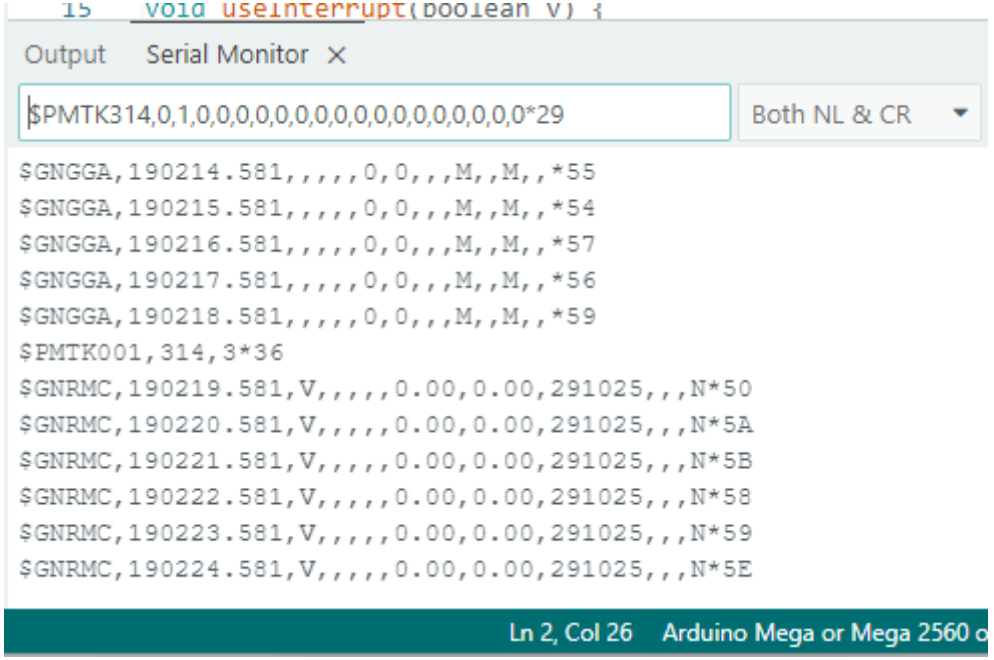
Using the Adafruit GPS Library

Communicating with the GPS

- All communication to and from the GPS is done via text Serial messages
- We can also connect our serial monitor directly to the GPS with the “Direct” switch
- This has the advantage of letting us look at the raw output in the Serial monitor
- We can also send commands using the Serial Monitor text box



With the switch in “Direct” can talk directly to GPS, including sending commands and seeing response (the PMTK001 line).

Input/Output Strings

Output (Data from GPS)

- NMEA Sentences
  - By default, transmits the GGA, GSA, RMC, VTG sentences every 1 second
  - You can change which sentences are output
  - You can also set the update rate to be faster or slower than 1 Hz
- The GPS will also transmit an acknowledgment response to any commands

Input (Data to GPS)

- Commands are also sent as text strings
- Just like NMEA starts with $, uses commas, and ends with a checksum and <cr><lf>
- Commands are called PMTK packets
- Full Command list reference
  - https://laspace.lsu.edu/laaces/wp-content/uploads/2021/10/PMTK_Packet_User_Manual.pdf

Adafruit_GPS Variable

- The library adds a new variable type **Adafruit_GPS**
  - These are called classes
- When you create the Adafruit_GPS variable, you must give it the Serial Port where the GPS is connected
  - The **&** is required (this is sending the memory location of Serial1 rather than sending a copy to the function)
- In this example, GPS is a variable name
- We often want to use a define statement like the lower example
  - This tells the compiler that everywhere we use **GPSSerial,** we mean **Serial1**
  - That way, if we decide to change the serial port, we only have to change the #define statement

```
#include <Adafruit_GPS.h>
Adafruit_GPS GPS(&Serial1);
```

```
/* Adafruit Ultimate GPS Logge
#define GPSSerial Serial1
Adafruit_GPS GPS(&GPSSerial);
```

GPS Functions Functions and Variables

- Similar to the RTC library we used in the I2C activity, the library defines functions that act on the **Adafruit_GPS** variable (in this example, GPS)
  - Notice the parentheses
- The library also defines variables inside the Adafruit_GPS variable that allow you to access pieces of data from the GPS
  - These will have no parentheses
- The documentation
  - https://adafruit.github.io/Adafruit_GPS/html/class_adafruit___g_p_s.html#ae39fbc538a1ee3ba1c8108bf49065c3f

```
GPS.read();
//Here the read() function is
//being called on the GPS variable
```

```
GPS.latitude;
//In this case latitude is variable
//Inside of the GPS variable
//It is this case it the last
//parsed Latitude value
```

Important GPS Functions

(In all of these, assume a Adafruit_GPS variable named GPS)

- GPS.begin()
  - Initialized the GPS
- GPS.sendCommand()
  - Sends a command to the GPS
- GPS.read()
  - Reads one character from the GPS serial port
- GPS.newNMEAreceived()
  - Tells you if a complete NMEA sentence has been received
- GPS.lastNMEA()
  - Gives the last complete, valid NMEA received
- GPS.parse()
  - Pulls the information out of the NMEA and updates the variable values **for the data in that NMEA** (e.g. GPS.hour, GPS.altitude)
  - Can't update data that is not in NMEA you parse

begin()

- GPS.begin(*baudrate*);

- This works similarly to Serial.begin()
- Should be part of your setup()

- Initializes the GPS variable and tells the Arduino what baudrate to use when talking to the GPS
  - Default GPS baudrate is 9600

```
Adafruit_GPS GPS(&GPSSerial);

void setup() {
  // put your setup code here,
 GPS.begin(9600);
```

Just like the serial ports, you need to call begin before using the GPS

sendCommand()

- GPS.sendCommand(*command*);

- The argument is the character string to send to the GPS as a command
- You can manually type the command string into the argument (top example)
  - This can be difficult, especially since you need to calculate the checksum characters at the end
- The library defines strings for most of the common commands, so those can be used like the lower example
  - These are contained in the Adafruit_PMTK.h file
  - https://github.com/adafruit/Adafruit_GPS/blob/master/src/Adafruit_PMTK.h
- Commands should normally be sent in setup() after the GPS.begin() to ensure the GPS is configured with your desired settings

```
//Sending a Command by manually typing the string
GPS.sendCommand("$PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*28");

//Sending a Command Using the Library Constants
GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCGGA);
```

Both the above lines are sending the same command, the top manually and the bottom using one of the library strings

read()

- GPS.read();
  - No argument
- Works similarly to Serial.read()
- Reads one character (which is 1 byte) from the GPS Serial Port and returns it
  - Will return 0 if there are no characters to read
- But does two other important things
  - Copies the character to the internal string used by the LastNMEA() function
  - Then it checks if a complete and valid NMEA has been received
  - When a complete NMEA has been read, the values for newNMEAreceived() and lastNMEA() get updated

```
char c;
c=GPS.read();
```

In this case we are copying the read character into the variable c, but because read also copies the character

newNMEAreceived()

- GPS.newNMEAreceived();
  - No Arguments
- Returns True or False depending if a new NMEA has been received
- Reading the last character with GPS.read() changes it to True
- Outputting a NMEA with lastNMEA() changes it to false
- Usually you use newNMEAreceived(), lastNMEA(), and parse() in combination to update the GPS data

```
  if (GPS.newNMEAreceived()){
    GPS.parse(GPS.lastNMEA());
  }
}
```

First use newNMEAreceived() to check for a NMEA sentence

lastNMEA()

- GPS.lastNMEA();
  - No arguments
- Returns the character string of the last complete and valid NMEA received
  - This includes the checksum and the <NL> and <CR> at the end
- This turns the newNMEA() back to FALSE
- Every time read() gets to the end of an NMEA, this value gets overwritten

```
  if (GPS.newNMEAreceived()){
    GPS.parse(GPS.lastNMEA());
  }
}
```

Then you use lastNMEA to get the NMEA string

parse()

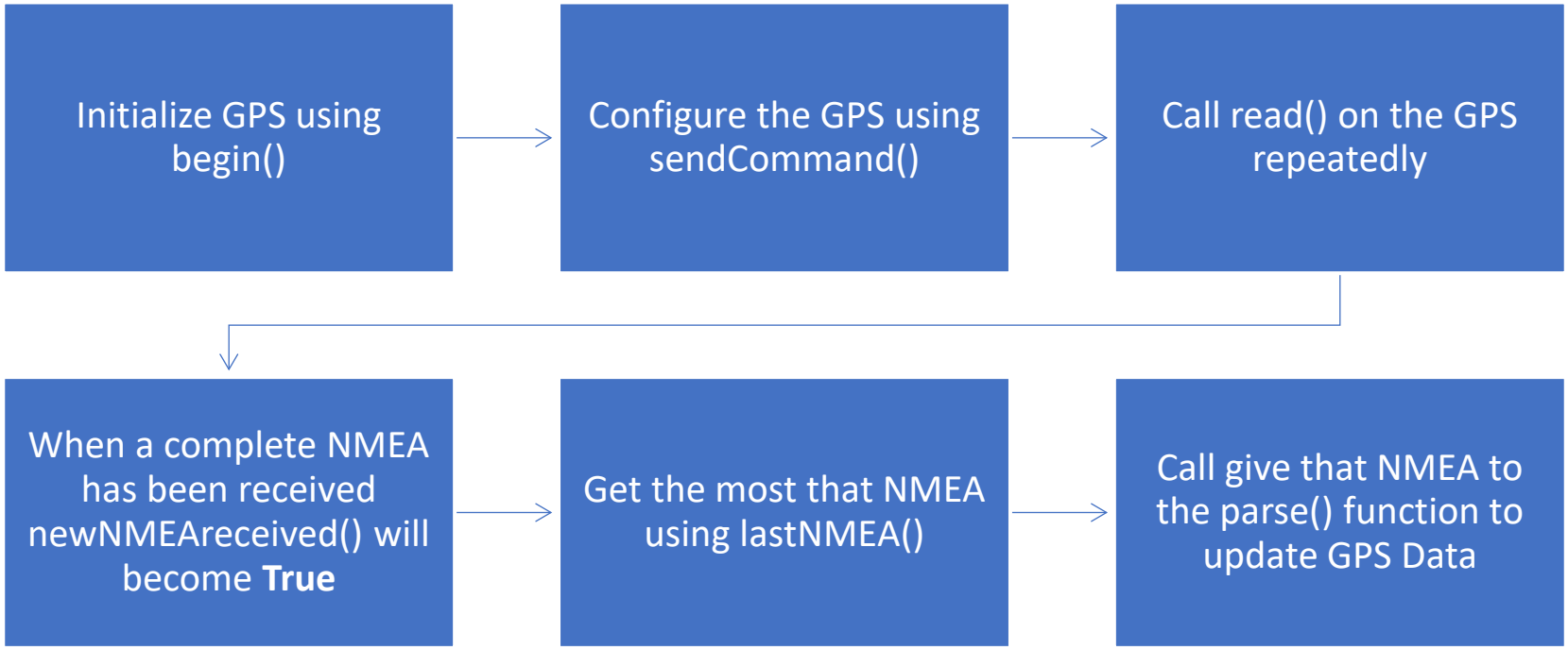
- GPS.parse(*NMEA String*);
- Give the NMEA text string as
- This breaks up the text string and updates the values of internal variables like GPS.hour or GPS.latitude
- But it can only update data that is in that NMEA
  - Ex. calling parse on an RMC will update Time, Date, Latitude, Longitude, but not Altitude
  - Also, if the data is blank (when the GPS does not have a fix), it will not be updated
  - In both cases, the value will remain unchanged, so if the GPS loses a fix on ascent, it will look like it stopped moving

```
if (GPS.newNMEAreceived()){
  GPS.parse(GPS.lastNMEA());
}
}
```

Finally, you give parse() the NMEA string and it will extract the data

Common Useful Internal GPS Variables	
GPS Variable	**Value**
GPS.latitude	Latitude (in Degrees and Minutes Format DDMM.MMMM, does not tell hemisphere)
GPS.longitude	Latitude (Degrees and Minutes Format DDMM.MMMM, does not tell hemisphere)
GPS.latitudeDegrees	Latitude (Decimal Degrees, - sign for South)
GPS.longitudeDegrees	Longitude (Decimal Degrees, + sign for West)
GPS.altitude	Height in meters above MSL
GPS.year	Year (two-digit format, 25 for 2025)
GPS.month	Month (1, 2, 3 .. 12)
GPS.day	Day of the Month (1, 2, 3 .. 31)
GPS.hour	Hour (Times with GMT/UTC so +5/6 Hours from Central depending on DST)
GPS.minute	Minutes
GPS.seconds	Seconds
GPS.satellites	Number of Satellites Tracked

GPS Data Flow



GPS Timing Concerns

- With all NMEA outputs on, we can use the serial monitor timestamp to see that it takes the GPS ~0.75 seconds to transmit all of that data
  - Since each character takes ~1ms to send and there are ~700 that makes sense
- The Serial receive buffer only holds 64 characters
- We need to parse each sentence (or save it another way)
- GPS will just continuously automatically transmit its data every 1 second; it will not wait for the Arduino

```
13:50:35.140 -> $GNGGA,185035.000,3024.7
13:50:35.238 -> $GNGLL,3024.7653,N,09110
13:50:35.271 -> $GPGSA,A,3,03,16,26,,,,,
13:50:35.336 -> $GLGSA,A,3,68,80,,,,,,,,
13:50:35.369 -> $GPGSV,3,1,11,16,67,172,
13:50:35.434 -> $GPGSV,3,2,11,04,41,323,
13:50:35.500 -> $GPGSV,3,3,11,09,06,310,
13:50:35.566 -> $GLGSV,2,1,07,69,70,011,
13:50:35.631 -> $GLGSV,2,2,07,78,18,032,
13:50:35.697 -> $GNRMC,185035.000,A,3024
13:50:35.764 -> $GNVTG,334.02,T,,M,1.95,N
```

Strategies to Avoid Missing Data

- Turn off unnecessary NMEA Sentences
  - By getting rid of the data we do not need, it will be less likely that the data we do need will be overwritten
- Call GPS.read() often
  - Since each character ~takes about 1 ms to transmit, we should call read about that often
- Check for newNMEAreceived() and call GPS.parse() often
  - If we have 2 sentences transmitted every 1 second, we should check for a new NMEA at least twice a second, but 3 or 4 times would be
  - Because of this, you want to avoid large using the delay() function with the GPS

Interrupts

- We can use a special function called an Interrupt to call read() every 1 ms
- An interrupt is a function that gets executed automatically when a certain condition occurs
  - They “interrupt” the normal flow of the program
  - This is the timers for the delay() and millis() functions work
- The trigger is tied to hardware, such as a pin changing from high to low or an internal clock reaching a certain value
- When a trigger occurs, the program stops what it is doing, runs the function called an Interrupt Service Routine (ISR)
- When the ISR finishes the program resumes where it left off
- https://docs.arduino.cc/language-reference/en/functions/external-interrupts/attachInterrupt/

Interrupt Example

- The code to the right will set up an interrupt to call GPS.read() ~1 ms
- The top box is the ISR
- The useInterrupt() function at the bottom enables/ disables the interrupt
  - The interrupt is off by default, so useInterrupt(true); should be called in setup()
  - The actual operations involve manipulating the setting registers on the AtMega2560
- the same timer is used for millis() and delay(), so if you have a typo, you may cause those functions to work incorrectly
- This code is available on the LaACES website and can be inserted at the top of your code below the #include statements

```
boolean usingInterrupt = false;    // This variable can be used to
ISR(TIMER0_COMPA_vect) {
/******* Interrupt Service Routine ********************************************
 *  This is the interrupt service routine. It reads a character from the GPS.
 ******************************************************************************/
  GPS.read();      // Read a character from the GPS
}
```

```
void useInterrupt(boolean v) {
/******* useInterrupt *********************************************************
 *  This function is enables or disables the SIGNAL(TIMER0_COMPA_vect) interrupt.
 ******************************************************************************/
  // millis() uses Timer0, so we'll interrupt in the middle and call the function abo
  if (v) {
    OCR0A = 0xAF;
    TIMSK0 |= _BV(OCIE0A);
    usingInterrupt = true;
  }

  // Do not call the interrupt function COMPA anymore
  else {
    TIMSK0 &= ~_BV(OCIE0A);
    usingInterrupt = false;
  }
}
```

The top function that is triggered by the interrupt, in this case it is triggered by a timer. Because the timer is the trigger, the function does not have a name and will not be called in the loop() or setup()

The bottom function turns the timer trigger on and off. Calling useInterrupt(true); turns it on and useInterrupt(false); turns it off