Wireless security

WarDog

the new war driving in miniature

Filip Pynckels
Director-general ICT
Federal Public Service Home Affairs

August 1, 2019

Table of contents

Introduction

This paper is written to show that it is not necessary to be an electronics genius to create a device that is so small that it can be hidden in plain sight, and that can get organizations in trouble.

The author urges the reader to see this paper as an inspiration to white hat hackers, and certainly to do nothing that is not legal. It goes without saying that the author cannot be held responsible for any damage that results from using the information in this paper or from implementing this information in real hardware/software.

Abstract

The expression wardriving is a generally known concept. It means loading your car with the necessary equipment (such as there are one or more notebooks, a couple of antenna's, a good GPS device, etc.), driving to a place that is (estimated to be) close enough to one or more wireless antenna's, and starting to scan the wireless antenna's and networks that belong to them.

The first thing to do is a passive (invisible) scan of the environment. Just 'sniffing' the wifi spectrum, or another RF band for that matter, depending if you'd like to know more about the wireless networks, the garage door openers, or what have you. For a wifi scan, what you like to know first is as much information as possible about the access points that are available in the vicinity. For instance:

- *The name of the wifi network (SSID - service set identifier)*
- *The mac address of the wifi access point (BSSID - basic service set identifier).*
- *The channel that the wifi antenna uses to send and receive information*
- *The signal strength (RSSI – received signal strength indicator)*

There are, however, major drawbacks to wardriving, such as:

- *Visibility: sitting in a car that is not known to the neighbors, while playing with a notebook that is connected to a less than usual antenna (directional, telescopic, ...) doesn't really make you blend into the environment.*
- *Approachability: when the wifi access points are placed in a shopping mall, in a train station, in a restaurant or café, a library, a ministry, a bank, etc., it will be very difficult to get close to these access points in order to guarantee a signal that is strong enough, or even just a signal without interference of RF noise.*

So, the question is, are there other options to gather information, and later on, to assess the security of a network, without getting spotted at first sight?

After a discussion with my (favorite) dog, we came to the conclusion that we'd like to try wardog. Would it be possible to create a device that is small enough to fit into a dog necklace, that is versatile enough to permit us to get wifi access point information while walking, and that is energy efficient enough to function during multiple hours. The device

should be cheap, it should be easy to build and use, and it should be programmable by using a main stream open source development environment.

Guess what: it's possible to develop such a dog necklace, and we will explain you how it can be done.

Components

The components that are needed to make our wardogging necklace are:

- A programmable processing unit
- A configurable wifi unit
- A configurable global positioning unit
- A storage device that provides an easy interface with a notebook to read the gathered data after walking the dog.
- A battery pack that can provide enough power to the used components
- A discrete dog necklace that is just large enough to contain all components, and that doesn't make my dog stand out like a lighthouse.

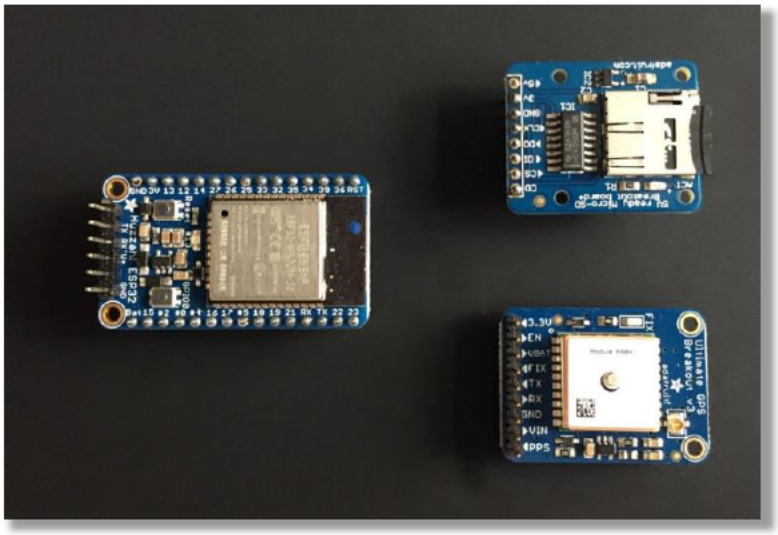

FIGURE 1 - COMPONENTS

The processing unit and the wifi unit

We have opted for the ESP32 processor in a configuration where it is combined with a wifi antenna and a serial interface to program it. This combined unit is the Adafruit ESP32-WROOM-32 breakout board (see Figure 1, component on the left).

The price is more than reasonable for a 240 MHz dual core. The size is 44.0mm x 25.5mm x 4.8mm and the important IC's are covered with a metal cover and hence protected against external pressure on the combined unit.

Some more technical specifications:

- 240 MHz dual core Tensilica LX6 microcontroller with 600 DMIPS,
- integrated 520 Kb SRAM,
- integrated 802.11b/g/n HT40 Wi-Fi transceiver,
- 4 Mb flash include in the WROOM32 module,
- on-board PCB antenna, SD-card interface support.

The latter (SD-card interface) will prove to be very useful to keep the dog necklace small. It prevents the need for supplementary IC's between the processing unit and the storage component.

The global positioning unit

As GPS unit, we have chosen for an Adafruit M3339-GPS breakout board (see Figure 1, component on the lower right side).

This component is not really cheap, but acceptable when one looks at the price/quality ratio. The important part is that this component contains an on-board antenna making the connection of an external antenna (although possible) unnecessary, which reduces the total price and the total size.

Some more technical specifications:

- Satellites: 22 tracking, 66 searching,
- Patch Antenna Size: 15mm x 15mm x 4mm
- Update rate: 1 to 10 Hz
- Position Accuracy: < 3 meters (all GPS technology has about 3m accuracy)
- Warm/cold start: 34 seconds
- Acquisition sensitivity: -145 dBm
- Tracking sensitivity: -165 dBm
- Vin range: 3v0-5v5
- Operating current: 25mA during tracking

It's important that this breakout board is able to function with as well 3v0 as with 5v0. The power consumption is low, the number of satellites it can follow is substantial for a good precision (less error prone), and the size is within the limits we had in mind for the dog necklace.

The storage device

As storage unit, we have chosen for an Adafruit 5v ready Micro-SD breakout board (see Figure 1, component on the upper right side).

The price is reasonable for what you get, the size is 31.85mm x 25.4mm x 3.75mm and hence small enough.

This breakout board gets SPI protocol signals on its breakout pins. Those signals are buffered by a HC4050M hex buffer towards and from the micro SD card. The input voltage is passed through an LP298x ultralow dropout regulator. This makes the breakout board especially suited as a battery powered device.

Alternative boards like the Sparkfun level shifting µSD breakout board (see figure 5&6) are alternatives that can function just as well.

The battery pack

The only requirements for this component are that it can power the other components with 5v0 and that is has enough capacity to keep the other components running during a number of hours. On top of that, being not extremely large would be very helpful.

A viable alternative is a number of button cell batteries (e.g. L type or Z type battery of 1V5) battery sockets connected in series with a total voltage of e.g. 4V5. There are button cell batteries of 3V0, but we consider this a nominal voltage for which the endpoint voltage will be too low to drive the used hardware.

The dog necklace

Since the size of the assembled configuration is not exactly know for the moment, we cannot make any prediction about the size and shape of the dog necklace, nor about the buckle, apart from the fact that it must fit snug and comfortable around the neck of the dog. Preferably, it should be adaptable to different sizes of neck circumference.

Software development environment

The ESP32 can be programmed in a number of ways. We have chosen to use the Arduino IDE with the ESP32 board add-on installed. This permits us to offer a simple way of reprogramming the ESP32-WROOM-32 in an environment that is as well easy to use as flexible enough to use C++, C or the ESP32 assembly languages. The latter has not been tried by the author, however. This environment also permits us to use the second core of the ESP32 should this be necessary.

Loading the compiled and/or assembled code to the ESP32 is included into the ESP32 board add-on. The only thing that is less flexible is the fact that there is no way to do instruction stepping or variable peeking to debug the written program. The only way to debug the written program is to let it write output to the standard serial port, which is connected to the Arduino IDE Serial Monitor.

The settings to use in the Arduino IDE are:

- Choice of board type ..... Adafruit ESP32 Feather
- Upload speed 921600
- Flash frequency ..... 80 Mhz
- Core debug level..... none

Schematics and proof of concept

Hardware

Figure 2 shows the connections that have to be made for a proof of concept. Note that, for this proof of concept, we will use a 220v AC to 5v0 DC convertor and not a battery pack.

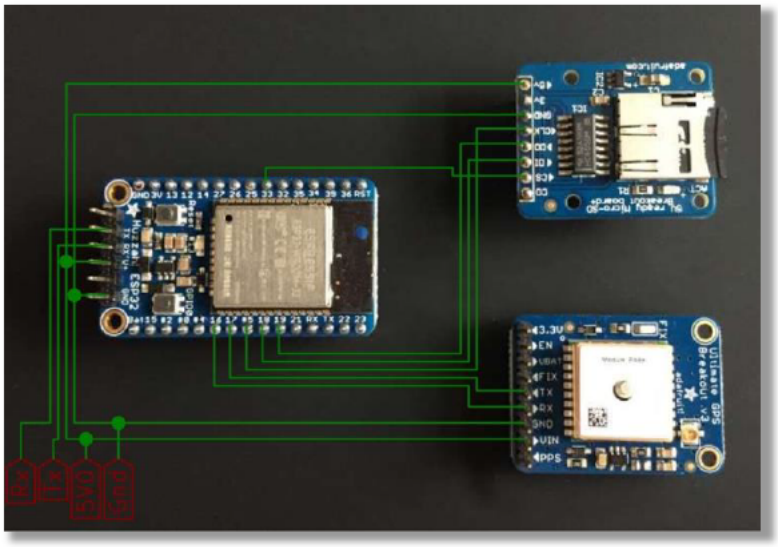

FIGURE 2 - SCHEMATICS

Figure 3 shows the assembled proof of concept. Note that at the left side of the image, 4 wires are connected with the "outside world". These are a red wire (5v0), a black wire (Gnd), and two purple wires (Rx and Tx serial connection to the Arduino IDE).

The components are respectively: the ESP32 & wifi at the left side of the breadboard, the micro-SD card breakout board in the middle and the global positioning unit at the right side of the breadboard.

All wires are connected as shown in Figure 2. It took some trial and error to have all wires connected as they should be. Standard serial to Arduino IDE, the second serial to the GPS unit, and the SD-card interface to the micro-SD card unit. But all's well that ends well, and after some programming, everything functions just fine. It evens seems that the ESP processing unit is so performant that it's not necessary to use the second core.

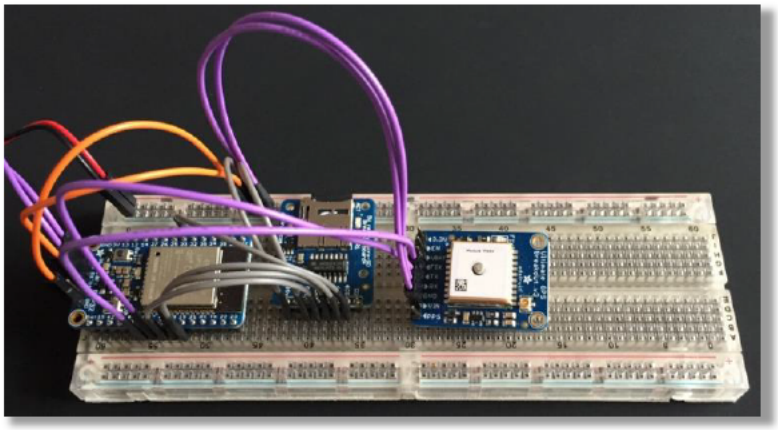

FIGURE 3 – PROOF OF CONCEPT

All connections are explicitly mentioned in the source code WarDog.ino (see Appendix 1).

Software

As already mentioned, the base software is the Arduino IDE. The advantage of this development environment is that is can be used as well under Linux, Windows as Mac OSx since it is written in Java. Within this development environment, one can add particular board configurations, as that for the ESP32 board add-on.

The main software used within the frame of this publication is an own development that can be found below (see Appendix 1) in the file WarDog.ino, where ino is the default file format of the development environment. The used programming language is pretty much C with some little C++ quirks to make life easier by using string and vector classes.

At a high level, since the GPS can be configured to wake up once per 10 seconds, the program functions as follows:

- Wait for GPS information,
- Get access point information,
- Write combined information to SD card,
- Blink a LED to show that all activity was performed successful
- Return to wait state

The function dependency graph is visualized in Figure 4. Nota that a dependency graph shows which functions depend on which other functions and that it is not a flow graph that would show the consecutive program flow.

An effort has been made to limit the number of used library functions to the strict minimum. The less different library functions one uses, the smaller the compiled code can be. To visualize the limited number of library functions the function dependency graph is extended to a depth of 4 levels.

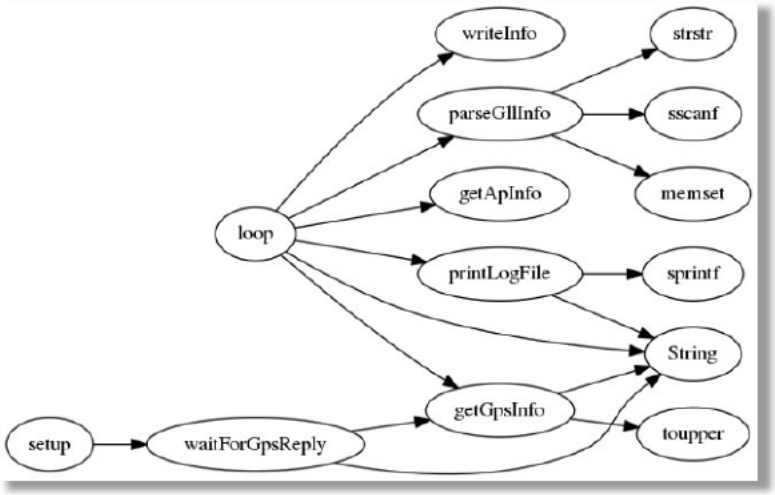

FIGURE 4 – FUNCTION DEPENDENCY GRAPH

The final version

Hardware

Figure 5 shows the final version of the hardware to put into a dog necklace. Note that the choice of SD-card breakout board has been changed to the Sparkfun level shifting µSD breakout board. The programming device is connected to the serial port of the processing unit by the wires going to the bottom of the figure. The hardware is shown on a flat surface but can be bend due to the independent boards that are only connected with isolated wires.

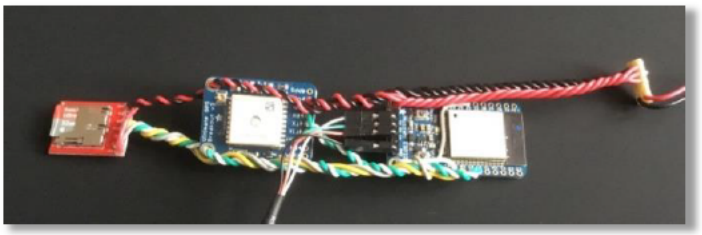

FIGURE 5 – FINAL HARDWARE WITH PROGRAMMER CONNECTION

The power for the programming phase is provided by the programming device. The power coming from the programming device is 5V0. Since all boards can handle a power range of at least 3V5 to 5V5, this is ok. The red and black wire at the right of the figure are not connected.

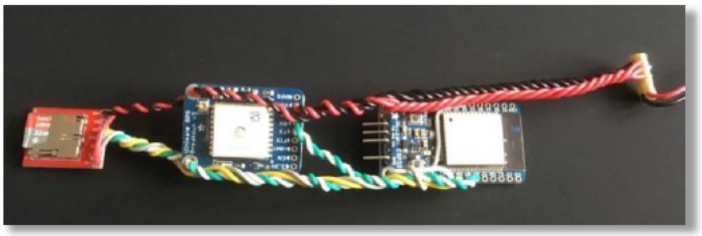

FIGURE 6 – FINAL HARDWARE FUNCTIONING INDEPENDENT

Figure 6 shows the same hardware without the cable to the programming device. In this case, the power comes through the red and black wire at the right of the figure. The power comes from three batteries (nominal voltage 1V5) that are connected in series so to deliver nominal 4V5 in total. The nominal 4V5 gives a connected voltage that is still pretty much in the middle of the allowed voltage range for all used components.

Software

As already mentioned, the base software is the Arduino IDE. We inserted the final (version 1.0) source code at the end of this text (see Appendix 1). Everything that was said in the software paragraph of the proof of concept is valid for the final version too.

Please note that most write operations to the console (default serial connection) are put in comment in order not to flood the console with messages. The write operations that are kept uncommented are those indicating an error. If more information is needed, the comment tokens (//) can be removed to have a better view on the program flow and the eventual place of malfunctioning.

Recorded information

It is, of course, an arbitrary choice how to write the recorded information to the SD-card. An easy way out is chosen in this case. A single file in the root directory called wardog.log and containing a dump of respectively the GpGllInfo and the ApInfo data.

Each record contains a version of the ApInfo information of a sensed access point, prefixed with the accompanying GpGllInfo information at that moment and place. The entire record has a total length of 64 bytes and is appended at the end of the log file.

The way that the records are stored has the disadvantage to contain a lot of redundancy due to the repeated storage of the GpGllInfo information. The advantage, on the other side, is that the file can be treated in an easy way without taking into account two different types of records that are intermingled. On top of that: records of equal length are easier to dump on the screen and easier to read in a hex dump if so needed.

The record structure is as follows:

```
1.  typedef struct {
2.
3.      // ----- GPGLL information ---------------------------------------------------
4.
5.      unsigned long latitude;       // Latitude
6.      char latitudeHemi;            // N/S
7.      unsigned long longitude;      // Longitude
8.      char longitudeMerid;          // E/W
9.      char active;                  // A (active record) / V (void record)
10.
11.     // ----- WIFI access point information ---------------------------------------
12.
13.     char ssid[33];                // Following documentation: max length = 32 + \0
14.     unsigned char bssid[6];       // Access point MAC address
15.     unsigned char channel;        // Access point channel
16.     int rssi;                     // Singal stringth in db
17.     wifi_auth_mode_t encryption;  // Access point encryption type
18.
19. } Info;
```

The data type wifi_auth_mode_t is an enumeration type that indicates the type of authentication the access point is using to protect itself and its users against unwanted access.

To come back on the redundancy issue: records of 64 bytes are written about every 50 seconds. If a 32Gb SD card is used, on the file level, an estimated 450000 records can be written. So, we can keep walking for a very long time before the SD-card will be filled up. The SD card reader also supports SD cards of 64Gb and 128Gb (both tested by the author), so lack of storage is not really an issue, even with the substantial redundancy of information in almost each record.

Note however that each time the hardware is restarted or reset (which is equivalent with restarting), the log file is erased (see Appendix 1 line 392). Commenting out the code line in question (//) prevents the log file from being erased, meaning that after a restart/reset old records will still be there and new records will be appended at the end of the log file.

With the recorded information bssid-latitude-longitude-rssi (at least 3 different GPS places for one bssid), it is possible to triangulate the location of the access point. More points can augment the accuracy of the calculations.

Note that a ssid can have several bssid's (access points). Note also that the same ssid can be used by different independent networks at different places, which makes treating the log file a bit more mathematically challenging. A characteristic that can be helpful in the triangulation is that the bssid is, in most cases, equal to the Mac address of the access point device, making it (in theory) globally unique.

If, however, you want to focus on one specific network, filtering can be built into the code of Appendix 1, or filtering can be built into the software treating the recorded information. This method of working will substantially simplify the mathematics necessary for finding the exact spot where the access points of the network are located. Focusing on one access point will of course simplify things even more.

The finishing touch

So, we have the hardware, we have the software, and we can record information. In short: we have the power. Now we need the elegance: making a necklace (see Figure 7) and convincing my dog that wearing it (see Figure 8) makes her look even sexier than she is without it.

FIGURE 7 – PREPARATION OF DOG NECKLACE

FIGURE 8 – FINAL HARDWARE IN DOG NECKLACE

Conclusion

It should be clear, after reading the above, that lately, all sorts of components exist that permit to make compose hardware solutions rather than designing them from scratch. This gives more and more power to less specialized people, as long as they can persevere searching the Internet for the right components. Once the right components are chosen, it all comes down to connecting the dots, or in this case, the breakout contacts.

The above text explains how a flexible and almost invisible (hidden in plain sight) war driving configuration can be created within a couple of days. This text stops at the stage of the silent data gathering of information, but nothing prevents to develop other software for the same hardware that takes things a step further and breaks into one or more networks.

Of course, for breaking into a network, off-line work is from times to times necessary. But even if everything must be done on-line, the developed device is small enough to be hidden and do its job without being noticed.

The reason why we stopped at the silent gathering of public data in the spectrum is that we don't want to give ammunition to black-hat script kiddies, and that we like to stay within the legal boundaries (by means of test, we have gathered data available in the public spectrum to check the content of this paper, but we didn't publish nor use it any further, and erased as soon as possible). Our goal was to point out the fact that less and less knowledge is required to break into the electronics of other people/companies. And hence, that electronic and ICT security is more and more an issue to be taken with the utmost seriousness. Seriousness in the sense of budgets, in the sense of hiring or recruiting knowledgeable people, but at least as important: seriousness in the sense of knowledge and attention paid to the subject by the private and the public sector.

What we could have done: walk in close proximity of the network access point we want to compromise and continue with an "airmon-ng/airodump-ng/aireplay-ng/aircrack-ng" alike cycle until we get at least one handshake from the targeted network. To explain it more structured:

- *First walk: the so to say "airmon-ng/airodump-ng" alike silent data gathering,*
- *Analyze the gathered data: get a position and authentication protocol,*
- *Change the software: load an "aireplay-ng" alike configuration in the ESP32,*
- *Second walk: walk your dog near the network until you receive a handshake,*
- *Off-line: use an "aircrack-ng" alike software to brute-force the network password*

PLEASE NOTE THAT THE LAST PARAGRAPH IS MERELY AN EXAMPLE OF WHAT CAN BE DONE, WHICH WE DIDN'T.

The last steps are intentionally part of a rather deprecated method in order not to inspire people with "non security research" aspirations. But it goes without saying that the same hardware can also be used to do more modern on-line attacks on networks. As long as you can get and stay unnoticed in the neighborhood of the network, Bob's your uncle.

Once you're in the network, the same hardware can be reused to do network mapping, and all kinds of other things. As long as you write the software, the dog brings it to the network access point…

Appendix 1 - Source code

```
1.      //================================================================================
2.      // Project:    wardog
3.      // File:        WarDog.ino
4.      // Author:      Filip Pynckels
5.      // Version:     1.0
6.      // Description: Main program
7.      //
8.      // Development environment:
9.      //              Arduino IDE 1.8.9
10.     //              ESP32 by Espressif Systems Version 1.0.3-rc1
11.     //              ESP32-WROOM-32 breakout board
12.     //              Micro-SD card breakout board (SPI interface)
13.     //              M3339-gps card breakout board (UART interface)
14.     //              Adafruit USB to TTL Serial Cable UART/USB
15.     //
16.     // USB to TTL Cable
17.     //
18.     //      UART TTL   |    ESP32 UART default pins
19.     //          Red   ----- 5V
20.     //          Black ----- GND
21.     //          Green ----- 34 default Rx (Rx UART0)
22.     //          White ----- 35 default Tx (Tx UART0)
23.     //
24.     // GPS card connection schema:
25.     //
26.     //      GPS Card   |  ESP32 UART default pins
27.     //          5V  ------- 5V  ----------------- Red
28.     //          GND ------- GND ----------------- Black
29.     //          Rx  ------- 16 (Tx UART1) ------- Green
30.     //          Tx  ------- 17 (Rx UART1) ------- White
31.     //
32.     // SD card connection schema:
33.     //
34.     //      SD Card    |  ESP32 SPI default pins
35.     //          5V  ------- 5V ------------------ Red
36.     //          3V          -
37.     //          GND ------- GND ----------------- Black
```

```
38.     //          CLK ------- #5 (CLK) ------------ Yellow
39.     //          DO  ------- 19 (MISO) ----------- Green
40.     //          DI  ------- 18 (MOSI) ----------- White
41.     //          CS  ------- 33 (Chip Select) ---- Gray
42.     //          CD          -
43.     //
44.     // Build settings:
45.     //      Board:              Adafruit ESP32 Feather
46.     //      Upload speed:       921600
47.     //      Flash frequency:    80 Mhz
48.     //      Core debug level:   none
49.     //
50.     // SD card formatting (for 1Gb card)
51.     //      File system:            FAT32
52.     //      Allocation unit size:   512 bytes
53.     //
54.     // Remarks: To upload the compiled code to the ESP32 follow the next steps:
55.     //      1. push <GPIO0 button>
56.     //      2. Hold <GPIO0 button> and push <RESET button>
57.     //      3. Hold <GPIO0 button> and release <RESET button>
58.     //      4. Release <RESET button>
59.     //      5. Upload from the Arduino IDE Sketch menu (Ctrl+U)
60.     //
61.     //================================================================================
62.
63.     #include <FS.h>
64.     #include <HardwareSerial.h>
65.     #include <SD.h>
66.     #include <WiFi.h>
67.
68.     //------------------------------------------------------
69.
70.     #define TIME_BETWEEN_CYCLES             (15 * 60)       // GPS time between cycles in milliseconds
71.                                                             // Gps interaction definitions
72.     #define PMTK_CMD_FULL_COLD_START        "$PMTK104*37\r\n"
73.     #define PMTK_CMD_FULL_COLD_START_ACK    "$PMTK011,MTKGPS*08"
74.     #define PMTK_SET_NMEA_UPDATERATE        "$PMTK220,10000*2F\r\n"
75.     #define PMTK_API_SET_NMEA_OUTPUT        "$PMTK314,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*2D\r\n"
76.                                                             // Gps valid parsing symbols
77.     #define VALID                           "$,*ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
78.
79.     #define LOG_FILE                        "/wardog.log"   // SD card log file
80.     #define SD_CS_PIN                       33              // SD card chip select pin
```

```
81.
82.     #define UART_GPS                        1               // UART 1 used as a connection to the GPS module
83.     #define UARTG_RX                        17              // Pin number for receive
84.     #define UARTG_TX                        16              // Pin number for transmit
85.
86.     #define SD_MISO                         19              // SD-card Master In Slave Out
87.
88.     typedef struct {                                        // WIFI access point info structure
89.        char ssid[33];                                       // Following documentation: max length = 32 + \0
90.        unsigned char bssid[6];                              // Access point MAC address
91.        unsigned char channel;                               // Access point channel
92.        int rssi;                                            // Singal stringth in db
93.        wifi auth mode t encryption;                         // Access point encryption type
94.     } ApInfo;
95.
96.     typedef struct {                                        // GPGLL record type
97.         // char header[10];                                 // Record description
98.         unsigned long latitude;                             // Latitude
99.         char latitudeHemi;                                  // N/S
100.        unsigned long longitude;                            // Longitude
101.        char longitudeMerid;                                // E/W
102.        char active;                                        // A (active record) / V (void record)
103.    } GpGllInfo;
104.
105.    //-------------------------------------------------------
106.
107.    HardwareSerial Gps(UART_GPS);                           // Gps serial connection
108.    int NrCardWrites = 0;                                   // Number of correct SD card write transactions
109.
110.    //-------------------------------------------------------
111.
112.        // Morse light...
113.
114.    const int ledPin = 13;
115.
116.        // Morse delays...
117.
118.    const int delay_long  = 400;
119.    const int delay short = 200;
120.    const int delay_space = 600;
121.
122.        // Morse sounds...
123.
```

```
124.    void dash  () { digitalWrite(ledPin, HIGH); delay (delay_long);  digitalWrite(ledPin, LOW); delay (delay_short); }
125.    void dot   () { digitalWrite(ledPin, HIGH); delay (delay_short); digitalWrite(ledPin, LOW); delay (delay_short); }
126.
127.        // Morse characters...
128.
129.    void space () { delay(delay_space);                                  }
130.    void cA    () { dot();  dash();                          space(); }
131.    void cB    () { dash(); dot();  dot();  dot();           space(); }
132.    void cC    () { dash(); dot();  dash(); dot();           space(); }
133.    void cD    () { dash(); dot();  dot();                   space(); }
134.    void cE    () { dot();                                   space(); }
135.    void cF    () { dot();  dot();  dash(); dot();           space(); }
136.    void cG    () { dash(); dash(); dot();                   space(); }
137.    void cH    () { dot();  dot();  dot();  dot();           space(); }
138.    void cI    () { dot();  dot();                           space(); }
139.    void cJ    () { dot();  dash(); dash(); dash();          space(); }
140.    void cK    () { dash(); dot();  dash();                  space(); }
141.    void cL    () { dot();  dash(); dot();  dot();           space(); }
142.    void cM    () { dash(); dash();                          space(); }
143.    void cN    () { dash(); dot();                           space(); }
144.    void cO    () { dash(); dash(); dash();                  space(); }
145.    void cP    () { dot();  dash(); dash(); dot();           space(); }
146.    void cQ    () { dash(); dash(); dot();  dash();          space(); }
147.    void cR    () { dot();  dash(); dot();                   space(); }
148.    void cS    () { dot();  dot();  dot();                   space(); }
149.    void cT    () { dash();                                  space(); }
150.    void cU    () { dot();  dot();  dash();                  space(); }
151.    void cV    () { dot();  dot();  dot();  dash();          space(); }
152.    void cW    () { dot();  dash(); dash();                  space(); }
153.    void cX    () { dash(); dot();  dot();  dash();          space(); }
154.    void cY    () { dash(); dot();  dash(); dash();          space(); }
155.    void cZ    () { dash(); dash(); dot();  dot();           space(); }
156.    void c0    () { dash(); dash(); dash(); dash(); dash(); space(); }
157.    void c1    () { dot();  dash(); dash(); dash(); dash(); space(); }
158.    void c2    () { dot();  dot();  dash(); dash(); dash(); space(); }
159.    void c3    () { dot();  dot();  dot();  dash(); dash(); space(); }
160.    void c4    () { dot();  dot();  dot();  dot();  dash(); space(); }
161.    void c5    () { dot();  dot();  dot();  dot();  dot();  space(); }
162.    void c6    () { dash(); dot();  dot();  dot();  dot();  space(); }
163.    void c7    () { dash(); dash(); dot();  dot();  dot();  space(); }
164.    void c8    () { dash(); dash(); dash(); dot();  dot();  space(); }
165.    void c9    () { dash(); dash(); dash(); dash(); dot();  space(); }
166.
```

```
167.        // Morse verbs...
168.        // There should be a char_space after
169.        // each real char, and a supplementary
170.        // char_space after each word.
171.
172.    void morse_SETUP_OK ()
173.    {
174.        pinMode(ledPin, OUTPUT);
175.        cS(); cE(); cT(); cU(); cP();
176.        space();
177.        cO(); cK();
178.        space();
179.    }
180.
181.    void morse_TRANSACTION_OK()
182.    {
183.        pinMode(ledPin, OUTPUT);
184.        cT(); cR(); cA(); cN(); cS(); cA(); cC(); cT(); cI(); cO(); cN();
185.        space();
186.        cO(); cK();
187.        space();
188.    }
189.
190.    //--------------------------------------------------
191.
192.    std::vector<ApInfo> getApInfo()
193.    {
194.
195.        // Get a list of information structures of the access
196.        // points that are available in the vincinity.
197.
198.        ApInfo apInfo;
199.        std::vector<ApInfo> apInfoVect;                    // Prevent buffer overrun by using a vector type
200.
201.        for (int i = 0; i < WiFi.scanNetworks(); ++i) {     // Iterate through found networks
202.            memset(&apInfo, 0, sizeof(apInfo));            // Zero out access point infromation structure
203.            strncpy(apInfo.ssid, WiFi.SSID(i).c str(), sizeof(apInfo.ssid)-1);
204.            memcpy(apInfo.bssid, WiFi.BSSID(i), sizeof(apInfo.bssid));
205.            apInfo.channel = WiFi.channel(i);
206.            apInfo.rssi = WiFi.RSSI(i);
207.            apInfo.encryption = WiFi.encryptionType(i);
208.            apInfoVect.push_back(apInfo);                  // Append access point information structure
209.        }
```

```
210.
211.        return apInfoVect;                                  // Return list of access point information structures
212.    }
213.
214.    //---------------------------------------------------
215.
216.    String getGpsInfo()
217.    {
218.
219.        // Read gps data up until the first \n
220.        // character. Translate all input to
221.        // uppercase and skip all invalid characters.
222.
223.        String gpsInfo = "";
224.        String valid = VALID;
225.
226.        while (true) {                                      // Compose the received gps output record
227.            if (Gps.available()) {
228.                char c = toupper(Gps.read());               // Translate to upper to be on the safe side
229.
230.                if (c == '\n') break;                       // Finish when end of line received
231.                if (valid.indexOf(c) == -1) continue;       // Skip undesired characters (such as \r)
232.                gpsInfo += c;                               // No buffer overflow risc due to use of String
233.            }
234.        }
235.
236.        return gpsInfo;
237.     }
238.
239.    //---------------------------------------------------
240.
241.    void parseGllInfo(const String gpGllStr, GpGllInfo* gpGllPtr)
242.    {
243.
244.        // Parse a gps record into its GPGLL components.
245.        // Take the necessary steps to avoid wrong
246.        // resulting data fields in case of empty source
247.        // data fields.
248.
249.        if (strstr(gpGllStr.c_str(), "$GPGLL,") == NULL) {  // Handle non GPGLL records
250.            memset(gpGllPtr, 0, sizeof(GpGllInfo));
251.            return;
252.        }
```

```
253.                                                    // Handle void GPGLL records
254.         if (strstr(gpGllStr.c_str(), "$GPGLL,,,,,") != NULL) {
255.             memset(gpGllPtr, 0, sizeof(GpGllInfo));
256.             gpGllPtr->active = 'V';
257.             return;
258.         }
259.
260.         #define ptr gpGllPtr                               // Notation to shorten sscanf statement
261.         if (9 != sscanf(gpGllStr.c str(), "%*9[^,],%lu,%c,%lu,%c,%lu,%c,%c*%hxu",
262.                 &ptr->latitude, &ptr->latitudeHemi, &ptr->longitude, &ptr->longitudeMerid, &ptr->active ))
263.         {
264.             memset(ptr, 0, sizeof(GpGllInfo));
265.             ptr->active = 'V';
266.             return;
267.         }
268.         #undef ptr
269.                                                    // You can only arrive here with
270.         return;                                    // a correctly parsed GPGLL record
271.     }
272.
273.     //-------------------------------------------------------
274.
275.     void printLogFile()
276.     {
277.
278.         // Show content of the log file on the
279.         // console in HEX format.
280.
281.         File logFile = SD.open(LOG_FILE);
282.
283.         if (!logFile)
284.             Serial.println("[e] Card: unable to open log file for reading");
285.         else {
286.             Serial.println("[ ] Card: log file dump");
287.
288.             while (logFile.available()) {
289.                 String hexLine = "          ";
290.
291.                 for (int i = 0; i < (sizeof(GpGllInfo) + sizeof(ApInfo)); i++) {
292.                     char buffer[3];
293.                     sprintf(buffer, " %02x", logFile.read());
294.                     hexLine += String(buffer);
295.                     if (!logFile.available()) break;
```

```
296.                 }
297.
298.                 Serial.println(hexLine);
299.             }
300.         }
301.
302.     }
303.
304.     //------------------------------------------------------
305.
306.     void waitForGpsReply(const String requiredReply)
307.     {
308.
309.         // Wait for the required ACK message
310.         // on the SoftwareSerial connection.
311.
312.         String gpsString;
313.
314.         do {
315.             gpsString = getGpsInfo();                       // No buffer overflow risc due to use of String
316.
317.             #ifdef DEBUG GPS
318.                 Serial.println(gpsString);
319.             #endif // DEBUG GPS
320.
321.         } while (gpsString != requiredReply);
322.     }
323.
324.     //------------------------------------------------------
325.
326.     int writeInfo(const char* path, const GpGllInfo gpGllInfo, const std::vector<ApInfo> apInfoVect)
327.     {
328.
329.             // Write all access point information to
330.             // the connected SD card.
331.
332.             File file = SD.open(path, FILE APPEND);
333.             if(!file) {                                      // Handle unable to open file case
334.                 return -1;
335.             } else {                                         // Iterate through found networks vector
336.                 for (int i = 0; i < apInfoVect.size(); ++i) {
337.                     const ApInfo*    ap = &(apInfoVect[i]);
338.                     const GpGllInfo* gp = &(gpGllInfo);
```

```
339.
340.                     delay(500);                             // Wait half a sec to be on the safe side
341.                     if(!file.write((byte*) gp, sizeof(GpGllInfo))) { return -2; }
342.                     delay(500);                             // Wait half a sec to be on the safe side
343.                     if(!file.write((byte*) ap, sizeof(ApInfo)))    { return -2; }
344.                 }
345.             }
346.
347.             delay(500);                                     // Wait half a sec to be on the safe side
348.             file.close();
349.             return 0;
350.     }
351.
352.     //-----------------------------------------------------
353.
354.     void setup()
355.     {
356.
357.         // Setup the connection to the console,
358.         // the connection to the gps, the gps,
359.         // and the connection to the SD card.
360.
361.         // ===== Setup console ============================
362.
363.         Serial.begin(115200);                               // Open serial connection to console
364.         while (!Serial) {}                                  // Wait for connection to be established
365.         Serial.println("[ ] Cons: connection established");
366.
367.         // ===== Setup general positioning system ==========
368.
369.         Serial.println("[ ] Gps:  establishing connection");
370.         Gps.begin(9600, SERIAL_8N1, UARTG_RX, UARTG_TX);    // Set config to prevent errors
371.         while (!Gps) {}
372.         Gps.write(PMTK_CMD_FULL_COLD_START);                // Force a gps cold start
373.         waitForGpsReply(PMTK CMD FULL COLD START ACK);      // Wait for cold start to be finished
374.         Gps.write(PMTK_SET_NMEA_UPDATERATE);                // Set a gps update rate of 1 per 10 seconds
375.         delay(500);                                         // Wait half a sec to be on the safe side
376.         Gps.write(PMTK_API_SET_NMEA_OUTPUT);                // Only get GPGLL records every 5 updates
377.         delay(500);                                         // Wait half a sec to be on the safe side
378.         Serial.println("[ ] Gps:  connection established");
379.
380.         // ===== Setup wifi in station mode ================
381.
```

```
382.        Serial.println("[ ] Wifi: establishing connection");
383.        WiFi.disconnect();                                  // Disconnect from an access point should a connection exist
384.        WiFi.mode(WIFI_STA);                                // Set wifi to station mode
385.        Serial.println("[ ] Wifi: connection established");
386.
387.        // ===== Setup SD card ============================
388.
389.        Serial.println("[ ] Card: establishing connection");
390.        while (!SD.begin(SD_CS_PIN)) {}                     // Open SPI connecion to SD card
391.        delay(500);                                         // Wait half a sec to be on the safe side
392.        SD.remove(LOG_FILE);                                // Erase log file
393.        Serial.println("[ ] Card: connection established");
394.
395.        // ===== Signal OK ================================
396.
397.        morse_SETUP_OK();
398.    }
399.
400.    //-----------------------------------------------------
401.
402.    void loop()
403.    {
404.
405.        // Main loop: Wait for gps info, get access
406.        // point information, write to SD card, and
407.        // loopie loopie loopie...
408.        // Handle eventual demand to dump the log file
409.        // to the console.
410.
411.        GpGllInfo gpGllInfo;
412.        std::vector<ApInfo> apInfoVect;
413.        String gpsInfoStr;
414.
415.        // ===== Wait for new GPGLL record to arrive =======
416.
417.        while (true) {
418.            gpsInfoStr = getGpsInfo();
419.            parseGllInfo(gpsInfoStr, &gpGllInfo);           // Wait for arrival of gps data record
420.            if (gpGllInfo.active != 0)
421.                break;                                      // Only fall through if the record is a GPGLL record
422.            else {
423.                Serial.println("[E] Gps:  received a non GPGLL record");
424.                // Serial.println("          " + gpsInfoStr);
```

```
425.            }
426.        }
427.
428.        // Serial.println("[ ] Gps:  received new coordinates");
429.        // Serial.println("          " + gpsInfoStr);
430.
431.        // ===== Get new access point information ==========
432.
433.        apInfoVect = getApInfo();                           // Get information about access points in the vincinity
434.        // Serial.println("[ ] WiFi: received new access point information");
435.        // Serial.print  ("          ");
436.        // for (int i = 0; i < apInfoVect.size()-1; i++) {
437.        //     Serial.print(String(apInfoVect[i].ssid) + ", ");
438.        // }
439.        // if (0 < apInfoVect.size())
440.        //     Serial.println(apInfoVect[apInfoVect.size()-1].ssid);
441.
442.        // ===== Write information to SD card ==============
443.
444.        int wrc = 0;
445.        wrc = writeInfo(LOG_FILE, gpGllInfo, apInfoVect);   // Write a record per capted access point to the SD card
446.
447.        switch (wrc) {
448.            case -1:
449.                Serial.println("[E] Card: SD card transaction error (RC=" + String(wrc) + ",file open error)");
450.                break;
451.
452.            case -2:
453.                Serial.println("[E] Card: SD card transaction error (RC=" + String(wrc) + ",file write error)");
454.                break;
455.
456.            default:
457.                NrCardWrites++;
458.                // Serial.println("[ ] Card: SD card transaction numer " + String(NrCardWrites));
459.                Serial.println("[ ] Tran: " + String(NrCardWrites));
460.                morse_TRANSACTION_OK();                     // Signal OK
461.                break;
462.        }
463.
464.        // ===== Show log file on console if required ======
465.
466.        if (Serial.available()) {                           // If there is console input
467.            printLogFile();                                 // Dump the log file to the console
```

```
468.             while (Serial.available()) Serial.read();       // Empty console input buffer
469.         }
470.     }
```