Table of Contents

1 Introduction

Version 1.3 August 2016

The Hondata Bluetooth protocol is a packet based binary protocol using RFCOMM and Bluetopia RFCOMM over BLE emulation.

The protocol covers all Hondata products which use Bluetooth.

Elm327 style text commands are also supported by means of using the first packet byte as the protocol designator with a value below that of any ASCII character.

2 Packets

All packets start with a TDLHeader structure. This contains a protocol byte, a packet id and the overall packet size, include the header structure.

The following examples use a Bluetooth communications class similar to this:

```
class TBluetooth
{
  public:
    bool IsConnected;
    DWORD LastErrorCode;
  public:
    TBluetooth();
    ~TBluetooth();
    bool GetLocalAddress(unsigned char * pAddress);
    bool Open(void);
    bool Close(void);
    bool Connect(bool ComPortConnection, unsigned long long DeviceAddress);
    bool Disconnect(void);
    int Send(const char * Buffer, unsigned int Size);
    int Receive(char * Buffer, unsigned int Size);
    int Purge(void);
};
```

2.1 Protocol Packets

Protocol commands & packets

Currently the only command is to retrieve the device type and determine if the vehicle ignition is off or on.

```
// Hondata binary datalogging protocol
#define PROTOCOL_HD 0x00

// Header for all incomming and outgoing packets
typedef PACKED struct {
  unsigned char Protocol;  // = PROTOCOL_HD
  unsigned char ID; //
  unsigned short Size; // little endian, size includes this header
} TDLHeader;

#define DL_DeviceInfo 0x02
typedef PACKED struct {
  TDLHeader Header;
  unsigned char DeviceType;
  unsigned char IgnitionOn;
} TDeviceInfoReply;

// DeviceType
#define DT_S300 0xC0
#define DT_KPro 0xC1
```

```
#define DT_FlashPro 0xC2
```

Get Info example code

```
unsigned int BytesSent, BytesReceived;
BYTE InfoCmd[] = { PROTOCOL_HD, DL_DeviceInfo, 0x04, 0x00 };
BytesSent = Bluetooth->Send(InfoCmd, sizeof(InfoCmd));
if(BytesSent != sizeof(InfoCmd))
  printf("Bluetooth->Send failed");

TDeviceInfoReply InfoReply;
BytesReceived = Bluetooth->Receive((char *)&InfoReply, sizeof(InfoReply));
if(BytesReceived != sizeof(InfoReply))
  printf("Bluetooth->Receive failed\r\n");
else
{
  String Device;
  switch(InfoReply.DeviceType)
  {
    case DT_S300: Device = "S300"; break;
    case DT_KPro: Device = "KPro"; break;
    case DT_FlashPro: Device = "FlashPro"; break;
    default: Device = "?"; break;
  }
  printf("Device: %.02X (%s)\r\n", InfoReply.DeviceType, Device);
  printf("Ignition: %s\r\n", InfoReply.IgnitionOn ? "On" : "Off");
}
```

Get Info example transmit packet

```
{ 0x0, 0x2, 0x4, 0x0 }
```

Get Info example receive packet

```
{ { 0x0, 0x2, 0x6 }, 0xC0, 0x1 }
= device = S300, ignition = on
```

2.2 DTC Packets

DTCs

There are two DTC commands - one to read DTCs, and one to reset the DTCs. The maximum number of DTCs is fixed at 20.

```
#define DL_GetDTCs 0x20
// static const int MAXDTCS = 20;
#define MAXDTCS 20
typedef PACKED struct {
  TDLHeader Header;
  unsigned char Result;
  unsigned short DTCCount;
  unsigned short DTC[MAXDTCS];
} TGetDTCReply;

#define DL_ResetDTCs 0x21
typedef PACKED struct {
  TDLHeader Header;
  unsigned char RetCode;
} TResetDTCsReply;
```

Get DTCs example code

```
unsigned int BytesSent, BytesReceived;
BYTE DTCCmd[] = { PROTOCOL_HD, DL_GetDTCs, 0x04, 0x00 };
BytesSent = Bluetooth->Send(DTCCmd, sizeof(DTCCmd));
if(BytesSent != sizeof(DTCCmd))
  printf("Bluetooth->Send failed\r\n");

TGetDTCReply DTCReply;
BytesReceived = Bluetooth->Receive((char *)&DTCReply, sizeof(DTCReply));
if(BytesReceived != sizeof(DTCReply))
  printf("Bluetooth->Receive failed\r\n");
else
{
```

```
  printf("# DTCs = %d\r\n", DTCReply.DTCCount);
  for(int i=0; i<DTCReply.DTCCount; i++)
    printf("  DTCs = P%.03X\r\n", DTCReply.DTC[i]);
}
```

Get DTCs example transmit packet

```
{ 0x0, 0x20, 0x4, 0x0 }
```

Get DTCs example receive packet

```
{ { 0x0, 0x20, 0x2F }, 0x1, 0x2, { 0x611, 0x505, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x
= result ok, two error codes, 0x0611 & 0x0505
```

Clear DTCs example code

```
unsigned int BytesSent, BytesReceived;
BYTE ClearDTCCmd[] = { PROTOCOL_HD, DL_ResetDTCs, 0x04, 0x00 };
BytesSent = Bluetooth->Send(ClearDTCCmd, sizeof(ClearDTCCmd));
if(BytesSent != sizeof(ClearDTCCmd))
  printf("Bluetooth->Send failed\r\n");

TResetDTCsReply DTCClearReply;
BytesReceived = Bluetooth->Receive((char *)&DTCClearReply, sizeof(DTCClearReply));
if(BytesReceived != sizeof(DTCClearReply))
  printf("Bluetooth->Receive failed\r\n");
```

Clear DTCs example transmit packet

```
{ 0x0, 0x21, 0x4, 0x0 }
```

Clear DTCs example receive packet

```
{ { 0x0, 0x20, 0x4 }, 0x01 }
= result ok
```

2.3 Channel Packets

Channels

There are two steps to determine the list of supported datalog channels:
1. Send `DL_GetDatalogInfo` and get the number channels and total size of the datalog packet.
2. Send `DL_GetDatalogChannelIDs` and get a list of the channel IDs, sizes and types.
This channel information should then be stored for the datalogging functions.

```
#define DL_GetDatalogInfo 0x30
typedef PACKED struct {
  TDLHeader Header;
  unsigned short ChannelCount;
  unsigned short PacketSize;
} TDatalogInfoReply;

#define DL_GetDatalogChannelIDs 0x31
typedef PACKED struct {
  unsigned short ChannelID;
  unsigned char ChannelSize;
} TChannelIdEntry;

typedef PACKED struct {
  TDLHeader Header;
  // TChannelIdEntry array
} TGetChannelIdReply;

// Same packet except include variable size list
typedef PACKED struct {
  TDLHeader Header;
  TChannelIdEntry ChannelList[];
} TGetChannelIdListReply;
```

Get channel Info example code

```
clear the list of channels
unsigned int BytesSent, BytesReceived;
```

```
BYTE SizeCmd[] = { PROTOCOL_HD, DL_GetDatalogInfo, 0x04, 0x00 };
BytesSent = Bluetooth->Send(SizeCmd, sizeof(SizeCmd));
if(BytesSent != sizeof(SizeCmd))
  printf("Bluetooth->Send failed\r\n");
TDatalogInfoReply DatalogInfoReply;
BytesReceived = Bluetooth->Receive((char *)&DatalogInfoReply, sizeof(DatalogInfoReply));
if(BytesReceived != sizeof(DatalogInfoReply))
  printf("Bluetooth->Receive failed\r\n");
else
{
  ChannelCount = DatalogInfoReply.ChannelCount;
  printf("Channels: " + String(ChannelCount));
  PacketSize = DatalogInfoReply.PacketSize;
  printf("Packet size: " + String(PacketSize));

  BYTE cmd[] = { PROTOCOL_HD, DL_GetDatalogChannelIDs, 0x04, 0x00 };
  BytesSent = Bluetooth->Send(cmd, sizeof(cmd));
  if(BytesSent != sizeof(cmd))
    printf("Bluetooth->Send failed\r\n");

  TGetChannelIdReply ChannelReplyHeader;
  BytesReceived = Bluetooth->Receive((char *)&ChannelReplyHeader, sizeof(ChannelReplyHeader));
  if(BytesReceived != sizeof(ChannelReplyHeader))
    printf("Bluetooth->Receive failed\r\n");
  else
  {
    if(DatalogSensors != NULL)
      delete[] DatalogSensors;

    DatalogSensors = new BYTE[ChannelReplyHeader.Header.Size];
    if(DatalogSensors != NULL)
    {
      TGetChannelIdListReply * pSensorReply = (TGetChannelIdListReply *)DatalogSensors;
      memcpy(DatalogSensors, &ChannelReplyHeader, sizeof(ChannelReplyHeader));
      unsigned int ListSize = ChannelReplyHeader.Header.Size - sizeof(ChannelReplyHeader);
      BytesReceived = Bluetooth->Receive((char *)pSensorReply->ChannelList, ListSize);
      if(BytesReceived != ListSize)
        printf("Bluetooth->Receive failed\r\n");
      else
      {
        int Offset = 0;
        for(int i=0; i<ChannelCount; i++)
        {
          String Name, Size, Type;
          int CID = pSensorReply->ChannelList[i].ChannelID;
          int SizeType = pSensorReply->ChannelList[i].ChannelSize;
          const TChannelInfo * Info = GetChannelInfo(CID);
          if(Info != NULL)
          {
            Name = Info->Name;
            switch(SizeType & CS_MASK)
            {
              case CS_BYTE: Size = "Byte"; break;
              case CS_WORD: Size = "Word"; break;
              default: Size = "?"; break;
            }
            switch(SizeType & CT_MASK)
            {
              case CT_BIT: Type = "Bit"; break;
              case CT_NUMBER: Type = "Num"; break;
              case CT_RPM: Type = "Rpm"; break;
              case CT_SPEED: Type = "Speed"; break;
              case CT_MBAR: Type = "Press"; break;
              case CT_KPA: Type = "Press"; break;
              case CT_TPS: Type = "TPS"; break;
              case CT_INJ: Type = "Inj"; break;
              case CT_IGN: Type = "Ign"; break;
              case CT_RETARD: Type = "Ign"; break;
              case CT_TEMP: Type = "Temp"; break;
              case CT_PCT: Type = "%"; break;
              case CT_PCT_SIGNED: Type = "%"; break;
              case CT_5V: Type = "Volt"; break;
              case CT_19V: Type = "Volt"; break;
              case CT_LAMBDA: Type = "Lam"; break;
              default: Type = "?"; break;
```

```
          }
        }

        add a new item to the list of channels
        set the item caption to the id (CID)
        set the item data to the id (CID)
        show the item offset (Offset)
        show the item name (Name)
        show the item size (Size);
        show the item type (Type);

        switch(SizeType & CS_MASK)
        {
          case CS_BYTE: Offset += 1; break;
          case CS_WORD: Offset += 2; break;
        }
      }
    }
  }
}
```

Get datalog info transmit packet

```
{ 0x0, 0x30, 0x4, 0x0 }
```

Get datalog info receive packet

```
{ { 0x0, 0x31, 0x8 }, 0x39, 0x3F }
= 57 channels, packet size 63 bytes
```

Get channel info transmit packet

```
{ 0x0, 0x31, 0x4, 0x0 }
```

Get channel info receive packet

```
{ { 0x0, 0x31, 0xAF, 0x0 { 0x0, 0x1, 0x83, 0x1, 0x1, 0x84, 0x2, 0x1, 0x42, 0x10, 0x1, 0x85, 0x20, 0x1, 0x47, ..
= protocol 0, command 0x31, size 0xAF
= channel id 0x0100 CID_RPM, size = word, type = rpm
= channel id 0x0101 CID_Speed, size = word, type = speed
= channel id 0x0102 CID_Gear, size = byte, type = number
= channel id 0x0110 CID_MAP, size = word, type = mbar pressure
= channel id 0x0120 CID_TPS, size = byte, type = tps
```

2.4 Datalogging Packets

Datalogging

Datalogging requires a knowledge of the channels. The datalog packet is a packed binary packet, broken down into channel data using the channel size information from `DL_GetDatalogChannelIDs`. See the example code for how the packet offset is calculated.

Datalogging can be performed continuously with no start or stop step necessary. The device will usually datalog to an internal buffer from the vehicle at high speed in order to reduce turn around time. Most devices will stop datalogging from the vehicle if no datalog requests (`DL_GetDatalogPacket`) have been received for 1 second.

Datalog packets for the first 1-10 ms may contain stale data and should be discarded.

```
#define DL_GetDatalogPacket 0x35
typedef PACKED struct {
  TDLHeader Header;
  // byte array of datalogging channels
} TGetDatalogReply;
```

Datalog example code

```
// memChannels->Clear();
if(DatalogSensors != NULL)
{
  unsigned int BytesSent, BytesReceived;
```

```
BYTE GetDatalogCmd[] = { PROTOCOL_HD, DL_GetDatalogPacket, 0x04, 0x00 };
BytesSent = Bluetooth->Send(GetDatalogCmd, sizeof(GetDatalogCmd));
if(BytesSent != sizeof(GetDatalogCmd))
  printf("Bluetooth->Send failed\r\n");

TGetDatalogReply GetDatalogReply;
BytesReceived = Bluetooth->Receive((char *)&GetDatalogReply, sizeof(GetDatalogReply));
if(BytesReceived != sizeof(GetDatalogReply))
  printf("Bluetooth->Receive failed\r\n");

unsigned int PacketSize = GetDatalogReply.Header.Size - sizeof(GetDatalogReply);
BYTE * Packet = new BYTE[PacketSize];
if(Packet != NULL)
{
  BytesReceived = Bluetooth->Receive((char *)Packet, PacketSize);
  if(BytesReceived != PacketSize)
    printf("Bluetooth->Receive failed\r\n");
  else
  {
    TGetChannelIdListReply * pSensorReply = (TGetChannelIdListReply *)DatalogSensors;
    BYTE * pPacket = Packet;
    for(int channel=0; channel<ChannelCount; channel++)
    {
      const TChannelIdEntry * ChannelEntry = NULL;
      for(int c=0; c<ChannelInfoLength; c++)
      {
        if(ChannelInfo[c].CID == pSensorReply->ChannelList[channel].ChannelID)
        {
          ChannelEntry = &pSensorReply->ChannelList[channel];
          break;
        }
      }

      if(ChannelEntry != NULL)
      {
        double RawValue;
        switch(ChannelEntry->ChannelSize & CS_MASK)
        {
          case CS_BYTE:
            RawValue = *(unsigned char *)pPacket;
            break;
          case CS_WORD:
            RawValue = *(unsigned short *)pPacket;
            break;
        }

        String Value, Unit;
        switch(ChannelEntry->ChannelSize & CT_MASK)
        {
          case CT_BIT: Value = RawValue ? "On" : "Off"; break;
          case CT_NUMBER: Value = RawValue; Unit = ""; break;
          case CT_RPM: Value = RawValue * 0.25; Unit = "rpm"; break;
          case CT_SPEED: Value = RawValue * 0.01; Unit = "kph"; break;
          case CT_MBAR: Value = RawValue / 10.0; Unit = "kPa"; break;
          case CT_KPA: Value = RawValue * 0.5; Unit = "kPa"; break;
          case CT_TPS: Value = (RawValue / 2.0) - 10.0; Unit = "%"; break;
          case CT_INJ: Value = RawValue / 1000.0; Unit = "ms"; break;
          case CT_IGN: Value = (RawValue - 20.0) / 2.0; Unit = "°"; break;
          case CT_RETARD: Value = RawValue / 2.0; Unit = "°"; break;
          case CT_TEMP: Value = RawValue; Unit = "°F"; break;
          case CT_PCT: Value = RawValue / 2.56; Unit = "%"; break;
          case CT_PCT_SIGNED: Value = (RawValue - 128) / 1.28; Unit = "%"; break;
          case CT_5V: Value = Format("%.02f", ARRAYOFCONST(((double)RawValue * 5.0 / 256.0))); Unit = "v"; br
          case CT_19V: Value = 6.0 + (RawValue / 20.0); Unit = "v"; break;
          case CT_LAMBDA: Value = RawValue / 32768.0; Unit = "l"; break;
          default: Value = "?"; break;
        }

        run through the channel list
        {
          if(the channel item cid == pSensorReply->ChannelList[channel].ChannelID)
          {
            get the channel item
            show the channel raw value (RawValue)
            show the channel value & unit (Value + " " + Unit)
```

```
              break;
            }
          }
        }

        switch(pSensorReply->ChannelList[channel].ChannelSize & CS_MASK)
        {
          case CS_BYTE:
            pPacket += sizeof(unsigned char);
            break;
          case CS_WORD:
            pPacket += sizeof(unsigned short);
            break;
        }
      }
    }

    delete[] Packet;
  }
}
```

Datalog transmit packet

```
{ 0x0, 0x35, 0x4, 0x0 }
```

Datalog receive packet

```
{ { 0x0, 0x35, 0x43, 0x0 } 0x20, 0x12, 0x0, 0x0, 0x1, 0x23, 0x1, 0x55 ...
= protocol 0, command 0x35, size 67 bytes (header 4, data 63)
= channel 0 (CID_RPM, size = word, type = rpm) 0x1220 = 1160 rpm
= channel 1 (CID_Speed, size = word, type = speed) 0x0000 = 0 kph
= channel 2 (CID_Gear, size = byte, type = number) 0x01 = 1st gear
= channel 3 (CID_MAP, size = word, type = mbar pressure) 0x0123 = 29.1 kPa
= channel 4 (CID_TPS, size = byte, type = tps) 0x55 = 32.5 %
```

3 Channels

In the example code a list of channel names as follows:

```
typedef PACKED struct {
  unsigned short CID;
  const char * Name;
} TChannelInfo;

extern const TChannelInfo ChannelInfo[];
extern const int ChannelInfoLength;

extern const TChannelInfo * GetChannelInfo(unsigned short CID);
extern const char * GetChannelName(unsigned short CID);

const TChannelInfo ChannelInfo[] = {
{ CID_RPM, "RPM" },
{ CID_Speed, "Speed" },
{ CID_Gear, "Gear" },
{ CID_MAP, "MAP" },
{ CID_MAPVoltage, "MAP voltage" },
{ CID_AFMVoltage, "AFM voltage" },
{ CID_AFMFlow, "AFM flow" },
{ CID_TPS, "TPS" },
{ CID_TPSVoltage, "TPS voltage" },
{ CID_ThrottlePlate, "Throttle plate" },
{ CID_Inj, "Injector duration" },
{ CID_InjPhase, "Injector phase" },
{ CID_Duty, "Injector duty" },
{ CID_Ign, "Ignition advance" },
{ CID_IgnDwell, "Ignition dwell" },
{ CID_IAT, "Intake air temperature" },
{ CID_ECT, "Coolant temperature" },
{ CID_ECT2, "Coolant temperature #2" },
{ CID_ECTCorrection, "IAT correction" },
{ CID_IATCorrection, "ECT correction" },
{ CID_PA, "Air pressure" },
{ CID_BatteryVoltage, "Battery" },
```

```
{ CID_VTS, "VTEC spool" },
{ CID_VTP, "VTEC pressure" },
{ CID_VTC_Cmd, "VTC command" },
{ CID_VTC_Actual, "VTC actual" },
{ CID_VTC_Duty, "VTC duty" },
{ CID_VTC_Phase, "VTC phase" },
{ CID_O2A_V, "O2A voltage" },
{ CID_O2A_C, "O2A current" },
{ CID_O2A_Heat, "O2A heater" },
{ CID_O2A_Heat_R, "O2A heater resistance" },
{ CID_O2B_V, "O2B voltage" },
{ CID_O2B_Heat, "O2B heater" },
{ CID_Lambda_Lambda, "Lambda" },
{ CID_Corr_Lambda, "Corrected lambda" },
{ CID_Target_Lambda, "Target lambda" },
{ CID_Wideband_V, "Wideband voltage" },
{ CID_Wideband_Lambda, "Wideband lambda" },
{ CID_ShortTermTrim, "Short term trim" },
{ CID_MedTermTrim, "Medium term trim" },
{ CID_LongTermTrim, "Long term trim" },
{ CID_ClosedLoopStatus, "Closed loop status" },
{ CID_KnockLevel, "Knock level" },
{ CID_KnockVoltage, "Knock voltage", },
{ CID_KnockThreshold, "Knock threshold" },
{ CID_KnockRetard, "Knock retard" },
{ CID_KnockLimit, "Knock ignition limit", },
{ CID_KnockControl, "Knock control" },
{ CID_KnockCount, "Knock count" },
{ CID_KnockCount1, "Knock #1" },
{ CID_KnockCount2, "Knock #2" },
{ CID_KnockCount3, "Knock #3" },
{ CID_KnockCount4, "Knock #4" },
{ CID_KnockCount5, "Knock #5" },
{ CID_KnockCount6, "Knock #6" },
{ CID_ACSwitch, "AC switch" },
{ CID_SCS, "Service connector" },
{ CID_BrakeSwitch, "Brake switch" },
{ CID_ClutchIn, "Clutch in" },
{ CID_PSP, "Power steering pressure" },
{ CID_ELD_Voltage, "Electrical load voltage" },
{ CID_ELD_Current, "Electrical load current" },
{ CID_ACClutch, "AC clutch" },
{ CID_CheckEngine, "Check engine" },
{ CID_FuelPump, "Fuel pump" },
{ CID_ReverseLock, "Reverse lock" },
{ CID_AltControl, "Alternator control" },
{ CID_IAB, "Secondary runners" },
{ CID_RadFan, "Radiator fan" },
{ CID_Datalogging, "Datalogging" },
{ CID_SecondaryTables, "Secondary tables" },
{ CID_RevLimiter, "Rev limiter" },
{ CID_IgnitionCut, "Ignition cut" },
{ CID_BoostCut, "Boost cut" },
{ CID_LaunchCut, "Launch cut" },
{ CID_LaunchRetard, "Launch retard" },
{ CID_ShiftCut, "Shift cut" },
{ CID_PurgeDuty, "Purge duty", },
{ CID_PurgeOpen, "Purge open", },
{ CID_BoostControl_Duty, "Boost control duty", },
{ CID_FTP, "Fuel tank pressure", },
{ CID_Nitrous1Arm, "Nitrous 1 arm" },
{ CID_Nitrous1On, "Nitrous 1 on" },
{ CID_Nitrous2Arm, "Nitrous 2 arm" },
{ CID_Nitrous2On, "Nitrous 2 on" },
{ CID_Nitrous3Arm, "Nitrous 3 arm" },
{ CID_Nitrous3On, "Nitrous 3 on" },
{ CID_AnalogInput1, "Analog 1" },
{ CID_AnalogInput2, "Analog 2" },
{ CID_AN0, "Analog input 0" },
{ CID_AN1, "Analog input 1", },
{ CID_AN2, "Analog input 2", },
{ CID_AN3, "Analog input 3", },
{ CID_AN4, "Analog input 4", },
{ CID_AN5, "Analog input 5", },
{ CID_AN6, "Analog input 6", },
```

```
{ CID_AN7, "Analog input 7", },
{ CID_DI0, "Digital input 0" },
{ CID_DI1, "Digital input 1" },
{ CID_DI2, "Digital input 2" },
{ CID_DI3, "Digital input 3" },
{ CID_DI4, "Digital input 4" },
{ CID_DI5, "Digital input 5" },
{ CID_DI6, "Digital input 6" },
{ CID_DI7, "Digital input 7" },
{ CID_TC_Speed_LF, "Traction control speed LF" },
{ CID_TC_Speed_RF, "Traction control speed RF" },
{ CID_TC_Speed_LR, "Traction control speed LR" },
{ CID_TC_Speed_RR, "Traction control speed RR" },
{ CID_TC_Slip, "Traction control slip" },
{ CID_TC_Turn, "Traction control turn" },
{ CID_TC_OverSlip, "Traction control over slip" },
{ CID_TC_Output, "Traction control output" },
{ CID_TC_Volts, "Traction control voltage" },
{ CID_TC_Retard, "Traction control retard" },
{ CID_TC_RevLimiter, "Traction control rev limiter" },
{ CID_FuelLevel, "Fuel level" },
{ CID_FuelEconomy, "Fuel economy" },
{ CID_FuelUsed, "Fuel used" },
};
const int ChannelInfoLength = sizeof(ChannelInfo) / sizeof(TChannelInfo);
```

3.1 Channel Constants

```
// Channel definitions

#define CID_Unknown 0x0000

// Vehicle / engine
#define CID_Engine 0x0100
#define CID_RPM                         (CID_Engine + 0x00)
#define CID_Speed                       (CID_Engine + 0x01)
#define CID_Gear                        (CID_Engine + 0x02)
#define CID_MAP                         (CID_Engine + 0x10)
#define CID_MAPVoltage                  (CID_Engine + 0x11)
#define CID_AFMVoltage                  (CID_Engine + 0x12)
#define CID_AFMFlow                     (CID_Engine + 0x13)
#define CID_BP                          (CID_Engine + 0x14)
#define CID_BPCMD                       (CID_Engine + 0x15)
#define CID_TPS                         (CID_Engine + 0x20)
#define CID_TPSVoltage                  (CID_Engine + 0x21)
#define CID_ThrottlePlate               (CID_Engine + 0x22)
#define CID_Inj                         (CID_Engine + 0x30)
#define CID_InjPhase                    (CID_Engine + 0x31)
#define CID_Duty                        (CID_Engine + 0x32)
#define CID_Inj_2                       (CID_Engine + 0x33)
#define CID_Ign                         (CID_Engine + 0x40)
#define CID_IgnDwell                    (CID_Engine + 0x41)
#define CID_IAT                         (CID_Engine + 0x50)
#define CID_IAT_2                       (CID_Engine + 0x51)
#define CID_ECT                         (CID_Engine + 0x60)
#define CID_ECT2                        (CID_Engine + 0x61)
#define CID_ECTCorrection               (CID_Engine + 0x62)
#define CID_IATCorrection               (CID_Engine + 0x63)
#define CID_PA                          (CID_Engine + 0x70)
#define CID_BatteryVoltage              (CID_Engine + 0x80)
// VTEC/VTC
#define CID_VTEC 0x0200
#define CID_VTS                         (CID_VTEC + 0x00)
#define CID_VTP                         (CID_VTEC + 0x01)
#define CID_VTC_Cmd                     (CID_VTEC + 0x10)
#define CID_VTC_Actual                  (CID_VTEC + 0x11)
#define CID_VTC_Duty                    (CID_VTEC + 0x12)
#define CID_VTC_Phase                   (CID_VTEC + 0x13)
#define CID_EXVTC_Cmd                   (CID_VTEC + 0x14)
#define CID_EXVTC_Actual                (CID_VTEC + 0x15)
// Lambda
#define CID_Lambda 0x0300
#define CID_O2A_V                       (CID_Lambda + 0x00)
```

```
#define CID_O2A_C                       (CID_Lambda + 0x01)
#define CID_O2A_Heat                    (CID_Lambda + 0x02)
#define CID_O2A_Heat_R                  (CID_Lambda + 0x03)
#define CID_O2B_V                       (CID_Lambda + 0x10)
#define CID_O2B_Heat                    (CID_Lambda + 0x11)
#define CID_Lambda_Lambda               (CID_Lambda + 0x20)
#define CID_Corr_Lambda                 (CID_Lambda + 0x21)
#define CID_Target_Lambda               (CID_Lambda + 0x22)
#define CID_Lambda_Lambda_2             (CID_Lambda + 0x23)
#define CID_Target_Lambda_2             (CID_Lambda + 0x24)
#define CID_Wideband_V                  (CID_Lambda + 0x28)
#define CID_Wideband_Lambda             (CID_Lambda + 0x29)
#define CID_ShortTermTrim               (CID_Lambda + 0x30)
#define CID_MedTermTrim                 (CID_Lambda + 0x31)
#define CID_LongTermTrim                (CID_Lambda + 0x32)
#define CID_ShortTermTrim_2             (CID_Lambda + 0x33)
#define CID_LongTermTrim_2              (CID_Lambda + 0x34)
#define CID_ClosedLoopStatus            (CID_Lambda + 0x40)
#define CID_ClosedLoopStatus_2          (CID_Lambda + 0x41)
// Knock
#define CID_Knock 0x0400
#define CID_KnockLevel                  (CID_Knock + 0x00)
#define CID_KnockVoltage                (CID_Knock + 0x01)
#define CID_KnockThreshold              (CID_Knock + 0x02)
#define CID_KnockRetard                 (CID_Knock + 0x10)
#define CID_KnockLimit                  (CID_Knock + 0x11)
#define CID_KnockControl                (CID_Knock + 0x12)
#define CID_KnockCount                  (CID_Knock + 0x20)
#define CID_KnockCount1                 (CID_Knock + 0x21)
#define CID_KnockCount2                 (CID_Knock + 0x22)
#define CID_KnockCount3                 (CID_Knock + 0x23)
#define CID_KnockCount4                 (CID_Knock + 0x24)
#define CID_KnockCount5                 (CID_Knock + 0x25)
#define CID_KnockCount6                 (CID_Knock + 0x26)
// Inputs
#define CID_Inputs 0x0500
#define CID_ACSwitch                    (CID_Inputs + 0x01)
#define CID_SCS                         (CID_Inputs + 0x02)
#define CID_BrakeSwitch                 (CID_Inputs + 0x03)
#define CID_ClutchIn                    (CID_Inputs + 0x04)
#define CID_PSP                         (CID_Inputs + 0x05)
#define CID_ELD_Voltage                 (CID_Inputs + 0x06)
#define CID_ELD_Current                 (CID_Inputs + 0x07)
// Outputs
#define CID_Outputs 0x0600
#define CID_ACClutch                    (CID_Outputs + 0x01)
#define CID_CheckEngine                 (CID_Outputs + 0x02)
#define CID_FuelPump                    (CID_Outputs + 0x03)
#define CID_ReverseLock                 (CID_Outputs + 0x04)
#define CID_AltControl                  (CID_Outputs + 0x05)
#define CID_IAB                         (CID_Outputs + 0x06)
#define CID_RadFan                      (CID_Outputs + 0x07)
// ECU
#define CID_ECU 0x0700
#define CID_Datalogging                 (CID_ECU + 0x00)
#define CID_SecondaryTables             (CID_ECU + 0x01)
#define CID_RevLimiter                  (CID_ECU + 0x10)
#define CID_IgnitionCut                 (CID_ECU + 0x11)
#define CID_BoostCut                    (CID_ECU + 0x12)
#define CID_LaunchCut                   (CID_ECU + 0x13)
#define CID_LaunchRetard                (CID_ECU + 0x14)
#define CID_ShiftCut                    (CID_ECU + 0x15)
#define CID_PurgeDuty                   (CID_ECU + 0x20)
#define CID_PurgeOpen                   (CID_ECU + 0x21)
#define CID_BoostControl_Duty           (CID_ECU + 0x30)
#define CID_FTP                         (CID_ECU + 0x40)
// Nitrous
#define CID_Nitrous 0x0800
#define CID_Nitrous1Arm                 (CID_Nitrous + 0x00)
#define CID_Nitrous1On                  (CID_Nitrous + 0x01)
#define CID_Nitrous2Arm                 (CID_Nitrous + 0x10)
#define CID_Nitrous2On                  (CID_Nitrous + 0x11)
#define CID_Nitrous3Arm                 (CID_Nitrous + 0x20)
#define CID_Nitrous3On                  (CID_Nitrous + 0x21)
// Analog / Digital
```

```
#define CID_Analog 0x0900
#define CID_AnalogInput1                    (CID_Analog + 0x00)
#define CID_AnalogInput2                    (CID_Analog + 0x01)
#define CID_AN0                             (CID_Analog + 0x10)
#define CID_AN1                             (CID_Analog + 0x11)
#define CID_AN2                             (CID_Analog + 0x12)
#define CID_AN3                             (CID_Analog + 0x13)
#define CID_AN4                             (CID_Analog + 0x14)
#define CID_AN5                             (CID_Analog + 0x15)
#define CID_AN6                             (CID_Analog + 0x16)
#define CID_AN7                             (CID_Analog + 0x17)
#define CID_DI0                             (CID_Analog + 0x20)
#define CID_DI1                             (CID_Analog + 0x21)
#define CID_DI2                             (CID_Analog + 0x22)
#define CID_DI3                             (CID_Analog + 0x23)
#define CID_DI4                             (CID_Analog + 0x24)
#define CID_DI5                             (CID_Analog + 0x25)
#define CID_DI6                             (CID_Analog + 0x26)
#define CID_DI7                             (CID_Analog + 0x27)
// Traction control
#define CID_TractionControl 0x0A00
#define CID_TC_Speed_LF                     (CID_TractionControl + 0x00)
#define CID_TC_Speed_RF                     (CID_TractionControl + 0x01)
#define CID_TC_Speed_LR                     (CID_TractionControl + 0x02)
#define CID_TC_Speed_RR                     (CID_TractionControl + 0x03)
#define CID_TC_Slip                         (CID_TractionControl + 0x10)
#define CID_TC_Turn                         (CID_TractionControl + 0x11)
#define CID_TC_OverSlip                     (CID_TractionControl + 0x12)
#define CID_TC_Output                       (CID_TractionControl + 0x20)
#define CID_TC_Volts                        (CID_TractionControl + 0x21)
#define CID_TC_Retard                       (CID_TractionControl + 0x30)
#define CID_TC_RevLimiter                   (CID_TractionControl + 0x31)
// Misc
#define CID_Misc 0x0B00
#define CID_FuelLevel                       (CID_Misc      + 0x00)
#define CID_FuelEconomy                     (CID_Misc      + 0x01)
#define CID_FuelUsed                        (CID_Misc      + 0x02)
#define CID_EthanolContent                  (CID_Misc      + 0x03)
#define CID_FuelTemp                        (CID_Misc      + 0x04)
```

3.2 Channel Sizes

```
// Channel sizes
// Highest two bits of type are used as the size; rest are the type

#define CS_MASK 0xC0
#define CT_MASK 0x3F

// 0x00 spare (bit)
#define CS_BIT 0x40 // 8 bits (same as byte currently)
#define CS_BYTE 0x40 // 8 bits
#define CS_WORD 0x80 // 16 bits
// 0xC0 spare (dword)
```

3.3 Channel Data Types

```
// Channel data types

#define CT_UNKNOWN 0x00
#define CT_BIT 0x01 // byte o=off, 1=on
#define CT_NUMBER 0x02 // byte or word
#define CT_RPM 0x03 // word lsb=1 unit=r/min
#define CT_SPEED 0x04 // word (lsb=0.01 unit=kph)
#define CT_MBAR 0x05 // word mbar pressure
#define CT_KPA 0x06 // byte kpa pressure
#define CT_TPS 0x07 // byte lsb=0.2% range -20 to 118%
#define CT_INJ 0x08 // word lsb=0.001 ms
#define CT_IGN 0x09 // also used for cam angle
#define CT_RETARD 0x0B // byte retard 0-64 lsb=0.25
#define CT_TEMP 0x10 // byte lsb=1 offset=0 unit=ºF
```

```
#define CT_PCT 0x11 // byte percentage 0 to +100 lsb=128/100
#define CT_PCT_SIGNED 0x12 // byte percentage -100 to +100 lsb=128/100
#define CT_MASSFLOW 0x16 // mg/s mass flow
#define CT_5V 0x18 // byte 0-5 volts
#define CT_19V 0x19 // byte voltage 6-18.8V lsb=0.05V
#define CT_LAMBDA 0x1E
#define CT_MBAR2 0x1F // word mbar pressure 2016 Civic

// max 0x3F
```

4 Hondata Specific PIDs

Since the OBDII protocol does not cover all the engine parameters, Hondata has added some Hondata specific PIDs, accessible from the Elm327 interface.

These are available from mode 0x91

PID 0x00 = PIDs supported (bitmap)
eg 0xD1 0x00 0x00 0x00 0x00 0x05 = PIDs 1 (injector duration) & 4 (boost control) supported

PID 0x01 = injector duration
Data = word, big endian, injector duration in microseconds
eg 0xD1 0x01 0x14 0x03 = 5.123 ms

PID 0x02 = cam angle target & actual cam angle
Data = byte, cam angle in degrees / 4
eg 0xD1 0x02 0x50 0x55 = target angle 20 degrees, actual angle 21.25 degrees

PID 0x03 = ELD (electrical load detector)
Data = byte, voltage 0-5V/256
eg 0xD1 0x03 0x59 = 1.738V

PID 0x04 = boost control duty cycle
Data = byte, duty cycle 0-100% / 256
eg 0xD1 0x04 0x59 = 34.7%

PID 0x05 = knock retard
Data = byte, knock retard / 4
eg 0xD1 0x05 0x10 = 4 degrees knock retard

PID 0x06 = knock level
Data = byte, knock level 0-100% / 256
eg 0xD1 0x06 0x59 = 34.7%

PID 0x07 = knock count
Data = word, big endian, knock count
eg 0xD1 0x07 0x00 0x04 = knock count 4

PID 0x08 = knock count cylinder
Data = byte, number of cylinders
Data = byte, knock count cylinders 1,2,3,4,5,6
eg 0xD1 0x08 0x04 0x01 0x02 0x00 0x00 0x00 0x00 = 4 cylinders, counts 1, 2, 0, 0

Unsupported PIDs
return 0x7F 0xD1 0x12

5 Revision History

Version 1.0 January 2014
Initial document.

Version 1.1 March 2014
Added Hondata Specific PIDs

Version 1.2 September 2014
Added ethanol content and fuel temperature.

Version 1.3 August 2016
Added 2016 Civic channel IDs