# A6GPRS Modem Services Library

Author: David Henry mailto:mgadriver@gmail.com

Download https://github.com/mgaman/A6General

| Version | Information | Date |
|---|---|---|
| 1.00 | A6GPRSDevice + A6GPRS + A6MQTT classes | 23-Jan-2017 |

## Licence



## Introduction

1. The A6GPRS is a low cost GSM modem with 2G capabilities.
2. This library aims to make general services such as telephony, SMS and TCP client communication easily available to the Arduino user.
3. The library is intended to be used on any hardware Serial port and will NOT work via SoftwareSerial.
4. The library is divided into a number of classes:
    a. A6GPRSDevice. This class handles low level tasks such as handling UART activity to and from the modem hardware. Broadly speaking the Arduino programmer does not need to be concerned with the methods and variables of this class. However some customization must be performed before use.
    b. A6GPRS. This class encapsulated the modem AT command language to perform a variety of basic services such as basic telephony, SMS, GPRS connections and TCP client connections.
    c. A6MQTT. This class implements the MQTT Messaging Protocol Version 3.1.1.
    d. HTTP. This class … (TBD)
    e. SMTP. This class … (TBD)
    f. Call. This class will implement basic telephony, initiating calls, accepting calls, Caller ID etc. (TBD)
    g. SMS. This class … (TBD)

## Design Constraints

1. The documentation available for the A6GPRS is incomplete and sometimes just incorrect. The code developed so far is a mix of reading the book and simple observation.
2. A particular problem is missing descriptions of unsolicited result codes. When connected as a TCP client to a server, data from the server may be mixed up with various modem result codes and it is up to the various classes to separate the 2 streams.
3. For example, the MQTT class does not expect to have to deal with incoming calls or SMS messages. You may wish to add that yourselves.
4. The modem works, by default, at a baud rate of 115200 bps. No other baud rate is supported.

# Customization

## A6GPRSDevice

Edit the A6Modem.h file to determine which hardware serial port is used and the maximum size of the circular buffer used to hold incoming data.

1. Edit the macro HW_SERIAL to define which UART is used i.e. Serial, Serial1, Serial2 or Serial3.
2. Edit the macro HW_SERIAL_EVENT to define the matching serial event to HW_SERIAL, e.g. Serial2 is handled by serialEvent2.
3. Edit the macro COMM_BUF_LEN to define the size of the circular buffer holding incoming data.

## A6MQTT

Edit the A6MQTT.h file to determine the maximum length of an incoming publish event (topic + message parts). Publish events that are too long are discarded.

1. Edit the macro MAX_MESSAGE_LENGTH to be large enough to accommodate the largest expected topic + message.
2. If a message is too long it will be discarded. This might be a problem for those with QOS > 0 as their receipt will never get to be acknowledged.

# Design Philosophy – A6GPRSDevice

Various debugging messages are present in the code. If the application wishes to see the messages it must implement 'helper' methods to display the message. For example the line

gsm.DebugWrite("got a reply");

may appear anywhere in the library.

To see the message a method A6GPRSDevice:: DebugWrite(char *) must be implemented. If the message goes to a serial port or LCD screen or whatever is up to the developer.

# Design Philosophy – MQTT

While the class handles the formatting and parsing of all interaction between the client and the broker, it raises events (a.k.a Callbacks) that should be handled by the application.

While the application may choose not to handle events raised by the MQTT class, it makes the application a lot simpler to do so.

The provided examples show how this is done.

# Design Philosophy – A6MQTT Examples

All of the examples follow the same format and are divided into 3 source files, reflecting their part in the overall solution.

1. Name.ino – This is the actual Arduino application and contains the setup() and loop() functions plus others as needed.
2. A6Helpers.cpp – These are the optional methods defined for the A6GPRS class. Currently DebugWrite and HWReset are supported.

3. MQTTHelpers.cpp – These are the optional events (callbacks) defined for the ASMQTT class. You may make this as simple or complex as needed. For example OnPubAck may enforce a mechanism to wait for one message to be published before publishing another.

# API

## A6GPRS:

Add the following line to your source file.

```
#include "A6Services.h"
```

An instance of A6GPRS called gsm is automatically created.

All methods return true if completed successfully, else false.

### Methods

1. `bool begin()`
    a. Initialize the A6 modem after hardware reset. The modem is left detached from the Packet Radio service and ready to accept AT commands.
    b. A baudrate of 115200 bps is assumed.
2. `Void HWReset()`
    a. This is an optional function to reset the modem by forcing the A6 RS pin low for 50ms. Note that you cannot do this directly from an Arduino GPIO pin as it takes too much current. See http://electronics.stackexchange.com/questions/82222/how-to-use-a-transistor-to-pull-a-pin-low
3. `bool getIMEI(char* buffer);`
    a. Retrieve the modems IMIE value and write it to buffer. Buffer must be large enough to hold the text.
4. `bool getCIMI(char* buffer);`
    a. Retrieve the SIM card CIMI value and write it to buffer. Buffer must be large enough to hold the text.
5. `bool getRTC(char* buffer);`
    a. Retrieve the modems Real Time Clock value and write it to buffer. Buffer must be large enough to hold the text.
    b. The value is in the format yy/mo/dd,hh:mm:ss,+tz where
        i. yy is year since 2000
        ii. mo month number
        iii. dd day number
        iv. hh hour
        v. mm minute
        vi. ss second
        vii. tx – timezone expressed in ½ hours before/after GMT
        viii. Example "17/01/08,21:33:19,+02" is 8th January 2017 21 hours 33 minutes 19 secs GMT. Local timezone 1 hour before GMT.
6. `bool setRTC(char* buffer);`

a. Set the modems Real Time Clock value and from buffer.
b. The format of buffer is as described in getRTC.

7. `enum eCIPstatus getCIPstatus()`
   a. Get the current connection state (see A6 AT commands section 10.1.4).
8. `bool startIP(char *apn);`
   a. Start a GPRS connection with the Access Point named apn. No userid or password needed.
9. `bool startIP(char *apn,char*userid,char *password);`
   a. Start a GPRS connection with the Access Point named apn using the credentials userid and password.
10. `ePSstate getPSstate();`
    a. Get the current PS (Packet Domain Service) state (see A6 AT commands section 9.1.4)
11. `bool setPSstate(ePSstate);`
    a. Set the Packet Domain Service state (see A6 AT commands section 9.1.4)
12. `bool stopIP();`
    a. Disconnect a TCP session. Note this must also be called if the server disconnected the TCP session.
13. `bool getLocalIP(char *);`
    a. Retrieve the IP address assigned by your APN and write it to buffer. Buffer must be large enough to hold the text.
14. `bool connectTCPserver(char* ipaddress, int port);`
    a. Start a TCP session with the IP address and port specified.
15. `bool sendToServer(char*);`
    a. Transmit a zero delimited string to the server.
16. `bool sendToServer(char* array, int length);`
17. `bool sendToServer(byte*array, int length);`
    a. Transmit a byte array of size 'length' to the server.
18. `void DebugWrite(int)`
19. `void DebugWrite(unsigned int)`
20. `void DebugWrite(char)`
21. `void DebugWrite(char *)`
22. `void DebugWrite(int)`
23. `void DebugWrite(const __FlashStringHelper*)`
    a. All of the DebugWrite methods are optional and are implemented by the application developer (if needed);

## Data

1. `unsigned long rxcount`
   a. Number of bytes received from the server during a TCP connection.
2. `unsigned long txcount`
   a. Number of bytes transmitted to the server during a TCP connection.
3. `bool enableDebug;`
   a. Used to enable or disable the action of DebugWrite in runtime code. For example you may scatter DebugWrite statements throughout your code but they can be enabled/disabled by use of enableDebug.

## Enums

1. `enum eCIPstatus {IP_INITIAL, IP_START, IP_CONFIG, IP_IND, IP_GPRSACT,IP_STATUS, TCPUDP_CONNECTING, IP_CLOSE, CONNECT_OK, IP_STATUS_UNKNOWN};`
    a. Enumerates the current connection state (see A6 AT commands section 10.1.4).
2. `enum ePSstate {DETACHED,ATTACHED,PS_UNKNOWN};`
    a. Evaluates the current PS (Packet Domain Service) state (see A6 AT commands section 9.1.4).

## A6MQTT:

Add the following line to your code

```
#include "A6MQTT.h"
```

Create an instance of the class

```
A6MQTT MQTT(int keepalive);  // keepalive is in seconds
```

## Methods

1. `bool connect (char *ClientIdentifier,  bool CleanSession);`
    a. Connects to the broker using the following parameters
        i. ClientIdentifier – this should be a unique string for the broker.
        ii. CleanSession – if true the broker will discard any previous session state for this ClientIdentifier, else the broker will resume any previous session state for this ClientIdentifier.
    b. No logon credentials.
    c. No Will defined.
2. `bool connect (char *ClientIdentifier,  bool CleanSession, bool UserNameFlag, bool PasswordFlag, char *UserName, char *Password, bool WillFlag, eQOS WillQoS, bool WillRetain, char *WillTopic, char *WillMessage)`
    a. Connects to the broker using the following parameters
        i. ClientIdentifier – this should be a unique string for the broker.
        ii. CleanSession – if true the broker will discard any previous session state for this ClientIdentifier, else the broker will resume any previous session state for this ClientIdentifier.
        iii. UserNameFlag – if false the UserName parameter is ignored
        iv. PasswordFlag – if false the Password parameter is ognored
        v. UserName – credential for logging on to the broker.
        vi. Password UserName – credential for logging on to the broker.
        vii. WillFlag – if false, ignore remaining parameters
        viii. WillQos – QOS of the Will message
        ix. WillRetain – If the broker should retain the message.
        x. WillTopic – Topic of the will message
        xi. WillMessage – Contents of the will message
3. `bool disconnect()`
    a. Disconnect from the broker. No parameters.

4. `bool publish(char *topic, char *message)`
    a. Publish a message. QOS 0, no RETAIN flag, no DUP flag.
5. `bool publish(char *topic, char *message, bool DUP, bool RETAIN, eQOS qos, int packetid)`
    a. Publish a message and specify the DUP, RETAIN and qos parameters. If qos > 0 packetid should be unique for each message so that its completion can be acknowledged.
6. `bool subscribe(int MessageID, char *Topic, eQOS qos)`
    a. Subscribe to the given topic at the maximum QOS. MessageID should be unique so that its completion can be acknowledged. Also so that it can be unsubscribed from.
7. `bool ping()`
    a. Ping the broker to keep alive the connection.
8. `bool unsubscribe(int messageid)`
    a. Unsubscribe from the topic with the corresponding messageid.
9. `Void Parse()`
    a. Must be invoked often in the main loop of the Arduino app. This processes data (if any) received on the UART from the A6 modem.

### Data

1. `bool waitingforConnack`
    a. A connect request was issued and the client is waiting for a CONNACK reply.
2. `bool connectedToServer`
    a. CONNACK was received and the connection request was accepted.

### Enums

1. `enum eConnectRC {CONNECT_RC_ACCEPTED, CONNECT_RC_REFUSED_PROTOCOL, CONNECT_RC_REFUSED_IDENTIFIER};`
    a. Indicates the outcome of a connection request.
2. `enum eQOS {QOS_0,QOS_1,QOS_2};`
    a. The 3 acceptable QOS values.

### Events (Callbacks)

The following events are implemented by the application developer. While not compulsory it makes life a lot easier to do so.

1. `void AutoConnect(void);`
    a. Called to implement the original connection to the broker and after any disconnection from the broker.
2. `void OnConnect(eConnectRC rc)`
    a. Called after receiving a CONNACK from the broker. It indicates the outcome of the CONNECT request.
3. `void OnDisconnect()`
    a. Called after disconnection from the broker.
4. `void OnSubscribe(int messageid)`
    a. Called after receiving a SUBACK from the broker. Messageid should be the same as in the previous subscribe request.
5. `void OnMessage(char *Topic, char *Message, bool DUP, bool RETAIN, eQOS qos)`

    a. Called after receiving a PUBLISH from the broker.
        i. Topic and Message are the contents of the message.
        ii. DUP indicates if this is a duplicate sending of the message.
        iii. RETAIN indicates if this is a retained message on the broker.
        iv. qos is the QOS of the message.
6. `void OnPubAck(int packetid)`
    a. Called after receiving a PUBACK or PUBCOMP from the broker. packetid should be the same as in the corresponding publish request (if QOS > 0).
7. `void OnUnsubscribe(int messageid)`
    a. Called when receiving an UNSUBACK from the brokere. Messageid should be the same as in the corresponding unsubscribe request.