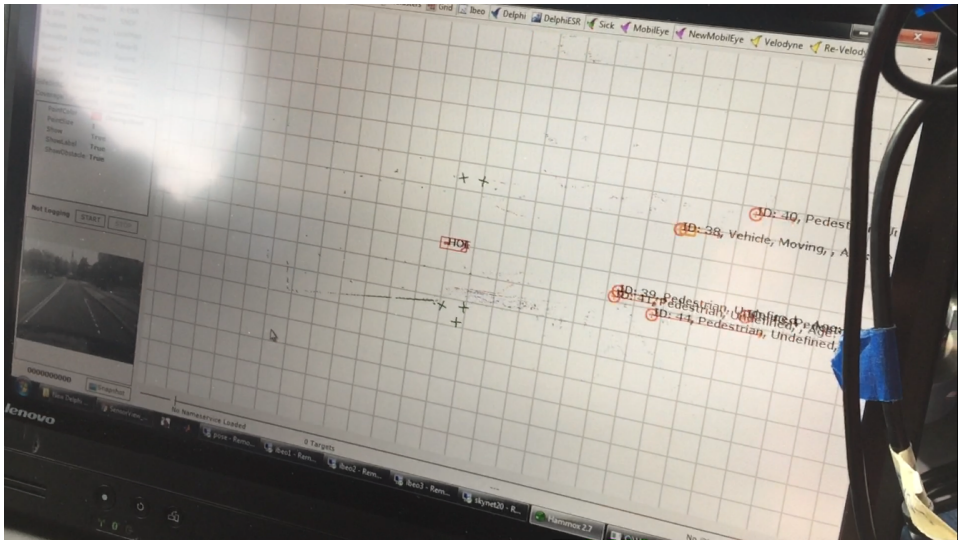

Figure 15 – SensorView with MobilEye Displayed

Observing figure 15, the rectangular box in the center is Skynet and the orange boxes to the right of Skynet are the objects MobilEye detects. Comparing MobilEye's data with Ibeo data shows that objects directly in front of Skynet are the most accurate but objects at angles or moving too quickly are not very accurate. More testing should be done to determine which exact scenarios produces what types of inaccuracies.

VI. Conclusion

The implementation scheme discussed in this report successfully integrated MobilEye into the Skynet system. The MobilEye data packets were captured completely, quickly, and accurately. Analyzing MobilEye's performance using SensorView, it was determined that MobilEye is excellent at extracting a lot of information about objects that are directly in front of Skynet that are moving at a moderate pace. Objects at an angle and moving too quickly produced results that did not match with Ibeo data and were significantly delayed. Therefore, the current performance of MobilEye using the implementation scheme discussed excels in environments such as highways but may lack in performance in heavily dense and quick paced environments such as an urban street in a city.

Areas to look into to improve the performance of the implementation scheme may include reducing the time it takes for the Ethernet Shield to send data to the network switch over SPI and reanalyzing the time synchronization to confirm it is accurate and has no time delay.

Overall, MobilEye provides an extra trove of information regarding the environment around Skynet and may have the most potential when paired with information from other sensors.

VII. Code

Please contact jyh37@cornell.edu for the code and corresponding libraries

```
#include "variant.h"
#include <due_can.h>
#include <SPI.h>
#include <Ethernet3.h>
#include <EthernetUdp3.h>

//[seconds msb][seconds lsb][ticks msb][ticks lsb][id msb][id
lsb][length][b0][b1][b2][b3][b4][b5][b6][b7]
byte packet[15];

unsigned int seconds; byte seconds_msb; byte seconds_lsb;
unsigned int ticks;   byte ticks_msb;   byte ticks_lsb;

unsigned long id;
byte id_msb;  byte id_lsb;

unsigned int time_ref;      //Arduino reference time upon time sync from timeserver (in
microseconds)

int begin_flag = 0;
int timer = 0;

//Arduino Due network info
byte mac[] = {0x90, 0xA2, 0xDA, 0x10, 0xE9, 0xF6};      //from ethernet shield sticker
(any MAC works)
IPAddress ip(192, 168, 1, 85);          //assign an IP that's not currently used by Skynet
IPAddress multiIP(239, 132, 1, 45);     //NewMobilEye multicast IP

unsigned int localPort = 30;      //Arduino port to listen on (timeserver sends to port 30 of
Arduino)
unsigned int multiPort = 30045;   //NewMobiLEye multicast port

EthernetUDP UdpRX;
EthernetUDP UdpTX;

byte receivePacket[5];    //[Type][Seconds MSB][Seconds LSB][Ticks MSB][Ticks LSB]

const int interruptPin = 2;     //pulse- from timeserver
int i = 0;                      //keeps track of status on pulse-

int rate;       //messages/second

int start;
```

```
int finish;
////////////////////////////////////////////////////////////////////////////////////////////////
void setup(){
  Can0.begin(CAN_BPS_500K);

  int filter;
  for (int filter = 3; filter < 7; filter++){
    Can0.setRXFilter(filter, 0, 0, false);
  }

  Ethernet.begin(mac, ip);

  UdpRX.begin(localPort);                    //timeserver (socket 0)
  UdpTX.beginMulticast(multiIP, multiPort);     //multicast (socket 1)

  pinMode(interruptPin, INPUT);
  attachInterrupt(interruptPin, pulseISR, CHANGE);     //UDP sent from timeserver when
pulse is turned off
                                        //ideally use FALLING, however FALLING doesn't
work as it should
  //Serial.begin(115200);
}

void loop(){
  //CAN
  CAN_FRAME incoming;

  if(begin_flag == 1){
  if (Can0.available() > 0) {
    Can0.read(incoming);
    packageFrame(incoming);
    //printPacket();
    multiCast();
    rate++;
  }
  }

  //TIMESERVER
  if(i == 2){     //i=0 when low, i=1 on rising, i=2 on falling
    time_ref = micros();

    int packetSize = UdpRX.parsePacket();       //205 us

    if (packetSize){
      UdpRX.read(receivePacket, packetSize);
```

```
      seconds_msb = receivePacket[1];       //only seconds changes, ticks is always 0
      seconds_lsb = receivePacket[2];

      unsigned int temp;
      temp = seconds_msb << 8;
      seconds = temp | seconds_lsb;

      begin_flag = 1;       //start program once timestamp received

      //Serial.print("Rate: "); Serial.println(rate);
      rate = 0;
    }
    i=0;
  }
}

void packageFrame(CAN_FRAME &frame){
//[seconds msb][seconds lsb][ticks msb][ticks lsb][id msb][id
lsb][length][b0][b1][b2][b3][b4][b5][b6][b7]
  getTicks();

  id = frame.id;
  id_msb = id >> 8;
  id_lsb = id & 255;

  //package packet
  packet[0] = seconds_msb;
  packet[1] = seconds_lsb;
  packet[2] = ticks_msb;
  packet[3] = ticks_lsb;
  packet[4] = id_msb;
  packet[5] = id_lsb;
  packet[6] = frame.length;

  for (int i=0; i<frame.length; i++){
    packet[i+7] = frame.data.bytes[i];
  }
}

void printPacket(){
//[seconds msb][seconds lsb][ticks msb][ticks lsb][id msb][id
lsb][length][b0][b1][b2][b3][b4][b5][b6][b7]
  unsigned int temp_seconds, temp_ticks, temp_id;
  temp_seconds = packet[0] << 8;
  temp_seconds = temp_seconds | packet[1];
  temp_ticks = packet[2] << 8;
```

```
  temp_ticks = temp_ticks | packet[3];
  temp_id = packet[4] << 8;
  temp_id = temp_id | packet[5];

  Serial.print("Seconds: "); Serial.print(temp_seconds);
  Serial.print(" Ticks: "); Serial.print(temp_ticks);
  Serial.print(" ID: "); Serial.print(temp_id);
  Serial.print(" Len: "); Serial.print(packet[6]);
  Serial.print(" Data: ");
  for(int i=0; i<packet[6]; i++){
    Serial.print(packet[i+7]); Serial.print(" ");
  }
  Serial.println();
}

void multiCast(){
  int x;
  //start = micros();

  UdpTX.beginPacket(multiIP, multiPort);         //47 us
  UdpTX.write(packet, 15);                       //145 us
  UdpTX.endPacket();                            //54 us

  //finish = micros();
  //Serial.println(finish-start);
  //Serial.println(x);
}

void getTicks(){
  ticks = (micros() - time_ref)/100;

  ticks_msb = ticks >> 8;
  ticks_lsb = ticks & 255;
}

void pulseISR(){
  i++;
}

/*NOTES*/
//10-18 microseconds from getting CAN message to being ready to send it out
```