Objective

Understand the UBX protocol to communicate with the Sparkfun gps.

Write I2C code to determine latitude, longitude, satellites in view, movement heading, and ground speed.

How does I2C work for this chip?

4.5.5 DDC Port

The Display Data Channel (DDC) bus is a two-wire communication interface compatible with the I²C standard (Integrated Circuit). See our on-line product selector matrix for availability.

Unlike all other interfaces, the DDC is not able to communicate in full-duplex mode, i.e. TX and RX are mutually exclusive. u-blox receivers act as a slave in the communication setup, therefore they cannot initiate data transfers on their own. The host, which is always master, provides the data clock (SCL), and the clock frequency is therefore not configurable on the slave.

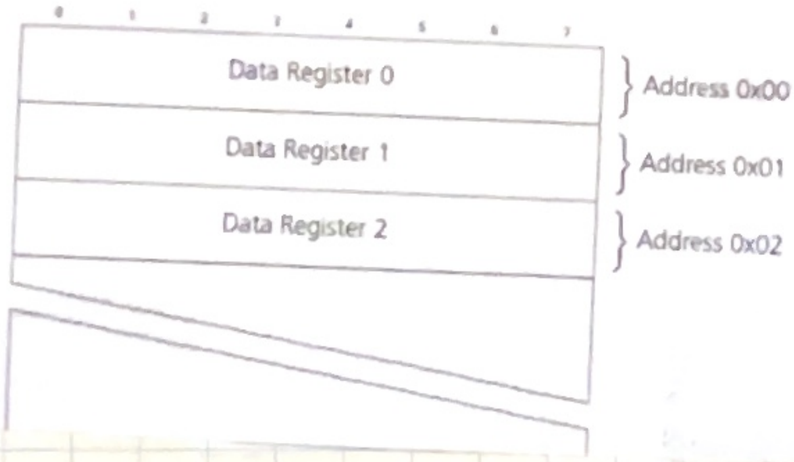
The receiver's DDC address is set to 0x42 by default.

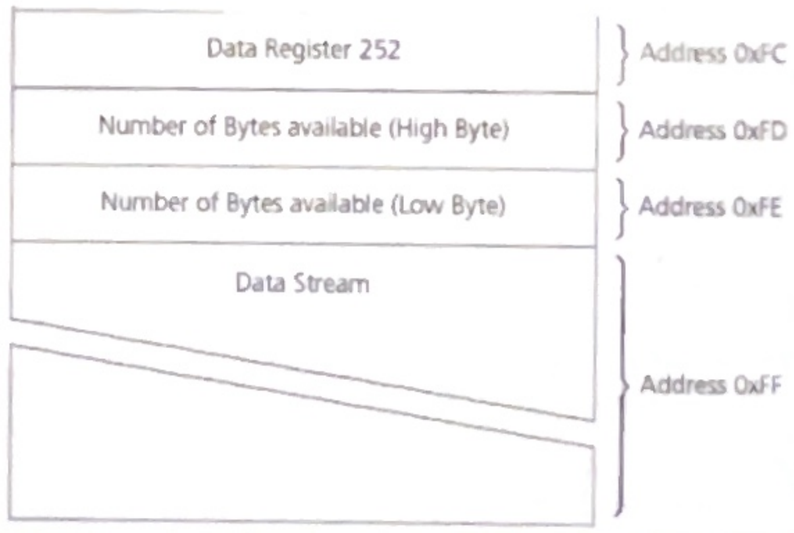
As the receiver will be run in slave mode and the DDC physical layer lacks a handshake mechanism to inform the master about data availability, a layer has been inserted between the physical layer and the UBX and NMEA layer. The receiver DDC interface implements a simple streaming interface that allows the constant polling of data, discarding everything that is not parse-able. The receiver returns 0xFF if no data is available. The TX-ready feature can be used to inform the master about data availability and can be used as a trigger for data transmission.

4.5.5.1 Read Access

The DDC interface allows 256 slave registers to be addressed. As shown in Figure *DDC Register Layout* only three of these are currently implemented. The data registers 0 to 252, at addresses 0x00 to 0xFC, each 1 byte in size, contain information to be defined later - the result of reading them is undefined. The currently available number of bytes in the message stream can be read at addresses 0xFD and 0xFE. The register at address 0xFF allows the data stream to be read. If there is no data awaiting transmission from the receiver, then this register will deliver the value 0xff, which cannot be the first byte of a valid message. If message data is ready for transmission, then successive reads of register 0xff will deliver the waiting message data.

☞ The registers 0x00 to 0xFC are reserved for future use and may be defined in a later firmware release. Do not use them, as they don't provide any meaningful data!





Essentially, Register 0xFF will act like a "UART". We will read and write to 0xFF.

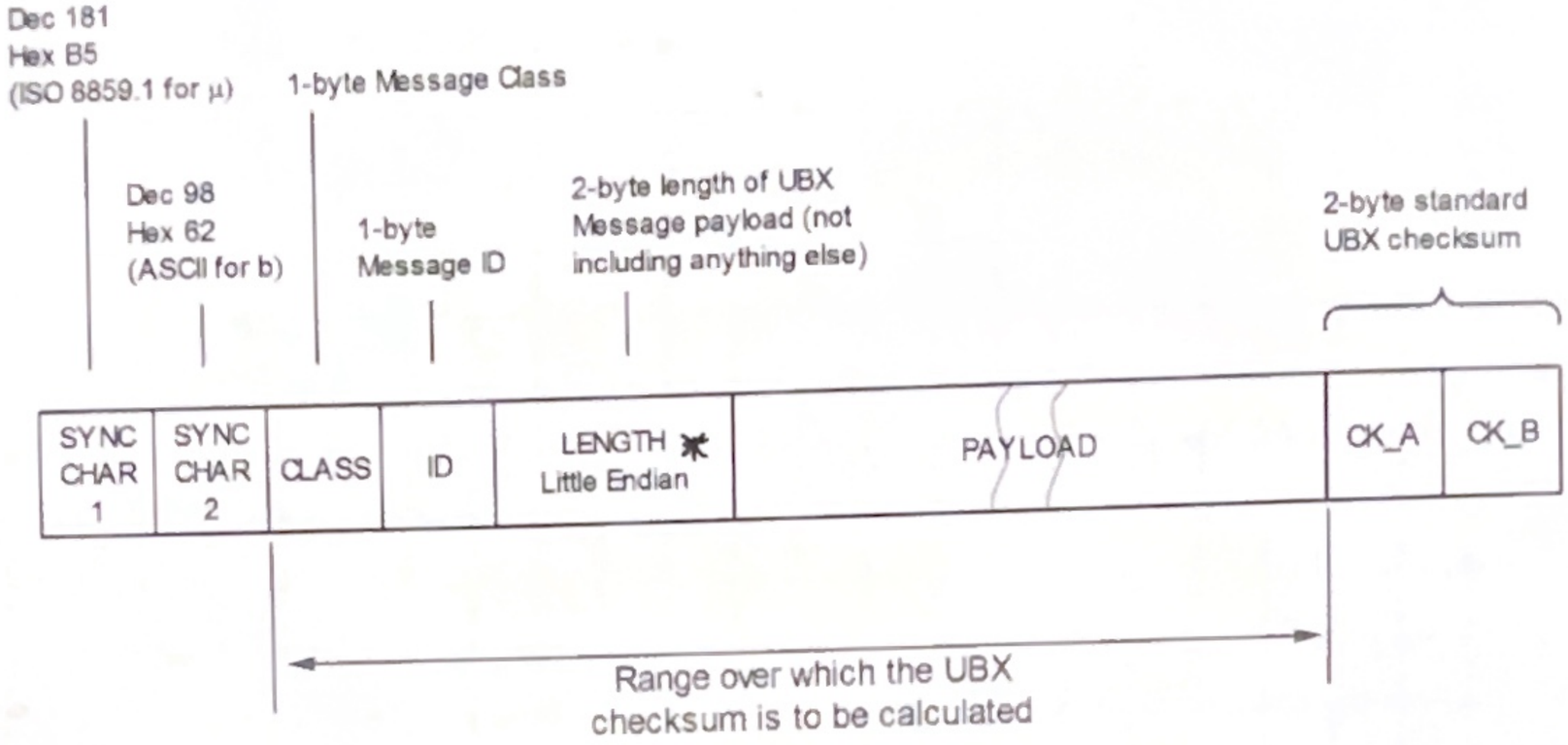
Revisiting the Sparkfun GPS

06/10/24
Wesley Code

What does a UBX message look like?

5.2 UBX Frame Structure

The structure of a basic UBX Frame is shown in the following diagram.



- Every **Frame** starts with a 2-byte Preamble consisting of two synchronization characters: 0xB5 0x62.
- A 1-byte Message **Class** field follows. A Class is a group of messages that are related to each other.
- A 1-byte Message **ID** field defines the message that is to follow.
- A 2-byte **Length** field follows. The length is defined as being that of the payload only. It does not include the Preamble, Message Class, Message ID, Length, or CRC fields. The number format of the length field is a Little-Endian unsigned 16-bit integer.
- The **Payload** field contains a variable number of bytes.
- The two 1-byte **CK_A** and **CK_B** fields hold a 16-bit checksum whose calculation is defined below. This concludes the Frame.

Little Endian means that the Lower bytes are stored at a smaller index in an array. The higher bytes are stored at the larger indexes.

Example: 0X 005C => 92 in decimal

Little Endian:	Index to Array	Value
	0	5C
	1	00
Big Endian:	0	00
	1	5C

06/10/24
Wesley Cook

Revisiting the Spark fun GPS

UBX Variable name short hand

5.3.5 Number Formats

All multi-byte values are ordered in Little Endian format, unless otherwise indicated.

All floating point values are transmitted in IEEE754 single or double precision.

Variable Type Definitions

Short	Type	Size (Bytes)	Comment	Min/Max	Resolution
U1	Unsigned Char	1		0..255	1
RU1_3	Unsigned Char	1	binary floating point with 3 bit exponent, eeeb bbbb, (Value & 0x1F) << (Value >> 5)	0..(31*2^7) non-continuous	~ 2^(Value >> 5)
I1	Signed Char	1	2's complement	-128 .. 127	1
X1	Bitfield	1		n/a	n/a
U2	Unsigned Short	2		0 .. 65535	1
I2	Signed Short	2	2's complement	-32768 .. 32767	1
X2	Bitfield	2		n/a	n/a
U4	Unsigned Long	4		0 .. 4'294'967'295	1
I4	Signed Long	4	2's complement	-2'147'483'648 .. 2'147'483'647	1
X4	Bitfield	4		n/a	n/a
R4	IEEE 754 Single Precision	4		-1*2^+127 .. 2^+127	~ Value * 2^-24
R8	IEEE 754 Double Precision	8		-1*2^+1023 .. 2^+1023	~ Value * 2^-53
CH	ASCII / ISO 8859.1 Encoding	1			

Check Sum Algorithm

The checksum algorithm used is the 8-Bit Fletcher Algorithm, which is used in the TCP standard (RFC 1145). This algorithm works as follows:

- Buffer[N] contains the data over which the checksum is to be calculated.
- The two CK_ values are 8-Bit unsigned integers, only! If implementing with larger-sized integer values, make sure to mask both CK_A and CK_B with 0xFF after both operations in the loop.

```
CK_A = 0, CK_B = 0
For(I=0;I<N;I++)
{
    CK_A = CK_A + Buffer[I]
    CK_B = CK_B + CK_A
}
```

- After the loop, the two U1 values contain the checksum, transmitted after the Message, which conclude the Frame.

Revisiting the SparkFun GPS

06/10/24

How does polling work?

5.5.2 Polling Mechanism

All messages that are output by the receiver in a periodic manner (i.e. messages in classes MON, NAV and RXM) and Get/Set type messages, such as the messages in the CFG class, can also be polled.

The UBX protocol is designed so that messages can be polled by sending the message required to the receiver but without a payload (or with just a single parameter that identifies the poll request). The receiver then responds with the same message with the payload populated.

We construct a UBX packet with the proper class and ID, but attach no payload.

We ~~~~ compute the checksum and send it also. Then we wait for a response.

What message do we want?

We want information related to position, velocity, and time.

⇒ UBX-NAV-PVT gives us this information.

5.14.11 UBX-NAV-PVT (0x01 0x07)

5.14.11.1 Navigation Position Velocity Time Solution

Message	**UBX-NAV-PVT**
Description	**Navigation Position Velocity Time Solution**
Firmware	Supported on: • u-blox 9 with protocol version 27.11
Type	Periodic/Polled
Comment	Note that during a leap second there may be more or less than 60 seconds in a minute. See the section Leap seconds in Integration manual for details. This message combines position, velocity and time solution, including accuracy figures

Message Structure	Header	Class	ID	Length (Bytes)	Payload	Checksum
	0xB5 0x62	0x01	0x07	92	see below	CK_A CK_B

Revisiting the Sparkfun GPS

Payload Contents:

Byte Offset	Number Format	Scaling	Name	Unit	Description
0	U4	-	iTOW	ms	GPS time of week of the navigation epoch. See the section iTOW timestamps in Integration manual for details.
4	U2	-	year	y	Year (UTC)
6	U1	-	month	month	Month, range 1..12 (UTC)
7	U1	-	day	d	Day of month, range 1..31 (UTC)
8	U1	-	hour	h	Hour of day, range 0..23 (UTC)
9	U1	-	min	min	Minute of hour, range 0..59 (UTC)
10	U1	-	sec	s	Seconds of minute, range 0..60 (UTC)
11	X1	-	valid	-	Validity flags (see graphic below)
12	U4	-	tAcc	ns	Time accuracy estimate (UTC)
16	I4	-	nano	ns	Fraction of second, range -1e9 .. 1e9 (UTC)
20	U1	-	fixType	-	GNSSfix Type: 0: no fix 1: dead reckoning only 2: 2D-fix 3: 3D-fix 4: GNSS + dead reckoning combined 5: time only fix
21	X1	-	flags	-	Fix status flags (see graphic below)
22	X1	-	flags2	-	Additional flags (see graphic below)
23	U1	-	numSV	-	Number of satellites used in Nav Solution
24	I4	1e-7	lon	deg	Longitude
28	I4	1e-7	lat	deg	Latitude
32	I4	-	height	mm	Height above ellipsoid
36	I4	-	hMSL	mm	Height above mean sea level
40	U4	-	hAcc	mm	Horizontal accuracy estimate
44	U4	-	vAcc	mm	Vertical accuracy estimate
48	I4	-	velN	mm/s	NED north velocity
52	I4	-	velE	mm/s	NED east velocity
56	I4	-	velD	mm/s	NED down velocity
60	I4	-	gSpeed	mm/s	Ground Speed (2-D)
64	I4	1e-5	headMot	deg	Heading of motion (2-D)
68	U4	-	sAcc	mm/s	Speed accuracy estimate
72	U4	1e-5	headAcc	deg	Heading accuracy estimate (both motion and vehicle)
76	U2	0.01	pDOP	-	Position DOP
78	X1	-	flags3	-	Additional flags (see graphic below)
79	U1[5]	-	reserved1	-	Reserved
84	I4	1e-5	headVeh	deg	Heading of vehicle (2-D)
88	I2	1e-2	magDec	deg	Magnetic declination
90	U2	1e-2	magAcc	deg	Magnetic declination accuracy

To fix the scaling on lat and lon, we will move the decimal 7 places to the right.

Like wise for ~~the~~ the heading we will move the decimal 5 places to the right.

46 | Revisiting the SparkFun GPS | 06/10/24

Example code

GC_GPS_UBX

```
#include <Wire.h>
unsigned long startTime;

typedef struct
{
  long lat;
  long lon;
  uint8_t SIV;
  long groundSpeed;
  long heading;
} NavData;
void setup()
{
  Serial.begin(9600);
  while(!Serial)
  Wire.begin();

  // Set the reading register to the data register.
  Wire.beginTransmission(0x42);
  Wire.write(0xFF);
  Wire.endTransmission();
  delay(100);

  startTime = millis();
}
void loop()
{
  if ((millis() - startTime) > 1000)
  {
    NavData data = readNavPVT();
    Serial.print("Lat: "); Serial.println(data.lat);
    Serial.print("Lon: "); Serial.println(data.lon);
    Serial.print("SIV: "); Serial.println(data.SIV);
    Serial.print("gSp: "); Serial.println(data.groundSpeed);
    Serial.print("Hea: "); Serial.println(data.heading);
    Serial.println("----------");
    startTime = millis();
  }
}
```

The goal of this code is to report the latitude, longitude, Sats in view, ground speed, and the heading of motion about every second.

We are using a struct to extract the data we want from the UBX-NAV-PVT message.

You can think of a Struct as a grouping of different variables that relate to each other.

Note: I am using an arduino giga because the gps is a 3.3V sensor and the giga has a 3.3V operating voltage.

code cont

```
NavData readNavPVT()
{
  // This method will ask the GPS for Position, Velocity, Time

  // This array is all the data before the "payload" data
  // u, B, class, id, length (LSB), length (MSB);
  uint8_t header[]=
  {
    0xB5, 0x62, 0x01, 0x07, 0x00, 0x00
  };

  // Start the checksum at 0
  uint8_t CK_A = 0;
  uint8_t CK_B = 0;
  // Array for the payload to go into
  uint8_t payload[92];

  // Part of the 8-bit fletcher algorithm.
  // The class and ID must be counted as with the payload
  for (int i=2; i<6; i++)
  {
    CK_A = CK_A + header[i];
    CK_B = CK_B + CK_A;
  }

  // This is an empty PVT packet
  // Per the data sheet:
  // "messages can be polled by sending the message
  // required to the receiver but without a payload"
  Wire.beginTransmission(0x42);
  Wire.write(0xB5); // Header
  Wire.write(0x62);
  Wire.write(0x01); // Class
  Wire.write(0x07); // ID
  Wire.write(0x00); // Length
  Wire.write(0x00);
  Wire.write(CK_A); // Checksum
  Wire.write(CK_B);
  Wire.endTransmission();

  // Now we expect the GPS to send out a similar
  // packet with it's own header. Everything should be the same
  // except for the length. The length should be 92 -> 0x5C.

  // Let's request the data.
  // 6 bytes for the header
  // 92 bytes for the payload
  // 2 bytes for the checksums
  Wire.requestFrom(0x42, 100);
  for (int i=0; i<6; i++)
  {
    header[i] = Wire.read();
  }
  for (int i=0; i<92; i++)
  {
    payload[i] = Wire.read();
    CK_A = CK_A + payload[i];
    CK_B = CK_A + CK_B;
  }
  uint8_t checksumA = Wire.read();
  uint8_t checksumB = Wire.read();

  // Checking for -> 0xB562, 0x0107, and 0x00C5
  uint16_t head_ = header[0]<<8 | header[1];          //Header
  uint16_t class_id = header[2]<<8 | header[3];       // Class/ID
  uint16_t payLoadLength = header[5] <<8 | header[4];// Length

  // Here, we check if the data is what we expect:
  // head_          -> 0xB562
  // class_id       -> 0x0107
  // payLoadLength -> 0x005C
  // If so, we can proceed to read the payload
  if (head_ == 0xB562 && class_id == 0x0107 && payLoadLength == 0x005C)
  {
    CK_A = 0;
    CK_B = 0;
    // Get the first part of the checksum using the header we read in.
    for (int i=2; i<6; i++)
    {
      CK_A = CK_A + header[i];
      CK_B = CK_A + CK_B;
    }
    for (int i=0; i<92; i++)
    {
      CK_A = CK_A + payload[i];
      CK_B = CK_A + CK_B;
    }
    if (checksumA == CK_A && checksumB == CK_B)
    {
      NavData data;
      data.lat = payload[31] <<24 | payload[30] << 16 | payload[29]<<8 | payload[28];
      data.lon = payload[27] <<24 | payload[26] << 16 | payload[25]<<8 | payload[24];
      data.SIV = payload[23];
      data.groundSpeed = payload[63] << 24 | payload[62] << 16 | payload[61] << 8 | payload[60];
      data.heading = payload[67] << 24 | payload[66] << 16 | payload[65] << 8 | payload[64];
      return data;
    }
  }
}
```

part of CK_sum

// we got a valid checksum.

// we got a valid PVT-packet!

little endian length

COM53

```
gSp: 651
Hea: 6456971
----------
Lat: 334679783
Lon: -819906045
SIV: 10
gSp: 651
Hea: 6456971
----------
Lat: 334679889
Lon: -819905717
SIV: 8
gSp: 405
Hea: 6456981
----------
```

Autoscroll Show timestamp