UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

PROYECTO FIN DE CARRERA

SISTEMA DE TELE-EMERGENCIA PARA PERSONAS CON NECESIDADES ESPECIALES (SIEMERG)

Guillermo Montaner Andrés

Julio 2014

SISTEMA DE TELE-EMERGENCIA PARA PERSONAS CON NECESIDADES ESPECIALES (SIEMERG)

AUTOR: Guillermo Montaner Andrés
TUTOR: Bazil Taha Ahmed

Grupo de Radiofrecuencia: Circuitos, Antenas y Sistemas (RFCAS)
Dpto. de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio 2014

Resumen

En este proyecto se plantea la necesidad de crear un dispositivo de tele-asistencia con las siguientes características:

a) Ha de permitir su empleo en casi cualquier parte. De esta manera ha de ser un dispositivo de un tamaño más o menos reducido.

b) Ha de ser capaz de aprovechar la red móvil ya desplegada, evitando así la necesidad de realizar nuevos despliegues de redes de comunicación.

c) Ha de ofrecer localización mediante alguno de los sistemas globales de navegación por satélite (GNSS) ya desplegados.

d) Ha de ser muy fácil de manejar, abstrayendo al usuario totalmente del funcionamiento interno.

Para cumplir los objetivos del proyecto se ha desarrollado un primer dispositivo, a modo de prototipo. Este prototipo no implicó mucho desarrollo HW pero nos ha permitido tener un primer contacto con las diferentes tecnologías implicadas.

Para realizar el aviso de emergencia mediante el prototipo tuvimos que programar una aplicación. Una vez concluido el trabajo con dicho prototipo, nos proponemos la creación de un dispositivo más compacto y más cercano al modelo final de usuario. Esta tarea implicará desarrollo Hardware y Software.

Una vez se hayan desarrollado ambos dispositivos, se realizarán pruebas para analizar su funcionalidad y poder realizar una comparativa

Finalmente, exponemos una serie de conclusiones e ideas de cara al trabajo futuro sobre dispositivos de aviso de emergencia.

Palabras clave

GSM, GNSS, GPS, GLONASS, microcontrolador, emergencias, PCB, SMS, SIM,

Abstract

In this Project it is coming up the necessity to develop a tele-assistance device with the following properties:

a) It must allow almost anywhere its use. Thus it must be a device of a more or less reduced size.

b) Should be able to use the mobile networks that are already deployed, avoiding the need of deploying new communication networks.

c) Has to offer location through some of the Global Navigation Satellite Systems (GNSS) just deployed.

d) It should be very easy to use, abstracting completely the user about the inner workings.

To meet the project objectives we have devoloped a first device, as a prototype. This prototype didn't imply much Hardware developing, but has allowed us to have a first contact with the different technologies involved.

To make the emergency warning by means of the prototype we had to program an application. Once the work with the prototype is concluded, we propose the creation of a more compact and closer to the user final device model. This task will involve hardware and software development.

Once both devices are developed, tests are conducted to examine its functionality and to make a comparative.

Finally, we present some conclusions and ideas for future work on emergency warning devices.

Keywords

GSM, GNSS, GPS, GLONASS, microcontroller, tele-asistance, PCB, SMS, SIM,

Agradecimientos

Terminado este proyecto de fin de carrera, quisiera poner la vista atrás y dar las gracias a todos aquellos con los que he compartido esta etapa, y que me han permitido llegar hasta aquí. Yo no suelo ser una persona muy afectiva, por lo que éste va a ser para mí el apartado más difícil de redactar.

En primer lugar, quiero dar las gracias a mi tutor Bazil Taha por haberme dado la posibilidad de realizar este PFC y formar parte del grupo RFCAS. A Bazil Taha le debo el tener este proyecto terminado y funcionando, puesto que sin su implicación esto no hubiese sido posible. Gracias Bazil, por haberte comprometido durante tanto tiempo con este proyecto y haberme atendido siempre que lo he necesitado. Gracias, también, por todas las palabras de ánimo que me has expresado en todo este tiempo, y por depositar en mí tu confianza, incluso cuando aparecían los mayores contratiempos. Quiero agradecer a José Luis Masa su inestimable ayuda en toda esta etapa y el interés mostrado. Gracias José Luis por: atenderme en tu despacho tantas y tantas veces, darme ánimos en los momentos difíciles y proporcionarme soluciones y consejos de lo más útiles de cara a afrontar este proyecto. Agradecer también al resto de profesores que han pasado por mi etapa de estudiante, aportándome en ocasiones no sólo conocimiento, sino también valores.

Quiero mostrar mi agradecimiento a toda la gente de RFCAS por su cercanía y simpatía, gracias a las cuales he tenido una estancia muy agradable en el laboratorio C-107. Agradecer especialmente a Pablo Sánchez por haberme resuelto dudas y enseñado a emplear algunas de las herramientas del laboratorio.

Quisiera dar las gracias a los técnicos del edificio C: Manuel Vázquez y Conrado López, por tener que soportarme todo este tiempo. Gracias a ellos se han desarrollado las PCBs fabricadas en la escuela y he podido acceder al laboratorio de fabricación de circuitos siempre que lo he necesitado. Dar las gracias también a todos los estudiantes que están trabajando en las peceras de los laboratorios, especialmente a Dompi, por: prestarme material, interesarse en el proyecto, abrirme los laboratorios en horas “intempestivas”, etc.

Quisiera dar las gracias a todos mis compañeros de la facultad. Además de haber compartido tanto juntos, muchos hemos llegado a ser buenos amigos. En este párrafo quiero dar las gracias de forma especial a Charlie no sólo por estar siempre ahí, al igual que el resto de compañeros, sino por haberme proporcionado herramientas y soluciones para afrontar algunos aspectos del proyecto. Es momento también de acordarse del resto de amigos, y de los incontables momentos que hemos compartido. No sólo quiero agradecer la convivencia, sino que lo más importante es la amistad que me han brindado. También tengo muy presentes los consejos que muchos me habéis ido proporcionando a lo largo de todo el camino.

Por último, y más importante quisiera agradecer a toda mi familia por su apoyo, interés incondicional y confianza. En primer lugar quiero agradecer a mis abuelos: Luis, Carmen y María Luisa por estar siempre atentos a mis estudios. Quiero dar las gracias muy especialmente a mi madre, porque siempre ha hecho todo lo que estaba en sus manos para ayudarme, además de estar día tras día interesándose en saber cómo estoy y cómo iba el trabajo. A tal punto ha llegado su implicación y preocupación con este proyecto, que incluso intentaba buscar soluciones a algunos de los problemas técnicos que han tenido lugar. Quiero dar las gracias a mis hermanos por la preocupación mostrada, pero, sobre todo, por estar siempre ahí en los momentos más difíciles. Quisiera poder agradecer a mi padre todo lo que ha luchado para que yo estudiase en mi no corta etapa de rebeldía, y todo el esfuerzo dedicado a convertirme en un hombre de provecho, y lo más importante, por todo lo que me ha enseñado acerca de la vida. Sin duda alguna, es a la persona que más debo el haber llegado hasta aquí.

Acrónimos

GNSS: Global Navigation Satellite System
SMS: Short Message Service
CellID: Identificador de Celda
GSM: Groupe spécial mobile
PCB: Printed Circuit Board
DoD: Department of Defense (EEUU)
GPS: Global Positioning System
NAVSTAR-GPS: Navigation Satellite Timing and Ranging GPS
GEO: Geoestacionaria
SBAS: Satellite Based Augmentation System
DOP: Dilution Of Precission
AM: Modulación en Amplitud
FM: Modulación en Frecuencia
1G: Primera Generación
PLMN: Public Land Mobile Network
CEPT: Conferencia Europea de Administraciones de Correos y Telecomunicaciones
ETSI: European Telecommunications Standards Institute
MS: Mobile Station
TM: Terminal Móvil
SIM: Subscriber Identity Module
MS: Estación Móvil
TA: Adaptador de Terminal
TE: Equipo Terminal de datos
BSS: Subsistema de Estaciones Base
TSC: Controlador de Estación Base
BTS: Estación Base
MSC: Mobile Switching Center
NSS: Network Switching Subsystem
GMSC: Gateway Mobile Switching Center
VLR: Visitor Location Register
AuC: Authentication Center
EIR: Equipment Identification Register
OMSS: Operation and Maintenance Subsystem
OMC: Operation and Maintenance Center
SMSC: Short Message Service Center
HLR: Home Location Register
RSS: Received Signal Strength
AoA: Angle of Arrival
BGA: Ball Grid Array
UART: Transmisor-Receptor Universal Asíncrono
GPRS: General Packet Radio Service
NMEA: National Marine Electronics Association
USB: Universal Serial Bus
PC: Personal Computer
PDU: Protocol Data Unit
MCC: Mobile Country Code
MNC: Mobile Network Code
LAC: Local Area Code

PIN: Personal Identifier Number
BCCH: Broadcast Control Channel
ASCII: American Standard Code for Information Interchange
UTC: Coordinated Universal Time
HDOP: Horizontal Dilution Of Precission
LED: Ligth-Emmiting Diode
FIFO: First In, First Out
SPI: Serial Peripherical Interface
RF: Radio-Frecuencia
LGA: Land Grid Array
EVK2: Evaluation Kit versión 2
RS232: Recommended Standard 232
CMOS: Complementary Metal-Oxide-Semiconductor
DC: Direct Current
CTS: Clear To Send
RTS: Request To Send
DTR: Data Terminal Ready
DSR: Data Set Ready
DCD: Data Carrier Detect
ROM: Read-Only Memory
RAM: Random Acces Memory
Li-ION: Ion de Litio
THD: Through-Hole Device
SMD: Surface Mount Device
TEM: Transversal Electromagnético
FR-4: Fiberglass Reinforced Epoxy
UAM: Universidad Autonóma de Madrid
VSWR: voltage standing wave ratio
UMTS: Universal Mobile Telecommunications System
API: Application Programming Interface
SMA: Sub-Miniature version A

Índice de Contenido

Índice de Figuras

Índice de Tablas

Capítulo 1
Introducción

1.1 Introducción

El proyecto propuesto se centra en la tele-atención a personas con patologías de cierta importancia y que puedan requerir de asistencia rápida y eficaz en cualquier momento. Generalmente, estas personas son de avanzada edad y con un natural rechazo a las nuevas tecnologías. La propuesta se centra en este tipo de personas, intentando proporcionarles una herramienta fácil de aviso a los sistemas de emergencia y salud pública ante cualquier eventualidad. Para ello, se trata de aprovechar la red móvil celular y la localización GNSS, para poder realizar un aviso eficaz y preciso de su posición, circunstancias clínicas, etc. El empleo de la red móvil garantizará cobertura para este servicio casi en la totalidad de la superficie del territorio nacional, y por lo tanto, no limitaría el aviso de emergencia al interior de la vivienda del paciente, como los actuales servicios de tele-asistencia que se prestan hoy en día.

Así mismo, la utilización de dicha red móvil permitiría una primera localización “gruesa” del sujeto para poder informar, ya en el primer mensaje de aviso, de la zona donde se encuentra. La incorporación de un receptor GNSS básico, permitiría en un espacio temporal en torno al minuto, dar una posterior localización más exacta de la persona que precise asistencia.

1.2 Motivaciones y Objetivos

Existe un gran número de personas cuya situación hace que puedan requerir de asistencia sanitaria en caso de emergencia. Permitir a estas personas cierta independencia, como puede ser el hecho de salir a la calle de su ciudad, es la motivación fundamental de este proyecto.

El objetivo del proyecto será el desarrollo de un prototipo sencillo de transmisor, que simplemente mediante el accionamiento de un pulsador, realice un aviso de emergencia a través de la red móvil.

Para ello se enviará un primer mensaje a través del servicio de mensajes cortos (SMS) la ubicación “gruesa”, más tarde se enviará otro SMS con una localización más precisa del sujeto. El número de teléfono, propio de cada usuario, nos permitirá identificar a la persona que solicite el aviso de emergencia.

Como se ha comentado anteriormente, la ubicación del sujeto se tendrá de dos maneras distintas. En primer lugar, y de forma rápida, conociendo el número identificador de estación base móvil a la cual está conectado el terminal (CellID); finalmente, y en un espacio inferior a un minuto desde que el sujeto accione el botón del transmisor, se espera tener una localización bastante precisa por parte del alguno de los sistemas GNSS ya desplegados.

En cuanto al primer mensaje de texto, la asociación entre los posibles ID de célula y la ubicación geográfica exacta de las diferentes estaciones base correrá a cargo de los operadores del servicio. En el segundo mensaje de texto se enviarán las coordenadas obtenidas directamente del módulo GNSS; este proceso requiere más tiempo debido a que estos módulos necesitan bastante información por parte de los diferentes satélites que estén al alcance (órbita, reloj, etc.) para ofrecer a su salida la localización del sujeto.

1.3 Estado del arte

En este apartado veremos algunos sistemas similares al propuesto en este proyecto y haremos un pequeño análisis de la tecnología en la cual se basan. Mostraremos sus características y extraeremos ideas básicas de cara al futuro diseño. También atenderemos a sus limitaciones, desde el punto de vista de los objetivos de nuestro proyecto.

En primer lugar, hablaremos del Servicio de Tele-asistencia en España, el cual se implantó en el año 1991. Este servicio es prestado gracias a un dispositivo portátil, el cual se interconecta de forma inalámbrica al teléfono fijo del domicilio o a otro terminal específico. Este dispositivo es muy fácil de usar (se acciona pulsando un único pulsador) y es fácil de llevar siempre encima (forma de pulsera o colgante). El dispositivo portátil tiene una cobertura suficiente como para poder dar un aviso de emergencia desde cualquier parte de la casa. Cuando se activa dicho botón se realiza una llamada a un centro de atención, desde el cual personal preparado tomará las medidas de actuación oportunas. Además, también se realizan llamadas periódicas al domicilio de la persona que accede a este servicio, para realizar un seguimiento y almacenar información actualizada sobre su situación. El terminar con la **limitación del acceso** a los usuarios de este servicio únicamente desde su **domicilio**, como hemos visto anteriormente, es una de las mayores motivaciones de este proyecto.

Pasaremos ahora a hablar de los dispositivos portátiles de localización que emplean la red móvil para comunicarse. Algunos de ellos llevan un receptor GNSS incorporado, el cual permite además disponer de una localización precisa del dispositivo en la mayoría de casos. Estos dispositivos fueron desarrollados en paralelo al curso de este proyecto, y los primeros modelos comerciales pudieron verse a mediados de 2013. Estos dispositivos están pensados para ser empleados por personas enfermas, o ancianas, o ser utilizados en excursiones, para dar un aviso de emergencia u ofrecer servicios de seguimiento. La notificación del aviso de emergencia, en la mayoría de dispositivos, se realiza a una serie de números de teléfono previamente almacenados. Este aviso suele constar de un mensaje de texto con las coordenadas del sujeto acompañado de una llamada telefónica (se irán llamando a los números de teléfono almacenados, por orden hasta que uno dé respuesta). La mayoría de estos dispositivos contienen a grandes rasgos, desde un punto de vista electrónico, un módulo celular, un receptor GNSS y un micro-controlador el cual permite el manejo, configuración y procesado de datos de estos dos módulos.

La mayoría de estos dispositivos permiten múltiples configuraciones, las cuales ha de ser programadas previamente por el usuario. Un aspecto interesante de algunos de estos dispositivos es la incorporación de un servicio denominado ‘geo-fencing’. Este servicio permite delimitar zonas o áreas mediante coordenadas, para lo cual se deberá programar el dispositivo. Si el dispositivo rebasa estas delimitaciones el aviso de emergencia se da automáticamente, y se mantiene un seguimiento del dispositivo mediante el envío de SMS cada cierto período de tiempo. En la siguiente figura podemos contemplar varios modelos de dispositivos de localización comerciales.

Figura 1.1: Modelos comerciales de dispositivos de seguimiento celulares.

Como ya hemos visto, la mayoría de los dispositivos de localización poseen un receptor GNSS y un módulo celular. Estos dos módulos, normalmente, se gestionan y controlan por medio de un microcontrolador. Antes de poder empezar a usarse, el usuario necesita realizar una configuración previa. Esta configuración, normalmente se realiza por medio de un software específico que se proporciona junto con el dispositivo. La arquitectura básica de estos dispositivos puede observarse en la figura siguiente.

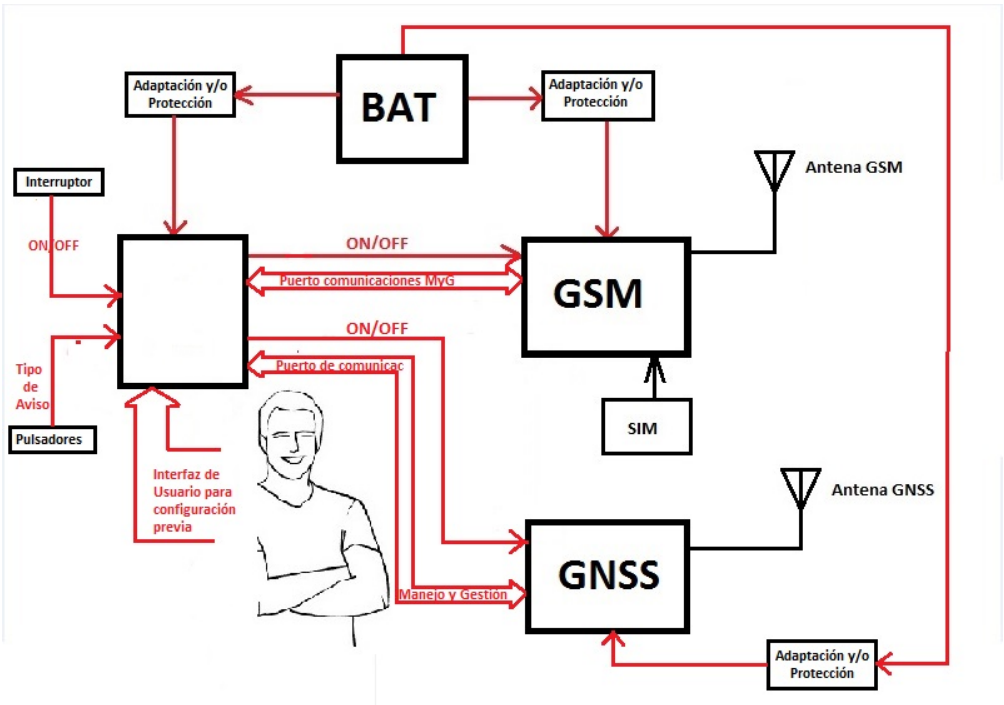


Figura 1.2: Arquitectura básica en dispositivos de seguimiento celulares.

En la figura superior podemos apreciar cómo el microcontrolador es el encargado del encendido y apagado de los módulos GNSS y celular (en esta figura el módulo celular es representado mediante el bloque “GSM”). El microcontrolador es el encargado, así mismo, del manejo y gestión de ambos módulos mediante sus puertos de comunicación. Podemos observar que el usuario puede interaccionar con el microcontrolador para realizar la configuración previa del

dispositivo, a través de una interfaz dispuesta para tal propósito. Para realizar el aviso de emergencia, el dispositivo puede poseer una serie de pulsadores. El encendido del dispositivo suele realizarse mediante un interruptor. Una misma batería alimentará tanto a los módulos celular y GNSS así como al microcontrolador; para ello habrá que interponer circuitos de adaptación y protección, ya que lo más usual será encontrarnos con exigencias distintas para la alimentación de estos tres chips.

En este proyecto, a diferencia de los anteriores dispositivos, evitaremos al usuario la parte de configuración previa del dispositivo. De esta manera, ahorraremos esfuerzos innecesarios a los usuarios (los cuales, en ocasiones, serán reacios a las nuevas tecnologías) y evitaremos que se puedan cometer errores a la hora de configurar estos dispositivos. Además el aviso será realizado a una central de atención atendida por personal cualificado, al igual que el servicio de tele-asistencia español.

Respecto a los módulos celulares, existen multitud de fabricantes de productos para este sector, tales como: Telit, UBlox, Sierra Wireless o SIMCOM. Cada uno de estos fabricantes tiene disponibles en su web toda la información necesaria para trabajar con estos módulos (funcionalidad, manejo y gestión, características eléctricas y mecánicas, montaje sobre PCB, etc.…). Estas compañías comercializan módulos celulares compatibles con los distintos sistemas de telefonía móvil desplegados por todo el mundo, tanto de segunda como de tercera generación (normalmente los módulos de tercera generación permiten comunicación también mediante los sistemas de segunda generación). La gran mayoría de estas compañías también desarrollan receptores GNSS.

En cuanto a los módulos o receptores GNSS, también son numerosas las compañías que los fabrican. Como hemos observado en el párrafo anterior, algunas de estas compañías, tales como: Telit, u-Blox, Sierra Wireless y otras tantas; fabrican tanto receptores GNSS como módulos celulares. Algunas de estas compañías fabrican GNSS embebidos dentro de los módulos celulares, los cuales son gestionados mediante interfaces software, propias de cada fabricante.

Capítulo 2
Tecnologías implicadas en el desarrollo del proyecto

2.1 Sistema de Satélites para Navegación Global (GNSS)

2.1.1 Introducción

GNSS (Global Navigation Satellite System) engloba todas aquellas tecnologías de geoposicionamiento por satélite que, dando cobertura global, funcionan de manera autónoma.

A pesar de que durante la década de los 60 el ejército de Estados Unidos había desarrollado una serie de sistemas de posicionamiento basados en satélites como, por ejemplo el sistema 'Transit' (desarrollado por la Marina, era capaz de ofrecer posicionamiento a los submarinos con una precisión de unos 25 m. en un espacio de tiempo inferior a 10 minutos), el primer sistema GNSS completo nació en torno a la década de los 70, con el desarrollo del sistema GPS (Global Positioning System) por parte del Departamento de Defensa de Estados Unidos (DoD). Este sistema se había ideado para la localización de tropas u objetivos, el guiado de misiles, etc.

Con orígenes en aplicaciones militares secretas, la tecnología GPS ha entrado a formar parte de nuestra vida cotidiana, ya que el poder conocer la posición de un receptor GNSS en cuatro dimensiones (longitud, latitud, altitud y tiempo) ha dado lugar a multitud de aplicaciones civiles (sistemas de seguimiento de flotas, navegadores GPS para vehículos terrestres marítimos y aéreos, etc.).

2.1.2 Nacimiento e historia de los primeros GNSS

Cuando los soviéticos lanzaron en 1957 el primer satélite artificial en órbita terrestre, el Sputnik I, investigadores del Laboratorio de Física Aplicada de Johns Hopkins en Baltimore, descubrieron que debido a que ellos sabían su posición en la tierra, podían calcular la órbita exacta del satélite midiendo el corrimiento de frecuencia (efecto doppler) de la señal de radio transmitida por Sputnik según se acercaba y alejaba de ellos. Sólo tomó un pequeño salto de intuición concluir que lo opuesto también era posible; es decir, que uno podría determinar su posición exacta en la tierra conociendo la órbita del satélite [Ver Ref. 2].

El lanzamiento del satélite espacial estadounidense Vanguard en 1959 puso de manifiesto que la transmisión de señales de radio desde el espacio podría servir para orientarnos y situarnos en la superficie terrestre o, a la inversa, localizar un punto cualquiera en la Tierra.

Durante los años 60, varias organizaciones del ejército de Estados Unidos trabajaron de forma independiente en sus propias versiones de sistemas de navegación por satélite. En 1973 el DoD ordenó a las diferentes organizaciones que cooperasen en el desarrollo de un sistema único. Fue así como nació el NAVSTAR-GPS (NAVigation Satellite Timing And Ranging Global Positioning System), sistema que no estuvo operativo hasta 1993. Finalmente en 1995 el sistema GPS (Nombre por el cual es más conocido el sistema NAVSTAR-GPS) tuvo plena operatividad. A diferencia de otros antecesores, el cálculo de la posición no se basaba en el corrimiento Doppler de frecuencia, sino en el retardo de la señal que experimenta el receptor (este fenómeno fue probado con anterioridad en un proyecto de la Marina estadounidense llamado 'Timation'). Para poder disponer de unos cálculos precisos del retardo de la señal éstos satélites llevan relojes atómicos a bordo.

En la actualidad el sistema GPS cuenta con una red de 24 a 30 satélites (mínimo 24 distribuidos en 6 órbitas planares con una inclinación de 55 º respecto al ecuador terrestre), garantizando un mínimo de 5 satélites visibles en cualquier parte del mundo.

2.1.3 Estructura y breve introducción al funcionamiento de un Sistema GNSS

Cualquier sistema GNSS está dividido en 3 segmentos diferenciados:

1.- El **segmento espacial**, formado por el conjunto de satélites de navegación y comunicación. Los satélites de navegación son aquellos empleados para ofrecer un posicionamiento, mientras que los de comunicación, particulares de un país o región, se sitúan sobre órbitas GEO y sirven para aplicar correcciones y aumentar la precisión del sistema resultante (SBAS).

2.- El **segmento de control**, está formado por estaciones en tierra. Su objetivo es garantizar las prestaciones de los satélites, aplicándoles correcciones en sus órbitas y sincronización en sus relojes atómicos.

3.- El **segmento de usuario**, formados por los receptores. Éstos serán los encargados de recoger la información, presente en la señal de los diferentes satélites a su alcance, para calcular su posicionamiento.

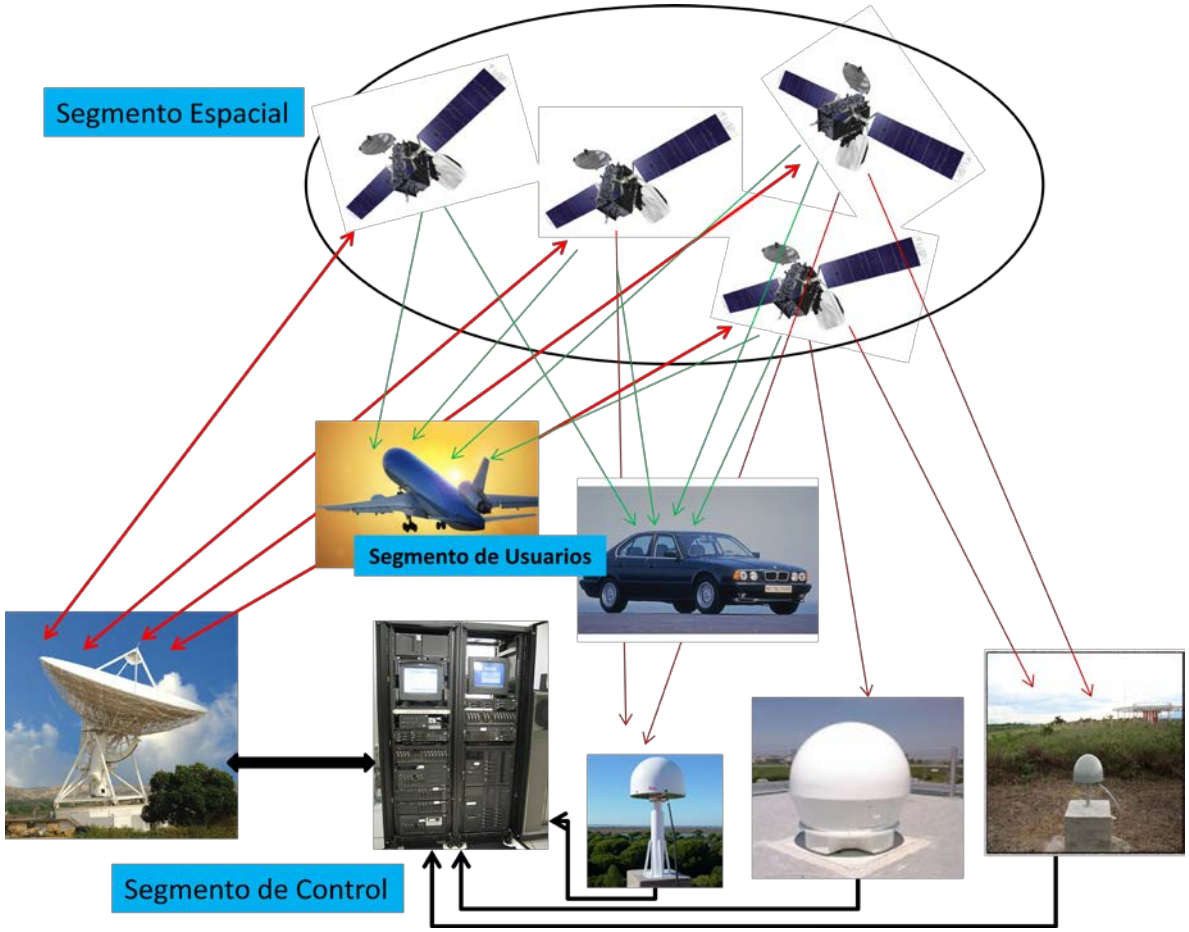


Figura 2.1: Estructura básica de un sistema GNSS.

En la imagen de la página anterior podemos observar cómo se estructura un simple sistema GNSS. Podemos observar como el segmento de control tiene varias estaciones fijas monitoras y de carga y una estación de control. Esta estación de control es capaz de enviar correcciones a los satélites, además de recibir la señal del segmento espacial. Las estaciones monitoras y de carga reciben la

señal del segmento espacial, las correcciones que consideren necesarias son enviadas al centro de procesamiento (en el caso de pertenecer a un sistema SBAS (Sistema de mejora de la precisión basada en satélites), éstas señales podrían tener como destinatario un satélite GEO, en vez del centro de procesamiento). El centro de procesamiento se encargará de calcular las correcciones a aplicar a los satélites y pasar esta información a la estación de control. El segmento de usuarios recibirá la señal del segmento espacial en sus receptores GNSS, los cuales calcularán su posición a partir de dichas señales. El segmento espacial envía las señales que serán recibidas en los segmentos de control y de usuarios. Sobre este modelo básico, se podría añadir SBAS, aumentando simplemente el segmento espacial con satélites sobre órbitas GEO (disponiendo para este fin de unas cuantas estaciones terrenas de control para comunicación con este satélite).

Funcionamiento resumido de un sistema GNSS

Para el cálculo de la posición del receptor es necesario que éste conozca la posición del satélite y el tiempo del reloj a bordo del mismo. Las señales enviadas por el segmento espacial recogen estos datos. La posición del satélite se da a conocer a partir de las efemérides, que son unas tablas astronómicas que describen las posiciones precisas de los satélites en función del tiempo. El receptor GNSS es el encargado de calcular la distancia que le separa del satélite gracias a una medición precisa del retardo que sufre la señal del satélite. Conocidas la velocidad de propagación de la onda electromagnética y el tiempo transcurrido desde que se emite la señal y ésta llega al receptor, sabremos la distancia que los separa. De esta manera cada satélite indicará al receptor que éste se encuentra en un punto de la superficie de una esfera con centro en el propio satélite cuyo radio es la distancia que le separa del satélite. Basándonos en este método, serían necesarios al menos tres satélites, para calcular la posición del receptor en tres dimensiones. No obstante, en estos sistemas GNSS se requiere tener al menos cuatro satélites “a la vista” para calcular la posición del receptor, ya que este satélite extra permitiría eliminar los errores de sincronismo.

2.1.4 Principales fuentes de error en un sistema GNSS

A pesar de que estos sistemas pueden proporcionar precisiones decimétricas (normalmente los receptores baratos ofrecen una precisión de unos pocos metros) en sus posicionamientos sobre usuarios civiles, la información procedente de cada satélite que viaja en una señal puede verse afectada por distintas fuentes de error. Las más destacadas son:

1.- Efectos atmosféricos: cuando una señal electromagnética atraviesa la ionosfera, su velocidad de propagación se ve reducida. Estos retrasos pueden introducir un error significativo en el cálculo de la distancia, ya que estos cálculos se basan en la velocidad de propagación de la señal procedente del satélite. Además, este retraso atmosférico no tiene un valor constante, sino que hay varios factores que influyen, tales como la elevación del satélite, el efecto del Sol sobre la densidad ionosférica o la cantidad de vapor de agua contenida en la atmósfera.

2.- Efectos multitrayecto: este error puede producirse cuando el receptor se sitúa cerca de una superficie electromagnéticamente reflectora. Ocurren cuando la antena del receptor detecta una señal satelital que previamente ha sido reflejada sobre la superficie reflectante. Esto puede suponen que sobre la señal directa se superpongan copias desplazadas en el tiempo (retardo) de ésta señal, produciendo errores a la hora de calcular la distancia. Una forma de subsanar en parte estos errores es el empleo de antenas receptoras que dispongan de una serie de filtros que eliminen aquellas señales cuyo ángulo de elevación esté por debajo de cierto umbral.

3.- Errores de efemérides: producido cuando un satélite se encuentra en una posición diferente de la que indica a los receptores. Estos errores se corrigen desde el segmento de control, como se ha comentado anteriormente.

4.-Errores de reloj y debidos a la relatividad: los relojes que llevan a bordo los diferentes satélites presentan una precisión nanométrica, pero a pesar de ello, algunas veces estos relojes presentan derivas de tiempo, las cuales afectan directamente al cálculo de la distancia satélite-receptor. Al igual que los errores de efemérides, es responsabilidad del segmento de control la subsanación o reducción de este efecto. También, gracias a la teoría de la relatividad especial y a la teoría general de la relatividad de Einstein, se pueden predecir algunas derivas de tiempo motivadas por el efecto del movimiento de los satélites o de la gravedad de la Tierra, las cuales afectan al sincronismo de los relojes del satélite y del receptor; teniendo esto en cuenta los efectos de la relatividad pueden ser subsanados casi en su totalidad.

5.- Dilución de precisión (DOP): es una medida de la fortaleza de la geometría de los satélites, relacionada con la distancia entre los satélites y su posición en el cielo. Cuando los satélites al alcance se encuentran muy próximos entre sí, la precisión puede verse degradada. En cambio, si los satélites están muy alejados entre sí y con buenas elevaciones, la precisión del posicionamiento aumenta. Además éste valor puede obtenerse en la mayoría de receptores GNSS, una vez obtenido un posicionamiento. En este proyecto este valor será incluido a la hora de enviar el mensaje de emergencia con el posicionamiento GNSS como indicativo de su posible precisión. Cuanto mayor sea el valor DOP, mayor será la probabilidad de que el posicionamiento sea impreciso, y mayor será a su vez el posible margen de imprecisión (la distancia entre el verdadero posicionamiento y el mostrado a la salida de nuestro receptor).

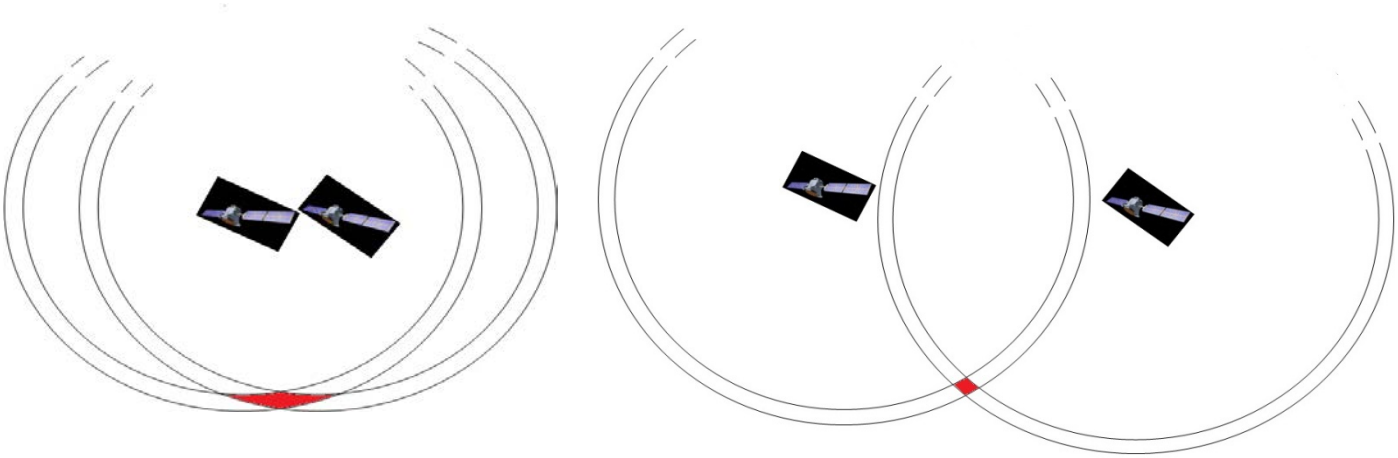


Figura 2.2: Ejemplo intuitivo DOP con 2 satélites

Como podemos observar en la figura de arriba, una mayor altitud y mayor separación entre los satélites visibles (imagen de la derecha) reduce el posible margen de error. Cada par de círculos concéntricos (cuyo centro es uno de los satélites) representa el posible margen de error de cada satélite. El margen de error resultante aparece coloreado en rojo. Esta imagen no pretende ser un reflejo de la realidad, puesto que como sabemos, se necesitan al menos cuatro satélites para poder ofrecer un posicionamiento; y estos círculos concéntricos deberían ser sustituidos por esferas concéntricas y el error sería en 3 dimensiones.

Es necesario contemplar los errores que pueden afectar a la precisión de nuestro posicionamiento GNSS, ya que la aplicación de nuestro proyecto es dar un aviso de carácter urgente. Muchos son los errores que podrían entrar en juego a la hora de obtener un posicionamiento. Por ejemplo, en zonas urbanas con calles estrechas, los errores multitrayecto sumados a la posibilidad de que el valor DOP sea elevado (los edificios bloquearían la visibilidad de multitud de satélites, aquellos visibles quedarían muy agrupados en el cielo y las señales

procedentes de los satélites no visibles que sean alcanzados seguramente estén afectados por el multitrayecto) pueden provocar desviaciones de hasta varios kilómetros en el posicionamiento, como veremos más adelante en esta memoria. En la imagen siguiente podemos apreciar cómo el despejamiento del cielo está íntimamente ligado al valor del DOP.

Figura 2.3: Cómo el despejamiento afecta al DOP.

Será por tanto necesario indicar al destinatario del aviso de emergencia (Servicios Sanitarios, Operadora de Emergencias, etc.) de posibles imprecisiones en el posicionamiento. Esto será implementado con el envío del valor DOP asociado al posicionamiento obtenido en nuestro receptor GNSS. También consideraremos cambios en el valor del DOP dentro de un mismo aviso de emergencia, por lo que si un posicionamiento es primeramente calificado de “posiblemente impreciso” (por su valor del DOP) y más tarde se obtiene un valor del DOP considerado como “posiblemente preciso” se procederá a enviar este nuevo posicionamiento, junto al nuevo DOP, en el momento en que se detecte en nuestro receptor GNSS.

2.2 Redes Telefónicas Públicas Móviles

2.2.1Nacimiento e Historia

El comienzo de la telefonía móvil puede tener su origen en los inicios de la Segunda Guerra Mundial, donde se apreció que era necesario poder establecer la comunicación a distancia de un lugar a otro. Los primeros sistemas de telefonía móvil civil empiezan a desarrollarse a principios de los años 40 en los Estados Unidos. Eran sistemas de radio analógicos que emplearon primero Modulación en Amplitud (AM) y posteriormente Modulación en Frecuencia (FM), la cual se popularizó por su mayor inmunidad a interferencias. No obstante, los primeros terminales o equipos eran grandes y pesados, por lo que estaban destinados casi exclusivamente a su uso a bordo de vehículos. Debemos remontarnos al 3 de Abril de 1973, para ver la primera llamada realizada desde un teléfono móvil portable. A principios de la década de los 80 puede decirse que nace la Primera Generación (1G) de telefonía móvil, cuando el fabricante Ericsson lanza el sistema Nordic Mobile Telephony 450 MHz (NMT 450), el cual seguía empleando canales de radio analógicos con FM. Lo que hacía que este sistema se englobase dentro de la primera generación de PLMN (redes móviles públicas terrestres) era la prestación de movilidad plasmada en las propiedades de localización, traspaso e itinerancia.

2.2.2 Red de Segunda Generación GSM

El desarrollo de las primeras PLMN, que llamamos de primera generación, fue anárquico, con numerosas incompatibilidades entre sistemas. Las prestaciones de estos servicios eran elevadas, y se registró una gran demanda que fue contenida mediante la aplicación de tarifas elevadas.

Ya en la década de los 80 se plantea en Europa el problema de las limitaciones e incompatibilidades de las redes de primera generación. En 1982 se crea dentro del seno de la CEPT un grupo de trabajo llamado GSM con el objetivo de crear un estándar paneuropeo para todos los países de la CEPT, que permitiese la itinerancia internacional. Finalmente, en 1989 recae sobre la ETSI la responsabilidad de esta normalización. Como resultado se desarrolló el sistema de segunda generación más extendido en todo el mundo, abarcando más de 200 operadoras en más de 190 países. Se decidió operar sobre la banda de los 900 y 1800 MHz con canales de radio totalmente digitales, mejorando la eficiencia del espectro radioeléctrico y brindando elevadas prestaciones en cuanto a cobertura, capacidad y calidad de servicio a un gran número de usuarios. Además este sistema fue creado para poder operar junto con las redes fijas preexistentes.

2.2.3 Resumen de la arquitectura del sistema GSM

El sistema GSM está compuesto por varios bloques funcionales, cuyas funciones e interfaces están completamente especificadas para lograr ofrecer el servicio de telefonía móvil a un gran número de usuarios finales. La red GSM está constituida por las siguientes entidades:

a) **Estación móvil (MS):** comprende todos los elementos empleados por el abonado del servicio, tales como: el terminal móvil (TM), la tarjeta SIM (Subscriber Identity Module) que es la tarjeta de abonado (proporcionada por el operador al usuario, al contratar su servicio), el adaptador de terminal (TA) para la interconexión del teléfono móvil con un equipo terminal de datos (TE), empleado para la transmisión de datos.

b) **Subsistema de estaciones base (BSS):** comprende las funciones de la capa física (interfaz radio) para la conexión con las MS. En el BSS pueden distinguirse dos bloques funcionales denominados: controlador de estación base (BSC) y estación base (BTS). Una BTS dispone de todos los dispositivos necesarios para la transmisión y recepción por radio para cubrir una celda o sector. Cada MS se conecta a una única BTS, aquella que ofrezca mejor cobertura y permita la conexión (generalmente del mismo operador con el que el MS contrató los servicios). Una BSC interconecta varias BTS con un elemento de conmutación denominado centro de conmutación móvil (MSC), para el encaminamiento de las llamadas hacia la red. El MSC pertenece al subsistema de red y conmutación. La BSC se encarga de la asignación y liberación de los canales de radio, así como de la gestión de traspaso de llamada cuando el MSC cambia a una BTS dependiente de la misma BSC.

c) **Subsistema de red y conmutación (NSS):** este subsistema se encarga de la interconexión con otras redes de telefonía y de la gestión de las bases de datos con la información relativa a todos los abonados al servicio. Las funciones de conmutación internas a la red GSM se realizan mediante los MSC, mientras que los centros de conmutación móvil pasarela (GMSC) son los elementos de conexión con otras redes. La gestión de las bases de datos es realizada por el registro central de abonados (HLR) y el registro de posiciones visitante (VLR), estas bases de datos contienen toda la información relativa a los abonados (perfil del servicio contratado, tarificación, llaves de autenticación y cifrado, localización del móvil). Para proteger la comunicación contra la intrusión, el fraude y el uso desautorizado de equipos, está el centro de autenticación (AuC) y el registro de identificadores de equipo (EIR).

d) **Subsistema de operación y mantenimiento (OMSS):** el elemento principal de este subsistema es el centro de operación y mantenimiento (OMC) que se encarga de la supervisión y mantenimiento del resto de bloques. Permite la solución de problemas y fallos que aparezcan, así como la optimización del rendimiento del sistema mejorando la configuración de los parámetros que controlan los procedimientos de comunicación.

En la siguiente imagen podremos apreciar la arquitectura del sistema GSM, con sus diferentes bloques funcionales.

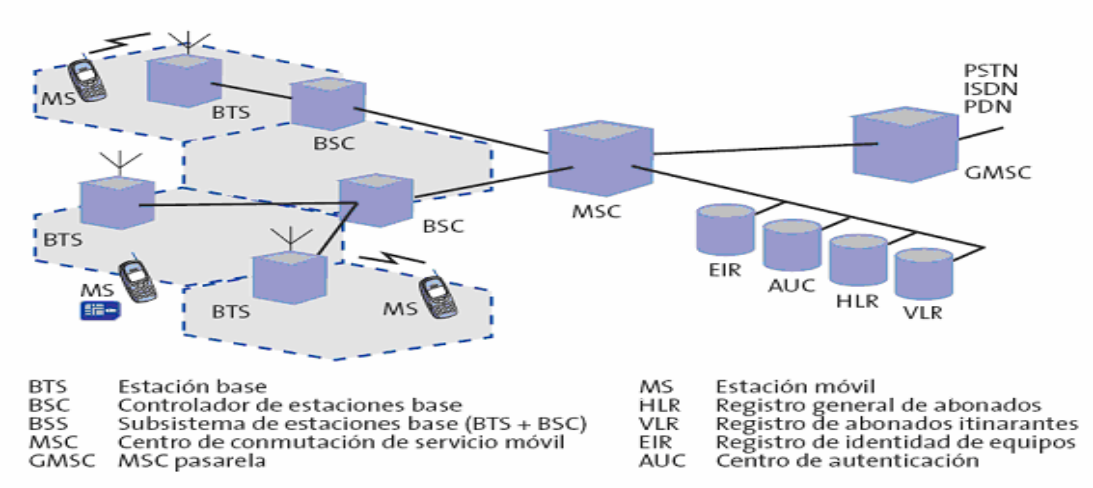


Figura 2.4: Arquitectura red GSM.

2.2.4 Envío de mensajes cortos de texto a través de la red GSM

Para hablar del envío de mensajes cortos a través de la red GSM es necesario hablar, en primer lugar, de un elemento perteneciente a la arquitectura GSM: el SMSC (Short Message Service Center). Este centro es el encargado de recibir, almacenar y enviar los SMS. Cada operador tiene un SMSC asociado a cada MSC. Este centro también se encarga de las tareas de autenticación y control necesarias para el envío de SMS, tales como: autenticación de usuario, comprobación de si puede o no enviar mensajes, verificación del estado del teléfono destino y verificación del estado de teléfonos con mensajes pendientes.

Cuando un MS envía un mensaje, lo primero que hace es enviarlo hacia el SMSC. Para que el mensaje llegue al SMSC se envía primero a la BTS a la cual está conectado dicho MS, para que esto suceda, previamente el MSC del nivel superior consultará al VLR si éste usuario tiene los permisos necesarios. Esta BTS lo reenvía a su BSC, la cual a su vez lo reenvía a su MSC. Es el MSC el que se encarga de pasarlo al SMSC, el cual va encolando los mensajes que este MSC le va pasando. Tan pronto llega un nuevo mensaje al SMSC, éste se pone el último de una cola. Cuando le toca el turno a este mensaje, el SMSC consulta al HLR por el MSC de destino y lo redirige allí. El SMSC de destino enviará un informe del estado del mensaje al MSC de origen para que lo reenvíe al usuario que originó el mensaje.

El SMSC de destino almacenará dicho mensaje en su propia cola, y permanecerá en ella hasta que sea entregado o deje de ser vigente (los SMS tienen un tiempo de caducidad, rebasado el cual son desechados). Cuando este mensaje pueda ser enviado, el SMSC consultará al HLR para obtener la información necesaria para encaminar el mensaje hasta su destino. También hará una consulta al VLR al cual está conectado el usuario destino (que puede ser de la misma operadora que el usuario del mensaje origen o no) para saber si el usuario destino está conectado (disponible). Si el usuario está disponible, entonces el SMSC envía este mensaje a su MSC, el cual lo pasa a la BSC y BTS hasta alcanzarle. Cuando éste lo recibe, el MSC informa al SMSC de que el mensaje se entregó y puede ser borrado de su base de datos (cola). Opcionalmente, el SMSC de destino informa al SMSC de origen de que el mensaje fue entregado, y si el usuario origen lo solicita, este SMSC origen le avisará de la recepción del mensaje por parte del usuario destino. Este aviso se realiza mediante los denominados informes de entrega.

2.2.5 Posicionamiento mediante Telefonía Móvil

Como puede deducirse de lo expuesto en el primer apartado de esta memoria, la red móvil existente puede ser empleada para ofrecer un servicio de posicionamiento. Por lo general, la precisión de este tipo de posicionamientos es peor que el realizado por los receptores GNSS. No obstante, en este proyecto combinaremos ambas técnicas. De hecho, el empleo de ambas técnicas puede tener cierto carácter de complementariedad, ya que es fácil ver que en densas áreas urbanas la calidad de señal GNSS puede llegar muy degradada (por la presencia de obstáculos, edificios, etc.) mientras que tendremos una amplia cobertura móvil (además de la reducción del radio de celda); por otro lado, en áreas rurales la recepción de la señal GNSS será buena, gracias a la ausencia de obstáculos que influyan en la señales GNSS, y por otro lado el número de estaciones base al alcance del receptor será limitado.

Existen multitud de variantes para realizar servicios de localización mediante la red móvil. A continuación se presentaremos los métodos más relevantes.

a) Identidad de Celda o Cell Id

Este método es el más simple de todos. Conocida la celda en la cual está registrado el terminal móvil, podremos asegurar que el usuario se encuentra dentro de su área de cobertura. La precisión de este posicionamiento depende de la longitud del radio de la celda, ya que el usuario puede estar en cualquier lugar dentro del área de celda. El error máximo de este método será igual al radio de cobertura, el cual en entornos urbanos densamente poblados puede estar en torno a los 150 ó 200 metros, mientras que para áreas rurales puede ser superior a los 4 Km.

b) RSS (Received Signal Strength)

En este método nos valemos de la potencia recibida de señal. Nos basamos en la pérdida de potencia que la señal sufre al atravesar el medio de propagación (para el espacio libre la potencia de señal decae con el cuadrado de la distancia). Si empleamos una sola estación base el resultado será un anillo en torno a dicha estación base; empleando dos estaciones base tendremos la intersección de dos anillos. Para obtener una solución única necesitaremos de al menos tres estaciones base.

No obstante, este método resulta bastante impreciso debido a que las señales pueden estar afectadas por obstáculos, inconveniencias en cuanto a la orientación de las antenas, multitrayectos, etc. Para aumentar la precisión es necesario trabajar con modelos de propagación más avanzados o estudiar la distribución de campo en el espacio en torno a la BTS.

c) Timing Advance o Cell Id++

Con este método, además de conocer el identificador de celda, se obtiene una estimación del tiempo de ida y vuelta de la señal. Este tiempo ha de traducirse a distancia, teniendo en cuenta la velocidad de propagación de la señal, de tal modo que

$$D = (½) * (c*TA)$$

Este método, a diferencia del RSS, no se ve afectado por la presencia de obstáculos. A pesar de esto, sí puede verse afectado por los multitrayectos.

Esta estimación del tiempo de ida y vuelta puede hacerse mediante el parámetro TA (Timing Advance) disponible para la celda de registro. Este parámetro es empleado por la BS para una correcta recepción de las ráfagas de datos para acceso inicial y la reducción de los tiempos de guarda de las ráfagas de datos (ya que la celda GSM puede tener un radio de hasta 35 Km, según el estándar). En GSM la tasa de transmisión de datos es de 270.833 Kbps, por lo tanto, la duración de un bit corresponde a 3’69 µs. El parámetro TA se aumenta en unidad por cada retardo (de ida y vuelta) de 3’69 µs, y al ser un registro de 6 bits puede almacenar valores comprendidos entre 0 y 63. Por lo tanto la distancia de error empleando éste método es de 553.5 metros, independientemente del radio de la celda.

d) AoA (Angle of Arrival)

Con este método se calcula el ángulo con que llega la señal a la antena de la BTS. Los multitrayectos que pueda sufrir la señal pueden afectar muy negativamente a la precisión de este método. Es por ello que se deben emplear al menos dos estaciones base equipadas con antenas multi-array y/o una gran diversidad y así conseguir una precisión adecuada. Uno de los principales inconvenientes de éste método es que se necesitaría equipar las actuales BTS con unos receptores especiales. Además un giro de estas antenas por el viento puede afectar notablemente a la precisión.

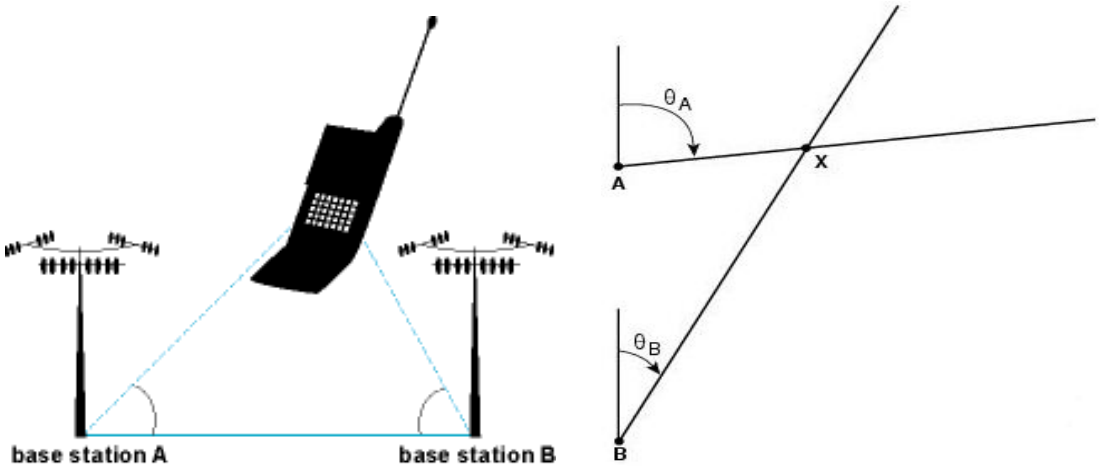


Figura 2.5: Esquema del cálculo del posicionamiento mediante AoA.

Capítulo 3
El prototipo del proyecto

3.1 Introducción

Normalmente, para establecer comunicaciones mediante la red GSM, se emplea un terminal móvil. De hecho, los terminales más modernos incorporan sistemas operativos que permiten correr multitud de aplicaciones con variadas funcionalidades, incluyendo normalmente, un receptor GPS que podrá ser gestionado por medio de alguna de estas aplicaciones.

No obstante, debemos recordar que el público al cual va destinado este proyecto, mayoritariamente compuesto por gente mayor, no tiene por qué estar al tanto del uso de estos modernos terminales móviles. Por este motivo, debemos abstraer al usuario del manejo de toda la tecnología que va detrás del dispositivo que diseñemos. Lo ideal sería que el usuario sólo tuviese que accionar un pulsador, y olvidarse de todo lo demás, a la espera de recibir atención por parte de los servicios sanitarios. Unos simples indicadores luminosos darán a conocer al usuario el estado de su aviso (imposibilidad de conocer las coordenadas GPS, posible fallo en la precisión del GPS, fallo al conectarse a una red GSM).

Debemos ser capaces de **cumplir todos los objetivos** del proyecto, **salvo, posiblemente, las restricciones en tamaño** (posteriormente, para el diseño final, puede realizarse una miniaturización del prototipo, o hacer un nuevo diseño basado en el prototipo con un tamaño más reducido). Para ello, lo mejor, será escoger un diseño simple, sobretodo en el aspecto Hardware, para poder comenzar a trabajar lo antes posible en el manejo y funcionalidad de los módulos GSM y GPS.

3.2 Elección de componentes y montaje

Para realizar el prototipo hemos escogido la placa GSM-GPS de la compañía alemana Antrax, compatible con Arduino.

Arduino es una plataforma de hardware libre, basada en una placa (que posee una serie de entradas y salidas digitales y analógicas, algunas de las cuales pueden implementar interfaces de comunicaciones estándar) con un microcontrolador y un entorno de desarrollo, empleada en el prototipado y el diseño final de multitud de proyectos. En la siguiente imagen puede apreciarse la placa Arduino Uno empleada en este diseño y su tamaño aproximado, el cual resulta algo excesivo.

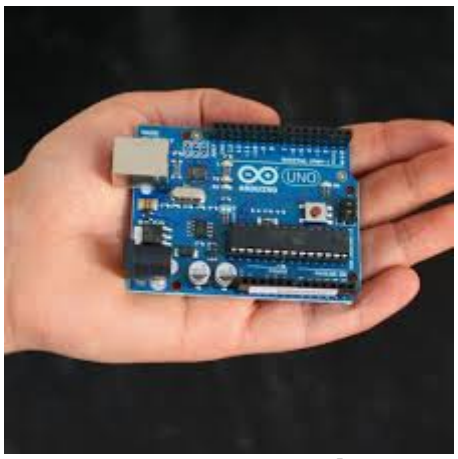


Figura 3.1: Placa Arduino y visualización de su tamaño.

Disponiendo de un Arduino y la placa GSM-GPS de Antrax, el montaje hardware se reduce a encajar una placa sobre la otra (para ello la placa GSM-GPS cuenta con una ‘cama de pinchos’

que podrá montarse a modo de ‘mochila’ sobre la placa Arduino) y enroscar una antena GSM sobre un conector dispuesto para tal propósito (esto puede observarse en la siguiente imagen).

Figura 3.2: Montaje de las placas Arduino y Antrax.

Además, el equipo técnico del proveedor ha desarrollado una librería de código, denominada ‘GSM_ GPS_Shield’, para un manejo más sencillo de la placa. La librería se compone de tres archivos:

1) El fichero ‘keywords’, que simplemente indica al entorno de desarrollo de Arduino cuáles serán las palabras clave que deberán ser subrayadas cuando se escriban en el código.
2) El archivo ‘GSM_GPS_Shield.h’ en el cual aparecen declaradas las clases GSM y GPS. Cada una de estas clases tendrá una serie de variables (públicas, si se permite su acceso desde cualquier parte del programa; y privadas, si sólo se tiene acceso desde ‘GSM_GPS_Shield.c’) y funciones internas.
3) El archivo ,‘GSM_GPS_Shield.c’ el cual contiene el cuerpo de todas las funciones de las clases GSM y GPS, y se encarga del guardado de datos importantes dentro de las variables internas de clase correspondientes.

En la librería se encuentran desarrolladas funciones tales como llamar por teléfono, obtener las coordenadas mediante GPS y mandar mensajes, entre otras muchas; que pueden abstraer al usuario del funcionamiento de los módulos GSM y GPS para funcionalidades básicas.

Esta placa monta: un módulo GSM, un módulo GPS, una ranura para la inserción de la tarjeta SIM, dos pulsadores, un conector para antena y multitud de componentes para la interacción de estos dispositivos entre sí y con Arduino.

En cuanto al módulo GSM, la placa posee el módulo GE865-QUAD de la compañía Telit. Este es uno de los módulos GSM más pequeños del mercado, cuyo encapsulado es del formato BGA (conexiones en una cara del componente mediante un array de bolitas de estaño). Este módulo es un Quad-band, es decir, que puede operar en redes GSM a 850, 900, 1800 y 1900 MHz (en Europa estas redes operarán a 900 y 1800 MHz). Dispone de una UART, la cual será empleada como interfaz de comunicaciones entre el módulo y el Arduino. La sensibilidad de este receptor en las bandas 850/900 MHz es de al menos -107 dBm, mientras que en las bandas 1800/1900 MHz es de al menos -106 dBm; con el módulo operando en condiciones normales. En cuanto a la alimentación y consumo, éste módulo ha de alimentarse con un voltaje comprendido entre los 3.4 y 4.2 Voltios (el fabricante no recomienda nunca sobrepasar los límites de 3.22 – 4.5 V.). El consumo de corriente del módulo depende de la funcionalidad que esté desempeñando, de tal manera que no consumirá la misma corriente, por ejemplo, cuando esté efectuando una llamada que cuando se encuentre en estado de espera (el fabricante habla de picos de corriente de muy corta duración de

hasta 2 Amperios mientras el módulo realice llamadas o transmisión de datos GPRS, lo cual se verá más adelante en esta memoria). En la siguiente figura puede observarse dicho módulo, con sus medidas exactas.

Figura 3.3: Módulo GSM montado sobre la placa Antrax.

Respecto al módulo GPS, la placa monta un UP501 GPS de la compañía Fastrax. Este módulo posee una antena de parche embebida, por lo que no necesitaremos conectar una antena GPS a la placa. Este simple módulo tan sólo presenta 6 pines, siendo únicamente 2 de ellos de comunicaciones (RXD y TXD), estos puertos servirán para la recepción y transmisión de sentencias NMEA (National Marine Electronics Association), gracias a las cuales podremos obtener la información deseada (coordenadas, hora, fecha, etc.) y controlar el módulo. En cuanto al consumo y alimentación del módulo, éste se alimenta con un voltaje comprendido entre 3 y 4.2 voltios; con un consumo medio en torno a los 75 mW. en estado de navegación. La sensibilidad del receptor es de -148 dBm en condiciones de ‘cold start’ o ‘arranque en frío’ (el módulo se enciende sin conocer ninguna información acerca de los satélites visibles, situación en la cual se encontraría el módulo cuando el usuario solicitase asistencia). Se adjunta a continuación una imagen del módulo GPS.

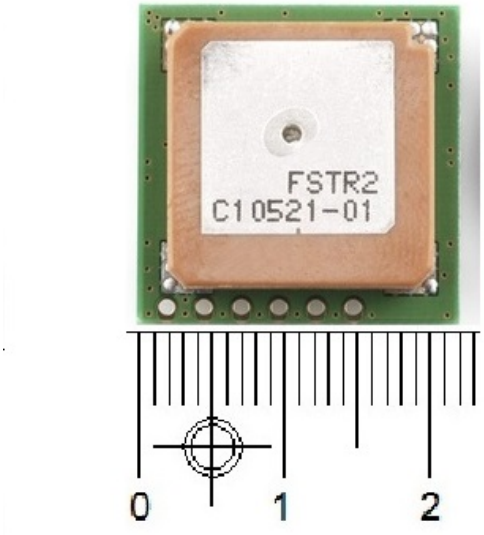


Figura 3.4: Módulo GPS montado sobre la placa y visualización de su tamaño.

Para que el dispositivo pueda encenderse, tras accionar un pulsador y sin tener que mantener al Arduino encendido muestreando una entrada (y por tanto consumiendo corriente de forma innecesaria); hemos decidido adquirir un interruptor de encendido/apagado comercial de la marca Pololu: el Pololu Pushbotton Power Switch. A continuación se muestran unas imágenes de dicho interruptor.

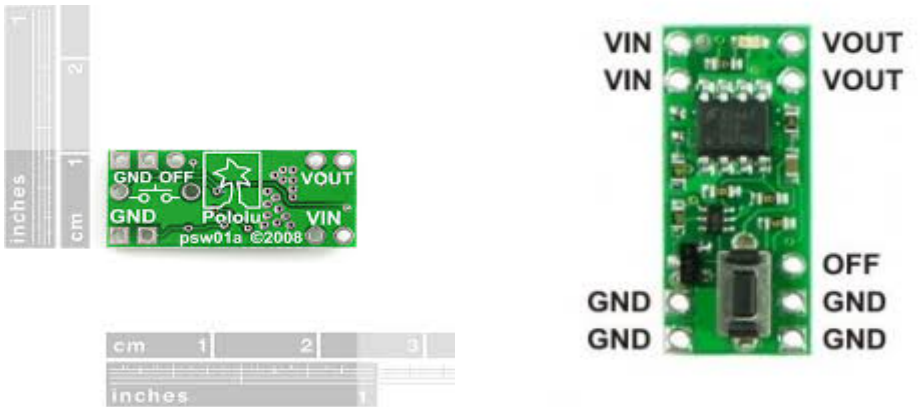


Figura 3.5: Interruptor de encendido Pololu.

Este interruptor impide el paso del voltaje que tiene a la entrada (puertos Vin) a su salida (puertos Vout) a menos que se accione el pulsador de la placa. Si se acciona dicho pulsador, el voltaje de Vout será igual a de Vin (estado ON). Para volver al estado inicial (estado de OFF) tenemos dos opciones: la primera sería volver a accionar el pulsador, y la segunda sería poner una entrada a nivel alto al puerto OFF.

Ahora, el montaje Hardware necesario para añadir este pulsador a nuestro diseño, se reduciría a interponer dicho pulsador entre la alimentación (batería) y nuestra placa Arduino (con la placa GSM-GPS ya montada).

3.4 Características principales y primeras pruebas

3.4.1 Alimentar el Arduino Uno

La placa Arduino puede alimentarse de tres formas diferentes: mediante una entrada USB, mediante un conector dispuesto para tal propósito, y mediante los dos pines de entrada 'GND' y 'Vin'. Cuando ésta se alimenta vía USB recibe 5 voltios de dicho puerto, con las limitaciones en corriente que tenga el dispositivo que provee la alimentación al Arduino (generalmente un PC). Cuando el Arduino se alimenta mediante los pines de entrada de alimentación o el conector, debe recibir un voltaje comprendido entre 7 y 12 voltios (nunca exceder los límites de 6 a 20 voltios).

La placa Arduino procede a encenderse (y a ejecutar el último código que se haya introducido en memoria) en cuanto recibe el voltaje de alimentación.

3.4.2 Encendido de la placa Antrax y de los módulos GPS y GSM

La placa GSM-GPS recibe el voltaje de alimentación por parte de la placa Arduino nada más encenderse esta última.

No obstante, el encendido y puesta en marcha del módulo GSM (el módulo GPS se enciende automáticamente) debe hacerlo Arduino mediante instrucciones software. Podemos observar entre las primeras líneas del código de la función de inicialización incluida en la librería `GSM_GPS_Shield` lo siguiente:

```
pinMode(_powerON_pin,                                                    OUTPUT);
// Pin 7 as Output
digitalWrite(_powerON_pin,                                                 HIGH);
// enable Power GSM-Modul
delay(3000);
```

Vemos que el circuito de encendido del módulo GSM, necesita tener una entrada a nivel alto, y esperar al menos 3 segundos para asegurar que el módulo esté encendido (el comando millis() detiene la ejecución del código tantos milisegundos como indique su argumento). A continuación ya podríamos comenzar a mandar comandos AT mediante el puerto serie del módulo.

3.5 Programación y depuración del código

En este apartado pasaremos a explicar la sintaxis de los comandos AT empleados para cada una de las tareas requeridas al módulo GSM. Detallaremos la estructura y significado de las sentencias NMEA y su empleo para el manejo del módulo GPS. También explicaremos y detallaremos la programación del código a ejecutar por parte del microcontrolador mediante el entorno de desarrollo de Arduino, así como las interfaces de comunicación empleadas para el envío de comandos a los módulos.

3.5.1 Comandos AT

3.5.1.1 Introducción e historia

Los comandos AT son instrucciones codificadas, que conforman un lenguaje de comunicación entre el hombre y un terminal (originariamente este terminal era un módem).

Estos comandos se desarrollaron en 1977 por Dennis Hayes como interfaz de comunicación con un módem, para solventar los problemas y limitaciones del marcado y operación manual de los módems por parte del usuario. La telefonía móvil también adoptó como estándar este lenguaje para poder interactuar con sus terminales, permitiendo acciones tales como realizar llamadas de voz o de datos; leer y escribir contactos en la agenda; leer, borrar y enviar mensajes SMS; además de otras muchas opciones.

3.5.1.2 Sintaxis y estructura de los comandos AT

Las letras AT son abreviatura de la palabra inglesa attention, y aparecerán encabezando todos los comandos. Una línea de comandos estará compuesta de: prefijo, cuerpo y carácter final.

El prefijo serán las letras ‘AT’ (o bien ‘A/’ o ‘AT#/’ para repetición del último comando enviado al módulo).

El carácter final indicará el final del comando y/o de algún argumento requerido por el módulo, como es el caso del texto cuando se solicita enviar un SMS mediante comandos AT.

El cuerpo denotará la tarea a realizar. Básicamente existen tres tipos de cuerpo: el de comandos básicos, el de comandos extendidos y el de comandos propietarios. Las tareas pueden ser de tres tipos: ejecución del comando enviado, solicitud del valor de un parámetro (usualmente denominado como comando de lectura) y solicitud de posibles valores que podría tomar un parámetro (también denominado como comando de test).

Los comandos básicos, son aquellos ya definidos por Hayes para la comunicación con módems. Se diferencian del resto por no incluir un carácter de separación con el prefijo AT. Un ejemplo de este tipo podría ser el comando para realizar una llamada de voz a un número determinado:

ATD 0034609381404;<CR>

Los comandos extendidos se diferencian del resto por llevar el carácter '+' como separación del prefijo. Un ejemplo de este tipo podría ser el comando para solicitar el estado de solicitud de clave de acceso al usuario (comando de lectura):

AT+CPIN?<CR>

Los comandos propietarios son aquellos definidos y desarrollados normalmente por el fabricante, que se suman al set de instrucciones AT. Se distinguen del resto por llevar los caracteres '#', '@', '$' ó'*' como separación del prefijo. Un ejemplo podría ser el comando para obtención del número de celdas GSM de las cuales podremos obtener su información (comando de test):

AT#MONI =?<CR>

Tras enviar un comando, normalmente, el módulo emite una respuesta (además de mostrar al usuario la información requerida, si procede) para informar al usuario si se produjeron o no errores. Aún así para todos los comandos se define un tiempo límite, denominado 'timeout', pasado el cual, si no tenemos respuesta por parte del módulo, supondremos que ha habido un error. No profundizaremos demasiado en el formato de las respuestas (dicho formato depende del ajuste de parámetros mediante comandos). En nuestro caso, se ha configurado el módulo para que indique si hubo éxito o no mediante las palabras OK y ERROR, respectivamente.

3.5.1.3 Comandos AT para inicialización del módulo GSM

Llamamos inicialización a la tarea de lograr conectar el módulo, tras haberse encendido, a una celda disponible de la red GSM. Mostramos a continuación la secuencia de comandos, en el orden que deben emitirse:

1.- ***AT<CR>*** Este es el primer comando que siempre debe emitirse, para ver si el módulo está correctamente encendido. Si se recibe la respuesta *OK*, podemos continuar.

2.- ***ATE0<CR>*** Aquí indicamos al módulo que en las respuestas que debe emitir no debe incluir el comando al cual responde (eco deshabilitado). Si se recibe la respuesta *OK* continuaremos.

3.- ***AT+IPR=9600<CR>*** Ajustamos la velocidad (9600 baudios) a la que enviaremos los datos al módulo mediante su puerto serie. La respuesta *OK* nos permitirá seguir avanzando.

4.- ***AT#SIMDET=1<CR>*** Aquí indicamos al módulo que ignore el pin 'SIMIN' de la ranura de la tarjeta SIM, y simule el estado de tarjeta SIM insertada.

5.- ***AT+CPIN?<CR>*** Aquí consultamos al módulo si se necesita introducir el código PIN para registrar el terminal a la red. Si es necesario introducirlo el terminal responderá con las palabras *SIM PIN*, en caso de que no sea necesario introducirlo, responderá con la palabra *READY*.

6.- ***AT+CPIN=1096<CR>*** Con esta línea de comandos introducimos el código PIN (en este caso indicamos que el PIN es 1096), si fuese necesario.

7.- ***AT+CREG?<CR>*** Con este comando comprobaremos si el terminal está registrado dentro de una de las celdas de red disponibles. Debemos esperar a la respuesta *0,1* ò *0,5* para que el terminal esté registrado (red nacional o roaming, respectivamente); mientras tanto, emitiremos repetitivamente este comando al terminal.

8.- ***AT+CMGF=1<CR>*** Este comando nos sirve para indicar al módulo que emplearemos el formato texto (representación de caracteres mediante 8 bits) para el envío de mensajes cortos de texto. Otra posibilidad sería emplear el formato PDU (7 bits por carácter), pero implica un manejo más avanzado. Este apartado queda concluido cuando recibimos la respuesta *OK* a este último comando.

3.5.1.3 Comandos AT para obtener la información de red

La información de red que queremos obtener será la siguiente:

1.- MCC (Mobile Country Code): Este el código de red asignado al país, para el caso de España éste es '214'.

2.-MNC (Mobile Network Code): Éste es el código de red asignado a cada operador, para el caso de Vodafone éste es '01'.

3.-LAC (Location Area Code): Este código lo asigna cada operador, dividiendo el país en zonas.

4.-CELL ID: Este es el número de identificación de celda. Cada celda dentro de un mismo LAC, tendrá un Cell Id único.

5.-PWR: ésta será la potencia de la señal, procedente de la celda, recibida en el terminal y medida en dBm. A veces, este parámetro puede ser empleado para estimar una posible distancia a la que se encuentre el paciente de la antena de la estación base (como por ejemplo en grandes áreas muy despejadas y sin obstáculos).

6.-TA (Timing Advance): éste parámetro, que estará disponible únicamente para la celda en la cual estemos registrados, es un indicador de la distancia a la cual nos encontramos respecto de la antena de la estación base. Cada unidad indica una distancia (aproximada) de unos 550 metros a la estación base.

Para conocer dicha información emplearemos dos comandos, dentro del amplio abanico de comandos para el 'escaneo' de la red:

1.- ***AT#MONI<CR>***: Este comando lo emplearemos para solicitar la información de la celda a la que estamos registrados (este comando también puede usarse para obtener información, algo más escueta, de hasta 6 celdas vecinas; estas celdas serán las que mejor calidad de señal ofrezcan, independientemente del operador de red). Al mandar este comando a través de la UART del módulo, en caso de no haber errores, nos esperará una respuesta de la forma:

```
#MONI: vodafone ES BSIC:64 RxQual:0 LAC:4C10 Id:C0FC
ARFCN:105 PWR:-75dbm TA:1
OK
```

Podemos observar en esta respuesta que se nos muestra el Nombre de Red (el operador de red y su país): *vodafone ES*, que indica que estamos en España conectados a una celda del operador Vodafone. En caso de que no se conociese el nombre del operador de la red, se nos mostraría el MCC y MNC, los cuales en conjunto ofrecen la misma información que el Nombre de Red (de tal modo que saber que la red es Vodafone España es lo mismo que saber que el MCC sea '214' y el 'MNC' sea '01'). También se muestra el resto de la información buscada.

Si no ha habido ningún error, el módulo responderá con la sentencia *OK*, tras mostrar la información del comando *#MONI*. En caso contrario el módulo responderá con la sentencia *ERROR*.

2.- ***AT#CSURV<CR>***: El segundo comando lo emplearemos cuando detectemos algún posible error en el receptor GNSS, para obtener la información de todas las celdas que se encuentren al alcance del terminal. Este comando realiza una inspección completa de la red, recorriendo cada uno de los canales existentes. Para cada portadora reconocida se muestra la información asociada. A continuación se muestra un ejemplo de respuesta del módulo a este comando:

```
Network survey started…
arfcn: 48 bsic: 24 rxLev: -52 ber: 0.00 mcc: 610 mnc: 1
lac: 33281 cellId: 3648 cellStatus:
CELL_SUITABLE numArfcn: 2 arfcn: 30 48 numChannels: 5 ar-
ray: 14 19 22 48 82
```

```
arfcn: 14 rxLev: 8
Network survey ended

OK
```

Podemos observar que se dan dos tipos de información, una muy extensa y otra bastante escueta. La información extensa se muestra para aquellas portadoras GSM de tipo BCCH (Broadcast Control CHannel) las cuales transmiten información de conectividad. Para el resto de portadoras GSM no se muestra tanta información puesto que estas portadoras son empleadas por las estaciones base para control de otras funciones. Los terminales móviles de usuario actualizan de forma periódica la calidad de señal de las portadoras BCCH para la toma de decisión de a qué estación base (o celda de red) nos conectaremos.

Cabe destacar que la ejecución completa de este comando (envío del comando por el puerto serie y recepción de la respuesta) puede tomar mucho tiempo. Durante todo este tiempo, y al igual que con el resto de comandos, el módulo no podrá procesar nuevas instrucciones hasta haber dado su respuesta, o, en caso contrario, se produzca un timeout.

De la información transmitida por cada estación base en su canal radio BCCH, obtendremos y almacenaremos en una variable el 'LAC', el 'CELLID' y 'PWR' de todas aquellas celdas Vodafone de España (MCC igual a 214 y un MNC igual a 01).

3.5.1.4 Comandos AT para el envío de SMS

Para enviar un SMS emplearemos el siguiente comando:

AT+CMGS=0034609381404,129<CR>

A lo que el módulo responderá con el carácter '>'. Tras recibir dicho carácter, podremos introducir el texto a enviar. Para indicar al módulo el fin del texto enviaremos el carácter <Ctrl+Z>.

3.5.2 Sentencias NMEA (National Marine Electronics Association)

3.5.2.1 Introducción

Este será el lenguaje empleado por el módulo GPS, y en general, el lenguaje de muchos receptores GPS y estándar de comunicaciones electrónicas máquina-máquina en navegación marítima.

El protocolo NMEA fue creado por la Asociación Nacional de Electrónica Marina de los Estados Unidos. El estándar empleado aquí es el NMEA 0183, el cual emplea un protocolo serie de transmisión de caracteres ASCII que define cómo los datos son enviados en sentencias o frases.

Desde el momento en que se inicialice el receptor GPS, éste comenzará a enviar a través de su puerto serie una serie de sentencias NMEA de diversos tipos con un período de tiempo determinado (la configuración por defecto, es de cuatro tipos de sentencias NMEA por segundo). El valor de este período de tiempo y las sentencias que se deseen a la salida del receptor, pueden ser a su vez ajustadas (dentro de los límites impuestos por el fabricante) por el usuario mediante el envío de sentencias NMEA de configuración al módulo.

3.5.2.2 Estructura de frase

Una sentencia NMEA 0183 puede ser de tres tipos: informativa, imperativa o de consulta.

Las sentencias informativas son aquellas que presentan información de algún tipo (por ejemplo, las coordenadas del posicionamiento, información de los satélites encontrados, etc.) y presentan el siguiente formato:

```
$GPAAM,A,A,0.10,N,WPTNME*32
```

Donde el primer carácter '$' indica el comienzo de la sentencia. Las dos siguientes letras indican la identidad del emisor de la frase, en este caso las siglas 'GP' denotan que el origen de la sentencia es una unidad GPS. Los caracteres siguientes (hasta la primera coma) indicarán el tipo de mensaje, en el ejemplo 'AAM' es un aviso de alarma de llegada. Los siguientes caracteres, separados por comas, hasta llegar al carácter '*' son los datos (información) de la sentencia. Los dos últimos números hexadecimales son el checksum de la sentencia (OR exclusiva bit a bit de todos los códigos ASCII de todos los caracteres comprendidos entre el '$' y el '*').

Las sentencias imperativas son aquellas que sirven para dar una orden al Módulo (por ejemplo, realizar un reset del módulo o ajustar la velocidad de envío/recepción de datos), mientras que las de consulta se emplean para conocer el estado o valor de los parámetros de ajuste del receptor (por ejemplo, para saber si la funcionalidad SBAS está habilitada). Estos comandos son de la forma:

```
$PMTK000*32
```

Donde PMTK es la identidad, que denota un comando hacia Media Tek, el procesador del receptor GPS. Los tres dígitos siguientes indican el tipo de comando. Finalmente se añade el checksum, de la misma manera que para las sentencias de información.

3.5.2.3 Sentencias NMEA empleadas

Los datos que nos interesan conocer del módulo GPS son las coordenadas del sujeto, y un indicador de la posible precisión de dicha medida. Vemos a continuación la sentencia

GGA la cual puede proporcionarnos toda la información requerida.

$GPGGA,hhmmss.dd,xxmm.dddd,<N|S>,yyymm.dddd,<E|W>,v,ss,d.d,h.h,M,g.g,M,a.a,xxxx*hh<CR><LF>

La sentencia anterior nos proporciona varios argumentos, algunos de los cuales pasaremos a detallar:

a) ***hhmmss.dd*** : este argumento es la hora actual UTC, que es el principal estándar de tiempo por el cual el mundo regula la hora. Los dos primeros dígitos indican la hora, los dos siguientes indican los minutos, los dos siguientes los segundos y los dos últimos las décimas de segundo (separados del resto del argumento por un punto).

b) ***xxmm.dddd, <N|S>***: este argumento denota la latitud. Los dos primeros dígitos son los grados, los dos siguientes los minutos, y los cuatro últimos las décimas de minuto (separadas del resto del argumento por el carácter '.'). Tras la coma, aparecerá una letra N o S, según sea una latitud norte o sur, respectivamente.

c) ***yyymm.dddd, <E|W>***: este argumento es la longitud. Los tres primeros dígitos indican los grados, los dos siguientes los minutos, y los cuatro últimos, separados del resto por un punto, son las décimas de minuto. Tras la coma, aparece una letra que puede ser E o W, denotando este u oeste, respectivamente.

d) ***v***: indicador del origen del posicionamiento.

e) ***ss***: número de satélites empleados en el posicionamiento. Es un número comprendido entre 00 y 12.

f) ***d.d***: este es el valor del HDOP (Horizontal Dilution of Precission). Más adelante en el apartado de pruebas explicaremos el sentido este parámetro. Simplemente, decir, que está bastante relacionado con la precisión del posicionamiento proporcionado.

g) ***h.h***: este valor representa la altitud.

h) ***M,g.g,M,a.a,xxxx*hh<CR><LF>*** resto de parámetros, más el retorno de carro y el cambio de línea (<*LF*>) que indican el fin de la sentencia.

Para controlar el módulo GPS, también es necesario el envío de comandos (NMEA) a través de su UART. Tendremos que habilitar la funcionalidad SBAS (que viene desactivada por defecto), hacer un reset del módulo (para evitar que guarde los datos de las efemérides, y simular así la situación de un 'arranque en frío' que tendría un usuario a la hora de solicitar asistencia). Estos comandos son, respectivamente:

$PMTK104*37<CR><LF>

$PMTK313,1*2E<CR><LF>

3.5.3 Estructura principal del programa

El dispositivo permanecerá apagado (ausencia de alimentación) hasta que el usuario presione el pulsador de aviso, independientemente del tiempo que el usuario mantenga el botón apretado. En este caso, si el usuario acciona dos veces el pulsador, anulará el aviso. Para cumplir este objetivo, incluimos entre los componentes a comprar, el pulsador de encendido/apagado de Pololu.

Cuando el dispositivo se encienda, debe activar el módulo GPS para que vaya 'buscando' nuestra ubicación. Ahora, lo primero que hay que hacer es registrarse dentro de una celda que esté al alcance del chip GSM. Para ello será necesario encender y llevar a cabo la tarea de inicialización del módulo GSM descrita en el apartado de comandos AT (más la espera de 3 segundos, tras encender el módulo). Si esto no fuera posible, el dispositivo seguirá intentándolo hasta que lo consiga. En caso de no conseguirlo (situación en la cual el dispositivo está fuera de la cobertura de la red GSM nacional) no se podrá realizar el aviso de emergencia. Tanto en caso de éxito, como en caso de fallo, se informará de ello al usuario mediante los LEDs del dispositivo.

Una vez el dispositivo esté registrado dentro de una celda, la cual será la que mejor calidad de señal ofrezca (siendo de la compañía Vodafone, o de otra compañía que permita la conexión a usuarios de Vodafone); debemos buscar la información de la misma, haciendo uso del comando 'AT#MONI'. Si se produce un error al ejecutar dicho comando (lectura de la palabra 'ERROR' en el puerto serie del Arduino) se seguirá intentando repetitivamente obtener la información, hasta que recibamos la respuesta 'OK' por parte del módulo GSM. Cuando recibamos la citada palabra 'OK' en el puerto serie, podremos pasar a la siguiente tarea.

Llegados a este punto procederemos a enviar mediante un SMS la información de la celda a la cual estamos registrados, dando un primer posicionamiento del usuario (posicionamiento 'grueso'). Para mandar dicho mensaje, emplearemos el comando 'CMGS' de la forma ya descrita, hasta que el módulo GSM nos devuelva la palabra de éxito 'OK'.

El siguiente paso será obtener las coordenadas del usuario mediante el módulo GPS, que ha estado recopilando toda la información necesaria sobre los satélites que tiene a 'la vista' desde que el dispositivo fue puesto en marcha. Si se produce algún error (por ejemplo, un posible fallo en la precisión), o el GPS rebasa una cantidad de tiempo predeterminada a la hora de obtener las coordenadas, pasaremos a buscar la información de todas las celdas Vodafone al alcance.

En caso de disponer de un posicionamiento mediante GPS de forma exitosa, se enviará un mensaje de texto con el comando 'CMGS'(esto se hará repetitivamente hasta obtener éxito en el envío del SMS) con la ubicación del usuario, en forma de coordenadas (latitud y longitud). En el mensaje se informará al número de emergencias de la posible precisión de este posicionamiento. Se informará también al usuario, mediante los indicadores lumínicos, del éxito al dar el aviso, y habremos llegado al fin del programa.

En caso de no disponer de las coordenadas, buscaremos la información de celda de aquellas celdas vecinas del operador Vodafone, que estando al alcance del dispositivo, tengan un nivel de señal por encima de cierto umbral. Este umbral vendrá marcado por la potencia de señal de la celda que nos atienda, un ejemplo de esto sería incluir la información de aquellas celdas Vodafone cuyo nivel de señal esté a menos de 15 decibelios del nivel de señal de la celda servidora. Para conseguir esta información haremos uso del comando

‘CSURV’. Se lanzará repetitivamente este comando en caso de que el módulo nos respondiese con mensajes de error, hasta lograr obtener el mensaje de éxito ‘OK’. Una vez obtenida la información de las celdas vecinas se procederá a mandar esta información en uno o varios mensajes de texto (repetitivamente hasta obtener éxito en el envío del SMS).

Una vez enviado el SMS con la información de las celdas vecinas, se pasará de nuevo a buscar las coordenadas del usuario, no volviendo a tener que buscar la información de celdas vecinas, en caso de nuevo fallo. Si se dispone de las coordenadas, éstas proceden a enviarse mediante SMS junto a un indicador de la precisión de esta medida. El programa llegará a su fin únicamente si el estimador de la precisión del posicionamiento se considera adecuado, sino el dispositivo seguirá buscando un mejor posicionamiento GPS (no enviando más SMS con posicionamientos posiblemente imprecisos).

En caso de llegar al fin del programa, una opción sería apagar el dispositivo, poniendo una entrada a nivel alto mediante Arduino, en el puerto ‘OFF’ del interruptor de Pololu.

Lo anteriormente descrito puede observarse de forma esquemática, en la siguiente imagen, por medio de la siguiente máquina de estados:

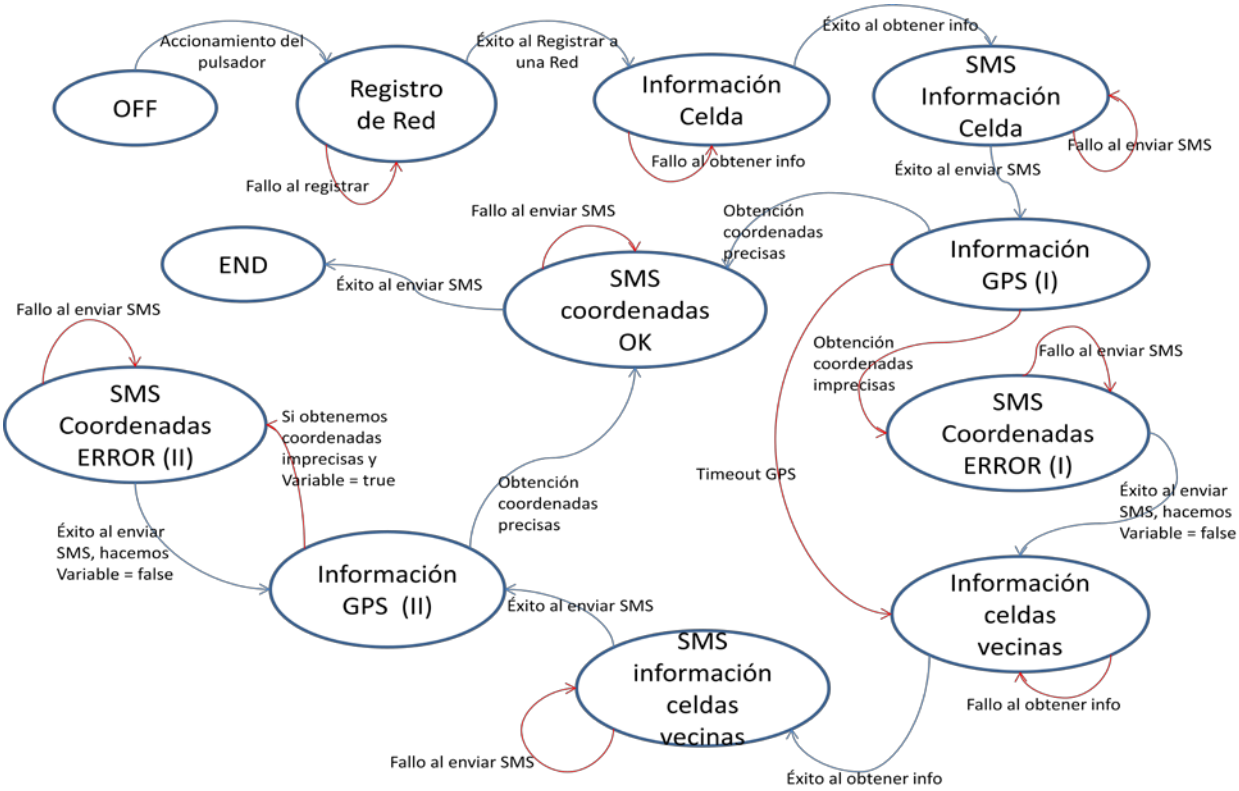


Figura 3.6: Diagrama de estados asociado al funcionamiento del programa.

Cabe destacar que se han creado dos estados para obtener las coordenadas del posicionamiento GPS (Información GPS (I) e Información GPS (II)) ya que en el primero de ellos puede darse la situación de ‘timeout’, mientras que en el segundo estado permaneceremos indefinidamente a la espera de las coordenadas.

También se han definido dos estados para el envío de coordenadas imprecisas (SMS coordenadas ERROR (I) y SMS coordenadas ERROR (II)) ya que tras el primero iremos en busca de la información de las celdas vecinas, mientras que el segundo estado (al cual sólo

se accederá si obtenemos una información imprecisa del receptor GPS tras haberse producido un error de tipo ‘timeout’ en el primer estado de obtención de información GPS) nos llevará al estado ‘Información GPS (II)’ en busca ya, únicamente de unas coordenadas cuyo indicador de precisión sea adecuado (valor del HDOP por debajo de cierto umbral). No obstante, se ha permitido el envío de mensajes con indicadores de DOP elevados, siempre y cuando el nuevo valor del DOP obtenido mejore notablemente, lo cual podrá observarse en la explicación del código del programa principal.

3.5.4 Entorno de desarrollo de Arduino y programación

Para programar el Arduino, debemos ejecutar el entorno de desarrollo. Esta aplicación posee un editor de texto, un área de mensajes, una consola de texto y un menú de opciones (algunas de las cuales, serán también presentadas en una barra de herramientas). A continuación puede observarse una imagen del entorno de desarrollo:

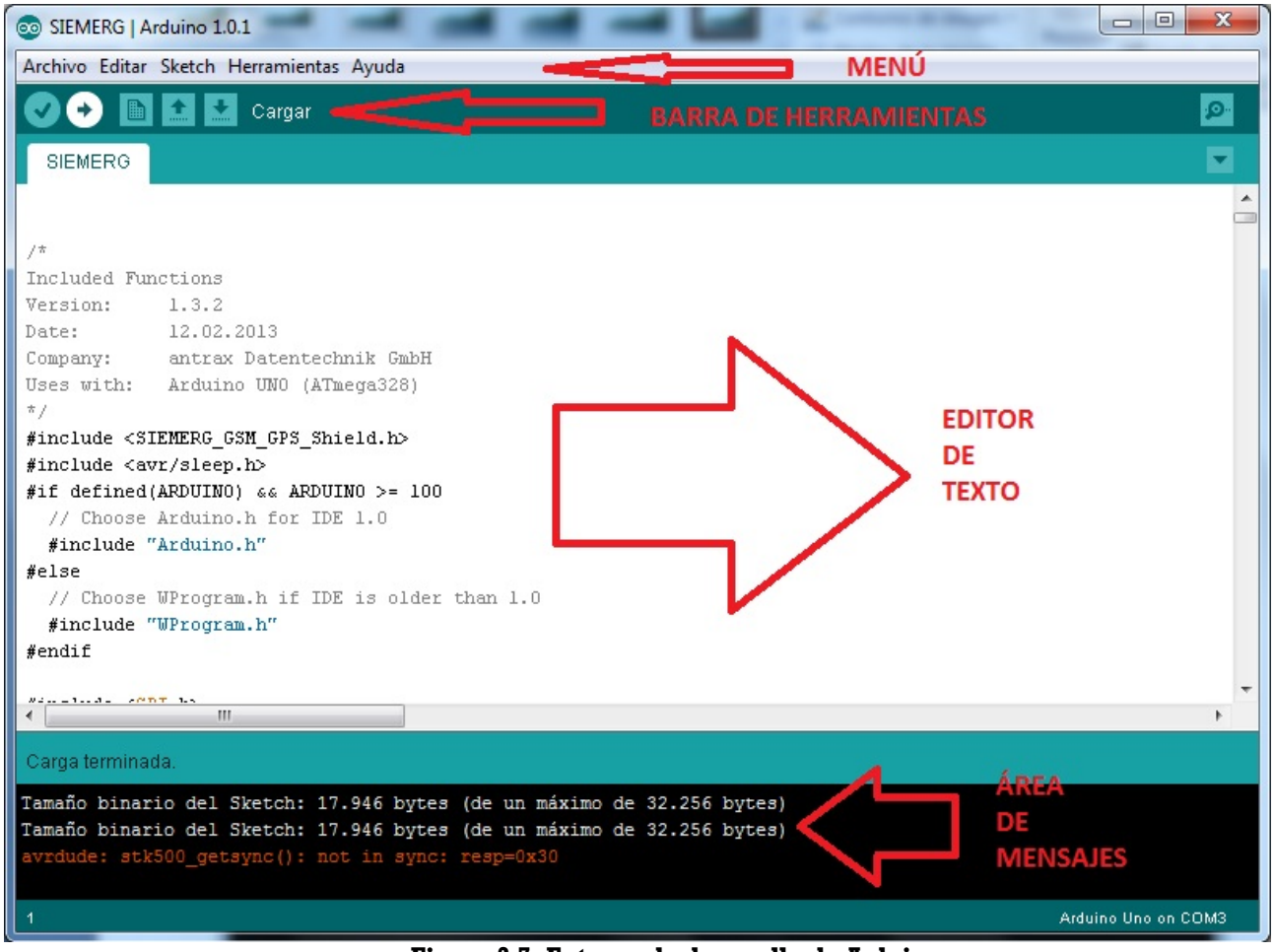


Figura 3.7: Entorno de desarrollo de Arduino.

Esta herramienta nos permite programar el Arduino de una forma cómoda y sencilla. El lenguaje de programación se denomina ‘Processing’, y está basado en C, soportando la mayoría de funciones del estándar.

Para llevar a cabo el desarrollo de la aplicación que correrá en nuestro Arduino desarrollaremos un programa principal y haremos uso de las librerías proporcionadas por

Antrax. En estas librerías modificaremos y desarrollaremos las funciones propias y las variables de las ya implementadas estructuras GPS y GSM. La aplicación principal será la encargada de llevar a cabo la secuencia de acciones que son necesarias para poder dar el aviso de emergencia y controlar el estado del mismo. Para ello necesitará realizar llamadas a las funciones de la librería, gestionar las variables de las estructuras, detectar posibles inconvenientes e informar al usuario mediante los indicadores lumínicos del estado del aviso. A continuación se mostrará el código desarrollado, con una explicación detallada del funcionamiento del mismo.

-Librerías del programa

Las librerías están formadas por dos archivos: el primero, que hemos renombrado como 'SiemergLib.h' a partir del archivo proporcionado por la compañía Antrax, nos servirá para declarar las estructuras de datos y las clases (una clase es una estructura de datos, que contiene, además, métodos o funciones) junto con sus variables y las cabeceras de las funciones que vayan a ser desarrolladas; el segundo, renombrado como 'SiemergLib.cpp', contendrá el cuerpo de todas las funciones.

Lo primero que debe hacerse en el archivo 'SiemergLib.h' es importar todas aquellas librerías de las que vayamos a hacer uso en la librería.

```
#ifndef SIEMERG_GSM_GPS_Shield_h
  #if defined(ARDUINO) && ARDUINO >= 100
    // Choose Arduino.h for IDE 1.0    #include "Arduino.h"
  #else
    // Choose WProgram.h if IDE is older than 1.0
    #include "WProgram.h"
  #endif

  #define SIEMERG_GSM_GPS_Shield_h

  #include <SPI.h>
```

Las librerías *Arduino.h* y *WProgram.h* son las que contienen todas las funciones propias del lenguaje de la placa Arduino, la única diferencia entre ambas es la versión del entorno de desarrollo que se emplee (versiones antiguas emplean la librería *WProgram.h*).

Lo siguiente que aparece en el archivo 'SiemergLib.h' es la declaración de dos estructuras. Estas estructuras servirán para almacenar la información de la celda de registro y la información de las celdas vecinas. A continuación se muestra dicha declaración:

```
struct ServingCellStruct{
  char Nombre[15];
  boolean NombreDisponible;
  unsigned int Mcc;
  byte Mnc;
  unsigned int Lac;
  unsigned int CellId;
  byte Pwr;
  byte Ta;
};
```

Podemos observar que almacenaremos el nombre de la red o los códigos MCC y

MNC, así como los valores de LAC, CELLID, PWR y TA. Como todos estos valores serán positivos, hemos decidido declarar aquellos que puedan exceder el valor de 256 como *unsigned int* (si los declarásemos como *int* perderíamos la mitad del espacio de almacenamiento puesto que *int* reserva este espacio para almacenar números negativos). Los parámetros que no puedan exceder el valor 256 los declararemos como byte, puesto que una variable de tipo *int* o *unsigned int* ocupa dos bytes, ahorrando así el máximo espacio posible de memoria (entonces una variable *int* permite almacenar valores comprendidos entre -2^15 y 2^15, mientras que una variable *unsigned int* permite almacenar valores comprendidos entre 0 y 2^16). Más abajo, se declara la estructura en la cual se almacenará la información de una celda vecina.

```
struct CellStruct{
  unsigned int Mcc;
  byte Mnc;
  unsigned int Lac;
  unsigned int CellId;
  byte Pwr;   };
```

Seguido a esto último, se declaran dos clases, las cuales serán el elemento principal de la librería. Estas clases contendrán una serie de variables y funciones, las cuales serán públicas o privadas, según se pueda tener acceso a ellas desde cualquier parte del código o únicamente desde el código desarrollado en el archivo 'SiemergLib.cpp', respectivamente. Las clases declaradas son GSM y GPS, se adjunta a continuación el código de estas declaraciones:

```
class GSM
  {
    public:
      GSM(int powerON_pin, int baud);
      void initializeGSM(char pwd[4]);
      long externalTime;
      void setExternalTime();
      int MandarSMS(byte tipoSms, byte numAnts, float _DOP, char
      _coordinates[40]);
      int getInfoCell();
      int getInfoAllCells();
    private:
long lastTime;
      void setTime();
      char buffer[25];
      int _powerON_pin;
      int _baud;
      ServingCellStruct ServingCell;
      CellStruct CellArray[30];
      void readGSMAnswer();
      void VaciarPuertoSerie ();
      void VaciarPuertoSerieCsurv ();
      unsigned int potencia(byte factor);
      unsigned int convHexaDec(char * Hexa, unsigned int tam);
  };
```

Podemos observar que en la clase GSM aparecen declaradas como públicas seis funciones y una variable; y como privadas, seis funciones y seis variables. Pasaremos a dar una breve descripción de todas ellas.

En primer lugar hablaremos de las variables privadas. La variable privada *_powerON_pin* denota el pin que ha de ser activado por Arduino para encender el

módulo GSM. La variable *_baud* será la velocidad, en baudios, de la comunicación serial que habrá entre Arduino y el módulo. La clase GSM contendrá también una estructura del tipo *ServingCellStruct* para almacenar la información de la celda de registro actual y un array de estructuras del tipo *CellStruct* para almacenamiento de la información de hasta 30 posibles celdas vecinas. También hemos declarado una variable del tipo array de caracteres, denominada *buffer*, a modo de cola FIFO (el primero en entrar será el primero en salir) para ir almacenando la respuesta del módulo a los comandos AT enviados para obtención de información de celdas y envío de mensajes. A ésta última variable se le da un tamaño más que suficiente (hasta 25 caracteres) para poder observar largas cadenas del texto de respuesta.

Pasaremos a hablar ahora de las funciones privadas. La función privada *readGSMAnswer()* será la encargada de leer e interpretar las respuestas del módulo GSM a los comandos AT enviados para la inicialización del módulo. Los métodos *VaciarPuertoSerie()* y *VaciarPuertoSerieCsurv()* sirven para eliminar los caracteres de respuesta del módulo GSM presentes en el buffer del puerto serie de Arduino, para liberar así espacio para futuras respuestas. Los métodos *potencia()* y *convHexaDec()*sirven para obtener la n-ésima potencia de diez y convertir una cifra expresada en hexadecimal a decimal, respectivamente.

El programa principal no tendrá acceso a estos recursos, que serán empleados internamente por la librería para poder proporcionar resultados al programa principal a través de las variables públicas.

Hay declarada una variable pública, *externalTime*, que es empleada para almacenar el tiempo de ejecución en un momento dado. Este registro es empleado para saber cuánto tiempo lleva ejecutándose cierta porción de código desde la rutina principal (para ello no tendríamos más que restar al tiempo actual dicho valor de tiempo almacenado). De esta manera podremos llevar un control de tiempo desde la rutina principal.

Hablaremos ahora de las funciones públicas. En primer lugar aparece declarado el constructor, que es la subrutina encargada de inicializar un objeto de una clase. Este método ha de denominarse como la clase, por tanto su nombre es *GSM()*, y recibe como argumentos la velocidad de transmisión del puerto serie y el pin de encendido del módulo GSM. El método *initializeGSM()* es empleado para llevar a cabo la tarea de inicialización del módulo GSM descrita en apartados anteriores, y recibe como argumento el código PIN (contraseña de acceso a la tarjeta SIM). Cuando se ejecute la función *setExternalTime()* se almacenará el tiempo actual en la variable *externalTime*. El método *MandarSMS()* se empleará para mandar los diferentes SMS de aviso y sus argumentos serán el tipo de mensaje (posicionamiento GSM, posicionamiento preciso GPS, posicionamiento impreciso GPS o información de celdas vecinas) el número de celdas vecinas, el valor del HDOP y las coordenadas. Las funciones *getInfoCell()* y *getInfoAllCells()* se emplearán para obtener la información de celda de registro y celdas vecinas, respectivamente, y almacenarla en la estructura de datos correspondiente dentro de la clase *GSM*.

A continuación mostramos la declaración de la clase GPS:

```
class GPS
  {
    public:
```

```
    GPS(int baud);
    char initializeGPS();
    void getGPS();
    void resetGPS();
    void habilitarSBAS();
    char checkS1();
    char checkS2();
    void setLED(char state);
    char gps_data[80];
    char gps_data2[80];
    char latitude[20];
    char longitude[20];
    char coordinates[40];
    float HDOP;
  private:
    void WriteByte_SPI_CHIP(char adress, char data);
    char ReadByte_SPI_CHIP(char adress);
    void Select_SPI(char cs_pin);
    void Release_SPI(char cs_pin);
    int _powerON_pin;
    int _baud;  };
```

En esta última clase aparecen declaradas como públicas ocho funciones y seis variables, mientras que otras 4 funciones y 2 variables son declaradas como privadas.

Las variables privadas *_powerON_pin* y *_baud* son empleadas para los mismos fines que en la clase GSM, la única diferencia radica en que la placa Arduino empleará el protocolo SPI como interfaz de comunicación serial con el módulo GPS.

Los métodos privados de la clase GPS: *WriteByte_SPI_CHIP()*, *ReadByte_SPI_CHIP()*,*Select_SPI()*y *Release_SPI()* serán empleados internamente por otros métodos de la librería para establecer la comunicación serial propia del estándar SPI.

Repasaremos ahora las variables públicas de la clase GPS. Las variables *gps_data* y *gps_data2* son empleadas como buffer de datos para ir almacenando los caracteres de respuesta del módulo GPS. Tienen un tamaño suficiente para poder evaluar hasta 80 caracteres de respuesta de forma simultánea. Por otro lado, las variables *latitude*, *longitude* , *coordinates* y *HDOP* son empleadas para almacenar la información que queremos obtener del posicionamiento GPS.

En cuanto a las funciones públicas, podemos ver en primer lugar la declaración del constructor de la clase: `GPS()`, el cual recibe como argumento la velocidad, en baudios, de la transmisión serial de datos entre el módulo GPS y Arduino. La función *initializeGPS()* será la encargada de inicializar el módulo GPS poniendo a nivel alto el pin de encendido, haciendo un reset al módulo y testeando el funcionamiento de la transmisión serial mediante el protocolo SPI, dependiendo de si el resultado es positivo o negativo se devuelve un valor de tipo `char`. La función *getGPS()* es la encargada de observar la sentencia GPGGA para ver si se encuentra disponible un posicionamiento y su HDOP asociada, para posteriormente almacenar esta información en las variables públicas correspondientes, para que estén accesibles desde cualquier parte del código del programa. La función `resetGPS()` se emplea para resetear el módulo GPS, lo cual elimina todos los datos de los satélites que puedan estar almacenados en memoria; simulando siempre una situación de ‘cold start’. Como su propio nombre indica, la función `habilitarSBAS()`

se encarga de habilitar la funcionalidad SBAS del módulo GPS. Las funciones `checkS1()` y `checkS2()` son empleadas para muestrear los puertos de entrada asociados a dos pulsadores dispuestos en la placa GSM-GPS de Antrax, y saber si éstos están accionados en el instante en que se ejecutan dichas funciones. La función `setLED()` es la encargada de encender o apagar (según el argumento de tipo `char` que recibe) uno de los leds que disponemos a modo de indicadores lumínicos para informar al usuario del estado del posicionamiento del módulo GPS en tiempo real.

El desarrollo de los métodos ya descritos ha de hacerse en el archivo 'SiemergLib.cpp'. En este archivo también se asignarán valores a las variables mencionadas anteriormente.

-Programa principal

En este apartado podremos ver la gestión implementada de los métodos y variables de la librería detallada anteriormente, llevada a cabo por la rutina principal de nuestro código.

El entorno de programación de Arduino divide el programa principal en tres bloques. El primero de estos bloques es empleado para incluir las librerías necesarias para el desarrollo del programa principal y la declaración de variables globales (accesibles y modificables desde cualquier parte del código) y desarrollo de funciones globales (accesibles desde cualquier parte del código). El segundo bloque se emplea para una función denominada `setup()` , el cual solamente se ejecuta una vez tras encender el Arduino. El último bloque es el reservado para la función `loop()` la cual será ejecutada repetitivamente y de forma indefinida (hasta que se produjese un reset del Arduino o éste mismo se apagase). A continuación se muestra el primero de estos tres bloques:

```
#include <SIEMERG_GSM_GPS_Shield. h>

#if defined(ARDUINO) && ARDUINO >= 100
  // Choose Arduino.h for IDE 1.0
  #include "Arduino.h"
#else
// Choose WProgram.h if IDE is older than 1.0
  #include "WProgram.h"
#endif

#include <SPI.h>

GSM gsm(7,9600);
GPS gps(9600);
float DOP, last_DOP;
byte error_gps=0;
byte num_antenas=0;

void(* resetFunc) (void) = 0;
```

Lo primero que se hace en el primer bloque es incluir las librerías necesarias para poder ejecutar el código. Entre estas librerías se encuentran la ya citada `SIEMERG_GSM_GPS_Shield. h`, declarada aquí para tener acceso a las variables y funciones de las clases GSM y GPS desde la rutina principal.

Más adelante comienza la declaración de clases y variables globales. Se declaran una

clase de tipo `GPS` y otra de tipo `GSM`, ambas con una velocidad de 9600 baudios para comunicación con el microcontrolador.

Se declara la variable `DOP` encargada de almacenar el valor de la precisión asociado al posicionamiento GNSS actual. También se declara una variable denominada `last_DOP` para almacenar valores anteriores del DOP, y poder establecer una comparativa con el posicionamiento actual. Como el valor del DOP arrojado por el receptor GPS es decimal, declaramos estas variables como float.

Se declara una viable de tipo byte (menor espacio posible), llamada `error_gps` para saber si se produjo un error en el GPS. Como vimos, los errores que pueden aparecer son: timeout, oscilación en el valor de las coordenadas o valor elevado del DOP. Esta variable nos servirá para saber si ya se produjo un fallo GPS el cual empujó a obtener la información de celdas vecinas. De esta forma evitaremos reenviar de nuevo la información de celdas vecinas tras varios fallos del módulo GPS.

Otra variable, denominada `num_antenas`, recogerá el número de celdas vecinas detectadas en un escaneo completo de red, si éste se realiza. Se declara como byte, ya que su valor nunca será elevado.

Por último se declara una función de reseteo del propio Arduino, por si se detecta algún fallo grave, poder reiniciar el proceso. Esta rutina se denomina resetFunc y es declarada como se indica en las páginas de soporte de Arduino.

Tras haber explicado el primer bloque de código, pasaremos a detallar el segundo. Iremos mostrando trozos de código y no pasaremos al siguiente bloque de código hasta comentar en profundidad el anterior. A continuación se muestran las primeras líneas de éste segundo bloque:

```
void setup(){
	if(gps.initializeGPS())
		Serial.println("******Initialization completed");
	else
		resetFunc()
	gps.habilitarSBAS();
	delay(200);
```

En primer lugar, procedemos a inicializa el módulo GNSS, para que desde el primer momento éste ya esté realizando las operaciones necesarias para ofrecer un posicionamiento a su salida. En caso de no poder inicializarse el módulo GPS pasamos a reiniciar el programa (cabe destacar que esta situación no se ha producido nunca en las múltiples pruebas de este dispositivo). Lo siguiente será activar la funcionalidad de SBAS del receptor y esperar 200 milisegundos, para pasar a ejecutar las siguientes líneas del código, las cuales se muestran a continuación:

```
gsm.initializeGSM("1096");
Serial.print("t conex = ");
Serial.println(millis());
```

Aquí pasamos a inicializar el módulo GSM, lo cual incluye la tarea de registrarse a una red al alcance del dispositivo. Cuando el módulo GSM esté inicializado pasaremos a mostrar el tiempo de ejecución del programa por el puerto serie. Esto sólo será útil para supervisar el funcionamiento del programa desde el PC, en el programa final de usuario evitaremos escribir por el puerto serie esta información.

```
gsm.setExternalTime();

while(gsm.getInfoCell()==0){
	//PRIMER SMS
	if ((millis()-gsm.externalTime)>40000){
		Serial.println("Fallo tras varias llamadas a MONI");
		resetFunc();
	}
}
Serial.print("t info = ");
Serial.println(millis());
```

En las líneas superiores realizamos la tarea de obtención de la información de red, tras estar ya registrados en una red. En esta tarea debemos supervisar el tiempo de ejecución, tanto externo, como interno. El tiempo externo será el que se actualiza desde la rutina principal, mientras que el interno se hace desde las funciones de las librerías. El tiempo externo nos servirá para supervisar que la ejecución de cierto trozo de código dentro de la rutina principal no exceda un determinado valor de tiempo. Por otro lado, el tiempo interno, será empleado para comprobar que la ejecución de cierto código dentro de determinadas funciones de las librerías no rebase un tiempo límite prefijado. La llamada a la función `gsm.setExternalTime()` sirve para actualizar el tiempo externo, o lo que es lo mismo, almacenar el instante actual de ejecución dentro de la variable pública de la clase GSM `externalTime`. En cuanto al control de tiempo interno, éste es llevado a cabo por la rutina `gsm.getInfoCell()`. Dentro de esta rutina se comprueba que la respuesta al comando AT#MONI no tarde demasiado tiempo en producirse, y en caso de producirse este timeout se devuelve el valor 0 a la rutina principal. Como podemos observar, en caso de que el valor devuelto por `gsm.getInfoCell()` sea 0, la rutina principal volverá a llamar a esta función mientras no se produzca el timeout externo. Si se rebasase este timeout externo, estaríamos ante un grave fallo, ya que tras 40 segundos de ejecución no se ha obtenido aún la información de celda. En este caso, no quedaría más remedio que reiniciar el programa dentro del dispositivo mediante la llamada al método `resetFunc()`. En caso de que obtengamos la información de celda, mostramos por pantalla el tiempo actual de ejecución (cosa que no habrá que hacer en el programa de usuario).

Cabe destacar que en este caso tanto el tiempo externo como el interno nunca fueron rebasados en las múltiples ejecuciones del programa. No obstante, se ha decido implementar este control de tiempo por seguridad, en caso de producirse una desconexión de red inesperada.

Lo siguiente que debemos hacer es proceder al envío del mensaje de texto con la información de celda. Para ello hacemos repetitivamente una llamada al método `gsm.MandarSMS()`, hasta que éste devuelva un valor distinto de 0. Para evitar saturar el puerto serie del módulo GSM en caso de fallo al enviar el mensaje, haremos esperar medio segundo entre llamadas. Para esto último nos valdremos de la función `delay`. El control de tiempos se realiza de forma similar al bloque de código anterior. Todo lo expuesto en este párrafo puede observarse en las siguientes líneas de código de la rutina principal:

```
gsm.setExternalTime();
while(!gsm.MandarSMS(1,0,0,"")){
	delay(500);
	if ((millis()-gsm.externalTime)>40000){
		Serial.println("Fallo tras varias llamadas a CSMG");
		resetFunc();
	}

}
```

Una vez mandado el primer mensaje del aviso de emergencia, debemos pasar a

prestar atención al módulo GPS, para ver si éste arroja un posicionamiento a su salida. También debemos controlar el tiempo de espera, para así poder detectar uno de los posibles fallos que puede tener el posicionamiento GPS. Para ello, se desarrollaron las siguientes líneas de código:

```
gps.getGPS();
gsm.setExternalTime();
while(gps.coordinates[0] == 'n') {
      gps.getGPS();
      if (error_gps==0){
            delay(20);
            gps.setLED(1);
            delay(20);
            gps.setLED(0);
      }
      if ((millis()-gsm.externalTime)>90000 && error_gps==0){
            //  El GPS tarda mucho
            Serial.println("GPS timeout");
            error_gps=1;
            Serial.println("");
            Serial.print("t Inicial = ");
            Serial.println(millis());
            delay(750);
            do{
                  num_antenas=gsm.getInfoAllCells();
                  delay(500);
           }while(!num_antenas);
      gsm.setExternalTime();
            while(!gsm.MandarSMS(3,num_antenas,0,"")){
                  delay(500);
                  if ((millis()-gsm.externalTime)>40000){
                        Serial.println("Fallo tras varias llamadas a CSMG");
                        resetFunc();
                  }

            }
        }
      Last_DOP=134;
}
```

En estas líneas de código podemos observar que lo primero que se hace es una llamada al método `gps.getGPS()`. Gracias a esto, tendremos en la variable pública `coordinates` de la clase `GPS` datos acerca del posicionamiento. Estos datos pueden ser, o bien las coordenadas del posicionamiento, o bien la frase '`not valid coordinates`' la cual indicará que el receptor GPS aún no puede ofrecer un posicionamiento. De este modo, comprobando que la primera letra de la variable `coordinates` sea distinta de 'n', sabremos que tenemos ya un posicionamiento por parte del receptor GPS. Mientas esta letra sea igual a 'n', nos mantendremos dentro del primer bucle `while`. En este bucle hacemos parpadear el segundo indicador lumínico del usuario (el manejo del primer indicador lumínico corre a cargo de las funciones de la clase GSM) para indicar que se está realizando el segundo posicionamiento. También, al principio del bucle, actualizamos la variable `gps.coordinates` mediante el empleo del método `gps.getGPS()`.

Por otro lado, dentro de este primer bucle `while`, comprobamos que el tiempo de espera para la obtención del posicionamiento GPS no lleve más de 90 segundos. En caso de rebasar dicho tiempo, pasamos a obtener la información de celdas vecinas, y damos el valor 1 a la variable `error_gps` para indicar que ya hemos pasado a buscar la información de celdas vecinas por timeout del GPS. Gracias al cambio de valor de la variable `error_gps` no volveremos a realizar la búsqueda de información de celdas vecinas ni dentro de este bucle ni en el código restante de este programa principal. Una vez encontrada la información de celdas vecinas, se procede a enviar uno o varios SMS con dicha información. En este caso el

indicador lumínico permanecerá apagado, para indicar al usuario que se produjo un error en el GPS. Las líneas de código encargadas de esta tarea ponen fin al segundo bloque de código de la rutina principal (bloque `setup()`).

Cuando salgamos del bucle anterior, tendremos ya un posicionamiento GPS. Cuando ya tengamos este posicionamiento, ya sea tras haber agotado el timeout o no, tendremos que enviar dicho posicionamiento mediante un SMS. Esta tarea se acomete desde el último bloque de código del programa principal. Para que el mensaje se envíe, debemos igualar el valor de `last_DOP` al máximo valor posible del DOP (que resulta ser 99’99) dividido entre el factor de mejora (este factor de mejora es el que permite enviar un mensaje con un DOP elevado, pero que supone una reducción considerable en comparación con el valor del DOP asociado al último posicionamiento GPS enviado mediante SMS). En el programa final, consideramos que el factor de mejora es 0.5 (lo que supone que se enviarán mensajes con un DOP elevado siempre y cuando este nuevo valor del DOP esté por debajo de la mitad del anterior), por lo tanto, debemos igualar `last_DOP` a 200.

Pasamos, ahora, a comentar este último bloque de código de la rutina principal. Esta última entidad ha de desarrollarse dentro de una función denominada `loop()`, la cual no devolverá nada (tipo `void`). A continuación se exponen las primeras líneas de este último bloque:

```
void loop(){
	gps.getGPS();
	DOP=gps.HDOP;
	Serial.print("t_salida = ");
	Serial.print(millis());
	Serial.print("--> ");
	Serial.print(gps.coordinates);
	Serial.print("--> DOP = ");
	Serial.println(DOP);
```

En estas primeras líneas de este último bloque, obtenemos las coordenadas del posicionamiento ya disponible y su valor HDOP asociado. También imprimimos por el puerto serie todos estos valores, junto con el tiempo en que se accedió a éstas líneas de código para facilitar la recopilación de datos para las futuras pruebas del prototipo. No será necesario incluir esto último en el programa final de usuario.

Lo siguiente que debemos hacer es distinguir si el posicionamiento obtenido puede tener posibles imprecisiones, o por el contrario, es preciso con una alta probabilidad. Para ello, realizando primeramente múltiples experimentos, fijaremos un umbral de HDOP. Estos experimentos consistirán en probar el receptor GNSS en multitud de condiciones. Como sabremos dónde nos encontramos en cada prueba, simplemente compararemos el error de posicionamiento con el valor del HDOP asociado al mismo. Fijaremos un máximo error admisible en el posicionamiento GPS (del orden de 50 metros), y observaremos los valores del HDOP asociados a posicionamientos con errores mayores y menores a este máximo admisible.

Si en nuestro primer posicionamiento GPS obtenido el HDOP asociado es superior a dicho umbral, procederemos a enviar un mensaje de texto con esta información, indicando en dicho mensaje de texto que el valor HDOP asociado es elevado. También, se pasará a buscar la información de las celdas vecinas, en caso de que ésta búsqueda no se hubiese producido anteriormente en la ejecución del programa principal (rebasamiento del timeout del receptor GPS del bloque `setup()` del programa principal). También se contemplarán posibles mejoras del valor del HDOP en futuros posicionamientos imprecisos. Para estos fines, se desarrollaron las siguientes líneas de código:

```
        if (DOP>2.5 && DOP < 0.5*last_DOP){
            // Aqui se manda un sms advirtiendo alta DOP con las coords
            //obtenidas y se buscan la info de todas las celdas
            last_DOP=DOP;
            gsm.setExternalTime();
            while(!gsm.MandarSMS(2,1,DOP,gps.coordinates)){
                  delay(500);
                  if ((millis()-gsm.externalTime)>40000){
                        Serial.println("Fallo tras varias llamadas a CSMG");
                        resetFunc();
                  }

            }
            if(error_gps==0){
                  Serial.print("t Inicial = ");
                  Serial.println(millis());
                  delay(100);
                  error_gps=2;
                  do{
                        num_antenas=gsm.getInfoAllCells();
                        delay(500);
                  }while(!num_antenas);
            gsm.setExternalTime();
                  while(!gsm.MandarSMS(3,num_antenas,0,"")){
                        delay(500);
                        if ((millis()-gsm.externalTime)>40000){
                              Serial.println("Fallo tras varias llamadas a
                              CSMG");
                              resetFunc();
                        }
                  }
            }
      }
```

Puede apreciarse que el primer `if` de estas líneas, compara el valor HDOP del posicionamiento con el umbral, además de comprobar que este valor del HDOP sea inferior a la mitad del valor almacenado en la variable `last_DOP` (como vimos anteriormente, se permite el envío de más de un SMS con posicionamientos imprecisos, si el HDOP asociado mejora significativamente).

Si se cumplen las dos condiciones del párrafo anterior, se pasará a ejecutar el cuerpo de este bucle `if`. Lo primero que se hace, dentro de este bucle, es actualizar el valor de la variable `last_DOP`. Gracias a la actualización de esta variable podremos distinguir en futuros posicionamientos imprecisos, si se produjeron mejoras significativas del HDOP. La tarea siguiente será mandar un mensaje de texto con este posicionamiento. A la función encargada del envío del mensaje se pasan los argumentos necesarios para indicar que el tipo de mensaje de aviso es de posicionamiento y que éste puede resultar impreciso (recordar que el primer argumento de la función indica el tipo de mensaje de aviso, y que el segundo, en un mensaje de tipo posicionamiento, indica la posible imprecisión de éste). También se pasan como argumentos las coordenadas del posicionamiento y el valor del HDOP asociado.

Lo siguiente será evaluar si ya se produjo un fallo del GPS, para que, en caso contrario (error_gps igual a 0), se pase a buscar la información de las celdas vecinas. Aquí, almacenaremos en la variable `error_gps` el valor 2, para indicar que esta información ya ha sido encontrada. De esta manera, si se vuelve a mandar un SMS con un posicionamiento GNSS elevado (caso en que la variable DOP sea superior al umbral y menor que la mitad de la variable last_DOP, variable que almacena el HDOP del último posicionamiento impreciso para el cual se envió un SMS) no se procederá a realizar un escaneo de red.

Cabe destacar que mientras estas líneas de código son ejecutadas, el segundo indicador lumínico permanecerá apagado, indicando que siguen encontrándose problemas con el posicionamiento GPS (el que mayores prestaciones en términos de precisión puede ofrecernos).

Para finalizar este último bloque del código del programa principal, se desarrollaron éstas últimas líneas de código:

```
        else if (DOP<2.5) {
              gsm.setExternalTime();
              while(!gsm.MandarSMS(2,0,DOP,gps.coordinates)){
                    delay(500);
                    if ((millis()-gsm.externalTime)>40000){
                          Serial.println("Fallo tras varias llamadas a CSMG");
                          resetFunc();
                    }
              }
              gps.setLED(1);
              Serial.print("Tiempo Final del programa completo --> ");
              Serial.print(millis());
              Serial.println(" segundos; ¡TODO OK!");
              while(1);
        }
}
```

Podemos apreciar, que este `else` es el encargado de actuar en el caso de que el HDOP asociado al posicionamiento sea inferior al umbral fijado. En este caso se mandará un mensaje de texto, con el posicionamiento y el valor del HDOP. Se informará al usuario del éxito al enviar este mensaje con un posicionamiento GPS preciso, mediante el encendido del segundo LED. Para las pruebas sobre el prototipo se incluye la impresión por el puerto serie del tiempo final del programa. El último bucle `while` dejará al programa principal en bucle infinito. Otra opción, útil en el programa final de usuario, sería sustituir este último bucle por el código necesario para apagar el dispositivo (llevar una salida del Arduino, conectada al puerto de apagado del interruptor Pololu, a nivel alto) o efectuar una especie de ‘rastreo’ buscando cambios en los valores de las coordenadas del posicionamiento. En este rastreo podrían enviarse mensajes tras encontrarse cambios significativos en los valores de las coordenadas del posicionamiento (posible cambio de posición del sujeto).

3.6 Pruebas definitivas sobre el prototipo

Se han realizado numerosas pruebas sobre el prototipo. Entre estas pruebas cabe destacar el cálculo del valor HDOP umbral, el análisis de tiempo necesario para realizar el aviso de emergencia, y el análisis del consumo de este prototipo. Puede observarse que las dos primeras pruebas se realizarán sobre el software del dispositivo, mientras que la última será, más bien, de carácter hardware.

Gracias a las primeras pruebas podremos determinar cuánto tiempo puede implicar cada uno de los pasos propios del aviso de emergencia completo y cuáles son y en qué medida afectan los factores externos (mala cobertura de red celular, despejamiento de la antena del receptor GNSS, etc.).

Por otra parte, la estimación del consumo del prototipo del proyecto nos ayudará a elegir una batería para alimentarlo. También podremos estimar el tiempo de autonomía de dicha batería a la hora de realizar avisos de emergencia.

3.6.1 Precisión del posicionamiento y valor del HDOP

Como ya comentamos anteriormente, la precisión de nuestro posicionamiento GPS está íntimamente ligada al valor del HDOP asociado. Siempre que se obtuvo un error excesivo (del orden de 50 metros o superior) en la estimación de nuestro posicionamiento, obtuvimos también un valor alto, generalmente superior a 3 unidades, del HDOP; mientras que para estimaciones precisas obtuvimos valores bajos del HDOP (generalmente no superiores a 2.5 unidades). La realización de múltiples experimentos en diferentes condiciones nos permitirá fijar un valor umbral del HDOP para nuestra aplicación.

La precisión del prototipo en su posicionamiento GPS y el valor del HDOP, están a su vez muy relacionados con la visibilidad acimutal de la antena del receptor GPS (en su orientación a los satélites del sistema GPS). De tal manera, que si se interponen obstáculos considerables entre la antena del receptor y los satélites del sistema GPS (edificios muy altos en calles estrechas, bajo puentes, dentro de una vivienda alejados del piso superior) tendremos una mayor probabilidad de obtener errores en nuestro posicionamiento, y esto si es que se obtiene posicionamiento alguno. En algunos casos, cuando no haya suficientes satélites (al menos 4) cuya calidad de señal supere el umbral marcado por la sensibilidad del receptor, veremos vacíos los campos correspondientes a las coordenadas del posicionamiento GPS en la sentencias NMEA a la salida del receptor.

Para escoger el valor de umbral del HDOP realizaremos multitud de pruebas con el receptor GPS bajo multitud de condiciones externas (sobre todo en condiciones desfavorables, con poco despejamiento acimutal de la antena del receptor), siempre en situación de arranque en frío. Para calcular el error cometido en el posicionamiento, nos valdremos de la plataforma web GOOGLE MAPS. Debemos saber en qué punto exacto nos colocamos a la hora de hacer el experimento, para medir de forma aproximada el error cometido por el receptor GPS.

Finalmente, pasamos a una tabla de Microsoft Office Excel los valores del HDOP frente al error aproximado cometido en cada uno de los posicionamientos. Recogeremos estos valores de la tabla mediante el software Matlab, y mostraremos una gráfica con el valor del DOP y el error cometido en cada experimento. De forma visual, podremos escoger

el valor de umbral del HDOP para que un posicionamiento sea considerado como preciso.

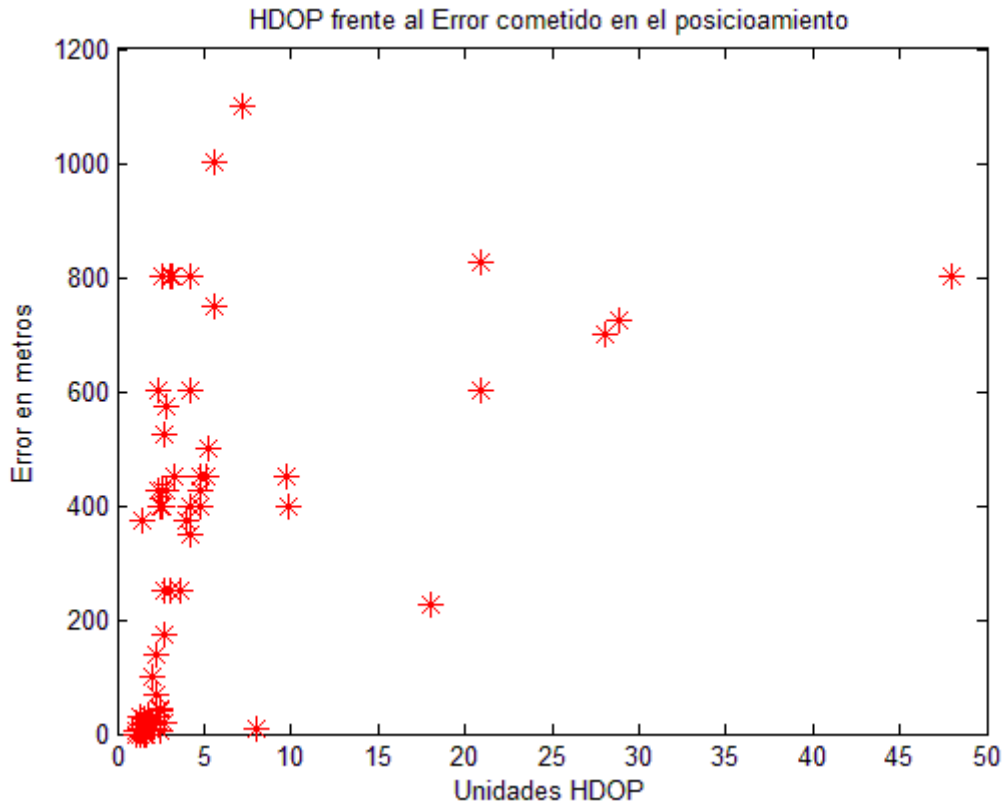


Figura 3.8: Gráfica I con el valor del error en el posicionamiento GPS frente al HDOP.

En la gráfica superior cada asterisco se corresponde con una prueba del receptor GPS. Este asterisco se sitúa sobre el eje X en función del valor del HDOP y sobre el eje Y en función del error cometido. Todos los asteriscos situados por encima de 50 unidades en el eje Y serán consideradas como pruebas cuyo posicionamiento resultó impreciso. A continuación se presenta una imagen ampliada de la zona de interés de esta gráfica.

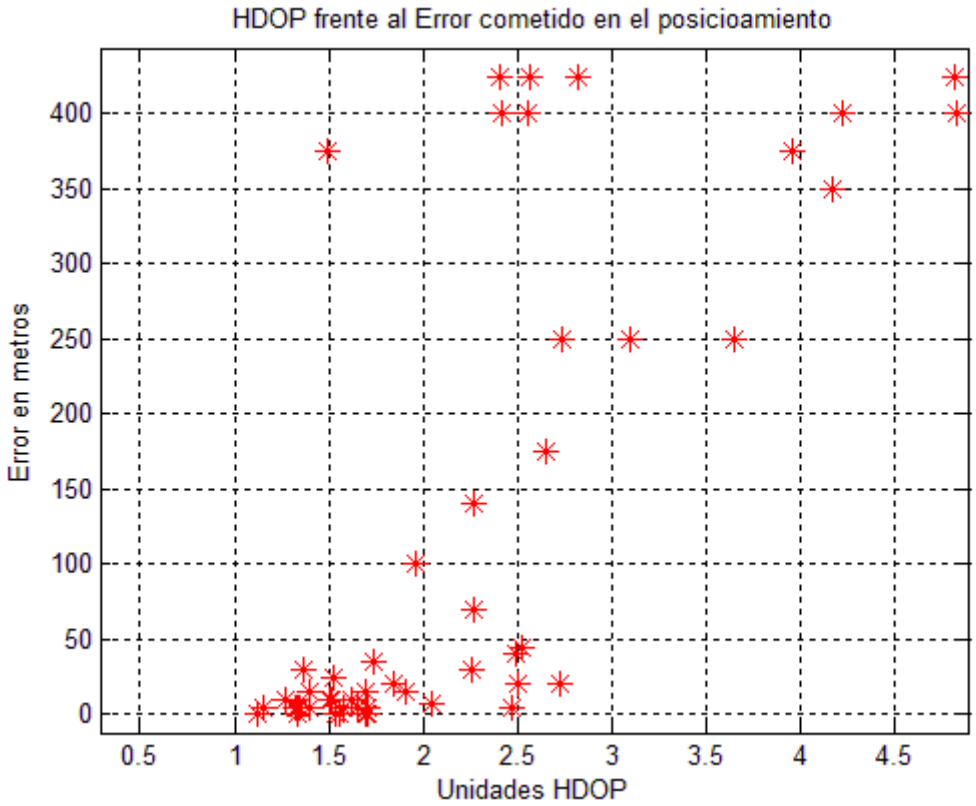


Figura 3.9: Gráfica II con el valor del error en el posicionamiento GPS frente al HDOP.

En la gráfica anterior podemos observar que si escogiésemos un valor umbral del HDOP igual a 2’5 unidades, sólo 6 experimentos hubiesen resultado imprecisos. En cambio, con un valor umbral de 2, sólo 2 experimentos producían un posicionamiento impreciso.

3.6.2 Tiempo necesario para dar el aviso de emergencia

En este apartado haremos un análisis temporal de cada uno de los procesos del aviso de emergencia. Dividiremos el aviso de emergencia en las siguientes tareas: registro de red, obtención de la información de celda, obtención de la información de celdas vecinas, obtención de un posicionamiento GNSS y envío de mensajes de texto. Haremos una estimación del tiempo que se emplea en ejecutar cada uno de estas tareas y más tarde veremos en qué situaciones se ejecutan. Este análisis temporal se ha realizado en tres escenarios diferentes: urbano, rural con buena cobertura y rural con mala cobertura.

Tiempo necesario para el registro de red

Para hallar este tiempo ejecutaremos las tareas de inicialización del receptor GPS y del módulo GSM, tras realizar estas tareas con éxito se imprime el tiempo obtenido mediante el método `millis()`. Lo siguiente que haremos será almacenar estos valores, y mediante el software Matlab los procesaremos, hallando su valor medio y desviación típica en los tres escenarios diferentes. En las pruebas realizadas se han encontrado diferencias significativas en estos tiempos en función del escenario donde se realizaron.

Tabla 3.1: Tiempo de registro de red del prototipo

Tipo de escenario	Tiempo Medio (s.)	Desviación (s.)	Tiempo máximo (s.)	Tiempo mínimo (s.)
Urbano	24.453	2.528	35.670	20.453
Rural	20.676	0.019	20.719	20.669
Rural (mala cobertura)	21.869	2.6837	26.670	20.669

Tiempo necesario para obtener la información de celda

Con el receptor GPS inicializado y el módulo GSM registrado, se realiza una llamada a la función encargada de conseguir la información de celda. Si la función falla se volverá a llamar repetitivamente, hasta obtener éxito. Una vez consigamos la información de celda mostraremos el tiempo actual. El tiempo necesario para obtener la información de red será el tiempo transcurrido desde que se consiguió el registro de red hasta que se imprime el nuevo valor de tiempo. Al igual que en el apartado anterior, hallaremos el valor medio y desviación típica del conjunto de valores de tiempo obtenidos en el experimento. Cabe destacar que no haremos distinción de escenarios para esta prueba, puesto que los tiempos obtenidos son muy similares.

Tabla 3.2: Tiempo de búsqueda de información de celda en el prototipo

Tiempo Medio (s.)	Desviación (s.)	Tiempo máximo (s.)	Tiempo mínimo (s.)
0.2397	0.0939	0.3390	0.1270

Tiempo necesario para hacer un 'escaneo' de red en busca de celdas vecinas:

Para realizar este cálculo ejecutamos una llamada a la función encargada de realizar la tarea mencionada. Imprimiremos por pantalla el tiempo, tanto al inicio como al final de esta tarea. El tiempo empleado en este proceso será la resta de ambos términos (tiempo final menos tiempo inicial). Al igual que en el resto de apartados, hallaremos el valor medio y desviación típica del tiempo empleado en realizar la búsqueda de la información de celdas vecinas. En las pruebas realizadas, hemos encontrado que el tiempo que toma esta tarea depende fuertemente del escenario donde sea realizada. Cuantas más estaciones base encuentre el terminal al alcance, mayor será la duración de esta tarea. Es decir, esta tarea implicará más tiempo de ejecución en entornos en los que haya muchas estaciones base al alcance (generalmente áreas urbanas) que en los casos en que se encuentren pocas (como es el caso de escenarios rurales).

Tabla 3.3: Tiempo de búsqueda de información de celdas vecinas en el prototipo

Tipo de escenario	Tiempo Medio (s.)	Desviación (s.)	Tiempo máximo (s.)	Tiempo mínimo (s.)
Urbano	129.09	33.066	186.14	96.24
Rural	57.39	5.53	63.31	51.52
Rural (mala cobertura)	25.03	5.51	29.915	10.049

Tiempo necesario para el envío de un sms

Cabe distinguir que en nuestro programa existen tres posibilidades de envío de diferentes SMS. Un primero con la información de la celda a la cual estamos registrados, un segundo con la información GPS y un tercero con la información de todas las celdas vecinas de una misma compañía. Tras realizar las pruebas, hemos concluido que los tiempos de envío de los mensajes de información de celda y posicionamiento resultan bastante similares. La única discrepancia en tiempos de envío de mensajes corresponde a los mensajes de información de celdas vecinas en algunas situaciones de buena cobertura, ya que la información transmitida en esta situación puede implicar el empleo de dos SMS.

Tabla 3.4: Tiempo de envío de un mensaje de texto en el prototipo

Tiempo Medio (s.)	Desviación (s.)	Tiempo máximo (s.)	Tiempo mínimo (s.)
3.879	0.832	8.020	2.590

Tiempo necesario para obtener un posicionamiento GPS

Este tiempo es el que mayor variabilidad va a presentar en nuestras pruebas, puesto que son múltiples los factores que afectan a la cobertura del sistema GNSS así como los posibles resultados que puede ofrecernos el receptor a su salida.

En la mayoría de casos, cuando la visibilidad acimutal de la antena del receptor GNSS resulta la adecuada, podremos disponer de un posicionamiento preciso en un tiempo bastante razonable. En estos casos el receptor puede conseguir el posicionamiento en un margen que va desde los 40 a los 80 segundos. En otros casos, el receptor no será capaz de

detectar la señal del mínimo número de satélites necesarios para ofrecer un posicionamiento a su salida. En esta situación, el receptor GNSS no ofrecerá posicionamiento alguno. En otras ocasiones, el receptor sí que podrá ofrecer un posicionamiento a su salida, pero éste será considerado como impreciso. En esta situación se considerará que la obtención del posicionamiento sigue pendiente (hasta encontrar un posicionamiento preciso). Es por estos motivos que no podremos hablar del tiempo que toma este proceso, pero sí podremos distinguir entre las tres situaciones antes mencionadas.

Tiempo total del aviso de emergencia:

La aplicación encargada del aviso de emergencia sólo termina de ejecutarse cuando se ha mandado un SMS de texto con un posicionamiento GPS con un valor HDOP inferior al umbral. A pesar de que el tiempo de ejecución tiene una fuerte dependencia con el tiempo necesario para obtener un posicionamiento GPS preciso, hemos realizado los anteriores cálculos para realizar una estimación de las diferentes fases por las que puede atravesar la aplicación y el tiempo que se implicará en cada una de ellas.

Como ya sabemos, pueden producirse situaciones tales como no obtener posicionamiento GPS alguno por un tiempo indefinido, o bien estar ante posicionamientos con un valor HDOP elevado indefinidamente. En estas situaciones la aplicación nunca tendrá fin.

Conocidos los tiempos necesarios para registrarse a una red, para obtener la información de la celda a la cual nos registremos, para el envío de un mensaje y para el escaneo de todas las celdas vecinas de Vodafone, podríamos hacer ya una estimación del tiempo necesario para ejecutar la aplicación en diferentes situaciones y el tiempo en que se enviarán los diferentes mensajes de aviso. A continuación presentaremos varias figuras en las que mostramos varias formas en que se podría ejecutar la aplicación mediante cronogramas. Por simplicidad y una mejor visualización, el cálculo se realiza para entornos rurales.

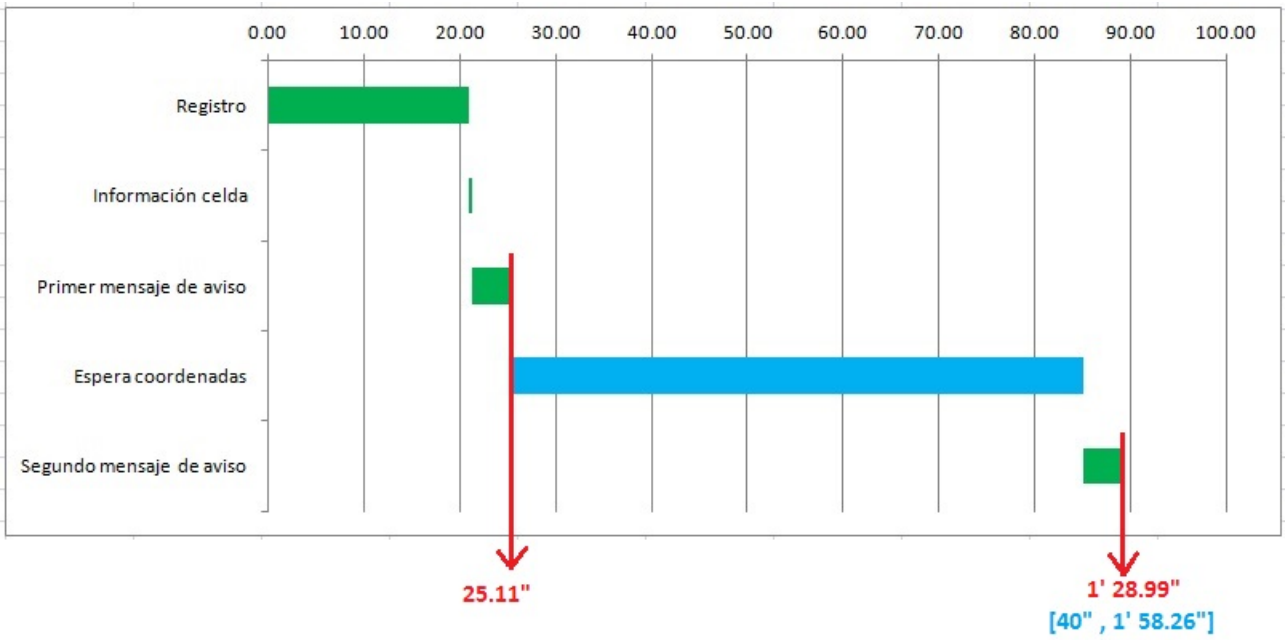


Figura 3.10: Cronograma para una ejecución correcta de la aplicación.

En la figura anterior puede apreciarse el cronograma de una ejecución correcta del código, es decir, que se mande el primer mensaje de aviso, y tras ello, antes de que se agote el tiempo de timeout del GPS se consigan unas coordenadas con un valor adecuado del HDOP. Podemos observar que el primer mensaje de aviso se manda algo por encima de los 25”. Mientas que el segundo mensaje puede enviarse desde unos 40” hasta 1’ 58.26” (límite impuesto por el timeout). Las barras verdes muestran tiempos que son fijos (se han empleado los valores medios de tiempo antes hallados), mientas las azules muestran tiempos que pueden variar, como es el caso del tiempo de espera al posicionamiento GPS. Tras cada envío de un mensaje se muestra una flecha roja con el instante de envío en la simulación (en rojo) y con el instante de tiempo inicial y final en que podría mandarse dicho mensaje (en azul). A continuación mostraremos el cronograma de caso en que se produce un timeout de la respuesta al GPS.

Figura 3.11: Cronograma para una ejecución con timeout GPS.

Como puede apreciarse en la figura superior, en caso de timeout del GPS, podremos tener el mensaje con la información de las celdas vecinas en un tiempo de casi 3'. Tras el envío de este mensaje, se procede, de nuevo a esperar las coordenadas del posicionamiento GPS, que aún podrían resultar imprecisas (pero la información de celdas vecinas no volvería a buscarse). Como podemos apreciar en el tiempo del último mensaje, éste puede que nunca llegue a mandarse, o puede darse el caso de que sólo se consiga un posicionamiento con HDOP elevado.

Por último mostraremos un cronograma para el caso en que se encuentre un posicionamiento GPS con elevado HDOP antes de que se produzca el timeout del GPS.

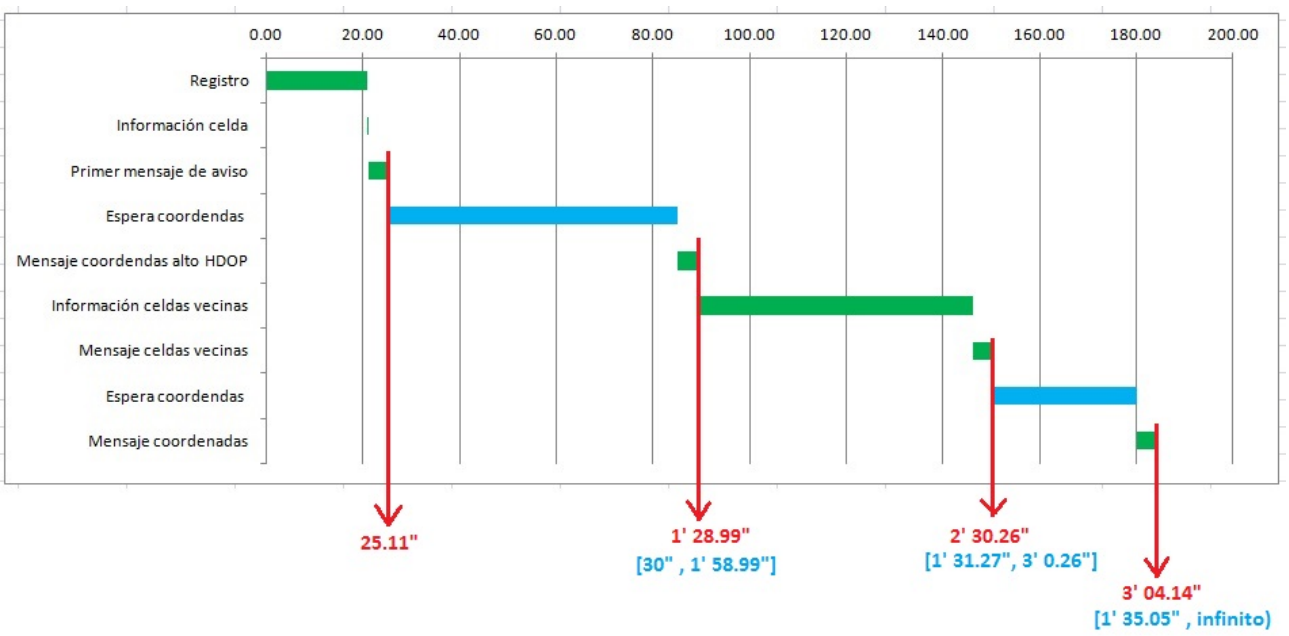


Figura 3.12: Cronograma para una ejecución con posicionamiento con HDOP elevado.

3.6.4 Aproximación al consumo eléctrico del dispositivo

Para testear el consumo eléctrico del prototipo hicimos varios experimentos. Observamos el consumo del módulo en diversas situaciones tales como: envío de un mensaje, registro de red, búsqueda de información de celdas vecinas. Conociendo el consumo de cada uno de estos procesos podremos aproximar el consumo final de nuestro prototipo (tendremos que tener en cuenta que el consumo dependerá a su vez de que porciones de código se ejecuten, ya que, por ejemplo, no supondrá el mismo consumo el caso en que es necesario obtener la información de las celdas vecinas que el caso en que no).

También se observó el consumo del dispositivo ejecutando el código final dentro del laboratorio, por lo que en estas condiciones, normalmente, no obtendremos un posicionamiento GPS preciso, o simplemente, no obtendremos posicionamiento GPS alguno.

Para medir la corriente que requiere el dispositivo introdujimos una resistencia de valor pequeño en serie con la entrada de alimentación de nuestro prototipo. El voltaje que caerá sobre esta resistencia será proporcional a la corriente que circula a su través, según la ley de Ohm (I = V / R). Analizando este voltaje con un osciloscopio podremos obtener el consumo de corriente a lo largo del tiempo. Debemos elegir un valor pequeño de resistencia para que la caída de tensión en sus bornes no sea significante. Una caída significante de voltaje en bornes de la resistencia puede provocar que la tensión restante que proporciona la fuente no sea suficiente para el dispositivo (además de que posiblemente no podamos detectar el malfuncionamiento del dispositivo, ante caídas de tensión en alimentación, a simple vista). Además de esto, debemos asegurarnos de que esta resistencia pueda aguantar grandes valores de corriente a su través, ya que como veremos en la siguiente imagen, proporcionada por el fabricante del módulo GSM, la corriente que puede exigir el GE865-QUAD es elevada ante determinadas situaciones.

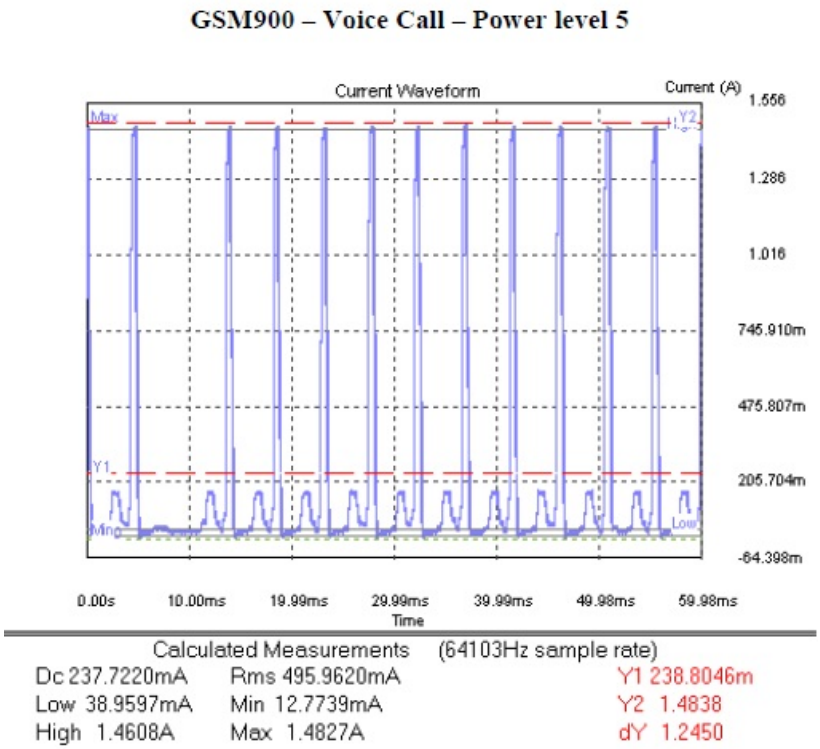


Figura 3.13: Picos de corriente indicados por el fabricante durante una llamada en GSM a 900MHz.

En la figura puede observarse que durante una llamada de voz, con el módulo trabajando en la banda de los 900 MHz de GSM, se producen picos de corriente de hasta casi 1.5 Amperios (el fabricante recomienda que de cara al diseño consideremos que pueden producirse picos de hasta 2 A.).

Por los motivos ya citados, compramos una resistencia bobinada de 1 Ohmio que puede soportar hasta 10 Watios. El valor de esta resistencia nos asegura que nunca restará suficiente voltaje a la fuente de alimentación, ya que una corriente de, por ejemplo dos amperios, reducirá tan sólo dos voltios a la alimentación de la placa (además situando el valor de alimentación elevado, pero dentro de los límites que especifican los fabricantes de Arduino, podremos asegurar que no se producirá nunca una caída de tensión suficiente como para que el prototipo no pueda funcionar adecuadamente). Con un valor de potencia nominal de 10 W. una resistencia de 1 ohmio puede aguantar hasta 3.33 amperios de corriente a su través, ya que:

$$P_{\max} = {I_{\max}}^2 \times R \Rightarrow {I_{\max}}^2 = \frac{P_{\max}}{R} \Rightarrow I_{\max} = \sqrt{\frac{P_{\max}}{R}} = \sqrt{10} = 3.33...\,\mathrm{A}.$$

A la hora de analizar el voltaje en bornes de la resistencia, debemos tener en cuenta que la corriente es igual a dicho voltaje partido del valor de la resistencia (Ley de Ohm). Para este caso, al tener una resistencia de un ohmio, el valor en amperios de la corriente será igual al valor en voltios del voltaje. No debemos olvidar la tolerancia de la resistencia (de un 5%), la cual afectará a la precisión de nuestra estimación de la corriente.

Dispuestos ya todos los materiales para realizar las medidas de corriente debemos realizar un montaje Hardware adecuado que nos permita medir con facilidad. Para ello empleamos 4 conectores para alimentar y aumentar la separación entre las placas Arduino y GSM-GPS. En el conector que se monta sobre la entrada de alimentación soldamos dos cables en las patillas correspondientes (**Vin** y **GND**). De este modo tenemos dos cables para alimentar ambas placas.

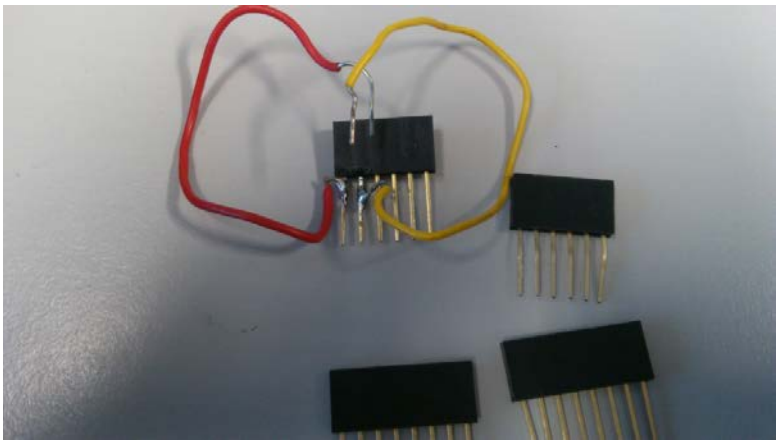


Figura 3.14: Conectores de expansión y alimentación para placa Arduino y GSM-GPS.

En la imagen superior pueden apreciarse dichos conectores, en la zona del centro puede apreciarse el conector que emplearemos para alimentar las placas.

Por otro lado, montamos en una placa 'Bread-board' el interruptor de Pololu, conectando a su entrada **Vin** la resistencia de 1 ohmio. También interconectaremos los pines duplicados de entrada (**Vin, GND**) y salida (**Vout, GND**).

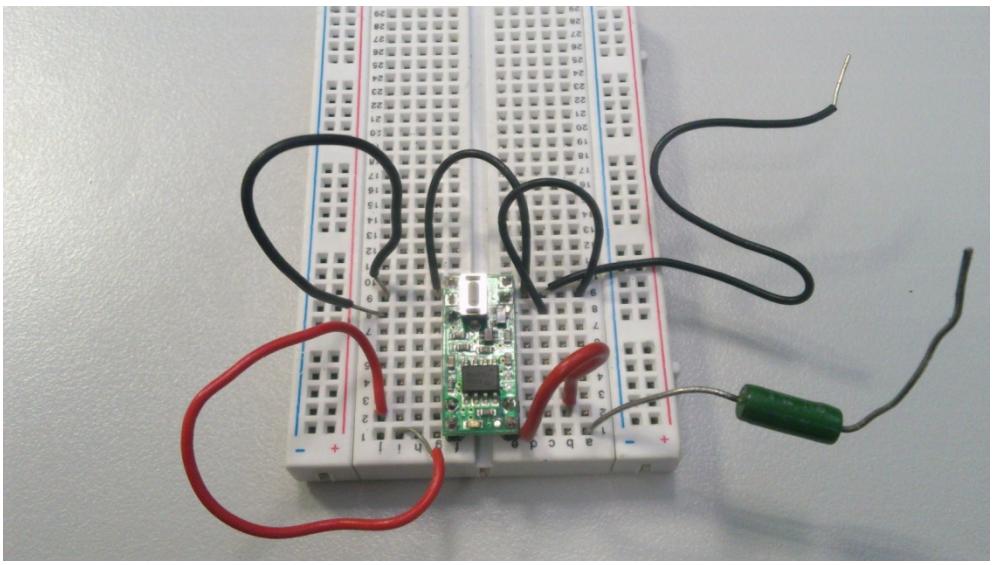


Figura 3.15: Interruptor Pololu con la resistencia conectada a Vin.

Ahora, simplemente debemos conectar los dos cables de alimentación a las salidas del interruptor de Pololu (**Vout** y **GND**). Con este montaje simplemente debemos encender la fuente de alimentación en configuración paralelo (para poder obtener el doble de corriente que podríamos obtener en configuración simple), situar su salida a 11.9 voltios y no limitar la cantidad de corriente que es capaz de suministrar. En la página web de Arduino podemos observar que nuestro modelo de placa acepta un voltaje de entrada comprendido entre 7 y 12 voltios, por lo que situando la alimentación de la fuente en 11.9 voltios, la resistencia podría tener una tensión en sus bornes de hasta 4.9 voltios (o lo que es lo mismo, que circule a través suya 4.9 Amperios, cosa que no esperamos que ocurra) sin que ello supusiese un posible malfuncionamiento del prototipo.

Una vez encendida la fuente conectamos la sonda del osciloscopio a los bornes de la resistencia, de tal modo que el conector de masa vaya al pin de entrada del interruptor Pololu y el conector positivo vaya a la salida de la fuente de alimentación; de esta manera estaremos recogiendo un voltaje siempre positivo (puesto que la corriente siempre será entrante al prototipo).

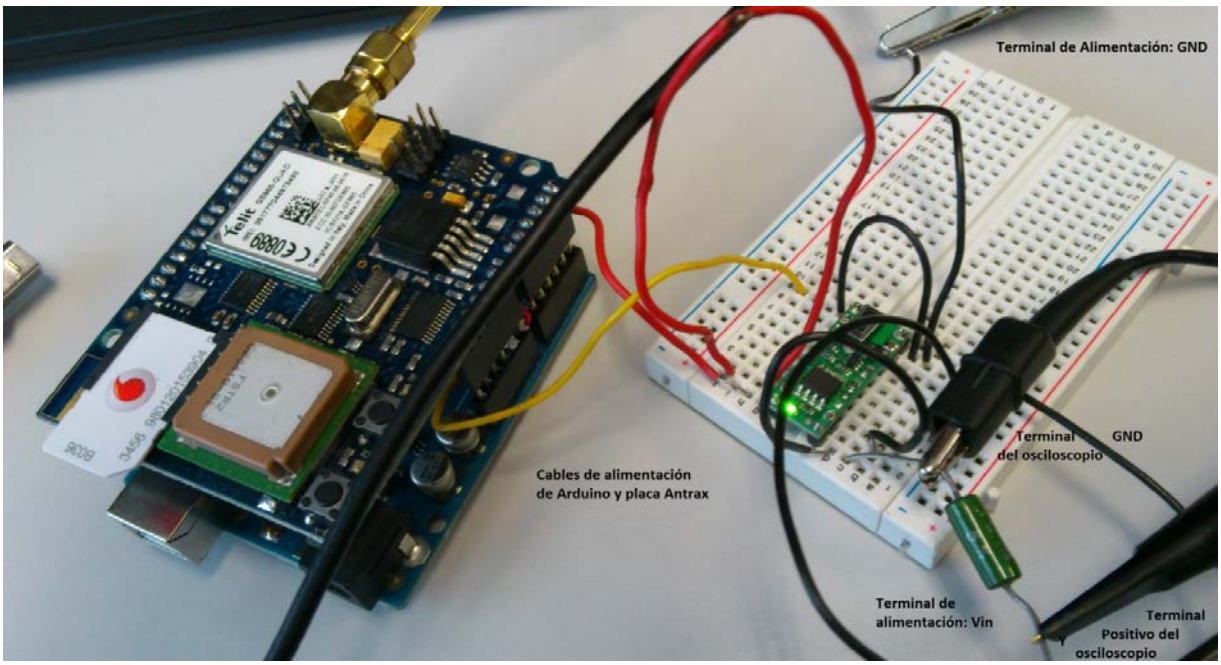


Figura 3.16: Montaje empleado para medición del consumo de corriente.

En la imagen superior puede apreciarse el montaje comentado de cara a medir el consumo de corriente a través de la medida del voltaje en bornes de la resistencia. Con este montaje veremos el consumo eléctrico de todo el prototipo (incluyendo el consumo residual del interruptor de Pololu con el prototipo apagado).

Ahora, simplemente debemos cargar un código para testeo del consumo del prototipo ante las situaciones propuestas. La forma más simple es desarrollar un programa que primero inicialice el GPS, y que cada vez que se accione un pulsador se pase a realizar una tarea concreta. Con la primera pulsación pasemos a inicializar el módulo GSM, con la segunda a realizar un escaneo de red en busca de la información de celda actual, con la tercera pulsación buscaremos la información de celdas vecinas y con la cuarta se realizará el envío de un SMS. Mediante el pulsador de Pololu apagaremos y encenderemos el prototipo.

Para cada tarea ajustaremos las escalas horizontal (tiempo) y vertical (voltaje) del osciloscopio para obtener una visualización completa del evento. Cada tarea será realizada varias veces, para cada una de las cuales guardaremos en un USB las imágenes de la pantalla del osciloscopio y un fichero 'CSV' en el cual tendremos las muestras de la forma de onda de la señal recogida.

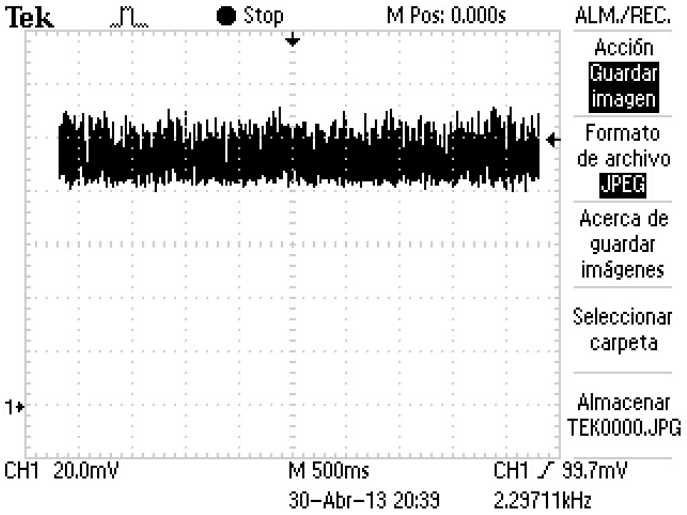


Figura 3.17: Forma de onda osciloscopio con tan sólo el módulo GPS inicializado.

En la imagen anterior podemos observar el consumo del módulo cuándo solamente está el GPS inicializado. Podemos ver que la forma de onda del voltaje varía unos 10 mV. (por arriba y por abajo) en torno a un valor medio de aproximadamente 94 mV. Esto quiere decir que el prototipo requiere una corriente media de unos 94 mA. con variaciones de aproximadamente ±10 mA.

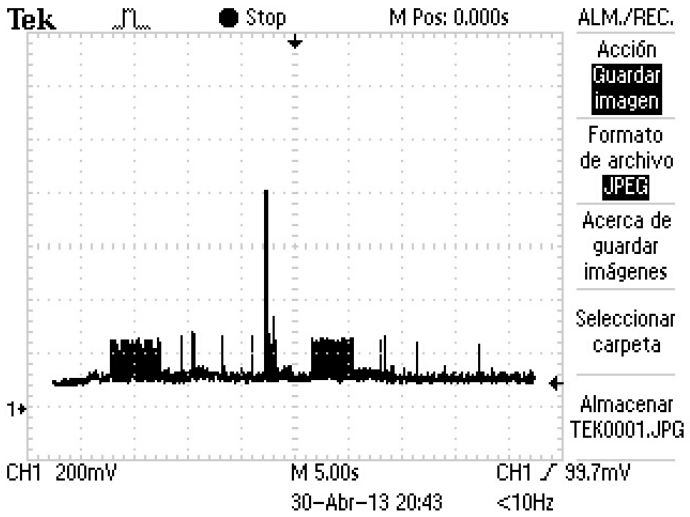


Figura 3.18: Forma de onda osciloscopio para la inicialización del módulo GSM.

En esta imagen podemos observar que el consumo en ciertos instantes aumenta drásticamente con respecto al caso anterior. Se puede apreciar un pico de consumo algo superior a los 800 mA. no obstante, la mayoría de picos se mantienen en torno a los 250 mA. Cabe destacar que en los primeros instantes (aproximadamente, los primeros tres segundos) de esta gráfica sólo el GPS está activo; tras esto vemos que se produce un pequeño aumento del consumo, el cual se correspondería con la ejecución de las primeras líneas de código del método de inicialización del módulo GSM.

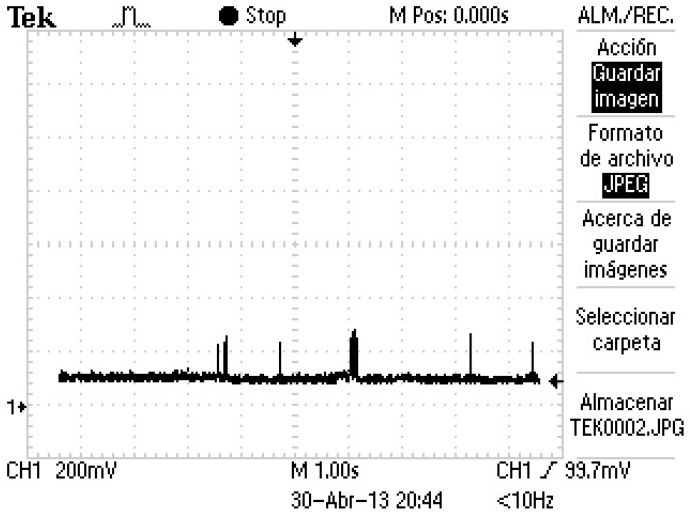


Figura 3.19: Forma de onda osciloscopio para la obtención de la información de celda servidora.

En esta imagen podemos apreciar que el consumo que representa la ejecución del comando `#MONI` supone unos pocos picos de corriente, situados en torno a los 250 mA., con respecto al caso de tener activados los módulo GPS y GSM, con este último en reposo.

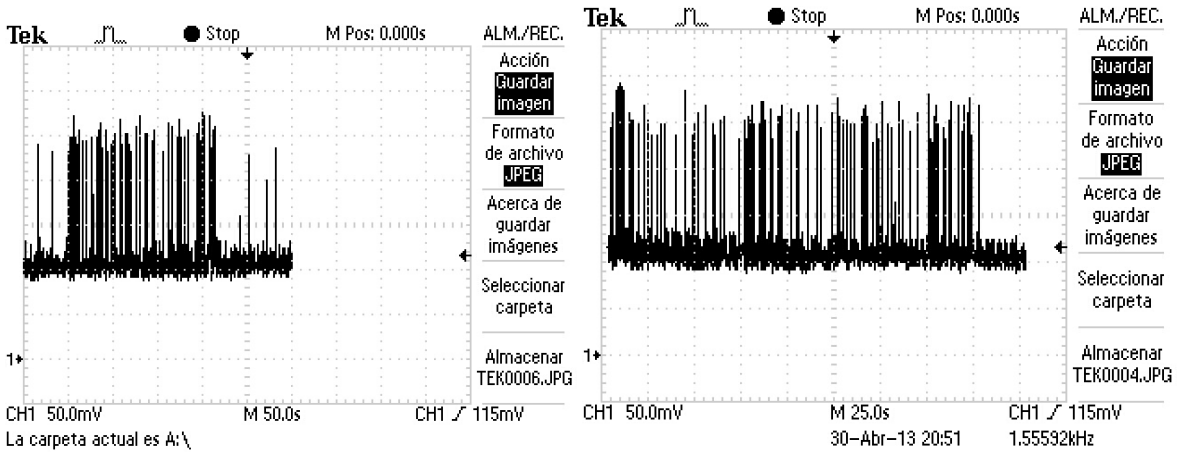


Figura 3.20: Forma de onda osciloscopio para 2 realizaciones diferentes del experimento de búsqueda de información de celdas vecinas, en diferentes escalas de tiempo.

Aquí podemos apreciar que la ejecución del comando `AT#CSURV` supone un mayor consumo de corriente que el caso del comando `AT#MONI`, simplemente porque éste primero necesita mucho más tiempo para proporcionar su respuesta. En cuanto a los valores que toma el consumo estos son muy similares a los obtenidos tras ejecutar el comando #MONI.

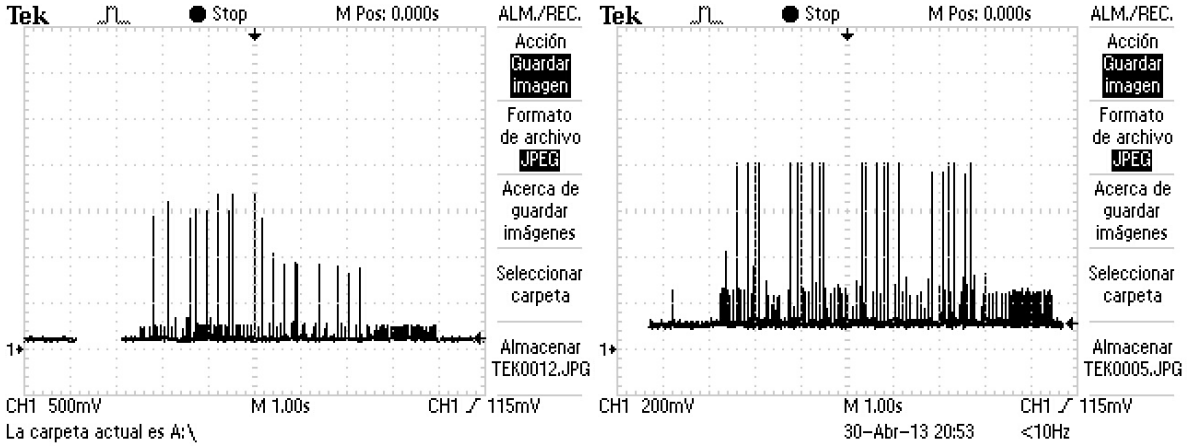


Figura 3.21: Forma de onda osciloscopio para 2 realizaciones diferentes del envío de un SMS.

Es a la hora de enviar un SMS cuando vamos a tener los mayores picos de consumo y, seguramente, el mayor valor del consumo medio. No obstante, el tiempo necesario para el envío de un mensaje de texto no conlleva más de 8 segundos de altos valores de corriente. Podemos apreciar que en la primera imagen los mayores picos de consumo son de aproximadamente 1.7 A. y que en la segunda son algo mayores a los 800 mA. En estas pruebas es donde hemos encontrado mayor variabilidad en los resultados de un experimento a otro, dentro del mismo estudio.

Decidimos crear un 'script' en Matlab que fuera capaz de cargar los archivos '.csv' de todos los experimentos de un mismo estudio para hallar la media de consumo de cada uno de los procesos analizados anteriormente. En este script indicaremos el comienzo y el fin del evento dentro de la gráfica, para que no incluya toda la forma de onda en los resultados (por ejemplo, en el caso del envío de un SMS sólo tendremos en cuenta unos 8 segundos de la gráfica del consumo).

Tabla 3.5: Consumo eléctrico del prototipo

Tipo Evento	Corriente media (mA.)	Consumo medio (mAh.)
GPS sólo	93.6	0.005
Registro Red	144.3	1.3
Registrado	112.5	—
Información de celda	112.5	0. 1
Mensaje	138.5	0. 3
Información celdas vecinas	112.6	7

Podemos observar en la tabla de arriba que cualquiera que sea el evento que tenga lugar, su consumo medio es muy reducido. Tras haber observado los picos de corriente que tienen lugar en ocasiones, y para garantizar que la batería entregue tales picos, lo más adecuado sería recurrir a una batería de Li-ION para alimentación del prototipo. El problema es que normalmente estas baterías son de 3’7 voltios, pero empleando dos de ellas en serie lograríamos ascender a los 7’4 voltios, más que suficiente para alimentar el prototipo. Si la capacidad de estas baterías es de, por ejemplo, 500mAh, podríamos estar ejecutando más de 3 horas el código de la tarea que más consumo de corriente representa (el registro de red).

Tras procesar estos valores con Matlab, decidimos comprobar si éstos se reflejaban en los valores arrojados por la fuente de alimentación en su pantalla. Revisando una por una cada prueba, siempre encontramos que los valores mostrados en pantalla se movían en torno a los obtenidos en Matlab (tanto por encima, como por debajo). De esta forma, concluimos que el simple uso de la fuente hubiese bastado para aproximar el consumo eléctrico del dispositivo. Lo que no seremos capaces de percibir en la fuente serán las abruptas variaciones de corriente, tales como picos que se producen en algunas de las pruebas. Para poder observar las variaciones instantáneas de corriente tendremos que recurrir siempre al osciloscopio.

Capítulo 4
Desarrollo del dispositivo final

4.1 Introducción

A pesar de que el prototipo cumple los objetivos de este proyecto, hemos decidido crear otro dispositivo más eficiente en términos de espacio y consumo. En este apartado iremos detallando el proceso completo del desarrollo de un dispositivo más compacto y cercano al modelo comercial final.

En primer lugar, una reducción del tamaño de este prototipo sería el objetivo primordial de este nuevo diseño. Por otro lado, resultaría útil emplear sólo aquellos recursos de los módulos GSM y GPS que sean necesarios, ahorrando en términos de espacio y consumo de corriente. Deberemos por tanto, escoger de nuevo una serie de componentes, pensando en cumplir los objetivos del proyecto consiguiendo el ahorro de espacio.

También sería interesante que los indicadores lumínicos fuesen más entendibles, lo cual podría lograrse simplemente añadiendo unos pocos LEDs extra.

4.2 Elección del módulo GSM

Necesitamos disponer de un módulo, a ser posible similar al empleado en el prototipo (GE865-QUAD), de tamaño reducido, que nos permita realizar todas las operaciones desarrolladas en el prototipo y tenga una serie de características extra. Como hemos comentado la reducción de tamaño del dispositivo es esencial en este aparatado, por lo tanto será fundamental comenzar adquiriendo un módulo con un tamaño lo más reducido posible (sin perder las funcionalidades requeridas).

Como características o requisitos extra podrían estar los siguientes, en orden de importancia:

1.- No necesitar de un microcontrolador para el manejo de este módulo. Por suerte, la mayoría de módulos GSM cuentan con un intérprete de Python embebido y una memoria más que suficiente para cargar aplicaciones en ellos. Al no necesitar un microcontrolador ahorraremos mucho espacio (en el caso del prototipo, sería equivalente a decir que no necesitamos la placa Arduino). También supondrá una reducción en la complejidad del diseño, al no tener que preocuparnos por intercomunicar el módulo GSM con un microcontrolador y las necesidades que esto genera (posible adaptación de niveles lógicos, regulaciones de voltaje, etc.). Además, esto también supondrá un ahorro económico en el desarrollo del dispositivo final. Consideraremos este requisito como fundamental.

2.- Que el módulo GSM tenga a su vez un módulo GPS embebido, como es el caso de algunas modelos de Telit. De hecho la mayoría de las familias de módulos GSM de Telit incluyen una versión con GPS. De esta manera ahorraremos también mucho espacio y el diseño final se simplificará, puesto que no tendremos que añadir un módulo GPS independiente al diseño ni preocuparnos por cómo intercomunicarlo con el módulo GSM. Este requisito resulta bastante importante.

3.- Que el módulo posea unos conectores lo más sencillos posibles, tanto para su conexionado o soldado a una placa, como para conectar las antenas (GSM y GPS) y la tarjeta SIM. De esta manera el diseño puede simplificarse bastante, al no necesitar preocuparnos por la creación de pistas de radio-frecuencia (RF) para cortocircuitar el pin de señal RF del módulo con los conectores para las antenas. Si el módulo además posee una forma ‘amigable’ de conectarse a nuestro PCB, también reduciremos ligeramente la complejidad, ya que, por ejemplo, es más manejable un módulo con un conector MOLEX

universal que uno de soldadura BGA. Telit dispone de multitud de módulos con diferentes conectores.

A continuación se muestra una imagen de los dos módulos considerados, el GM862-GPS (izquierda) y el GE910-GNSS (derecha):

Figura 4.1: Módulos GSM (con GNSS embebido) candidatos para el diseño final.

Telit únicamente dispone de un módulo que cumple los tres requisitos extra, además de poseer la funcionalidad ya obtenida en el prototipo. Este módulo es el GM862-GPS. No obstante la adquisición de este módulo implica la aparición de dos problemas: el primero es que su tamaño ya no es tan reducido (43.9 mm x 43.9 mm, un área casi cuatro veces superior al GE865-QUAD); y el segundo, que el fabricante no recomienda su uso en nuevos diseños, puesto que en no mucho tiempo dejará de fabricarse.

Debido a lo descrito en el párrafo anterior, hemos decidido adquirir el módulo GE910-GNSS, que no cumple la tercera característica extra. Sus pines son del formato Land Grid Array (LGA, matriz de conexiones similar al de BGA salvo que no aparecen bolitas de estaño, sino normalmente, conectores planos de oro). Sus dimensiones son 28.2 mm x 28.2 mm (ligeramente superiores a los 22 mm x 22 mm del GE865-QUAD).

Por un lado, no podremos realizar un soldado ‘a mano’ de este componente, por lo que seguramente tengamos que recurrir a una empresa externa a la escuela. Por otro lado, debemos ser ‘cuidadosos’ a la hora de crear las pistas de RF tanto como para conectar la antena GSM como para la antena GPS. Las especificaciones de estas pistas vienen indicadas por el fabricante, en el manual hardware.

4.3 Experimentación con el ‘Evaluation Kit’

Para poder experimentar con el módulo GE910-GNSS sin tener que hacer ningún desarrollo previo, Telit vende un Kit de Evaluación (EVK2) que puede usarse con todos sus módulos. Para interconectar cada una de las diferentes familias de módulos al EVK2, Telit comercializa una serie de placas interfaz. Este kit de evaluación está desarrollado para conectarse de manera sencilla a un PC, para poder probar todas las funcionalidades disponibles.

Para la realización del proyecto, hemos adquirido el EVK2 y la placa interfaz correspondiente a la familia GE910. Esto nos ayudará a hacer casi todas las pruebas previas necesarias para el desarrollo software de nuestro dispositivo y puede servir para orientar nuestro diseño Hardware. El único inconveniente es que la versión del GE910 disponible para el EVK2 es la QUAD, la cual no incorpora un GPS embebido. A continuación se muestra una imagen del kit de evaluación con la placa interfaz ya montada.

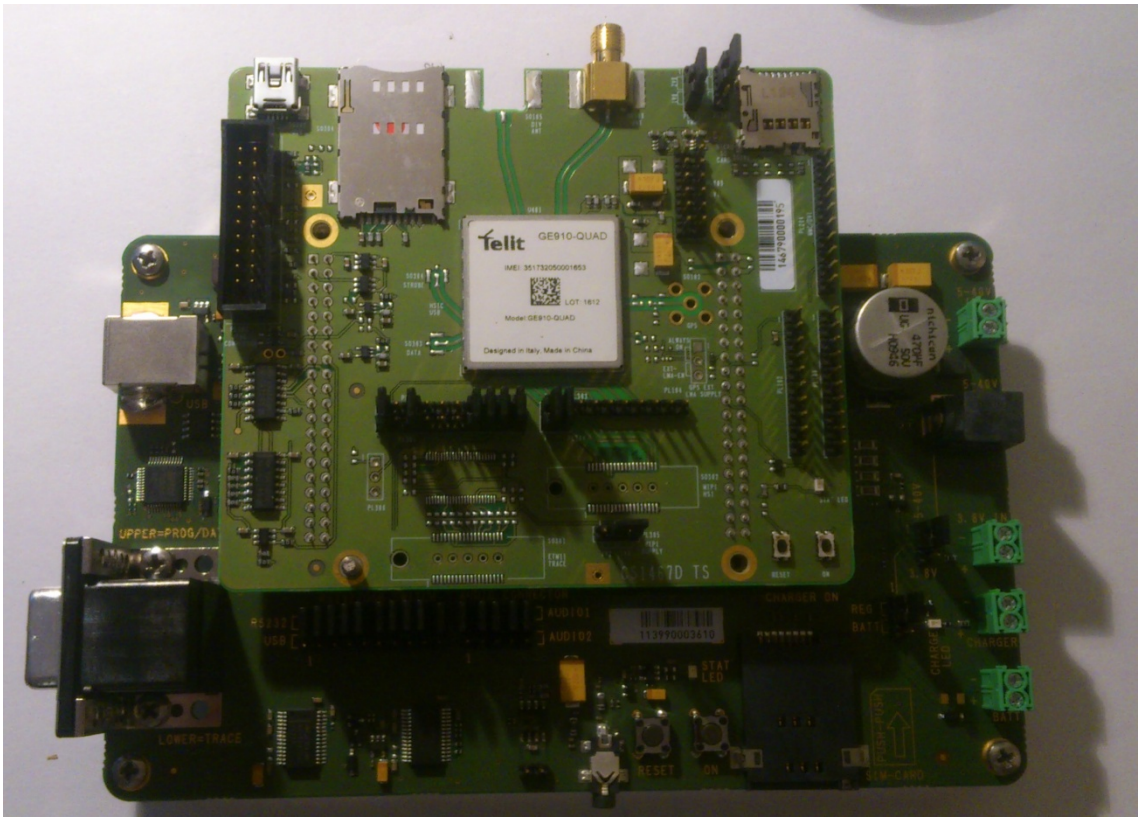


Figura 4.2: EVK2 con la placa interfaz GE910 montada.

4.3.1 Características principales del EVK2

Podremos interconectar este módulo con nuestro PC mediante el USB o el puerto serie. Hemos decidido hacerlo a través del puerto serie, puesto que es más sencillo en términos de desarrollo hardware y software para cuando decidamos hacer el dispositivo final. En caso contrario tendríamos que emplazar un chip para la conversión RS-232 a USB (por ejemplo, el chip FTDI) en vez de un sencillo conversor de niveles CMOS a los niveles lógicos del RS-232 del ordenador (como es el chip MAX3218) en nuestro diseño, además de tener que instalar correctamente en nuestro ordenador los drivers para emular un puerto serial a partir de un USB.

Esta placa dispone de varias entradas de alimentación, permitiendo incluso conectar una batería y su respectivo cargador. Podemos alimentar el dispositivo mediante una fuente de

alimentación que pueda entregar al menos 1 amperio de corriente operando en un rango de voltaje continuo (DC) de 5 a 40 voltios, para ello disponemos de una clema y un conector de tipo coaxial (deberemos elegir uno de ellos, si decidimos alimentar de esta manera al dispositivo). Otra opción sería emplear una fuente DC de 3.8 voltios que pueda entregar al menos 2 amperios de corriente. Por último, está la opción de alimentar el EVK2 mediante una batería típica de Li-ion de 3.7 voltios cuya capacidad sea superior a 500 mAh. Ésta última configuración, permite además la conexión de un cargador, para cargar la batería a través del EVK2.

El EVK2 permite elegir entre sus múltiples configuraciones mediante el emplazamiento de numerosos jumpers, presentes tanto en la placa interfaz, como en la propia placa evaluadora.

La placa evaluadora presenta dos indicadores lumínicos, uno para informar de que la batería se encuentra en el estado de carga y otro, llamado ‘status led’, que sirve para informar al usuario del estado del módulo (apagado, encendido sin registrar a una red, encendido y registrado, encendido y realizando una llamada). Este último led también está presente en la placa interfaz, ofreciendo la misma funcionalidad.

Tanto la placa interfaz, como la evaluadora poseen dos pulsadores, uno para encendido del módulo GSM y otro para realizar un reset externo. En nuestro caso, deberemos emplear siempre los pulsadores de la placa interfaz, como indica el manual de usuario del EVK2 en el aparatado dedicado a la interfaz GE910.

4.3.1 Puesta en marcha del EVK2:

En primer lugar debemos elegir cómo alimentaremos este dispositivo. En la facultad disponemos de multitud de fuentes FAC-662B de la empresa PROMAX las cuales pueden entregar hasta un amperio de corriente (se pueden llegara a obtener 2 amperios trabajando con una de estas fuentes en configuración paralelo, ya que cada aparato presenta internamente dos fuentes) y ofrecen un rango de voltaje de salida de 0 a 30 voltios (aunque en realidad pueden llegar a ofrecer 32.5 voltios con una mayor limitación en la corriente de salida).

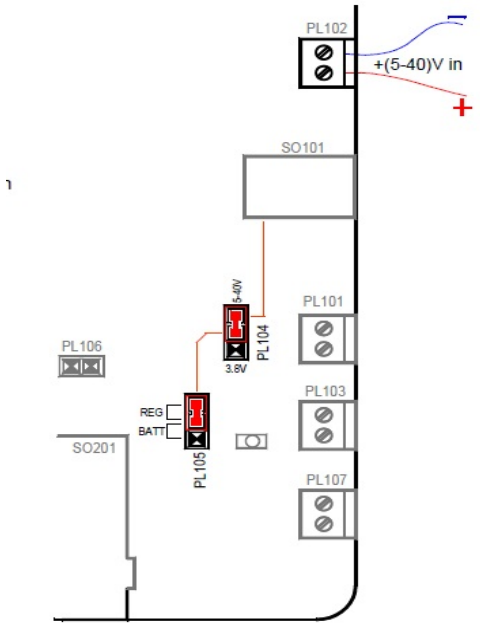


Figura 4.3: Opción escogida para alimentación del EVK2 y su respectiva configuración de jumpers.

Como puede observarse en la anterior imagen, la alimentación será proporcionada a través de la clema disponible en la placa evaluadora (PL102), y para ello colocaremos los jumpers de las líneas PL104 y PL105 sobre los pines 1 y 2.

Como emplearemos el puerto serie del EVK2 y no el puerto USB, debemos colocar una fila de 10 jumpers de la EVK2 en la posición RS232, como puede apreciarse en la siguiente imagen.

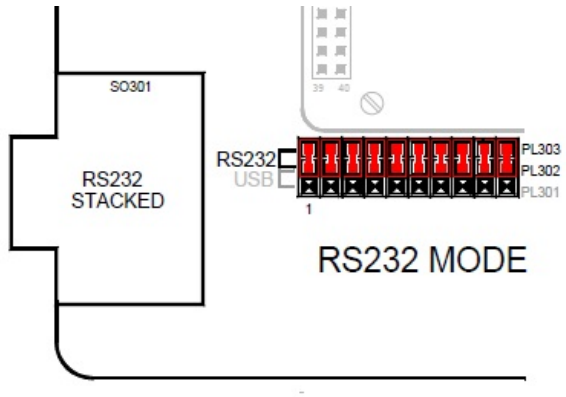


Figura 4.4: Configuración de los jumpers del EVK2 para comunicación con el PC a través del puerto serie.

Antes de empezar a mandar comandos al módulo GSM, procederemos a realizar una actualización de su firmware, para cargar en él la última versión disponible. Para ello debemos descargar dicho firmware junto al programa Xfp a través de Telit o su distribuidor en España: VENCO. Ejecutamos el programa Xfp e indicamos en qué directorio se encuentra el firmware obtenido, después alimentamos el EVK2 y conectamos su puerto serie al del ordenador. Hecho esto ya podemos actualizar el firmware, proceso que dura unos 10 minutos.

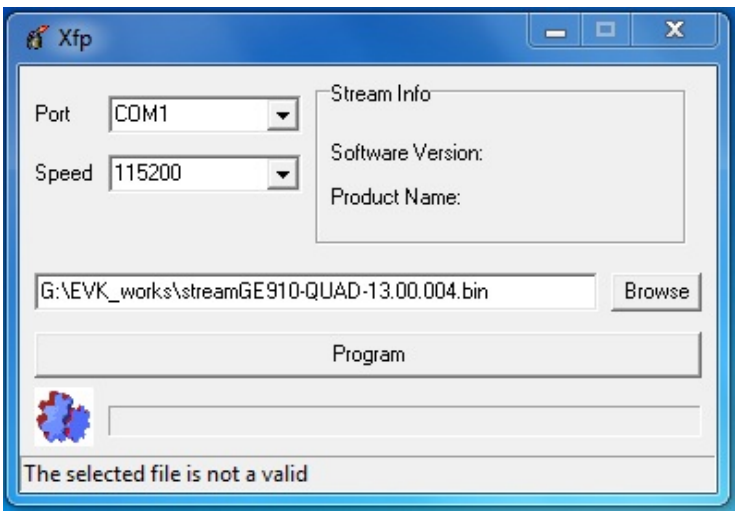


Figura 4.5: Interfaz del programa XFP.

En la imagen anterior se nos muestra la interfaz del programa Xfp, en la cual aparece seleccionado el firmware estable más reciente. Debemos indicar que la carga de este firmware se hará a través del puerto COM1 (el cual se corresponde con el puerto serie con conector macho de 9 pines del PC) a una velocidad de 115200 bps. Para actualizar el firmware debemos pulsar sobre el botón de 'Program' y, acto seguido, encendemos el módulo GSM presionando sobre el botón de ON del EVK2.

Ahora procedemos a insertar la tarjeta SIM en la ranura disponible en la placa interfaz. Conectamos la antena GSM al conector RF de la placa interfaz.

Ahora, ya podemos proceder a enviar comandos AT al módulo GSM a través del puerto serie de la EVK2. Para ello necesitaremos tener en nuestro PC un programa que nos permita establecer y monitorizar una comunicación serial, como puede ser el software Hyperterminal de Windows. Creamos una nueva conexión con este programa y, en la siguiente ventana indicamos que deseamos realizar una comunicación a través del puerto COM1. El último paso, previo a mandar comandos, será la configuración del puerto COM1 con Hyperterminal.

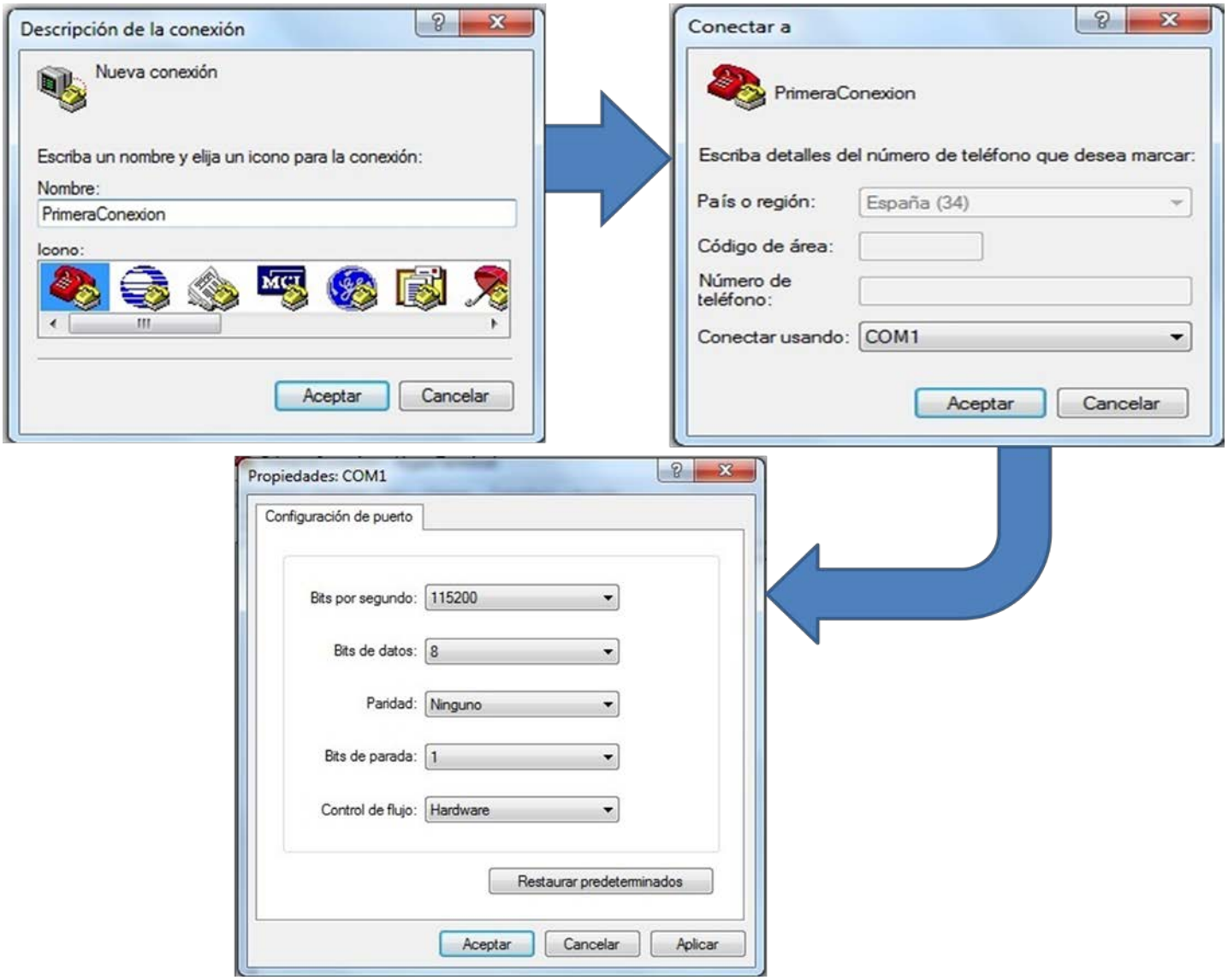


Figura 4.6: Flujo de ventanas de la configuración de Hyperterminal.

En la anterior figura podemos observar la secuencia de ventanas que debemos seguir para establecer de forma correcta la comunicación serial con el módulo GSM, a través del software Hyperterminal. En la última ventana de esta secuencia se puede apreciar la configuración del puerto COM1, con una velocidad de 115200 bps, 8 bits de datos, sin bits de paridad, 1 bit de parada y control de flujo vía hardware (manejado a través de las líneas CTS, RTS, DTR, DSR y DCD, propias del protocolo serial RS232).

Una vez llegados aquí ya podremos enviar comandos AT al módulo GSM, pudiendo, entre

otras cosas, probar tareas como la conexión a una red, obtención de la información de la celda de registro y celdas vecinas, etc.

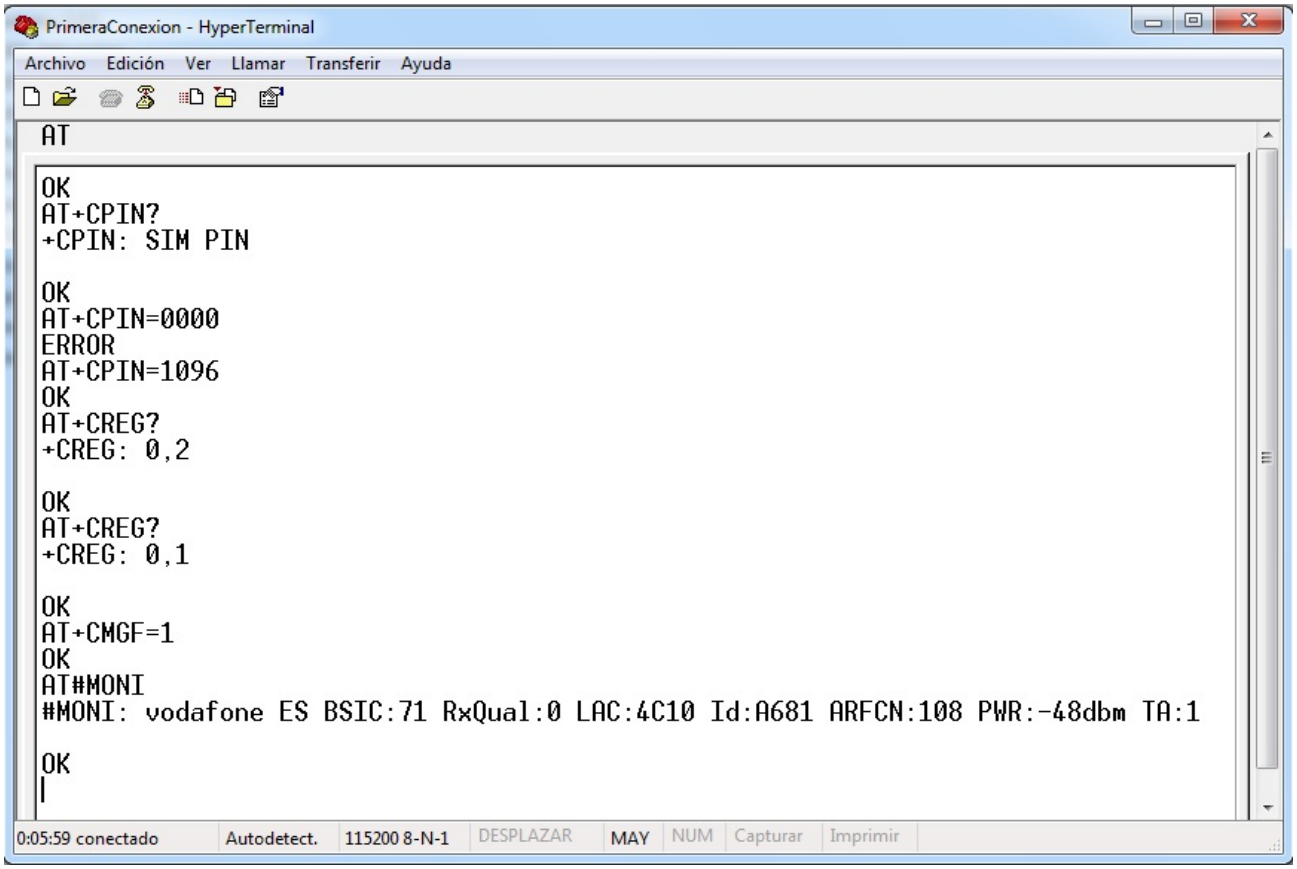


Figura 4.7: Primeras pruebas con envío de comandos AT mediante Hyperterminal.

En la figura anterior se muestran la secuencia de comandos y respuestas que necesitaríamos para garantizar la conexión a una red y obtención de información sobre la celda de registro. La conexión a la red está garantizada, como vimos anteriormente, cuando la respuesta al comando ***AT+CREG?*** es ***0,1*** (la respuesta ***0,2*** indica que el terminal móvil está buscando actualmente una celda a la que registrarse). Podemos observar que en la primera introducción del código PIN (comando ***AT+CPIN=0000***) el módulo nos devuelve la respuesta ***ERROR***, debido a que éste no es el código correcto.

4.3.1 Programación del módulo GE910 mediante el EVK2:

Realizadas unas primeras pruebas básicas, pasaremos a probar la extensión Easy Script disponible en todos los módulos de la familia GE910. Esta funcionalidad es la que nos permite cargar scripts, programados en lenguaje de alto nivel, como es Python; la cual nos permite tener el módulo en modo autónomo (o stand-alone), sin necesitar de un micro-controlador externo para su manejo.

Esta extensión cuenta con una serie de comandos AT para su manejo. Existe un comando para lectura de los archivos, lectura de los nombres de los archivos cargados, carga de archivos, etc. A continuación mostraremos los comandos empleados:

a) ***AT#LSCRIPT*****,** este comando es empleado para que el módulo liste el nombre de los scripts cargados en la memoria ROM del módulo, así como el tamaño que éstos ocupan y el espacio de memoria aún disponible. El formato de la respuesta es el siguiente:

```
#LSCRIPT: nombre1.py tamaño1
…
#LSCRRIPT: free bytes tamaño
OK
```

Donde se produce una respuesta *#LSCRIPT* por cada script cargado indicando el tamaño que éste ocupa. Finalmente se indica el número de bytes aún disponibles en memoria. Para indicar que no se produjo ningún error se manda la sentencia *OK*.

b) ***AT#RSCRIPT=<nombre>*** es el comando empleado para leer el contenido del script cuyo nombre es pasado como parámetro de esta función. Se muestra el contenido del archivo después de los caracteres <<<.

c) ***AT#WSCRIPT=<nombre>,<tamaño>[,<oculto>]*** comando empleado para indicar al módulo GSM que se quiere cargar un script en su memoria interna. Se indica el nombre del script, su tamaño en bytes (para consultar este dato de forma fácil en Windows 7 presionamos sobre el archivo con el botón derecho, y sobre la opción de propiedades veremos el tamaño, no debemos confundir este valor con el tamaño que ocupa el archivo en el disco) y, opcionalmente, se añade un parámetro denominado oculto. El parámetro oculto puede valer *0* o *1*, permitiendo la lectura del texto de este script con el comando *#RSCRIPT* o manteniendo oculto dicho texto, respectivamente.

Una vez ejecutado este comando y cuando recibamos los caracteres >>> en respuesta, pasaremos a enviar el archivo como un archivo de texto mediante el Hyperterminal. Si no hubo errores recibiremos la respuesta *OK* por parte del módulo. Podremos cargar con este comando, tanto scripts de Python como scripts precompilados de Python. Cargar un script precompilado hará que el módulo ahorre tiempo en el inicio de la ejecución de dicho script, tiempo que será tanto mayor cuanto mayor sea el tamaño del script.

d) ***AT#ESCRIPT=<nombre>*****,** comando empleado para indicar al módulo cual será el script que debe correr en el interprete de Python. Con esta opción, el script cargado en el módulo será ejecutado en cada encendido, siempre que la línea DTR del protocolo RS232 esté a nivel bajo, o lo que es lo mismo, que no haya interfaz de comandos AT conectada al puerto serie principal. La respuesta, por parte del módulo GSM, a este comando será la sentencia *OK* si no se produjeron errores, en caso contrario la respuesta será el texto *ERROR*.

e) ***AT#EXECSR*** con este comando el módulo pasará a ejecutar el script previamente habilitado con el comando *#ESCRIPT*. El tiempo transcurrido desde el envío de este comando y la ejecución del script dependerá de si hemos cargado un script precompilado o no; si hemos cargado un script precompilado, el intérprete de Python no tendrá que analizar dicho script, por lo que este tiempo se verá reducido al mínimo. En caso contrario, tras el envío de este comando AT propietario, el intérprete de Python empleará tiempo en anali-

zar dicho script antes de su ejecución; tiempo que será proporcional al tamaño del script (largos scripts no precompilados implicarán mayor tiempo de análisis por parte del intérprete).

f) ***AT#STARTMODESCR=<modo>[,<time_out>]*** este comando es empleado para selección del modo de ejecución del script en cada encendido del módulo. Si el modo es puesto a *1*, el script, previamente habilitado, se ejecuta en cada encendido del módulo, a no ser que el usuario envíe algún comando AT (por el puerto serie principal del módulo) antes de que se venza el tiempo fijado por el parámetro *time_out*, cuyo valor por defecto es de 10 segundos.

g) ***AT#DSCRIPT=<nombre>,*** este comando es el empleado para borrar el script, cuyo nombre es pasado como parámetro, de la memoria no volátil del motor de Python. Si se consigue borrar dicho script el módulo responderá con la sentencia *OK* a este comando, en caso contrario, la respuesta será *ERROR*.

Según el fabricante, la extensión Easy Script del módulo GE910 dispone de 2 MB de memoria no volátil, disponible para cargar nuestros scripts y ficheros de datos, así como 2 MB de memoria RAM para el programa que se ejecute.

El intérprete de Python que posee la familia GE910 de Telit soporta la versión 2.7.2 de este lenguaje e incorpora una serie de interfaces que serán accesibles en nuestros scripts como paquetes que debemos importar (al igual que el resto de librerías/paquetes que empleemos). Estas interfaces son la MDM, la MDM2, la SER, la GPIO y la GPS; las cuales permiten al script comunicarse con todos los recursos del módulo GSM (la última interfaz no estará disponible en nuestro EVK2, puesto que éste no posee GPS).A continuación se pasan a detallar las interfaces que incorpora éste intérprete:

1.- **Interfaz MDM**: es la que permite a nuestro script enviar comandos AT al módulo GSM. Es un puente software entre nuestro script y el motor de manejo de comandos AT interno del módulo GSM.

2.- **Interfaz MDM2**: es una interfaz secundaria con el motor de manejo de comandos AT del módulo. Puede emplearse cuando la principal esté ocupada.

3.- **Interfaz SER**: permite al script comunicarse (leer y escribir) en el puerto serie principal del módulo, ya que éste no es empleado para mandar comandos AT, al estar en uso la extensión Easy Script. Este puerto podría ser empleado para manejo de algún otro dispositivo (como podría ser un GPS externo). En nuestro caso, usaremos dicho puerto en nuestro EVK2 para tareas de 'debugging' de nuestro script. Para ello, una vez llamemos al comando #EXECSR, emplearemos el programa Hyperterminal para monitorización de dicho puerto serie (y así ver lo que el script hace que escriba el módulo en dicho puerto).

4.- **Interfaz GPIO**: la cual permite a nuestro programa acceder a las entradas y salidas de propósito general (GPIO) presentes en el módulo GSM de una forma más rápida que se haría por medio de comandos AT. El módulo GE910 posee 10 puertos GPIO.

5.- **Interfaz GPS**: la cual permitirá al código Python acceder al controlador GPS (si es que el módulo posee uno).

Cada una de estas interfaces posee sus propios métodos o funciones, los cuales nos permitirán gestionarlas de forma sencilla. Los métodos se verán con mayor profundidad en el punto siguiente, correspondiente al desarrollo software. No obstante, veremos un pequeño script a modo de ejemplo, en el cual se emplearán algunas de éstas funciones de alguno de los paquetes (interfaces) comentados arriba.

A continuación procederemos a cargar un simple Script para testear alguna de las interfaces narradas y los comandos AT de la extensión Easy Script. Para ello, en primer lugar, escribiremos un Script mediante un editor de Python (en nuestro caso PythonWin) teniendo previamente instalado el software Python de la versión 2.7.2.

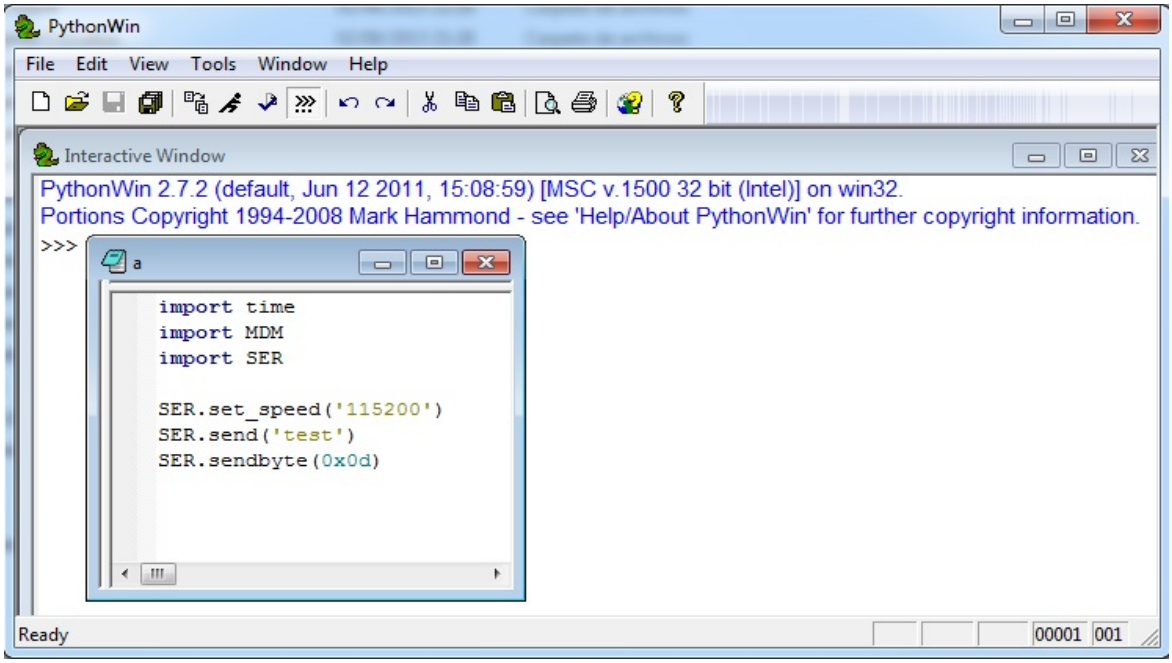


Figura 4.8: Primer Script en Python escrito con PythonWin.

En la imagen anterior puede apreciarse el texto del script que cargaremos en el módulo GSM mediante el EVK2. Podemos observar que se han importado tres paquetes: SER, MDM y time; de los cuales sólo emplearemos aquí la interfaz de comunicación serial SER. Si se produce un fallo a la hora de importar estas librerías no veríamos la palabra test en el puerto serie (si escribimos caracteres en el puerto serie puede que tampoco recibamos esta palabra, puesto que no hemos empleado el control de flujo para asegurar la transmisión). Las tres últimas líneas del script sirven para enviar la palabra test seguida de un retorno de carro a una velocidad de transmisión adecuada, a través del puerto serie. La función `set_speed()` nos permite fijar el régimen binario de transmisión del puerto serie, para ello recibe éste parámetro como argumento. Las funciones `send()` y `sendbyte()` (también disponibles en las interfaces MDM y MDM2 salvando la diferencia de que necesitan 2 argumentos) permiten enviar una cadena y un byte de texto ASCII, respectivamente, a través del puerto serie del GE910. Puede observarse que se anteponen los caracteres 0x al byte que tiene como argumento la función `sendbyte()` para indicar que éste está escrito en forma hexadecimal. En una tabla de caracteres ASCII, puede observarse que el carácter hexadecimal 0d representa el retorno de carro.

Mostramos a continuación una imagen con la secuencia de comandos seguidos para probar los comandos AT de la extensión Easy Script y cargar el script, comentado en el párrafo anterior, en el módulo GSM del EVK2.

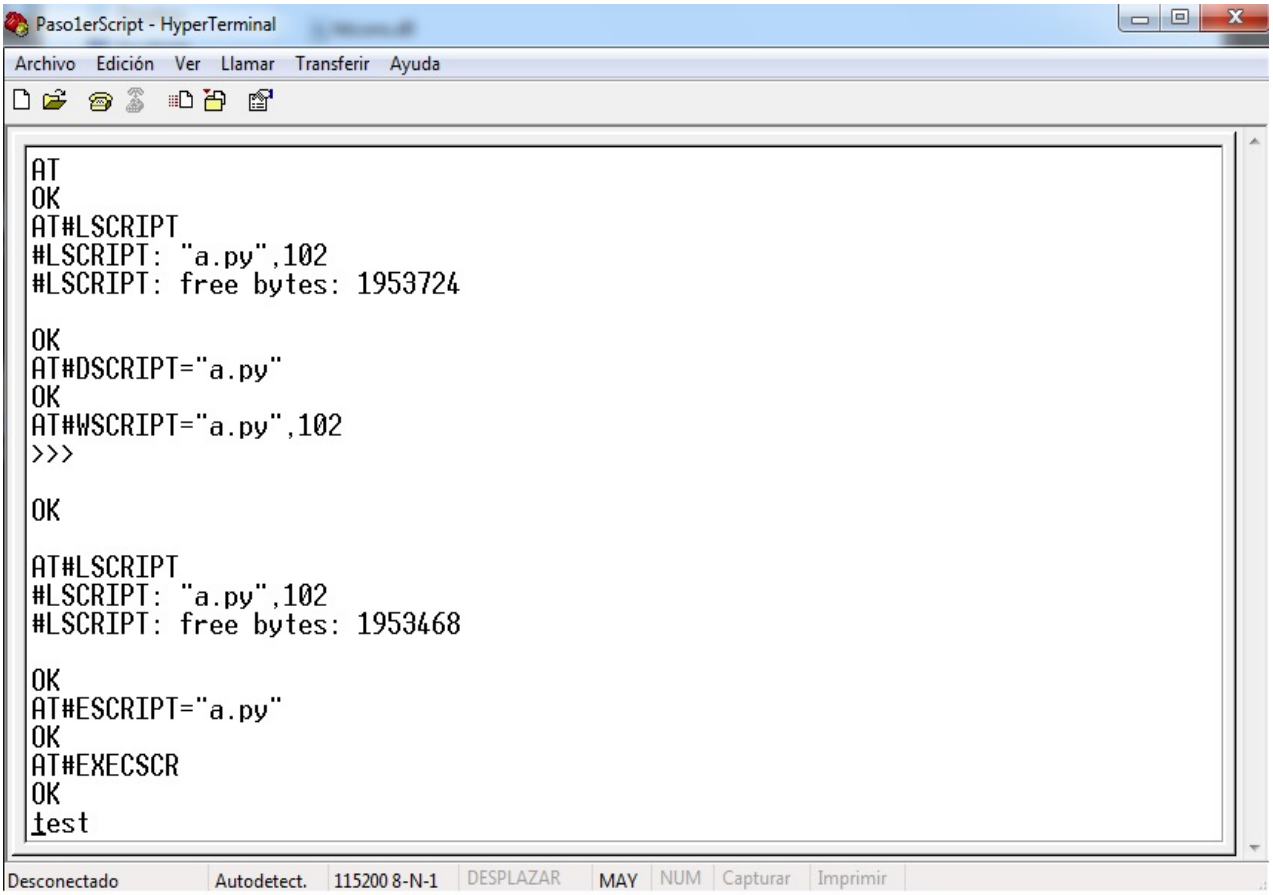


Figura 4.9: Secuencia de comandos AT empleados para probar la extensión Easy Script.

Lo primero que debemos hacer siempre es enviar el comando AT y observar la respuesta OK. Lo siguiente que hacemos aquí es emplear el comando *#LSCRIPT* para observar en la respuesta los nombres de los scripts que están cargados en el módulo GSM. Podemos ver que sólo hay un archivo, llamado a.py que ocupa 102 bytes (de hecho es el mismo archivo que queremos cargar, sólo que lo habíamos cargado previamente en alguna de las pruebas realizadas mientras se escribían éstas líneas). Lo siguiente que hacemos es borrar dicho script con el comando *#DSCRIPT*. Ahora, procedemos a cargar el script, con el comando *#WSCRIPT*. Una vez observamos los caracteres >>> en respuesta al comando *#WSCRIPT*, procedemos a enviar el archivo pinchando con el ratón sobre la opción Transferir del panel de opciones de Hyperterminal, eligiendo la acción de transferir archivo de texto. Podemos observar que en este proceso se crean lagunas de memoria en nuestro módulo, puesto que la respuesta al segundo comando *#LSCRIPT* muestra una situación idéntica a la inicial, sólo que con 256 bytes menos de espacio libre; no obstante este problema se resuelve tras apagarse la placa. Con el comando *#EXECSCR* ejecutaremos el Script, previamente habilitado con el comando *#ESCRIPT*.

Cabe destacar que la palabra *test* es el resultado de una correcta ejecución del script (observar que el retorno de carro deja el cursor en la primera posición de la línea). Al hacer esta prueba hemos observado que el tiempo que pasa desde que observamos la respuesta *OK* al comando *#EXECSCR* y que se ejecuta este simple script es de unos 6 segundos, por lo que consideramos a partir de ahora cargaremos scripts precompilados, para reducir este retardo al mínimo posible.

Como último, comentar que para que el script se transmita correctamente tras recibir la respuesta >>> al comando *#WSCRIPT*, debemos configurar adecuadamente la transmisión de

caracteres ASCII de Hyperterminal (pulsamos con el ratón sobre la opción Archivo, eligiendo Propiedades. En la ventana que aparece seleccionamos la pestaña de configuración y pulsamos sobre configuración ASCII) como muestra la siguiente figura:

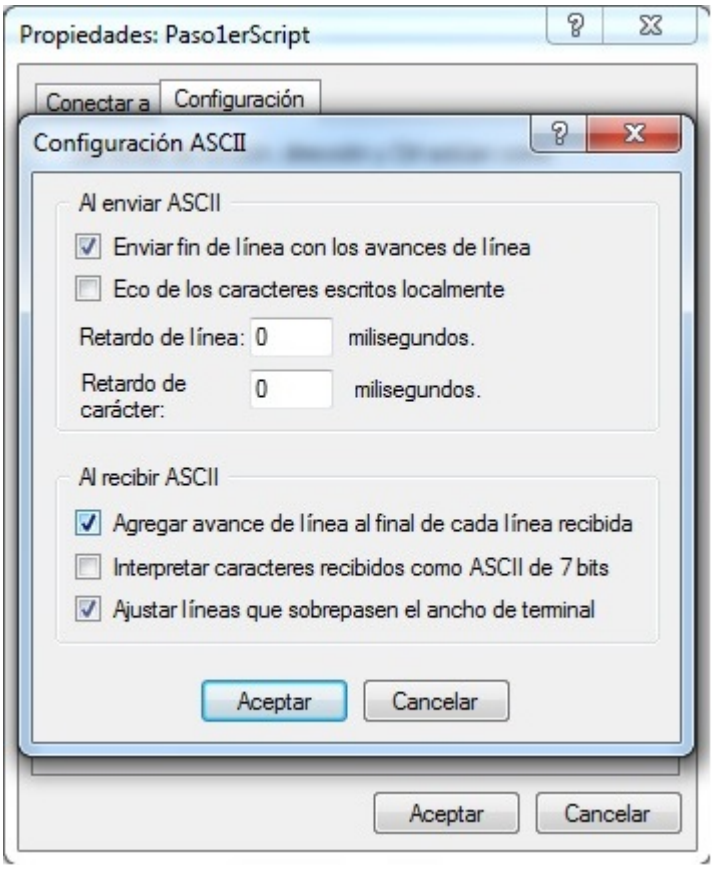


Figura 4.10: Configuración ACII en Hyperterminal para una correcta transferencia de archivos de texto.

4.4 Desarrollo Software

Gran parte del desarrollo software de nuestro dispositivo puede probarse en el EVK2, por lo que hemos decidido pasar a esta tarea, tras haber tenido un primer contacto con el kit de evaluación.

A partir de las interfaces comentadas en el aparatado de programación del módulo GSM del punto anterior, y de los paquetes que necesitemos de la API de la versión 2.7.2 de Python, desarrollaremos nuestro script. Realizaremos un programa, que salvo la funcionalidad GPS, cumpla con los objetivos del proyecto. De esta manera, cuando tengamos nuestro propio diseño terminado sólo tendremos que preocuparnos de la parte de código encargada de la gestión y manejo del receptor GNSS embebido en el GE910-GNSS.

Al no disponer de la funcionalidad GNSS, diseñaremos un programa que en primer lugar registre nuestro terminal a una red móvil. Una vez logrado el registro, pasaremos a obtener y almacenar la información de la celda de registro. Procedemos, ahora, a enviar dicha información mediante un mensaje de texto. Hecho esto, esperaremos una cantidad de tiempo determinada, para después obtener la información de las celdas vecinas (la cual, también, será enviada por medio de uno o varios SMS). El hecho de esperar un tiempo antes de pasar a obtener la información de las celdas vecinas nos va a permitir simular que nuestro dispositivo GPS tarda demasiado en ofrecernos unas coordenadas. Esto se puede implementar fácilmente con una función que simplemente espere cierto tiempo (timeout) cuando sea llamada. De esta manera, una vez que ya podamos programar la parte de código encargada de la gestión y manejo del GPS, simplemente podremos añadir la opción de encontrar unas coordenadas (válidas o no) antes de que se produzca el timeout. También tendremos que crear otra función para control del dispositivo GPS embebido para el caso en que la primera función falle, para que sea llamada tras encontrar la información de las celdas vecinas (así no volveremos a mandar la información de las celdas vecinas aunque no encontremos coordenadas válidas en esta segunda exploración) .

4.4.1 Estructura del Código

Dividiremos el cuerpo de nuestro programa en dos partes. Una primera parte estará formada por los métodos desarrollados, mientras que la segunda parte contendrá el código de la rutina principal. Esta rutina principal se encargará de llamar y procesar las respuestas de los diferentes métodos empleados.

4.4.2 Depuración del código

Gracias a la estructura del código empleada, el método para desarrollar nuestro programa se basará en la prueba-error al ejecutarse dentro del GE910 del EVK2. No podremos probar el programa en nuestro PC puesto que no hay disponible un código que nos permita simular las interfaces que están disponibles en el módulo (MDM, SER, etc.…). Aquellas partes de código que si puedan ser probadas en nuestro PC, tales como el manejo de ‘strings’, serán probadas en el PC con la herramienta PythonWin. Iremos programando una pequeña porción de código y no pasaremos a programar el código siguiente hasta saber que este pequeño trozo de código funciona. Cuando tengamos una porción de código lista para probar, adjuntaremos dichas líneas de código a nuestro script, el cual cargaremos en la memoria del módulo. Con el comando `AT#EXECSCR` pasaremos a ejecutar el script, el cual previamente habremos habilitado con el comando `AT#ESCRIPT`.

Para detectar posibles errores en el código cuando éste esté corriendo en el módulo GE910-

QUAD, emplearemos el puerto serie del módulo. Podremos acceder a dicho puerto desde nuestro script mediante la interfaz SER, y para ello diseñaremos dos funciones encargada del envío de caracteres por el puerto serie.

```
def pprint(cadena):
    SER.send(str(cadena))
    SER.sendbyte(0x0a)
    SER.sendbyte(0x0d)

def ppprint(cadena):
    SER.send(str(cadena))
```

Figura 4.11: Código de las funciones para depuración.

En la imagen anterior puede observarse que estas funciones recibirán como parámetro la cadena de texto que deberán imprimir. La diferencia entre una y otra función será el envío o no de los caracteres salto de línea y retorno de carro, es decir, que una función imprimirá una cadena y pasará a la siguiente línea de texto, y la otra no. En una tabla de caracteres ASCII puede comprobarse que el código del salto de línea es 0A (en notación hexadecimal) y el del retorno de carro es 0D. La función que imprime la cadena pero no realiza el cambio de línea será empleada para escribir cadenas en la misma línea de texto.

Para poder observar los caracteres que el script nos mande a través de su puerto serie, emplearemos el programa Hyperterminal, con la configuración ya detallada en el punto anterior de esta memoria. No deberemos intentar enviar caracteres desde nuestro PC al EVK2, puesto que al no haber implementado el control de flujo en nuestro programa, podríamos perder algunos de los caracteres que nos envíe el script; no obstante, con el script corriendo no tiene utilidad ninguna enviar caracteres al módulo, a través de su puerto serie.

4.4.3 Control del tiempo

El tiempo será un parámetro fundamental en nuestro diseño. La naturaleza de nuestro proyecto exige que el aviso de emergencia se produzca en el menor tiempo posible. Por otro lado, la comunicación con el módulo GSM a través de comandos AT exige que se vigilen cuidadosamente los tiempos, para evitar, por ejemplo, esperar un tiempo excesivo para una respuesta (la cual sobrepasado este tiempo límite seguramente no llegue, o en caso contrario, contenga errores).

Para el control del tiempo emplearemos una variable global, denominada tiempo, y una función que será usada para capturar en la variable tiempo el instante actual. También haremos uso del método clock() de la librería time; método que devuelve el tiempo, en segundos, que lleva ejecutándose el script que lo invoque.

```
def setTime():
    global tiempo
    tiempo=time.clock()
```

Figura 4.12: Código de la función responsable de la captura del instante actual.

Sobre estas líneas puede observarse el código de la función que recoge en la variable global `tiempo` el instante actual de ejecución. Para saber cuánto tiempo lleva el programa ejecutándose desde la última llamada a `setTime()`; simplemente restaremos al valor de retorno del método `time.clock()` el valor de la variable global `tiempo`. Gracias a esto, podremos saber cuánto tiempo toma la ejecución de cualquier trozo de código comprendido entre la llamada al método

`setTime()` y la resta antes citada.

4.4.4 Inicialización del módulo

Esta será la parte del código encargada de conseguir que nuestro terminal quede registrado a una estación base de la red móvil.

A diferencia de la rutina de inicialización del prototipo, emplearemos el mínimo número de comandos posibles para conseguir el registro a una red. De esta manera, ahorraremos tiempo, y evitaremos posibles errores en la respuesta a comandos innecesarios. Ya no serán enviados los comandos: `ATE0`, `AT#SIMDET`, `AT+CPIN?` ni `AT+IPR=9600`.

En cuanto a la estructura de esta parte del código, hemos decidido crear dos funciones para conseguir los objetivos de este apartado. Una de estas funciones será la encargada del envío de unos u otros comandos AT, en función de una variable de estado, a través de la interfaz MDM. La otra función será la encargada de recoger e interpretar las respuestas del módulo (también a través de MDM) para actualizar la variable de estado (por ejemplo, si recibimos la respuesta `OK` al comando `AT`, podremos pasar al siguiente estado).

```
def respInicGSM():
    global estado
    global fallosRegistro
    global tiempo
    c=-1
    k=0
    setTime()
    while c==-1:
        c=MDM.readbyte()
        if time.clock()-tiempo > 10:
            fallosRegistro=fallosRegistro+1
            pprint('timeout_respInic')
            return
    d=MDM.read()
    if 'OK' in d:
        pprint('OK')
        if estado==1 or estado==2 or estado==4:
            estado=estado+1
        elif '0,1' in d:
            pprint('0,1')
            if estado==3:
                estado=4
        elif '0,5' in d:
            pprint('0,5')
            if estado==3:
                estado=4
        elif '0,0' in d and estado==3:
            pprint('0,0')
            estado=1
    elif 'ERROR' in d and estado!=2:
        pprint('ERROR')
        fallosRegistro=fallosRegistro+1
    elif 'ERROR' in d and estado==2:
        pprint('ERROR')
        estado=0
```

```
def inic_gsm():
    global estado
    global fallosRegistro
    a=-1
    while(estado<5):
        if fallosRegistro > 5:
            estado=1
            pprint('reboot')
            restart_program()
        if estado==1:
            while a==-1:
                a=MDM.send('AT\r\n',1)
                pprint('AT\n')
            a=-1
            time.sleep(0.005)
            respInicGSM()
        elif estado==2:
            while a==-1:
                a=MDM.send('AT+CPIN=1096\r\n',1)
                pprint('AT+CPIN=1096\r\n')
            a=-1
            time.sleep(0.005)
            respInicGSM()
        elif estado==3:
            while a==-1:
                a=MDM.send('AT+CREG?\r\n',1)
                pprint('AT+CREG?\n')
            a=-1
            time.sleep(2)
            respInicGSM()
        elif estado==4:
            while a==-1:
                a=MDM.send('AT+CMGF=1\r\n',1)
                pprint('AT+CMGF=1\n')
            a=-1
            time.sleep(1)
            respInicGSM()
        elif estado==5:
            return estado
        else:
            return 0
    return estado
```

Figura 4.13: Código de las funciones de inicialización del módulo GE910.

Aquí podemos observar las dos funciones, ya citadas. En el código de la función `inic_gsm()`, encargada de enviar los comandos AT necesarios para el registro de nuestro terminal a la red, podemos observar que se emplea el estado y el número de fallos de registro como variables globales. De esta forma estas variables serán accesibles y modificables desde esta función. En la función `respInicGSM()` también se emplean estas mismas variables globales, además de otra, denominada `tiempo`, que será empleada para controlar que las respuestas de nuestro módulo a los comandos AT no sobrepasen el timeout indicado por el fabricante.

En la función `inic_gsm()` podemos observar que se ejecutará el bucle while (que engloba casi todo el cuerpo ejecutable de la función) mientras el estado sea inferior a 5 (a excepción del caso en el cual el estado valga 0, lo cual significará que se deberá abortar el programa). Por lo tanto, lo que definirá la tarea a realizar en cada instante, por parte de la función `inic_gsm()`, será la variable global `estado`. El estado puede tomar 6 posibles valores:

1.- Estado igual a 1: en esta situación el módulo está recién encendido, y lo primero que debemos hacer es enviar al módulo el comando `AT`. Si el módulo está listo responderá con el texto `OK` a este comando. En caso de que el módulo no estuviese listo podríamos recibir la respuesta `ERROR` o ver superado el tiempo límite de respuesta (timeout). Si la función `respInicGSM()` detecta que el módulo está listo, hará que la variable global `estado` pase a valer 2; en caso contrario, ya se produzca un timeout o se reciba el texto `ERROR`, aumentaremos en una unidad la variable global `fallosRegistro`.

2.- Estado igual a 2: en esta situación el módulo está listo para recibir e interpretar comandos AT. No obstante, para poder registrarnos en una red, debemos desbloquear el acceso a nuestra tarjeta SIM mediante la introducción de su código PIN. Esto lo haremos mediante el comando AT+CPIN=1096. Sabemos que el código PIN de nuestra tarjeta es 1096, por lo que no se producirá nunca un fallo de introducción del código (si se produjese un fallo abortaríamos la ejecución del programa mediante la asignación del valor 0 a la variable `estado`). Lo que sí podría pasar es que el módulo exceda el timeout de respuesta a este comando, y en tal caso la función `respInicGSM()` aumentará en una unidad la variable global `fallosRegistro` y retronará a la función sin alterar el valor de `estado`. Si todo transcurre correctamente la función `respInicGSM()` observará el texto `OK` en la respuesta del módulo y hará que `estado` pase a tomar el valor 3.

3.- Estado igual a 3: en esta situación el módulo está preparado para poder registrarse a una estación base de la red. El registro de red se hace de forma automática sin la necesidad del envío de comandos AT. A pesar de ello, sólo pasaremos al estado siguiente en caso de estar seguros que estamos registrados en una BTS; para ello el método `respInicGSM()` deberá observar la respuesta `0,1` o `0,5` al comando `AT+CREG?` previamente enviado por la función `inic_gsm()`. En caso de no estar registrados a una red recibiremos la respuesta OK si, a pesar de no estar registrados, todo va bien. Si se produjo un fallo a la hora de intentar registrarnos a la red la respuesta observada será 0,0 y la función `respInicGSM()` hará que el estado pase a ser el inicial (estado 1).

4.- Estado igual a 4: a estas alturas el módulo ya está registrado en una red, y simplemente debemos indicar el formato de texto empleado a la hora de enviar SMS. Esto se hará por medio del comando `AT+CMGF=1`, a lo que el módulo responderá con el texto `OK`, en caso de que todo vaya bien, y podremos pasar al quinto estado. En caso de fallo se podrá producir un timeout o recibir el texto `ERROR`, y la función `respInicGSM()` no alterará el valor de `estado` y aumentará en una unidad la variable `fallosRegistro`.

5.- Estado igual a 5: llegados aquí la función `inic_gsm()` nos retorna a la rutina principal, indicando que el valor de estado es igual a 5.

6.- Estado igual a 0: este estado indica que el programa debe abortarse, ya que se produjo un fallo a la hora de introducir el código PIN (segundo estado). No llegaremos nunca a este estado, a no ser que a la hora de programar el código de este programa nos equivoquemos en el código PIN. Con una simple ejecución del programa sabremos si el PIN escrito es el correcto, y ya no deberemos preocuparnos de llegar a este estado con el dispositivo que emplee esta tarjeta SIM.

Como ha podido observarse, en el cuerpo de la función `inic_gsm()`, el envío de los comandos AT se hace por medio de la función `MDM.send()`. Esta función envía a la interfaz `MDM` una cadena de caracteres, la cual toma como primer argumento; su segundo argumento será un número que denota la cantidad de décimas de segundo que puede esperar como mucho para enviar la cadena de caracteres a la interfaz. Esta función terminará cuando se haya podido enviar a la interfaz `MDM` la secuencia de caracteres, o cuando se haya sobrepasado el tiempo límite para hacerlo. El valor que retornará esta función será `-1` en caso de que el tiempo máximo haya expirado, o `1` en caso contrario. Es por este motivo que se ejecuta dentro de un bucle `while`, que comprueba que el valor de retorno sea distinto de `-1`, y en caso contrario sigue llamando a dicha función hasta obtener éxito. Los mismos caracteres que se envían a la interfaz `MDM`, son enviados a la interfaz `SER` (puerto serie del EVK2 monitoreado con el Hyperterminal) para observar la sucesión de estados.

Cabe destacar que en las múltiples ejecuciones de este código aún no se ha observado que la función `MDM.send()` sea llamada dos veces con los mismos argumentos (caso en que el tiempo límite expira) y, de hecho, lo mismo hemos observado en los múltiples métodos implementados que hacen uso de dicha función. No obstante, haremos uso del valor de retorno de esta función por prevención, ya que no queremos, debido a la naturaleza de este proyecto, que nuestra aplicación falle bajo ningún concepto.

La responsable de la modificación del valor de la variable estado es la función `respInicGSM()`, la cual es llamada cada vez que se envíe con éxito un comando (secuencia de caracteres específica) a la interfaz `MDM`. Esta función permanece a la espera, vigilando que no se sobrepase un tiempo límite de respuesta, de recibir el primer carácter de respuesta de la interfaz `MDM`. Una vez leído este primer carácter se recogen el resto de los caracteres en una variable de tipo string llamada `d`.

Como hemos citado en el párrafo anterior, esta función vigila que no se sobrepase un tiempo límite (10 segundos) hasta recibir una respuesta por parte del módulo a través de la interfaz MDM. Entenderemos el recibir respuesta como que se nos presente un primer carácter a través de MDM. Si se sobrepasase el tiempo de respuesta, la función `respInicGSM()` informará a través de la interfaz SER, y retornará a la función `inic_gsm()` sin modificar la variable de estado , pero aumentando en una unidad el número de fallos de registro. La función `MDM.readbyte()` es la que nos permite saber si hay un carácter presente en el buffer de respuesta de `MDM` . Esta función recoge dicho carácter en la interfaz y lo devuelve. Si no se encontrase ningún carácter devolvería el valor `-1`. Por lo tanto, lo implementado en las primeras líneas de código de la función es el mantenerse a la espera de encontrar que la función `MDM.readbyte()` devuelva un valor distinto a `-1`, siempre y cuando ésta lo haga antes de que pasen 10 segundos.

Una vez la función `respInicGSM()` tiene almacenada la cadena de texto de la respuesta en la variable `d` pasaremos a interpretar el texto de dicha respuesta. Dependiendo del texto de esta respuesta y teniendo en cuenta el estado actual, variaremos o no el valor de las variables `estado` y

`fallosRegistro`. A continuación exponemos las posibilidades de respuestas:

1.- La respuesta incluye el texto OK: en este caso esta respuesta nos permitirá avanzar (aumentar en una unidad el valor de la variable estado) dependiendo de varias situaciones. Esta respuesta permite avanzar directamente si estamos en el estado 1, 2 y 4. En el caso de estar en el estado 3, sólo avanzaremos si además se incluye el texto '`0,1`' ó '`0,5`' en la respuesta. Si se encontrase el texto '`0,0`' aumentaremos en una unidad el número de fallos (variable fallosRegistro) y haremos que la variable estado vuelva a tener el valor 1 (el texto '`0,0`' indicaría que el terminal aún no está registrado y tampoco lo va a intentar).

2.- La respuesta incluye el texto ERROR: esta respuesta, en general, mantendrá la variable estado sin alterar e incrementará el contador de fallos de registro; a excepción de que nos encontremos dicha respuesta estando en el estado 2 (estado en el que previamente introducimos el código PIN de la tarjeta SIM). En tal caso, habríamos escrito mal el código PIN en el cuerpo de nuestro programa, y para evitar que el script agote los 3 intentos que tenemos abortaríamos el programa. En este caso hacemos que la variable estado tome el valor cero, y al volver a la función `inic_gsm()` ésta retornaría el valor `0` (si desde el programa principal detectamos que la función `inic_gsm()` devuelve el valor 0 deberemos informar al usuario a través de `SER` y detener la ejecución del script, o mantener el programa inactivo en un bucle infinito de código).

4.4.5 Obtención de información de celdas

Para estos propósitos se han desarrollado dos funciones, la primera de las cuales servirá para obtener la información de la celda de registro, mientras que la segunda será empleada para conseguir la información de las celdas vecinas.

Veremos, únicamente aquí, el código de la función encargada de obtener la información de la celda de registro. Esto se debe a que la otra función resulta similar, pero mucho más extensa.

```
def obtenerInfoCell():
    global tiempo
    a=-1
    while a==-1:
        a=MDM.send('AT#MONI\r\n',1)
        pprint('AT#MONI\r\n')
    primerByte=-1
    setTime()
    while primerByte==-1:
        primerByte=MDM.readbyte()
        if time.clock()-tiempo > 20:
            pprint('timeout moni')
            return ()
    pprint('INFO:')
    info=MDM.read()
    if 'ERROR' in info:
        pprint('error moni')
        return ()
    pprint(info)
    inferior=info.find(' LAC:')+5
    superior=info.find(' Id:')
    LAC=info[inferior:superior]
    CELLID=info[info.find(' Id:')+4:info.find(' ARFCN:')]
    PWR=info[info.find(' PWR:')+5:info.find(' TA:')]
    TA=info[info.find(' TA:')+4:info.find(' TA:')+5]
    if '#MONI: Cc:' in info:
        MCC=info[info.find(' Cc:')+4:info.find(' Cc:')+7]
        MNC=info[info.find('Nc:')+3:info.find('Nc:')+5]
        informacion=(MCC, MNC, LAC, CELLID, PWR, TA)
    else:
        nombre=info[info.find('#MONI: ')+7:info.find('BSIC:')]
        informacion=(nombre, LAC, CELLID, PWR, TA)
    return informacion
```

Figura 4.14: Código de la función para obtención de información de la celda de registro.

Podemos observar que en primer lugar nos aseguramos de que se envíe el comando `AT#MONI` a través de la interfaz `MDM`. Una vez hecho esto esperamos a recibir el primer byte de la respuesta (de la misma forma que en la función `respInicGSM()`) vigilando que no expire un tiempo límite de 20 segundos. Si el tiempo expira devolvemos una lista vacía. Lo siguiente será almacenar la respuesta completa (a excepción del primer carácter) en la cadena `info`. Una vez tenemos la respuesta, debemos procesarla. Lo primero será comprobar que el texto `ERROR` no se encuentre en la respuesta. En caso afirmativo, se avisa del error a través de la interfaz `SER`, y se termina la ejecución de la función, devolviendo una lista vacía. En caso contrario, y gracias a que conocemos la estructura de una respuesta correcta, pasamos en primer lugar a almacenar el LAC, CellId, la potencia de señal recibida (PWR) y el valor del TA mediante el método `find()` de la clase String. Este método devuelve el índice dentro de la cadena de la primera ocurrencia del texto que recibe como argumento. Con este método conseguiremos indexar correctamente la cadena `info` para extraer los parámetros antes citados. No sabemos si en la respuesta al comando `AT#MONI` se nos presenta el nombre de la red o los valores MCC y MNC, pero para distinguir entre ambas situaciones nos valdremos del hecho de que, en caso de que se nos presenten los valores MCC y MNC, la respuesta contendrá el texto '`MONI: Cc:`'.

Esta función siempre devuelve una lista. Esta lista estará vacía en caso de que haya ocurrido un error. Podremos distinguir si la lista contiene el nombre de la red o los valores MCC y MNC o saber si el método `obtenerInfoCell()` falló, gracias al método `len()`. Este método devuelve el número de elementos contenidos en el argumento que recibe. En caso de que la lista contenga el nombre de red el número de elementos serán 5, si esta lista contuviese el MNC y el MCC el número de elementos serían 6, mientras que si el número de elementos es 0 (lista vacía) sabremos que se produjo un error.

4.4.6 Envío de mensajes

Para el envío de mensajes hemos decidido desarrollar un único método, denominado `mandarSms()`, el cual recibe como argumento la cadena de texto que aparecerá como cuerpo del SMS.

```
def mandarSms(cadena):
    global tiempo
    envioOk=-1
    while envioOk==-1:
        envioOk=MDM.send('AT+CMGS=0034609381404,129\r\n',1)
        pprint('AT+CMGS=0034609381404,129\r\n')
    respuesta=''
    setTime()
    while len(respuesta)==0:
        respuesta=MDM.read()
        if time.clock()-tiempo > 10:
            pprint('timeout1 en cmgs')
            return -1
    if '>' in respuesta:
        pprint('INFO:')
        envioOk=-1
        while envioOk==-1:
            envioOk=MDM.send(str(cadena),1)
            pprint(cadena)
        envioOk=-1
        while envioOk==-1:
            envioOk=MDM.sendbyte(0x1a, 1)
        respuesta=''
        setTime()
        while len(respuesta)==0:
            respuesta=MDM.read()
            if time.clock()-tiempo > 10:
                pprint('timeout2 en cmgs')
                return -1
        if 'OK' in respuesta:
            pprint('Resp Envio OK')
            return 1
        else:
            pprint('Resp Envio ERROR')
            return 0
    else:
        pprint('CMGS da error')
        return 0
```

Figura 4.15: Código de la función encargada de mandar SMS.

Como ya vimos en el capítulo anterior, para mandar un mensaje de texto emplearemos el comando `AT+CMGS` seguido del número de teléfono y el formato del mismo (precedidos por un signo igual y separados mediante una coma). Una vez enviemos dicho comando a la interfaz `MDM`, el módulo nos responderá con uno varios signos '`>`' indicando que podemos enviar el cuerpo del SMS a la interfaz. Tras enviar el cuerpo del mensaje, para indicar a la interfaz que hemos finalizado enviamos el carácter `Ctrl+Z` (en ASCII se representa mediante la codificación hexadecimal `1A`). Si antes de diez segundos el módulo no responde con la sentencia `OK`, entenderemos que ha habido algún error. Si se sobrepasan los diez segundos sin obtener respuesta, se devolverá el valor `-1`, si la respuesta es diferente a `OK` devolverá el valor `0`. Si no hubo errores se retorna el valor `1`.

4.4.7 Indicadores lumínicos

Como ya detallamos en el apartado de experimentación con el EVK2, el módulo GE910-GNSS posee diez puertos de entrada/salida genéricos. El empleo de estos puertos como fuente de alimentación o interruptor del circuito de aviso lumínico implica tener en cuenta una serie de consideraciones eléctricas, las cuales serán detalladas en el punto siguiente de esta memoria: "Circuitos necesarios para el diseño final".

Para manejar estos puertos de salida mediante Python disponemos en el módulo del paquete `GPIO`, el cual posee una serie de funciones. Al inicio del código deberemos importar la librería `GPIO`, al igual que las librerías `SER` y `MDM` ya detalladas con anterioridad. A continuación comentaremos aquellas funciones del paquete `GPIO` empleadas en nuestro código:

En primer lugar tenemos el método ***`GPIO.setIOdir(GPIOnumber, value, direction)`*** el cual nos permite establecer el sentido (entrada o salida) de un puerto genérico. El primer argumento será el número del puerto, seguido de un valor de inicialización, el cual sólo toma sentido cuando el puerto es fijado como salida. El último argumento es un valor que indicará si el puerto será empleado como salida (valor 1) o como entrada (valor 0), función alternante (valor 2) o, finalmente, como puerto triestado (valor 3).

Por otro lado, la función ***`GPIO.setIOvalue(GPIOnumber, value)`*** nos permitirá ajustar el valor lógico de un puerto configurado como salida. El primer argumento que recibe esta función es el número de puerto, mientras que el segundo argumento es el valor lógico que deseamos asignar al puerto en el momento de ejecución de esta función.

Sabiendo esto, ya podremos diseñar un pequeño programa de prueba. Volcaremos este código en nuestro EVK2 para observar el comportamiento de los puertos GPIO. Con un voltímetro, podremos observar el voltaje de los diferentes valores lógicos de salida de los puertos, verificando por tanto, el funcionamiento del código. Para comprobarlo, debemos localizar en nuestro EVK2 un pin conectado al puerto que deseemos emplear y otro pin con el valor de referencia de 0 voltios (GND).

```
import time
import GPIO
import SER

SER.set_speed('115200')

GPIO.setIOdir(7,1,0)
time.sleep(3)
GPIO.setIOvalue(7,1)
time.sleep(3)
GPIO.setIOvalue(7,0)
time.sleep(5)
GPIO.setIOvalue(7,1)
time.sleep(5)
GPIO.setIOvalue(7,0)
SER.send('test')
SER.sendbyte(0x0d)
```

Figura 4.16: Código de prueba para manejo del GPIO nº7.

En este código podemos observar que empleamos el puerto GPIO 7 como salida, comenzando con el valor lógico '0' (led apagado) e intercambiándose durante 3 segundos a '1' tras 3 segundos. Lo siguiente será otro 'toggle' pero con periodo 10 segundos (5 segundos por ciclo).

Para analizar el voltaje de salida del puerto 7, conectaremos el polo positivo del voltímetro a este puerto, o séase, al pin 6º del array de pines PL302 de la placa interfaz GE910. El polo negativo del voltímetro lo conectaremos a un pin de tierra (por ejemplo el pin 10º del array PL102). Una vez hecho esto, podemos encender el voltímetro y pasar a ejecutar este programa. Esperamos unos instantes (ya que el código volcado no ha sido precompilado) y observamos que el comportamiento es el esperado: el voltímetro muestra un valor de cero voltios al principio (el tiempo necesario para compilar el código más tres segundos), y pasará a valer 1’8 voltios en las dos ocasiones en que la salida esté a nivel alto.

4.4.8 Cuerpo del programa final

Desde el cuerpo del programa final se gestionan y manejan todos los métodos antes comentados. Comentaremos detalladamente este código, mostrando pequeñas porciones del mismo. Estas porciones mostradas realizaran una operación concreta.

Cabe destacar que el manejo de indicadores lumínicos ha sido implementado, pero las líneas encargadas de ello han sido comentadas, puesto que decidimos no simular esta funcionalidad desde el EVK2 (ya vimos que esta característica funcionaba correctamente en el apartado anterior).

A continuación expondremos las primeras líneas del código de la rutina principal:

```
estado=1
tiempo=time.clock()
fallosRegistro=0
#Inicialización receptor GNSS
#GPIO.setIOdir(9,1,0)
#GPIO.setIOdir(8,1,0)
#GPIO.setIOdir(10,1,0)
#GPIO.setIOdir(15,1,0)
#time.sleep(0.5)
#GPIO.setIOvalue(9,1)
SER.set_speed('115200')
```

En estas primeras líneas podemos observar que se realizan los pasos previos a la llamada de la función encargada del registro de red. Estos pasos previos consisten en la inicialización de variables (en el lenguaje Python las variables no se declaran), la configuración de puertos y la puesta en marcha del dispositivo GNSS embebido, si fuese necesario. Se inicializa la variable `estado` igualándola a 1 (puesto que, como vimos anteriormente, la función encargada del registro de red se vale de una variable global denominada `estado`, la cual denota la situación del registro de red). De esta manera indicamos las condiciones iniciales del dispositivo. También se inicializa la variable global `tiempo` así como `fallosRegistro`, para que las funciones que accedan a ellas por vez primera desde este programa, las vean ya inicializadas. Así mismo, se configuran como salida todos los GPIOS necesarios para los indicadores lumínicos del montaje final. Cabe destacar que la elección de los números de puertos GPIO responde a la idea de seleccionar aquellos pads más accesibles del encapsulado LGA del módulo GE910-GNSS (en general, lo más separados posible de pads empleados y lo más cerca del borde del LGA). Una vez configurados todos los GPIOS empleados, se procede a encender el primero de éstos para indicar al usuario que el dispositivo está encendido y listo para actuar. Finalmente, se configura la velocidad de tranferencia de datos del puerto serie del GE910-GNSS, que será empleada para analizar el estado del programa desde un PC y poder depurar el código mientras éste se fue desarrollando.

Llegados a este punto, lo siguiente que debe realizar el dispositivo, son las tareas necesarias para llevar a cabo el registro de red. Esto, simplemente, se realiza por medio de una llamada a la función `inic_gsm()`la cual vimos en profundidad en el apartado **4.4.4** (Inicialización del módulo). Una vez logremos el registro de red, debemos encender el segundo indicador lumínico, como puede verse a continuación:

```
res=inic_gsm()
#GPIO.setIOvalue(8,1)
```

Una vez estemos conectados (registrados) a una red, lo que debemos hacer es obtener la información de la celda de registro. La información de registro se obtiene mediante la llamada a la función `obtenerInfoCell()`. Esta función nos devolverá una lista con la información de celda en caso de éxito, o una lista vacía en el caso que surja algún error (timeout de comienzo de respuesta, error en la respuesta, etc.). Para evitar que el programa principal se quedase ejecutando indefinidas llamadas a la función `obtenerInfoCell()` se ha implementado un contador de fallos, que actúa de forma muy similar a la variable `externalTime` empleada en el código del prototipo de este proyecto. Si se realizan diez llamadas infructuosas a ésta función (cosa que sin duda denotaría un error excepcional, ya que aún no se ha registrado tal situación con el EVK2 del GE910), se reinicia el programa mediante la llamada a la función `restart_program()`. Así evitamos, de forma definitiva, posibles situaciones que impidan realizar el aviso de emergencia. La llamada a la función encargada del reinicio del programa, se realizaría antes de una nueva llamada a `obtenerInfoCell()` para evitar perder más tiempo. La información de la celda de registro es almacenada en la variable info. Con el método `pprint()` enviaremos los caracteres de esta lista por el puerto serie, para poder observar el comportamiento del programa desde un PC.

```
info=()
fallos=-1
while len(info)==0:
    fallos=fallos+1
    if fallos==10:
        restart_program()
    info=obtenerInfoCell()   #con len(info) sabemos de que tipo es
pprint(str(info))
```

Una vez se haya obtenido la información de celda, procederemos a enviar dicha información mediante un mensaje de texto. Para ello, tendremos que realizar una llamada exitosa a la función `mandarSms()`. Si esta función devuelve el valor 1 habremos enviado con éxito el mensaje de texto. Por tanto, y de forma similar a la llamada a la función `obtenerInfoCell()`, inicializaremos un contador de fallos y realizaremos hasta un máximo de 10 llamadas a `mandarSms()`. La función encargada del envío de texto, como vimos anteriormente, ha de recibir como argumento la cadena de texto del mensaje, que en este caso está almacenada en la lista `info`. Una vez obtengamos éxito al enviar el mensaje, informaríamos al usuario mediante el encendido del tercer indicador lumínico. A continuación se muestra la porción de código encargada de llevar a cabo esta tarea:

```
fallos=-1
while(mandarSms(info)!=1):
    fallos=fallos+1
    if fallos==10:
        restart_program()
    time.sleep(1)
#GPIO.setIOvalue(10,1)
```

Llegados a este punto, tendremos que conseguir las coordenadas del posicionamiento GNSS y el valor del DOP asociado a las mismas. Como vimos que en el EVK2 no disponíamos de la funcionalidad GNSS, simplemente, realizaremos un bucle que se encargará de vigilar el tiempo y el estado de la variable `HDOP` asociada al posicionamiento. La actualización de la variable `HDOP` correrá a cuenta del código encargado de obtener el poscionamiento GNSS y cuando aún no se tenga un posicionamiento disponible, esta variable tendrá un valor superior a 100 unidades.

De esta forma, podremos desarrollar casi todo el código del diseño final, a excepción de una función encargada de conseguir el posicionamiento GNSS, y el envío del mensaje de texto con dicha información (aún debemos considerar la forma en que agruparemos en una cadena de texto el

mensaje de aviso con un posicionamiento). Podremos, por tanto, desarrollar el código encargado de actuar en caso de fracaso por tiemeout y elevado HDOP durante la obtención del posicionamiento GNSS. Para estos fines se desarrolló el último bloque de código, el cual exponemos a continuación:

```
timeout = 0
fin_bucle=0
Coordinates=0
HDOP=101
lastHDOP=143
externalTime=time.clock()
while fin_bucle==0:
    #Conseguir coordenadas y HDOP
    if HDOP >= 2.5 and HDOP < 100 and HDOP <= 0.7*lastHDOP:
        lastHDOP=HDOP
        #mandar sms alto DOP
        if timeout==0:
            t=0
            fallos=-1
            while t==0:
                fallos=fallos+1
                if fallos==10:
                    restart_program()
                time.sleep(10)
                t=obtenerInfoAllCells()
            while mandarSms(t)!=1:
                time.sleep(1)
            timeout=1
    elif HDOP < 2.5:
        #mandar sms coords OK
        fin_bucle=1
    elif time.clock()-externalTime > 70 and timeout==0:
        timeout=1
        t=0
        fallos=-1
        while t==0:
            fallos=fallos+1
            if fallos==10:
                restart_program()
            time.sleep(10)
            t=obtenerInfoAllCells()
        fallos=-1
        while mandarSms(t)!=1:
            fallos=fallos+1
            if fallos==10:
                restart_program()
            time.sleep(1)
        #Fin del programa para GE910 sin GNSS
    #GPIO.setIOvalue(5,1)
    time.sleep(1)
    #GPIO.setIOvalue(5,0)
    time.sleep(1)

#GPIO.setIOvalue(5,1)
pprint('fin')
```

Podemos apreciar, que dentro del bucle `while` principal, se engloban los posibles eventos que pueden tener lugar a la hora de obtener el posicionamiento mediante el receptor GNSS. No se saldrá de este bucle a menos que se obtenga un valor adecuado del HDOP asociado al posicionamiento obtenido. Cuando esto ocurra, la variable `fin_bucle` pasará a tomar el valor 1 (hecho que nos permitirá abandonar el bucle `while`). Los posibles eventos que pueden suceder son:

a) Exceso de tiempo al obtener un posicionamiento GNSS.
b) Elevado valor del HDOP por primera vez, o mejora significativa en su valor (sin ser considerado preciso).
c) Obtención de un posicionamiento con un HDOP adecuado.

Si la primera situación tiene lugar, pasaremos a obtener toda la información de las celdas vecinas y a enviarla mediante uno o varios mensajes de texto. Para evitar el envío repetido de esta información actualizaremos una variable denominada `timeout`, la cual indicará si ya se realizó esta búsqueda, tras haber superado el tiempo de espera. Cabe destacar que para evaluar este tiempo de espera, tenemos que recurrir a una variable distinta (`externalTime`), ya que tanto el método `getInfoAllCells()` como, seguramente, el método encargado de conseguir las coordenadas y HDOP hagan uso de la variable `tiempo`.

Si se produce la segunda situación, debemos enviar un mensaje de texto con las coordenadas del posicionamiento y el valor del HDOP asociado, advirtiendo que éste último es elevado. Tras haber hecho esto, debemos evaluar si ya enviamos anteriormente la información de las celdas vecinas (comprobar si la variable `timeout` es igual a 1) para, en caso contrario, proceder a realizarlo. De forma similar a la situación anterior, nos valdremos de una variable (`alto_HDOP`) para evitar ejecutar esta parte del código más de una vez.

Ante el tercer posible evento, simplemente enviaremos las coordenadas del posicionamiento GNSS y el valor del HDOP asociado. Se dará por concluido el aviso de emergencia, y se informará de ello mediante el encendido del cuarto indicador lumínico.

Es importante destacar el orden en que aparecen los `if` y `elif` encargados de comprobar las tres diferentes situaciones. El exceso de tiempo se evalúa en último lugar, por si en ese instante preciso se dispone del posicionamiento sea enviado, y no pasemos a buscar la información de celdas vecinas.

También podemos observar que mientas se ejecute el bucle `while` principal, el último led de información lumínica permanecerá parpadeando. Si dentro del bucle se pasa a buscar la información de las celdas vecinas, este led permanecerá apagado. Una vez abandonemos este bucle `while`, se encenderá el último led.

4.5 Circuitos necesarios para hacer funcionar el módulo GSM

En este apartado hablaremos de los circuitos electrónicos necesarios para desarrollar nuestro dispositivo. El módulo GSM-GPS ha de ser alimentado, encenderse tras la pulsación de un botón, ser comunicado con una tarjeta SIM, ser capaz de comunicarse con un PC e informar al usuario del estado del aviso. Por tanto, necesitaremos crear un circuito de alimentación, otro de encendido, uno de conexión con la ranura de inserción de la tarjeta SIM , otro de comunicaciones con el PC y otro de indicadores de aviso. Finalmente tendremos que crear un esquemático con el conexionado del módulo GE910.

En el manual hardware de usuario de la familia de módulos GE910 podremos observar recomendaciones sobre casi todos estos circuitos.

4.5.1 Circuito de alimentación

En el manual Hardware de Telit sobre la familia GE910, aparece una referencia a las posibilidades que tenemos para alimentar el módulo. La tensión de alimentación recomendada ha de encontrarse entre los 3.4 y los 4.2 voltios (no debemos nunca estar fuera del rango extendido de alimentación indicado por el fabricante, el cual está entre los 3.2 y 4.5 voltios). Por tanto, una buena fuente puede ser una batería Li-Ion de 3.7 voltios. El fabricante desaconseja el empleo de otras baterías, tales como las de Ni/Ca o Ni/MH, conectadas directamente al módulo, ya que en ciertas ocasiones éstas pueden suministrar un voltaje por encima de los límites del GE910, y por tanto, dañarlo.

Para alimentar el módulo desde una batería necesitaremos, al menos, dos elementos de protección. El primero es un diodo Zener a modo de regulador, el cual evitará inversiones de tensión y aumentos elevados de la misma. El segundo es un condensador de desacoplo, que evitará picos y variaciones de tensión (actúa como filtro paso bajo). Un esquema de dicho circuito puede observarse a continuación.

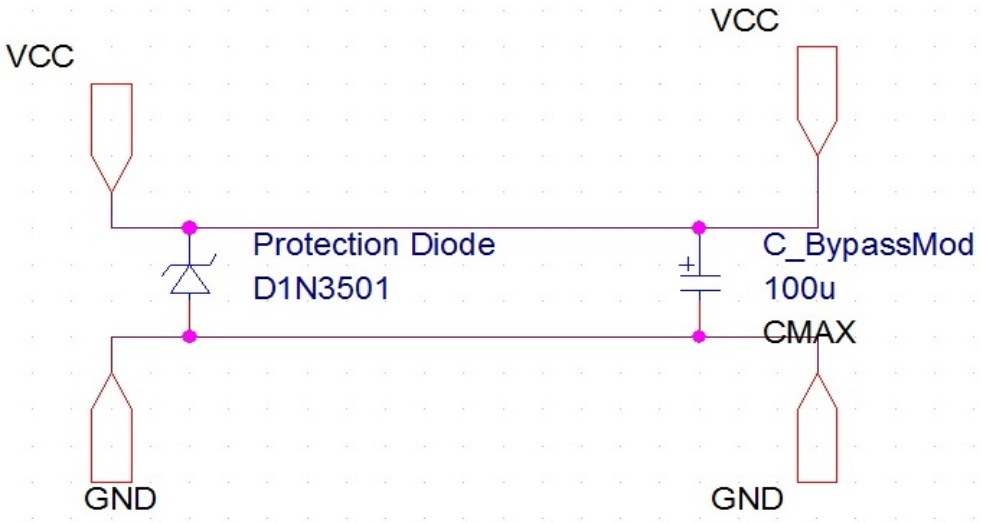


Figura 4.17: Circuito de alimentación del diseño final.

Podemos observar que el diodo sólo conducirá en dos situaciones, cuando el voltaje de polarización en inversa supere el umbral Zener, y cuando éste se polarice en directa. Si el diodo no conduce, el voltaje entregado al módulo GSM-GPS será directamente el proporcionado por la

batería (cuando un diodo no conduce, éste equivale a un circuito abierto). En caso de que el diodo conduzca, el voltaje entregado al módulo será el que cae en bornes del Zener. Si éste conduce en inversa, lo cual ocurrirá únicamente si la batería ha excedido el voltaje de salida, protegerá al módulo de dicho exceso de voltaje. El diodo conducirá en directa cuando la fuente tenga una inversión de voltaje (cambio de polaridad) e impedirá que dicho voltaje alcance directamente al módulo (puesto que el voltaje entregado a éste será aproximadamente la tensión de conducción en directa, muy cerca a los 0 voltios). El condensador de desacoplo sirve como camino para las corrientes de retorno de alta frecuencia, y filtra los picos de tensión.

Escogeremos un diodo Zener cuyo voltaje en inversa sea cercano (pero esté por debajo) del valor máximo de tensión que admite el módulo. Por otro lado, Telit recomienda el empleo de un condensador de desacoplo de 100 microfaradios (uF). Es importante que el diodo Zener se mantenga lo más cerca posible de la batería, y el condensador de desacoplo esté lo más próximo posible a los pads de alimentación del GE910.

4.5.2 Circuito de encendido

Como podemos observar en el manual hardware del módulo GSM-GPS, para encender el módulo necesitamos (además de proporcionarle alimentación) poner una entrada a nivel bajo durante cierto tiempo, transcurrido el cual regresará a nivel alto. Este pin de encendido del módulo se denomina ‘ON_OFF’ y está en la posición R12 de la matriz de conexiones. Para encender el módulo deberemos poner a nivel bajo el voltaje de dicho pin durante, al menos, cinco segundos. Ya que esta entrada ha de permanecer a nivel alto, tanto antes del encendido como tras haber superado el tiempo que ha de mantenerse a nivel bajo para encenderse, la línea dispone de una resistencia de pull-up interna (esta resistencia mantiene el pin a nivel alto a menos que llevemos a GND dicho pin) por lo que, por ejemplo, si esta entrada se conectase a una salida en estado de alta impedancia, el valor lógico de esta entrada sería el de nivel alto.

Ante una situación de emergencia, es posible que el usuario sólo pueda accionar el pulsador de aviso de forma súbita durante un ínfimo instante de tiempo. Por lo tanto, debemos ser capaces de proporcionar el aviso una vez el usuario pulse el botón, independientemente de la cantidad de tiempo que el usuario lo mantenga accionado.

Una buena solución, consiste en el empleo de un circuito monoestable. Estos circuitos mantienen la señal de salida en un nivel lógico específico (estado estable) mientras no reciban una excitación externa, recibida la cual, cambiarán el nivel lógico de salida (estado casi-estable) durante un período determinado por una constante de tiempo. Diseñaremos un circuito que lleve su salida a nivel bajo (estado casi-estable del monoestable) durante un tiempo superior a cinco segundos, tras percibir que el pulsador se ha presionado (excitación exterior del circuito monoestable), el resto del tiempo esta salida permanecerá a nivel alto.

Para diseñar el sistema de encendido nos valdremos de un circuito integrado del tipo 555 (familia de monoestables), como puede ser el TLC555 de la compañía Texas Instruments. Este chip lleva su salida a nivel alto durante un tiempo aproximado de 1.1*R*C (donde R y C son los valores de la resistencia y capacidad de dos elementos del circuito), si recibe una entrada instantánea a nivel bajo. Esta configuración sería la mostrada en la imagen inferior izquierda de la siguiente página.

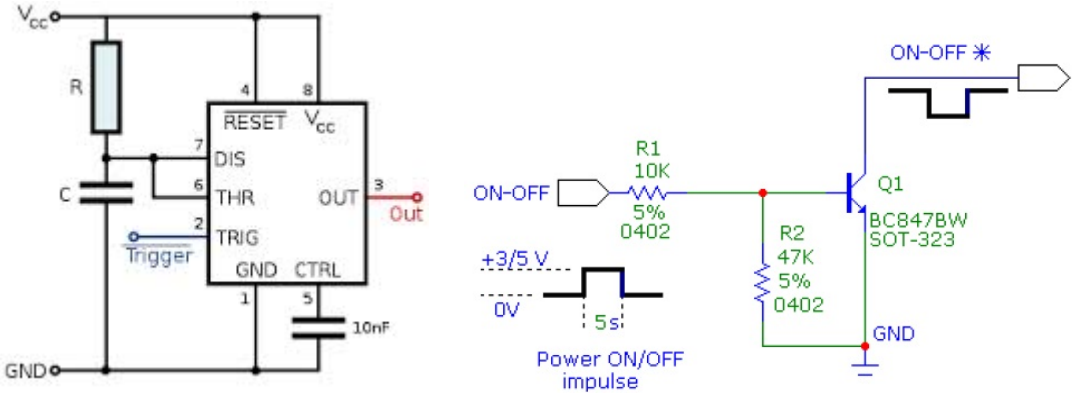


Figura 4.18: Circuitos necesarios para el encendido del dispositivo.

Será necesario invertir la señal de salida del TLC555, para que el estado estable proporcione una salida a nivel alto. La inversión de la señal puede hacerse mediante un transistor y dos resistencias, tal y como recomienda Telit (imagen superior derecha). Ahora, simplemente poniendo una resistencia de pull-up (de un valor adecuado) en la entrada trigger del TLC555 junto con el pulsador conectado a tierra, tendremos el circuito completo de encendido. Esta resistencia de pull-up conseguirá que la entrada del circuito esté a nivel alto a excepción de cuando apretemos el pulsador. En cuanto al valor adecuado de la resistencia de pull-up, debemos tener en cuenta que cuando activemos el pulsador conectaremos la entrada de alimentación (3'7 voltios) a tierra a través de dicha resistencia. Debemos elegir un valor de resistencia tal que el consumo de corriente, al activar el pulsador, no resulte excesivo. Con una resistencia de 22K Ohm, el consumo de corriente sería menor a 0'2 miliamperios (I= V/ R = 3'7/22000 = 0'000168181 A) lo cual resultaría suficientemente minúscula. El esquemático del circuito completo se muestra a continuación:

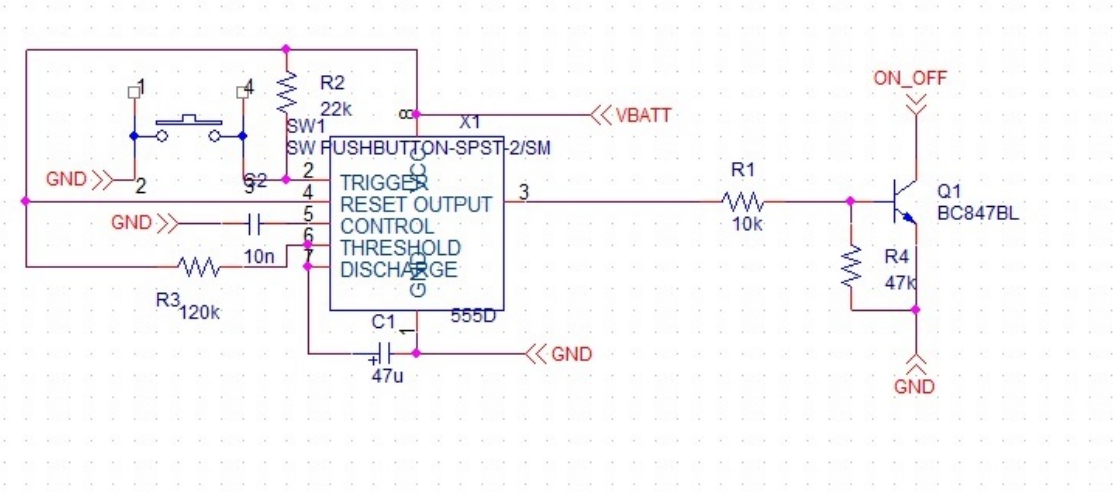


Figura 4.19: Circuito completo de encendido del dispositivo.

Para la elección de los valores de R y C de nuestro diseño tendremos que tener en cuenta las tolerancias de estos elementos y la propagación de errores en una operación de multiplicación. En cuanto a la propagación de errores, el error relativo que se cometería al multiplicar dos variables sería igual a la suma de los errores relativos de ambas variables. Los errores de estos componentes pasivos se expresan mediante las tolerancias, que no es más que una forma de mostrar el error relativo en términos de porcentaje. Es por todo esto que el posible error cometido en el producto,

expresado en términos de porcentaje (porcentaje de error sobre el valor total) será igual a la suma de las tolerancias de la resistencia y el condensador. La versión SMD más precisa de la resistencia que podemos adquirir posee una tolerancia del 1%, mientras que los condensadores SMD de Tántalo típicos poseen una tolerancia del 10%.

Una estrategia sencilla sería elegir unos valores de R y C tal que el valor 1’1*R*C superase al menos en un 11% el valor de 5, o lo que es lo mismo, que dicho producto sea igual o mayor que 5’6. También es aconsejable no escoger un condensador de valor demasiado elevado, puesto que esto aumenta drásticamente el tamaño del encapsulado (mientras que en las resistencias, su valor generalmente, no afecta al tamaño del encapsulado). Si escogemos unos valores de R y C iguales a 120K ohmios y 47 microfaradios respectivamente, el tiempo que se mantendría la señal a nivel alto a la salida del integrado 555 sería de 6.204. De esta manera, nos aseguramos, que el tiempo de encendido mínimo requerido sea cumplido con cierta holgura.

Como se pueden adquirir versiones SMD y THD del TLC555 y del resto de componentes de este circuito de encendido, adquirimos ambas versiones, una para el montaje del circuito de encendido en el PCB (Printed Circuit Board, placa de circuito impreso) que diseñemos, y otra para realizar pruebas con el osciloscopio, previo al montaje de la versión SMD en el PCB. A continuación mostramos la placa ‘breadboard’ empleada para probar el funcionamiento del circuito de encendido:

4.5.3 Circuito de comunicación con la tarjeta SIM

El módulo GSM-GPS dispone de 5 pines para la comunicación con la tarjeta SIM. La forma de comunicar éstos pines con la ranura para inserción de la tarjeta SIM se observa a continuación:

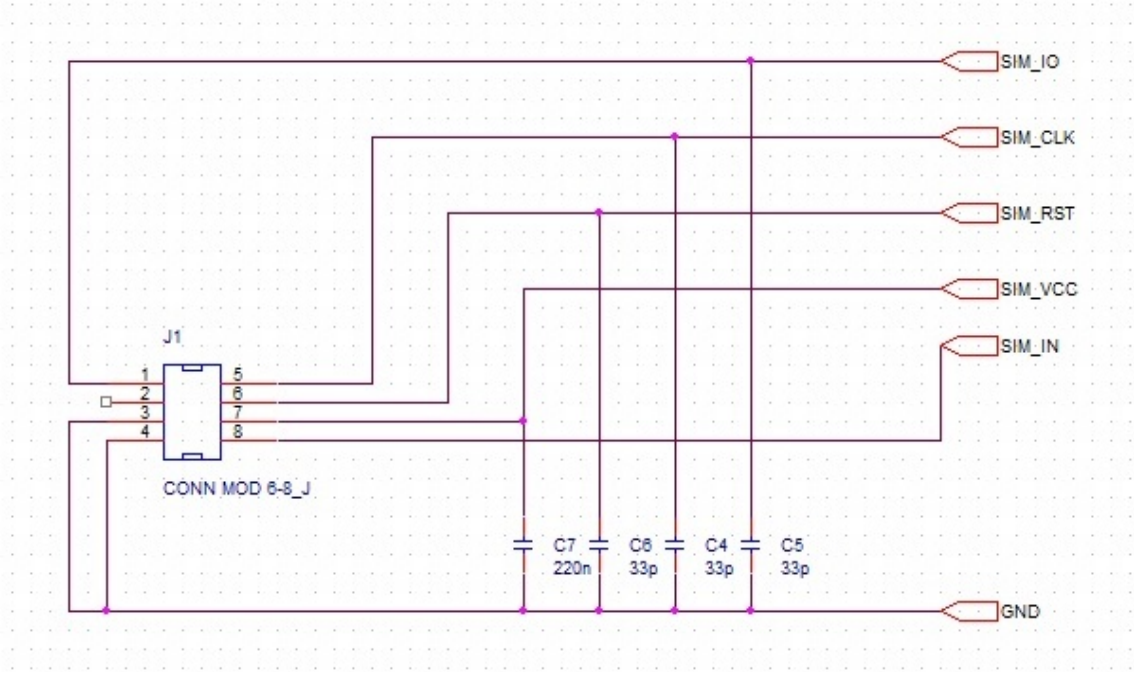


Figura 4.20: Circuito de comunicación del GE910 con la tarjeta SIM.

Para realizar este diseño, nos hemos basado en las recomendaciones de Telit en su manual para conexionado de sus módulos con la tarjeta SIM. Tendremos que prestar especial cuidado, en el aparatado de desarrollo hardware, para saber exactamente la disposición de pines dentro de la ranura para inserción de la tarjeta SIM que adquiramos.

4.5.4 Circuito de comunicación serial con el PC

El módulo dispone de una serie de interfaces de comunicaciones, entre las cuales están: USB 2.0 (estándar 2.0), y tres puertos serie (el principal, el auxiliar y el de la salida GNSS).

Por simplicidad, en nuestro diseño emplearemos el puerto serial principal, configurado como interfaz RS232. El puerto serie del módulo es una UART que dispone de las 8 señales del protocolo RS232, pero difiere en el RS232 del ordenador en la polaridad y nivel lógico de las señales. El módulo trabaja con señales cuyos niveles lógicos son de 0 y 1.8 voltios, mientras que el PC lo hace con unos valores de -15 y 15 voltios. Debemos ser capaces de convertir estas señales en ambas direcciones (señales que van del PC al módulo y señales que van del módulo al PC). En la guía hardware del módulo, nos muestran una imagen de un posible diseño, basado en el chip MAX218 de Maxim. En nuestro diseño empleamos el chip MAX3218, una versión más moderna del MAX218, el cual no se encontraba disponible. En la siguiente figura podemos apreciar una imagen sobre las recomendaciones de Telit en cuanto a la conversión de los niveles lógicos de su puerto serie para conexión con un PC:

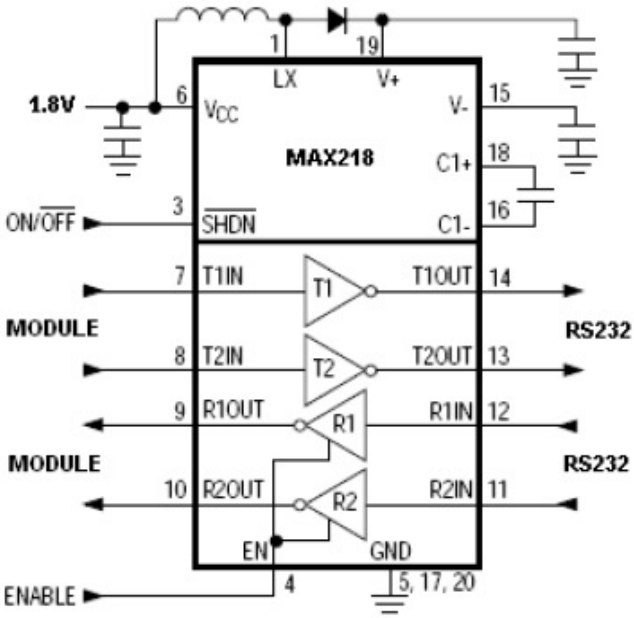


Figura 4.21: Configuración de pines del chip MAX3218.

Aquí la conversión se realiza sobre cuatro señales RS232, que son dos entradas y dos salidas del módulo. Esto se debe a que un único integrado MAX3218 posee dos drivers (traducción de salidas del módulo a entradas del PC) y dos receptores (traducción de salidas del PC a entradas del módulo). Esta conversión ha de realizarse sobre las 8 señales del protocolo RS232, conformadas por 5 salidas y 3 entradas del módulo, por lo tanto, necesitaremos, al menos, replicar 3 veces el circuito de la imagen superior (aunque algunos drivers o receptores del chip MAX3218 no sean empleados) para lograr la conversión total. A continuación mostramos una imagen del esquemático del circuito completo de comunicación con el PC:

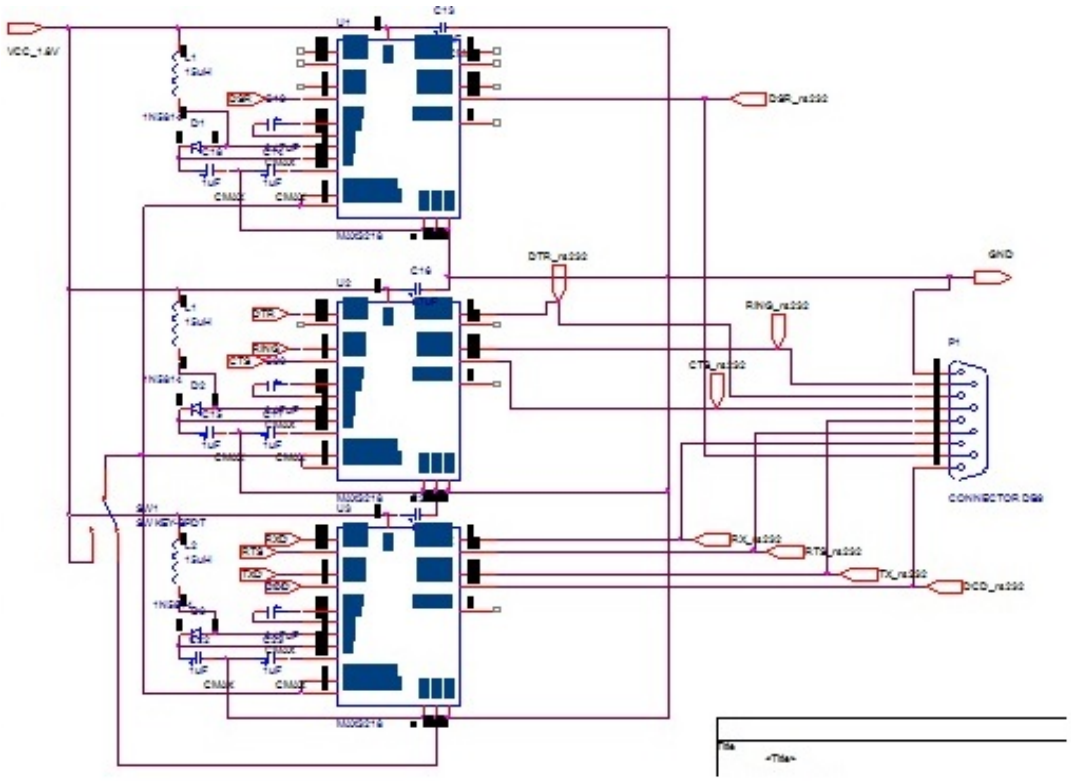


Figura 4.22: Esquemático del circuito de comunicación con el PC.

Puede observarse en el esquemático que la alimentación del chip es de 1.8 voltios, por lo que necesitaremos una fuente independiente, o un regulador para convertir los 3.7 voltios de la batería en 1.8 voltios. Para una mejor visualización del circuito de comunicación serial, mostramos una imagen ampliada del esquemático de la figura anterior:

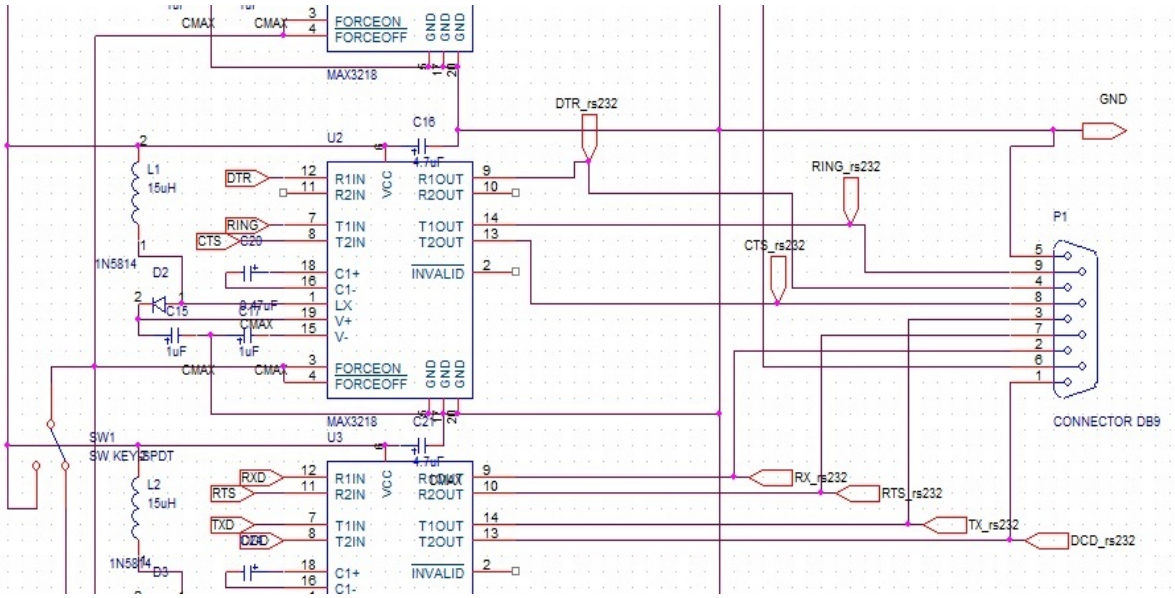


Figura 4.23: Detalles del esquemático del circuito de comunicación con el PC.

4.5.4 Circuito de información lumínica

Cada uno de los puertos GPIO presentes en el GE910 puede proporcionar un máximo de 1 mA de corriente, cuando se ha configurado como puerto de salida. Es por ello que se hace necesario adquirir indicadores de tipo LED con un bajo consumo de corriente. En caso de que la corriente necesaria para encender el LED sea superior a la proporcionada por el puerto, podremos alimentar el led mediante la fuente de 1’8 voltios del GE910 (señal VAUX/PWRMON). Esta fuente es capaz de surtir hasta 50 mA, como podemos apreciar en el manual “*xE910 global form factor application note*” disponible en la web de Telit. Mediante un transistor podríamos usar un puerto de salida como interruptor (conectado éste a la base del transistor), sin tener que emplear una corriente superior al miliamperio en la base.

Llegados a este punto ya podemos pasar a diseñar el circuito necesario para el manejo de los leds. Debemos tener en cuenta que los puertos de salida poseen niveles CMOS de 1’8 Voltios y son capaces de entregar un máximo de un miliamperio de corriente. Hemos adquirido unos leds rojos de bajo consumo, los cuales tienen una corriente directa típica de funcionamiento de un miliamperio a un voltaje de polarización de 1’6 Voltios. En la siguiente imagen se muestra un esquemático del circuito empleado.

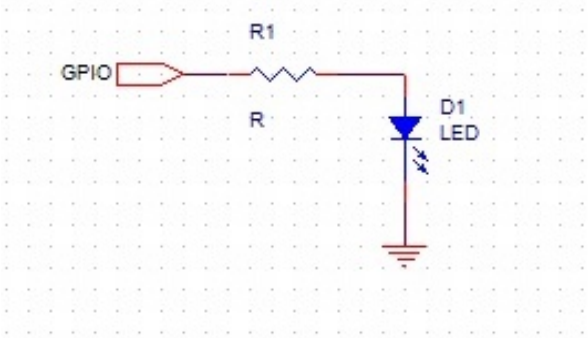


Figura 4.24: Circuito de manejo de indicadores lumínicos.

Para calcular el valor de la resistencia R1 debemos saber que la corriente que circulará por la rama es de un miliamperio. En estas condiciones, el voltaje de polarización en directa del led es de 1.6 voltios. Como la fuente proporciona 1.8 voltios, debemos de conseguir que en la resistencia caigan los 0.2 voltios sobrantes. Como la corriente es de un miliamperio, necesitaremos una resistencia de 200 ohmios (valiéndonos de la ley de ohm). Como 200 ohmios no es un valor comercial de resistencia, y teniendo en cuenta que la corriente de un miliamperio no debe ser excedida, cogeremos el valor comercial inmediatamente superior, 220 ohmios.

Una vez tenemos este diseño, montaremos en una placa 'breadboard' nuestro led con la resistencia en serie para comprobar si los puertos de salida podrán proporcionar la corriente necesaria al led. Emplearemos el código mostrado en el apartado de desarrollo software, donde se explica el manejo de los GPIO. Buscamos en la placa interfaz del EVK2 el pin correspondiente al GPIO_07, éste es el pin 6º del array de pines con referencia PL302. Conectamos este pin a la resistencia y buscamos un pin de tierra en la placa interfaz (por ejemplo el pin 10º del array PL102) para conectarlo al cátodo del led. Hecho esto podemos observar que el led se enciende adecuadamente. A continuación mostramos una imagen con el montaje realizado y el led encendido.

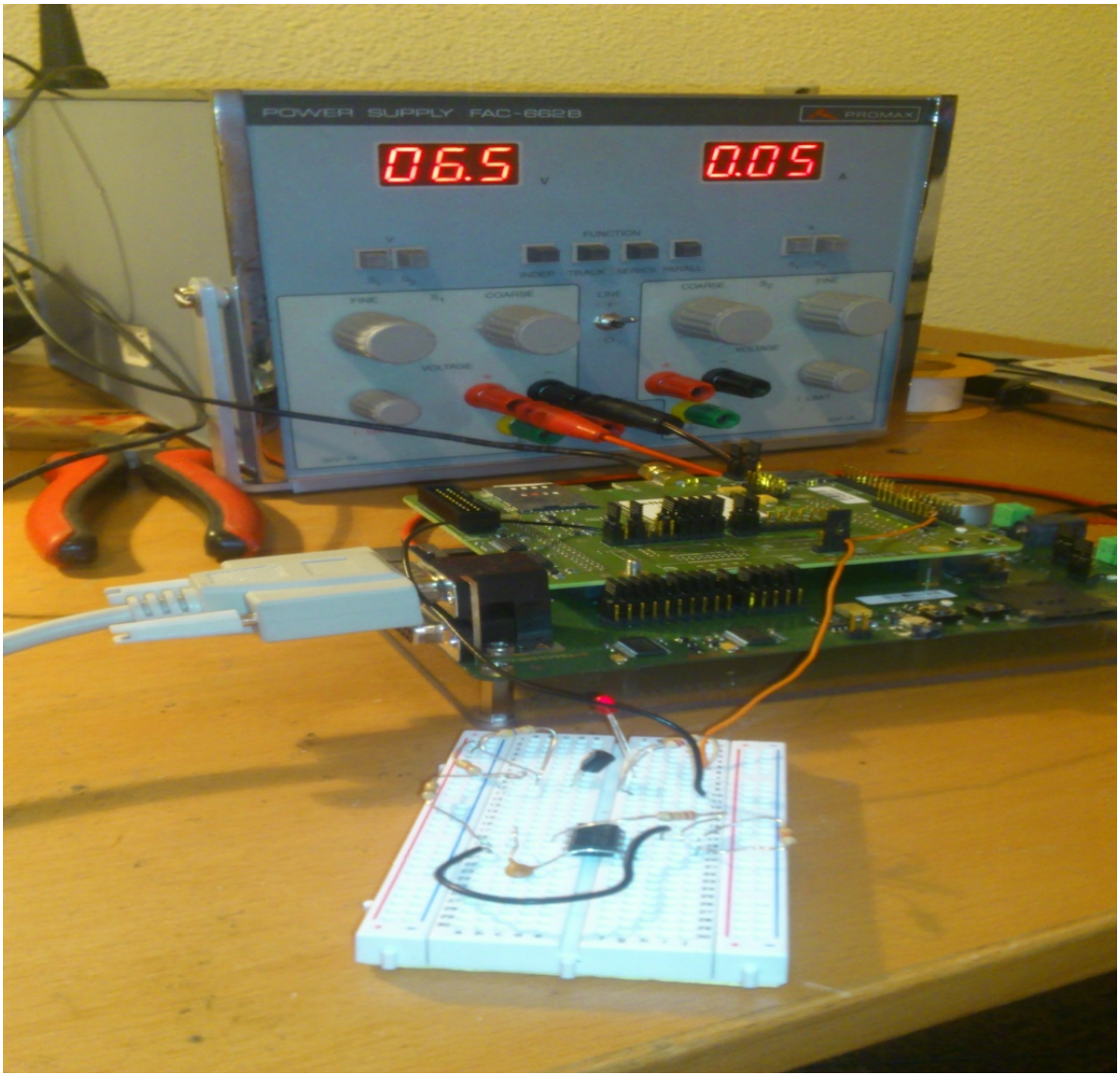


Figura 4.25: Montaje de prueba del circuito de indicadores lumínicos.

Podemos apreciar un ligero brillo en el diodo led, aunque es bastante más notorio el brillo de los leds del display de la fuente de alimentación. Para aumentar el brillo del led de la imagen

superior tendríamos que recurrir a la fuente independiente de 1’8 voltios presente en el GE910.

4.5.5 Circuito de conexionado del módulo GE910

En este apartado comentaremos el circuito necesario para la integración del módulo GE910 junto al resto de circuitos. En este montaje será necesario sacar una fila de pinchos con las señales para comunicación con el PC, otro par de filas de pinchos para comunicación con la placa inferior del dispositivo de usuario. También será necesario especificar las conexiones del módulo con las antenas GSM y GPS y con el resto de señales presentes en los circuitos anteriores. A continuación mostramos dicho esquemático:

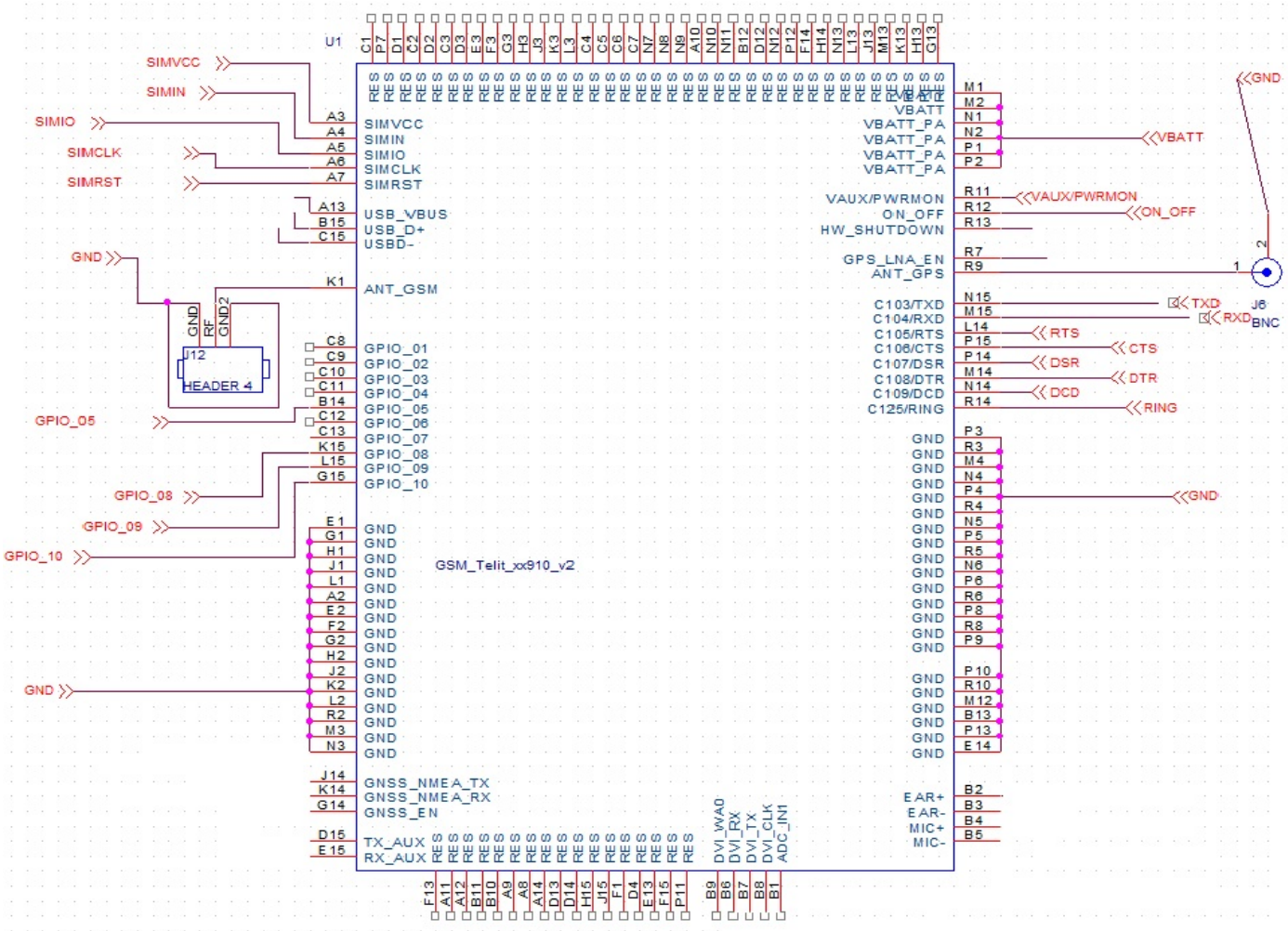


Figura 4.26: Circuito con el conexionado del GE910-GNSS.

En la imagen anterior podemos observar lo explicado en el párrafo anterior. Simplemente faltaría añadir las filas de pinchos para la comunicación con el resto de elementos circuitales y la entrada de alimentación. Estas filas de pinchos se encuentran en otra página de tipo esquemático perteneciente al mismo proyecto, y se muestran en la figura que se encuentra a continuación:

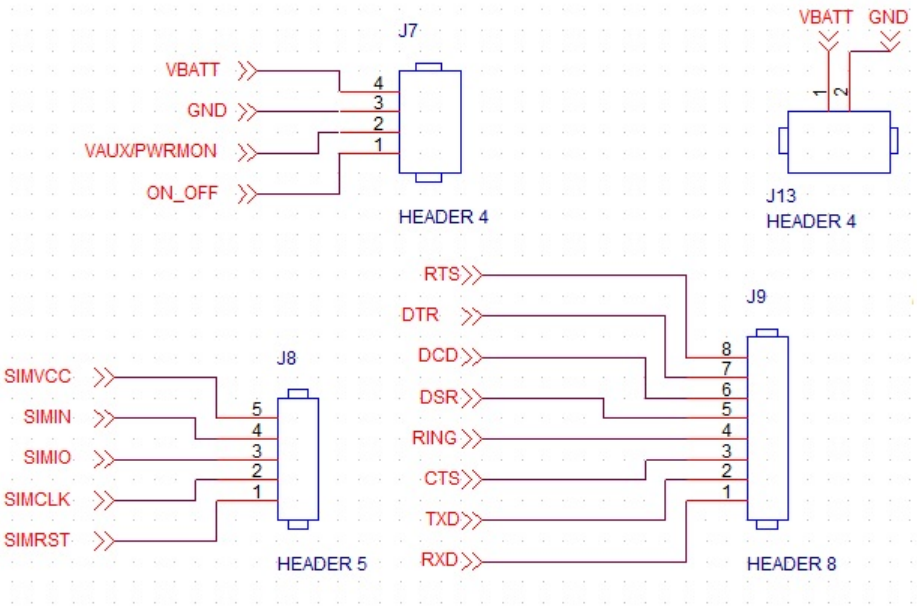


Figura 4.27: Resto del circuito de conexionado del GE910-GNSS.

4.5 Desarrollo Hardware

4.5.1 Introducción

En este apartado iremos viendo el desarrollo de las placas de circuito impreso (PCBs) que compondrán nuestro dispositivo. Hemos empleado las herramientas Capture y Layout de la plataforma Orcad para este diseño Hardware. Con el programa Capture crearemos los esquemáticos de nuestro diseño, ya mostrados en el aparatado anterior, mientras que el programa Layout será empleado para traducir dichos esquemáticos a las pistas y footprints de los componentes que compondrán nuestras placas de circuito impreso.

4.5.2 Solución final

Una vez vistos todos los circuitos necesarios para el desarrollo del dispositivo, debemos ser capaces de aunarlos en un mismo dispositivo, cuyo tamaño sea lo más reducido posible. Para cumplir nuestros propósitos, decidimos desarrollar tres placas distintas. Por un lado, el dispositivo de usuario estaría formado por la interconexión de dos de estas tres placas, aprovechando así algo de espacio en anchura (un diseño con un solo PCB tendría que incluir todos los elementos circuitales, ocupando aproximadamente la suma del área de cada uno de los PCBs). Ambos PCBs, a su vez, servirán para fijar la batería del dispositivo, la cual irá en el espacio entre placa y placa. Ha de desarrollarse un tercer PCB para comunicar el dispositivo de usuario con un PC (y así depurar posibles fallos, cargar nuestro script, etc.). Ésta nueva placa de circuito impreso no formará parte del dispositivo de usuario, consiguiendo ahorrar gran cantidad de espacio, puesto que este PCB requiere de muchos componentes, algunos de los cuales resultan ser bastante voluminosos. Así mismo no necesitaremos preocuparnos demasiado del tamaño de esta tercera placa.

En la placa superior del dispositivo de usuario soldaremos el módulo GSM-GPS con el circuito de protección de la alimentación y las antenas GSM y GPS junto con los indicadores lumínicos. De esta manera las señales de RF de las antenas no tendrían que ‘viajar’ entre placas distintas y estarían situadas en la parte superior del dispositivo. A su vez, cumplimos con las recomendaciones de Telit, las cuales indican que el condensador del circuito de protección ha de situarse próximo al GE910. Por otro lado, en la placa inferior soldaremos la ranura de inserción de la tarjeta SIM junto con el circuito de encendido. Podremos colocar la batería entre ambos PCBs, aprovechando el espacio existente al usar camas de pinchos. Además de esta manera será sencillo fijar la batería al dispositivo.

Como hemos comentado, en primer lugar debemos crear un proyecto de Orcad Capture por cada PCB que queramos diseñar. Desde este mismo programa podremos crear aquellos componentes que no se encuentren en las librerías (tales como el módulo GE910-GNSS, la ranura de inserción de la tarjeta SIM, etc.). Para poder trabajar en el desarrollo y emplazamiento de las pistas y footprints de nuestro PCB, debemos “traducir” los esquemáticos creados desde Orcad Capture para poder trabajar con Orcad Layout en el diseño de los circuitos impresos asociados a dicho esquemático. Tras realizar esta traducción (la cual será detallada más adelante) y cargarla desde el programa Layout, éste nos pedirá que asociemos un footprint a cada componente. Es por este motivo que desarrollaremos, antes de realizar esta traducción, todos los footprints no disponibles en las librerías del programa Layout de los componentes involucrados en el diseño.

4.5.3 Placa de comunicación serial

Lo que debemos hacer, una vez desarrollados todos los footprints de los elementos necesarios para este diseño, es crear un proyecto de Orcad Capture con el esquemático del circuito completo de comunicación serial (circuito de comunicación serial y conectores de dicho circuito).

El siguiente paso será la generación de un archivo, denominado Netlist, que será necesario para traducir los esquemáticos de Orcad Capture en footprints y pistas de un PCB físico para el programa Orcad Layout. Para crear este archivo, debemos seleccionar con el ratón el proyecto dentro de la ventana de archivos del Orcad Capture. Hecho esto pulsamos sobre el desplegable `Tools` del menú de opciones y elegimos la opción de `Create Netlist`, como podemos observar en la siguiente imagen:

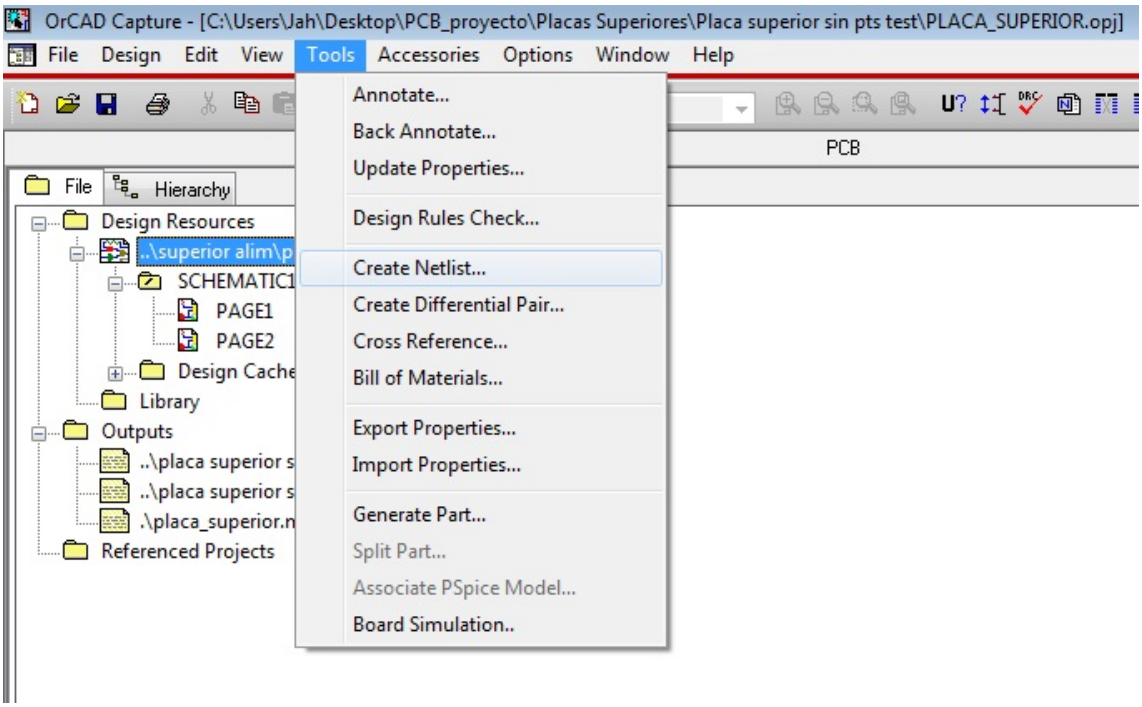


Figura 4.28: Creación del archivo Netlist.

Tras realizar esta acción se nos abrirá una ventana nueva. En esta ventana abriremos la pestaña que pone `Layout`, en la cual indicaremos el nombre del archivo y la carpeta que lo contendrá. Lo siguiente será darle a continuar.

Ahora, creada la netlist, debemos ejecutar la aplicación Orcad Layout. Escogemos crear un nuevo proyecto pulsando sobre `New` en el desplegable `File` del menú de opciones. Se nos abrirá una ventana, en la cual elegiremos un archivo de referencia, un archivo de tipo netlist y el emplazamiento donde se nos generará el archivo de tipo Layout (archivo .MAX) con el PCB asociado a dicha netlist. Seleccionadas estas tres opciones, pulsamos sobre el botón de `Apply ECO`. Tras hacer esto, el programa Layout nos pedirá que asociemos uno por uno aquellos footprints para los componentes que no tengan uno predeterminado. Vamos haciéndolo, seleccionando elementos de entre nuestra librería y las librerías estándar. Una vez hecho esto, nos aparecerá una pantalla con todos los componentes. A continuación mostramos una imagen con la ventana que nos aparecerá en el programa Layout con el nuevo proyecto.

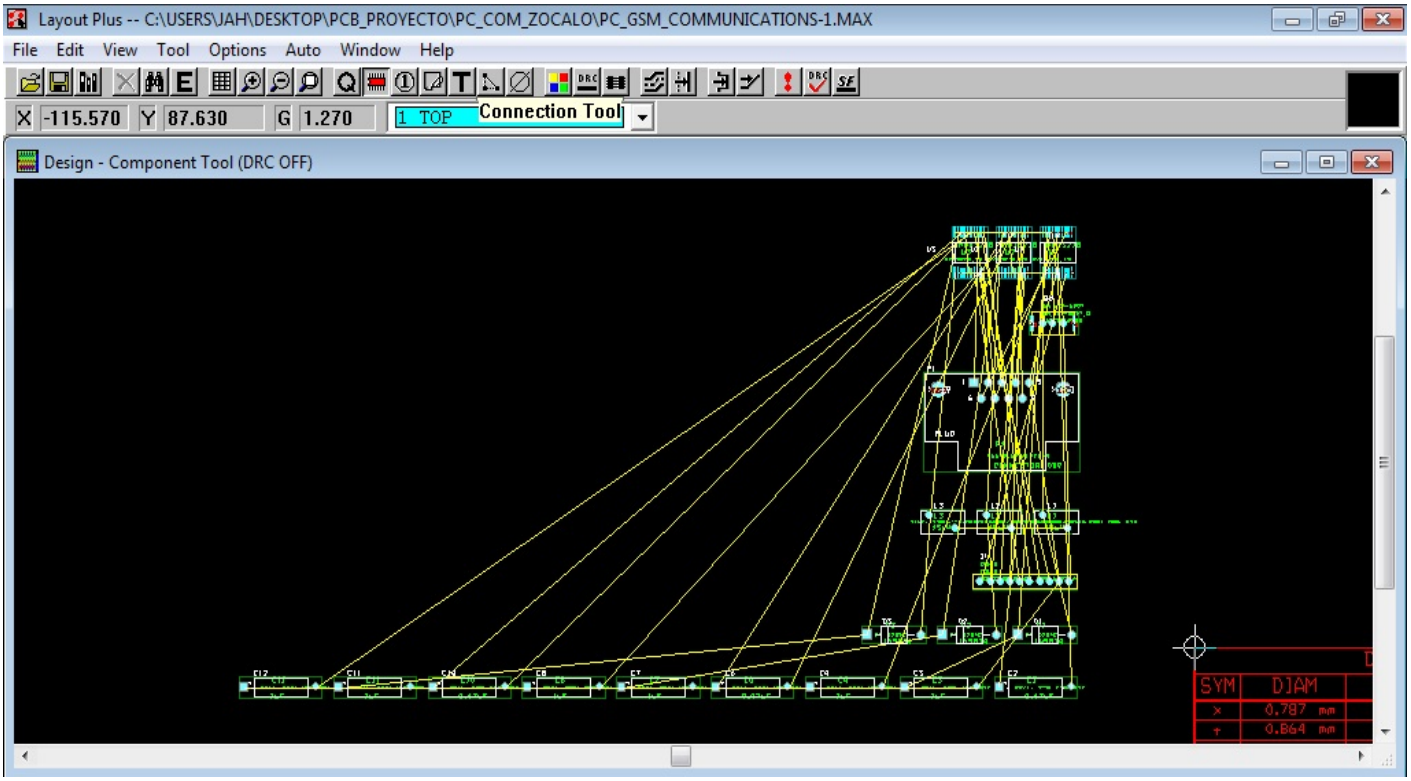


Figura 4.29: Creación del proyecto de placa de comunicación serial con Layout.

En la imagen superior podemos apreciar todos los footprints involucrados en esta placa, con sus respectivas conexiones representadas mediante líneas amarillas. Si aún quisiéramos cambiar el footprint de algún componente en particular, sólo tendríamos que acceder a sus propiedades pinchando sobre él con el botón derecho del ratón, tras haber seleccionado la utilidad `Component Tool` del panel de herramientas. En el campo `footprint` dentro de la ventana de propiedades, no tendríamos más que buscar un nuevo footprint de entre todos los que tenemos y seleccionarlo.

El proceso siguiente será el de emplazar estos componentes de la forma que consideremos más adecuada y crear el borde de placa (como ya vimos el tamaño de ésta no debe preocuparnos mucho). Una vez hayamos hecho esto, podremos proceder al ruteado de las pistas. Con la herramienta `track segment` dentro de la opción `tool` del menú de opciones, podremos realizar manualmente el ruteado de las pistas. Con esta misma herramienta, presionando la tecla `w` podremos definir la anchura de las pistas. En esta placa definimos una anchura de pista de 1 mm para las pistas de alimentación y 0’5 mm para el resto de pistas. Para aquellas pistas que tengan que pasar necesariamente entre dos pines a soldar reduciremos su anchura hasta llegar a los 0’35 mm para evitar posibles cortocircuitos. En la siguiente página se muestra una imagen de esta placa con todas las pistas ruteadas.

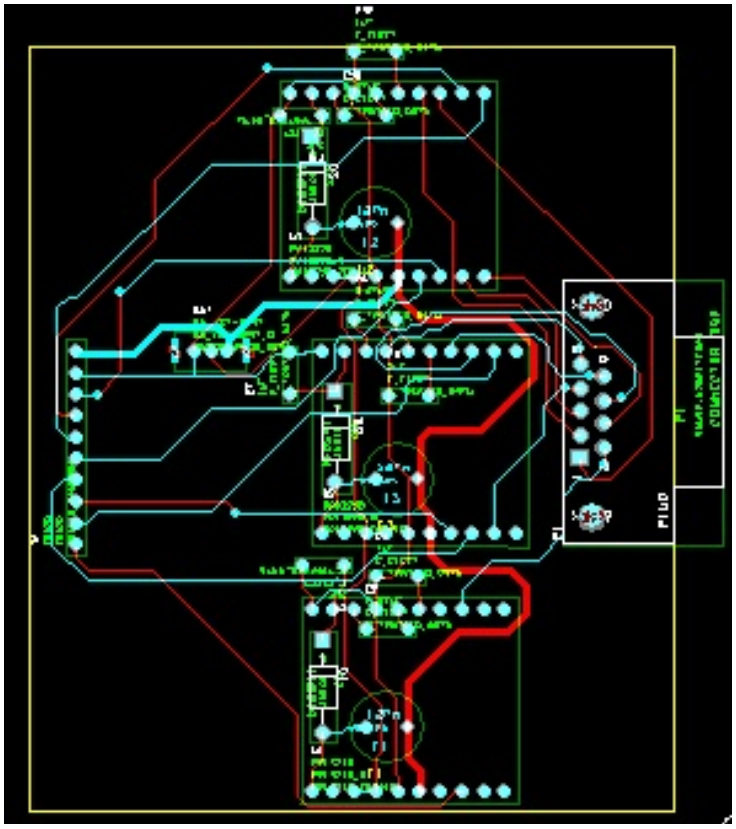


Figura 4.30: Placa de comunicación serial con todas las pistas ruteadas.

En la imagen superior podemos apreciar el ruteado y emplazamiento de componentes de esta placa. Las pistas de color rojo representan el ruteo de la capa inferior, mientras que las de color verdoso son las presentes en la cara superior del PCB. Podemos observar que la ancha pista de alimentación ha sido ruteada de tal manera que no pase cerca de los pines de los zócalos, lo cual, si sucediese, dificultaría la soldadura de éstos. Como el conector DB9 para conexión al PC imposibilita su soldado en la cara donde éste se sitúe, se ha tenido que llevar todas las pistas que lo interconectan a la placa inferior, puesto que hemos decidido situarlo sobre la capa superior.

Para finalizar el desarrollo, tras tener todas las pistas ruteadas, deberemos definir los planos de masa de la placa. Para ello escogeremos la herramienta `obstacle` del panel de herramientas, haremos 'click' con el botón derecho del ratón y escogeremos `New`, de nuevo presionamos con el botón derecho del ratón y escogemos `Properties`. Se nos abrirá una ventana, en la cual elegimos un obstáculo de tipo `copper pour`, con una resolución de 0.05 mm (dato denominado `width` en la ventana). Escogeremos en qué capa estará presente, en el desplegable `Obstacle Layer`, ya sea el plano de masa superior o inferior. Definiremos un espacio de 0'85 mm entre el plano de masa y el resto de pistas y pines (`clearance`). Por último, antes de presionar `ok`, asociaremos el plano de masa a la conexión GND (`net attachment).` Aunque a estas alturas ya podemos apreciar el diseño final de este PCB, esperaremos a generar los ficheros de fabricación para mostrar una imagen con el resultado final.

Los ficheros de fabricación que Orcad Layout genera son en formato Gerber, estándar de facto en la industria de la fabricación de placas de circuito impreso. Para fabricar el PCB empleando la fresadora de la escuela, se necesitarán cuatro ficheros:

1.- **Fichero de contorno de placa:** este fichero no se genera por defecto, por lo que debemos añadirlo dentro del cuadro que se nos abre al pichar sobre `Post Process Settings`,

dentro de `Options`. Escogemos las siglas ‘BRD’ como formato de este tipo de fichero.

2.- **Fichero de taladros:** estos ficheros tienen la terminación ‘.tap’, aunque en realidad su formato es el mismo que el de un fichero de texto. Indican a la fresadora la posición exacta (coordenadas cartesianas) y tamaño de los taladros presentes en la placa.

3.- **Fichero de cara superior**: este fichero contiene la definición de las zonas de cobre de la capa superior de la placa. Su formato será ‘.TOP’.

4.- **Fichero de cara inferior**: este fichero contiene la definición de las zonas de cobre de la capa inferior de la placa. Su formato será ‘.BOT’.

A continuación mostramos el diseño final de esta placa a través de los archivos Gerber ‘.TOP’ y ‘.BOT’ generados por Layout.

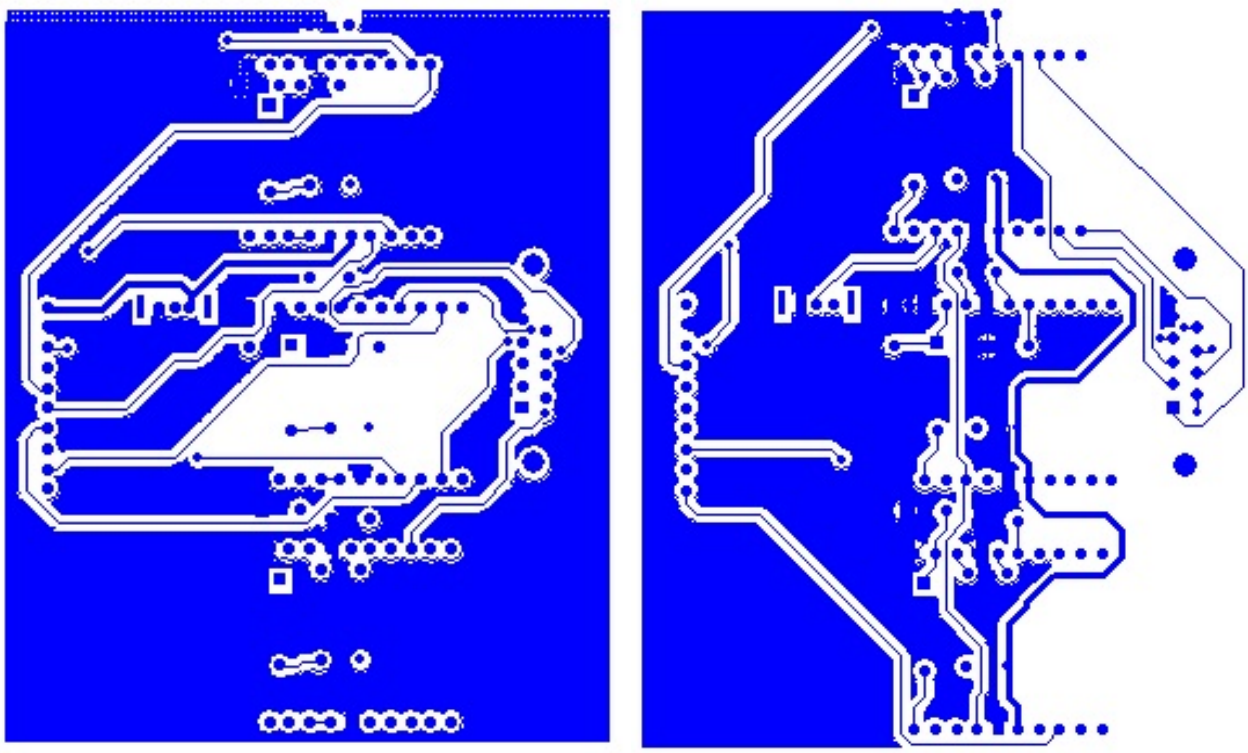


Figura 4.31: Placa de comunicación serial vista mediante los ficheros Gerber.

En las imágenes superiores podemos ver el aspecto del diseño final de la placa. A la izquierda tenemos la cara superior, y a la derecha la inferior. Finalmente, en las dos figuras siguientes podemos visualizar el aspecto de ambas caras de la placa, una vez fabricadas.

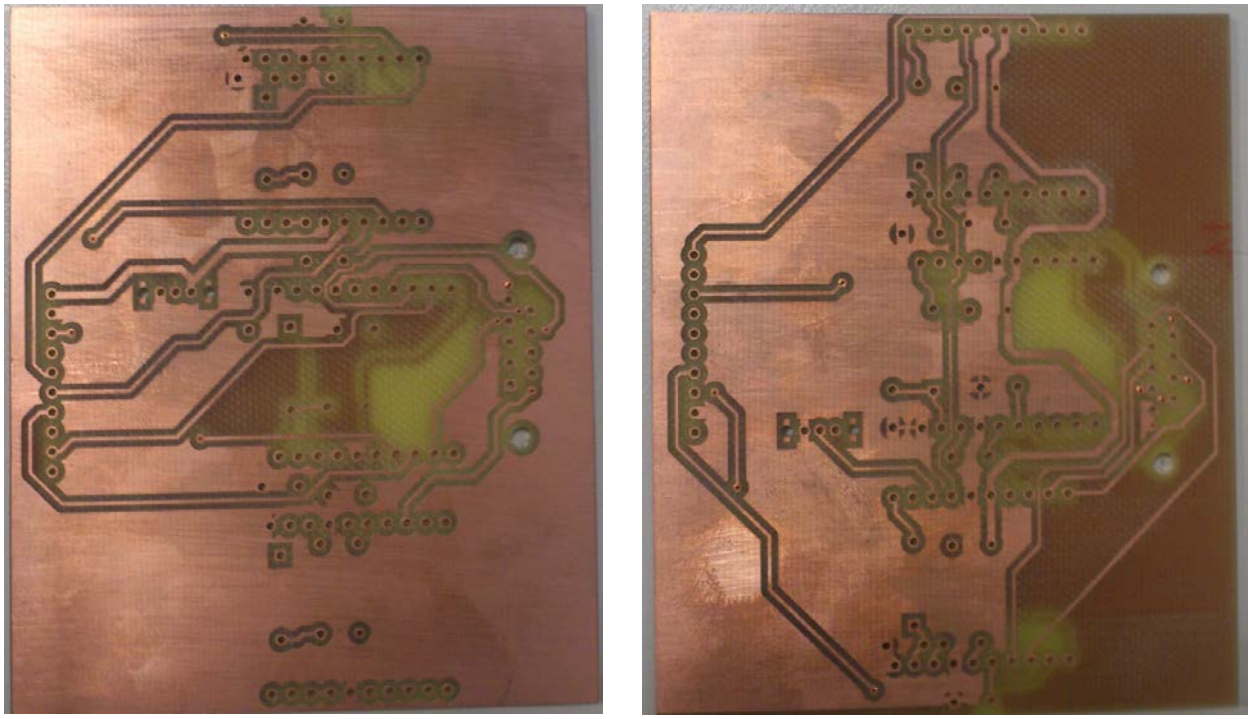


Figura 4.32: Fotografía de amas caras de la Placa de comunicación serial.

4.5.4 Placa del módulo GSM y antenas

Desarrollaremos antes la placa superior que la intermedia, ya que la placa superior contiene un mayor volumen de elementos. Como ambas placa han de interconectarse, a la hora de desarrollar la placa inferior, crearemos un borde de placa de las mismas dimensiones que el de la placa superior. También emplazaremos los conectores de la misma manera que aparecen sobre la placa superior.

En primer lugar debemos integrar los esquemáticos del circuito de alimentación y del circuito del módulo GE910 en un mismo proyecto de Orcad Capture, para desarrollar la placa superior como un mismo proyecto. Hecho esto, pasamos a crear el archivo Netlist del proyecto, procediendo de la misma manera que en el apartado anterior. Tras realizar esto, creamos un nuevo proyecto de Orcad Layout a partir de esta Netlist. A continuación mostramos el nuevo proyecto de Orcad Layout:

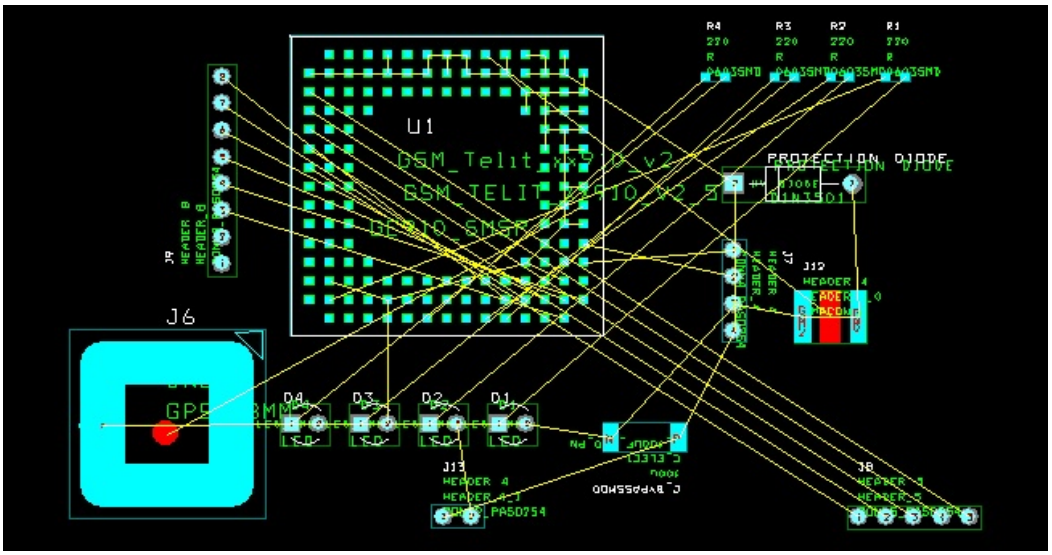


Figura 4.33: Creación del proyecto de placa superior con Layout.

En la imagen superior podemos apreciar todos los componentes involucrados en esta placa. Ahora deberemos emplazar estos componentes de forma óptima para que ocupen el menor área posible y permitan su ruteado, empleando únicamente dos capas en el PCB. En cuanto al ruteado de las pistas de RF (antenas GNSS y GSM), estas deben ser pistas de 50 ohmios. Lo más sencillo será desarrollar estas pistas en tecnología microstrip. La impedancia de una pista microstrip es función de cuatro variables. Estas variables son: el espesor del dieléctrico de la placa, el espesor del cobre, la constante dieléctrica del sustrato y la anchura de la línea. En el módulo GE910 los pines de RF están situados en los extremos, de tal manera, que la línea no tendrá que ir sorteando pads (a excepción de los dos pines vecinos al de RF). Debemos de elegir un sustrato que nos permita tener una anchura de pista microstrip de 50 ohmios menor del doble del pitch del GE910 (para evitar cortocircuitar los pads vecinos del de RF).

Una buena estrategia de emplazamiento sería situar el módulo GE910 en la cara inferior, de tal modo que las antenas se interconecten mediante pistas lo más cortas posible. Emplazando el GE910 en la cara inferior, juntaremos la antena GNSS a éste, pero en la cara superior (recordemos que la conexión RF se realiza en la cara opuesta de donde se sitúe este componente). Por otro lado, el conector RF de la antena GSM se situará lo más cerca posible del pin de RF del GE910. De esta manera ya tendemos casi definido el área que ocupará esta placa. A continuación se muestra una imagen de este PCB con las pistas ya ruteadas en el programa Orcad Layout.

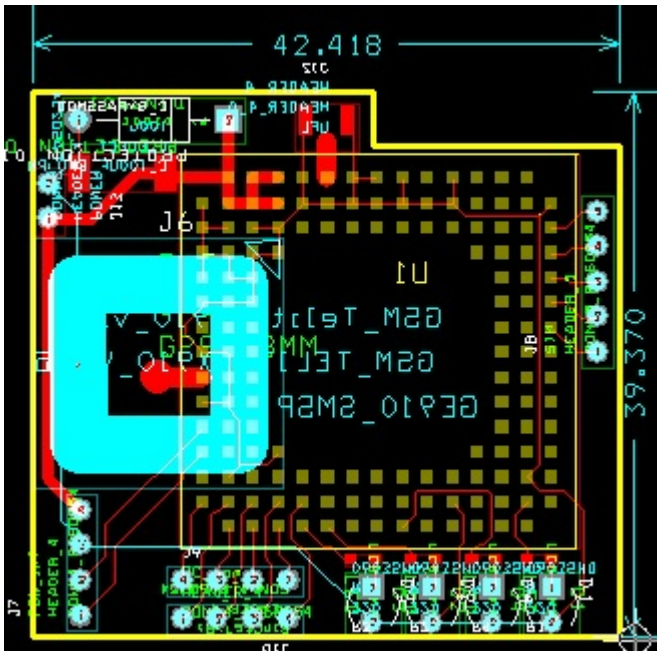


Figura 4.34: Placa superior con los componentes emplazados y las pistas ruteadas.

En la imagen superior podemos observar la estrategia de emplazamiento de los componentes descrita en el párrafo anterior. Los conectores de 4 y 5 pines más aislados son los empleados para interconexión con la placa inferior; éstos están lo más separados posible para lograr una mejor fijación mecánica y asegurar espacio para la batería entre placa y placa. Los dos conectores de 4 pines que aparecen juntos serán los empleados para la comunicación del GE910 con el PC. Podemos observar que el espacio ocupado en horizontal viene impuesto por emplazar el GE910, el conector de conexión con la placa inferior y la antena GNSS; mientras que el espacio en vertical viene impuesto por el emplazamiento del conector RF de la antena GSM y los conectores de comunicación con el PC. El emplazamiento de los leds y el circuito de protección de la alimentación se han realizado ocupando el espacio libre (espacio que quedaría emplazando el resto de componentes de la placa y creando un borde de placa rectangular que los contuviera). Se ha rebajado un poco el borde de placa (parte superior derecha del dibujo) en la zona donde iría situado

el pulsador, por si éste ha de situarse en la cara superior de la placa inferior, para poder tener acceso a él.

En este PCB resultará crucial un buen tratamiento del aspecto RF, el cual implica la creación de grandes superficies dedicadas a plano de masa conectadas entre capas mediante multitud de vías. El colocar estas vías entre planos de masa de capas distintas evitará que se genere un modo TEM espurio.

Cabe destacar que la fabricación y montaje de esta placa correrá a cargo de una empresa externa a la escuela, debida a su alta complejidad. Por este motivo, también debemos prestar especial atención a las capas de máscara de soldadura, imprescindibles para el correcto montaje de este circuito. Sin máscara de soldadura es muy probable que el soldado del encapsulado LGA se realice de forma incorrecta. También debemos tener en cuenta que las pistas de radiofrecuencia no deben cubrirse de éste material, puesto que estaríamos alterando su impedancia característica (de hecho, al depositar este material dieléctrico, convertiríamos nuestras pistas microstrip en oro tipo de pistas de RF llamadas embedded microstrip).

Como el software Orcad no permite realizar modificaciones a mano sobre la capa de máscara de soldadura, y para evitar que en los ficheros de fabricación aparezca máscara de soldadura sobre las pistas de RF, decidimos definir una vía cuadrada sin taladro. La estrategia consiste en definir únicamente la vía en la capa de la máscara de soldadura, para que su emplazamiento sólo suponga retirar la máscara de soldadura de su superficie. De este modo, definimos una vía cuadrada (Orcad no permite la definición de vías rectangulares) cuyo lado sea igual a la anchura de la pista de RF. Para retirar la máscara de soldadura de las pistas de RF emplazaremos tantas de estas vías como sean necesarias sobre dichas pistas.

Para el desarrollo de esta placa, se especificaron las características del sustrato requerido para que las pistas de RF tuviesen la impedancia deseada. Los cálculos de la anchura de pistas de RF fueron realizadas simulando un sustrato de FR4 de 0.8 mm de grosor con una anchura de cobre de 70 micras. En cuanto a los ficheros de fabricación necesarios, fueron generados los Gerbers de las caras superior e inferior así como los de las capas superior e inferior de máscara de soldadura y el fichero de taladros. A continuación se muestran dos imágenes en las cuales se pueden apreciar las dos caras de esta placa.

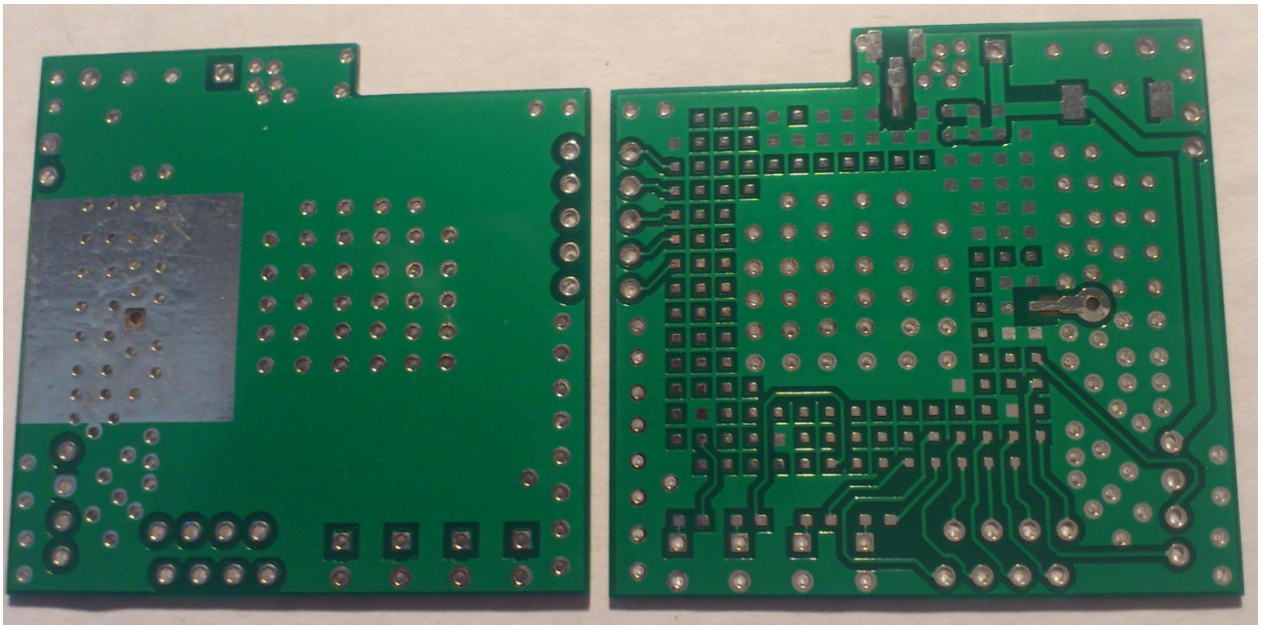


Figura 4.35: Placa superior encargada a PRODISA.

4.5.5 Placa de ranura de tarjeta SIM y encendido

Para el desarrollo de esta placa, crearemos el archivo Netlist desde un proyecto de Orcad Capture en el cual estén presentes los esquemáticos del circuito de encendido y comunicación con la tarjeta SIM. También será necesario añadir al proyecto los conectores que sirven para la conexión con la placa superior, prestando atención a que se presenten de la misma forma que se especificaron en el proyecto de la placa superior (mismo orden de conexión de señales). Generamos la Netlist y la empleamos para empezar un nuevo proyecto de Layout.

Ahora pasamos a crear un borde de placa con las mismas dimensiones que el de la placa superior. Para emplazar los conectores de interconexión tomaremos medidas de dónde se encuentran éstos dentro de la placa superior. Cuando éstos estén ya situados, podremos emplazar el resto de componentes presentes en la placa intermedia. Para facilitar al usuario el acceso al pulsador de encendido, situaremos éste sobre la cara inferior del PCB.

Presentamos ahora una imagen de la placa inferior, con los componentes emplazados y las pistas ruteadas.

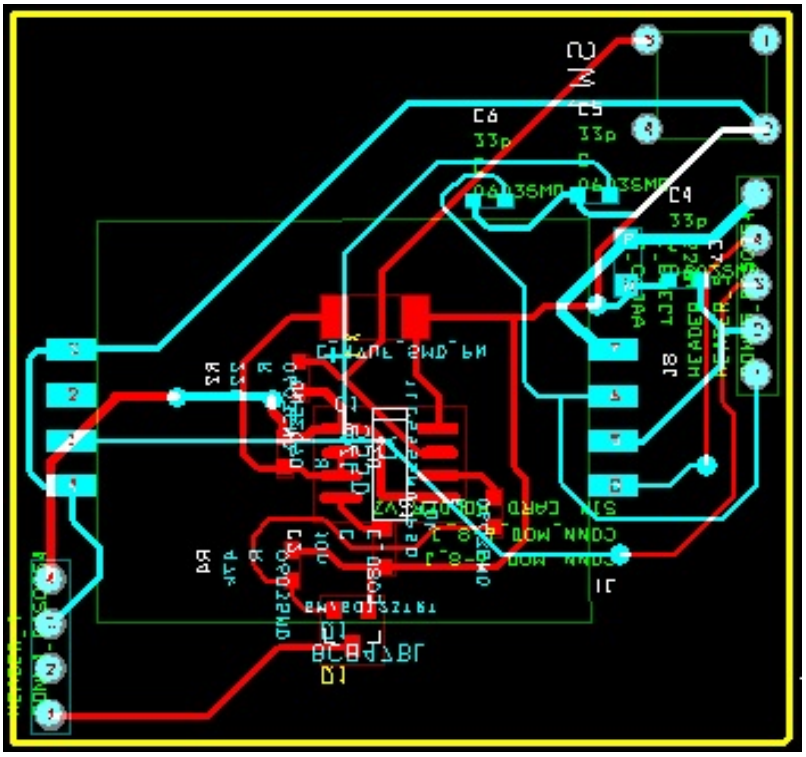


Figura 4.36: Placa inferior con los componentes emplazados y las pistas ruteadas.

Podemos observar que el circuito de encendido se encuentra emplazado sobre la cara inferior, mientras que el circuito de comunicación con la tarjeta SIM se sitúa sobre la cara superior. De esta forma aprovechamos espacio valiéndonos por un lado de que disponemos de versiones SMD de los componentes implicados, y por otro de que ambos circuitos son, en cierta forma, independientes.

A continuación se muestran dos imágenes con las caras superior (imagen de la izquierda) e inferior (imagen de la derecha) de esta placa, ya lista para su fabricación.

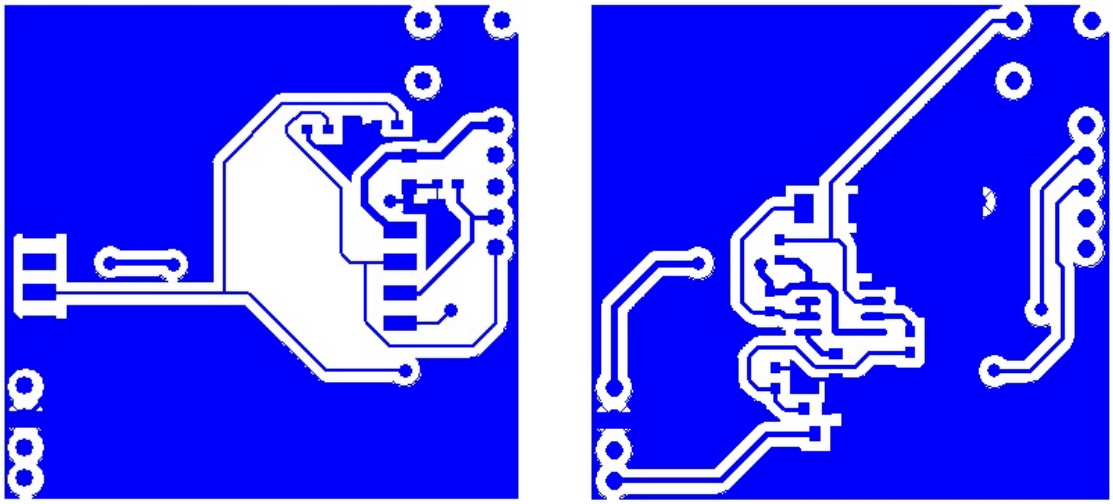


Figura 4.37: Caras superior e inferior de la placa inferior terminada.

Y, finalmente, mostramos una imagen de la placa inferior ya fabricada:

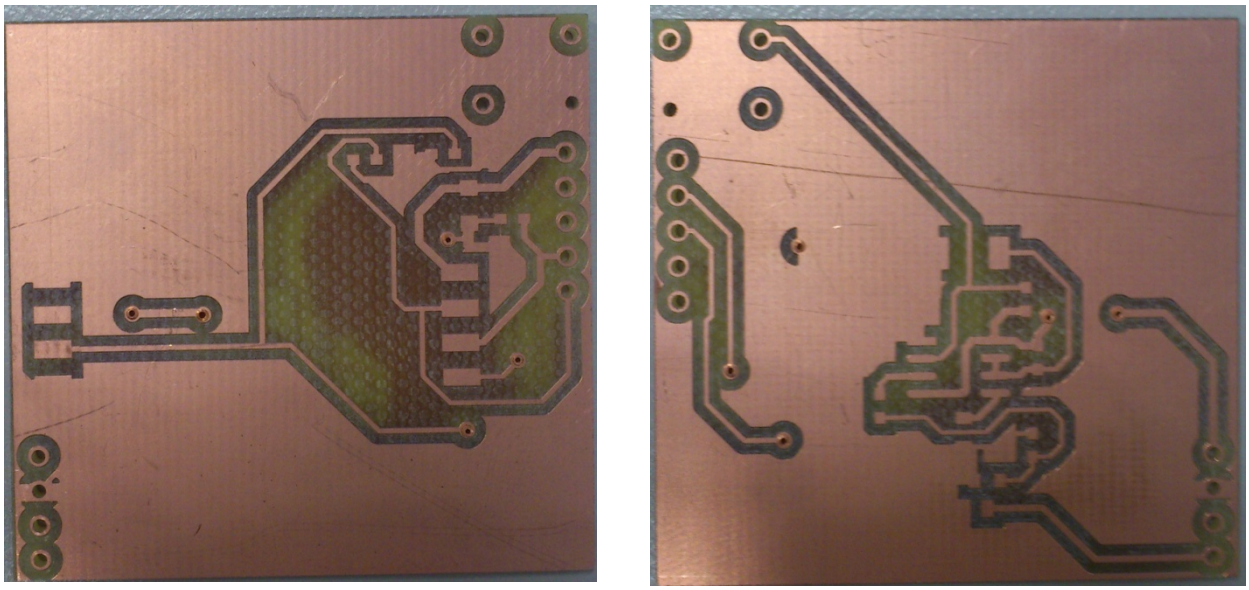


Figura 4.38: Caras superior e inferior de la placa inferior fabricada.

Para poder visualizar el tamaño de esta placa, mostramos a continuación una imagen de ésta frente a una moneda de 50 céntimos de euro:

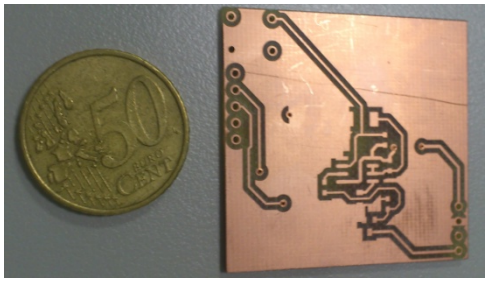


Figura 4.39: Placa inferior frente a moneda de 50 céntimos.

4.5.6 Montaje e Integración

Una vez tengamos las placas fabricadas, lo primero que haremos será colocar todas las vías del diseño con la remachadora del departamento. Una vez tengamos las placas remachadas, procederemos a poner un poco de estaño sobre ambos lados de las vías, con el soldador. De esta manera garantizaremos el contacto entre capas que implementan las vías. Cuando tengamos esto, podremos empezar a soldar los componentes sobre las placas de circuito impreso.

La mayoría de los componentes THD han de soldarse en ambas caras del circuito, a menos que hayamos hecho el ruteo de pistas de tal manera que se conecten al componente por una sola cara. Para facilitar la tarea del soldado de los PCBs emplearemos flux, que es una pasta la cual es una excelente conductora del calor, de esta forma el calor del soldador se repartirá mejor sobre la zona a soldar.

En cada placa, iremos comprobando progresivamente (según vayamos soldando), que el contacto esté bien y no existan derivaciones indeseadas. Para ello emplearemos el multímetro en la función de óhmetro. Iremos comprobando que la resistencia entre puntos que deberían estar cortocircuitados en nuestro diseño sea cero (o el valor residual del propio óhmetro), mientras que entre puntos aislados ésta sea la máxima medible por el aparato.

A continuación mostraremos mediante imágenes el resultado del proceso de montaje de cada uno de los PCB diseñados en este proyecto.

En primer lugar, comenzaremos mostrando el montaje del zócalo para el chip max3218, que fue desarrollado para evitar problemas en el software de Orcad durante el proceso de ruteo de pistas.

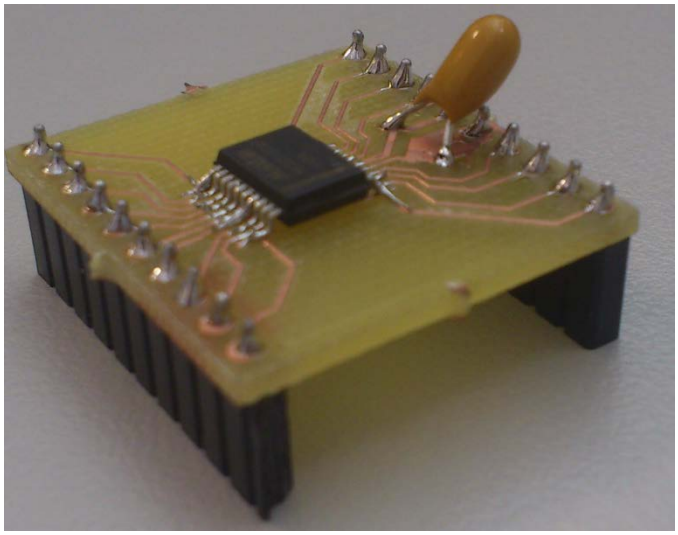


Figura 4.40: Placa/Zócalo para el chip MAX3218.

Como podemos ver arriba, este PCB es aprovechado para integrar el condensador de desacoplo lo más cerca posible de los pines de alimentación y tierra. Antes de proceder al soldado de esta placa, situamos de forma correcta el chip de tal modo que el pin de alimentación esté en contacto con su pista homóloga (se reconoce puesto que es la pista más ancha de esta placa y se conecta al condensador).

Cabe destacar que el soldado del chip SMD de Maxim es una tarea delicada, ya que el pitch (separación entre pines) de este encapsulado es de 0.65mm. Antes de soldar este circuito integrado, emplearemos estaño fino para aplicar flux tanto en las patas del chip, como en la superficie de contacto del PCB con éstas. Una vez aplicado el flux, soldaremos pata a pata de este

chip, de tal manera que no tengamos que pasar con la punta del soldador sobre superficies ya soldadas. Emplearemos la punta más fina del soldador y el microscopio.

Tras haber soldado el encapsulado SMD, procedemos a soldar el condensador de desacoplo. Finalmente, procedemos a soldar las tiras de pines que servirán de conector.

Una vez hayamos soldado esta placa, analizaremos mediante el polímetro el contacto de cada pata del MAX3218 con su respectiva pista, y comprobaremos que no hay interconexión entre pistas vecinas (es decir, que no haya pines vecinos cortocircuitados). De esta manera podemos concluir que las placas están bien soldadas.

Como vimos anteriormente, vamos a necesitar emplear 3 chips MAX3218 para la conversión de niveles lógicos del protocolo RS232 para comunicación del módulo con el PC. Es por ello, que tendremos que construir tres de éstas placas zócalo.

Respecto a la placa inferior del dispositivo final del usuario, el montaje fue realizado también en el laboratorio de desarrollo de circuitos, empleando el microscopio. A continuación se muestran dos imágenes con las caras superior e inferior de esta placa ya montada.

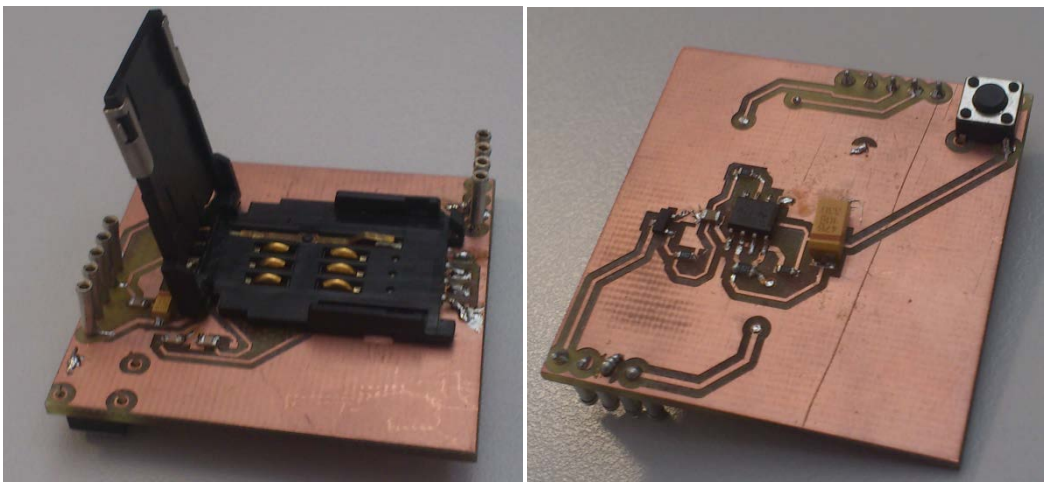


Figura 4.41: Placa inferior del dispositivo de usuario.

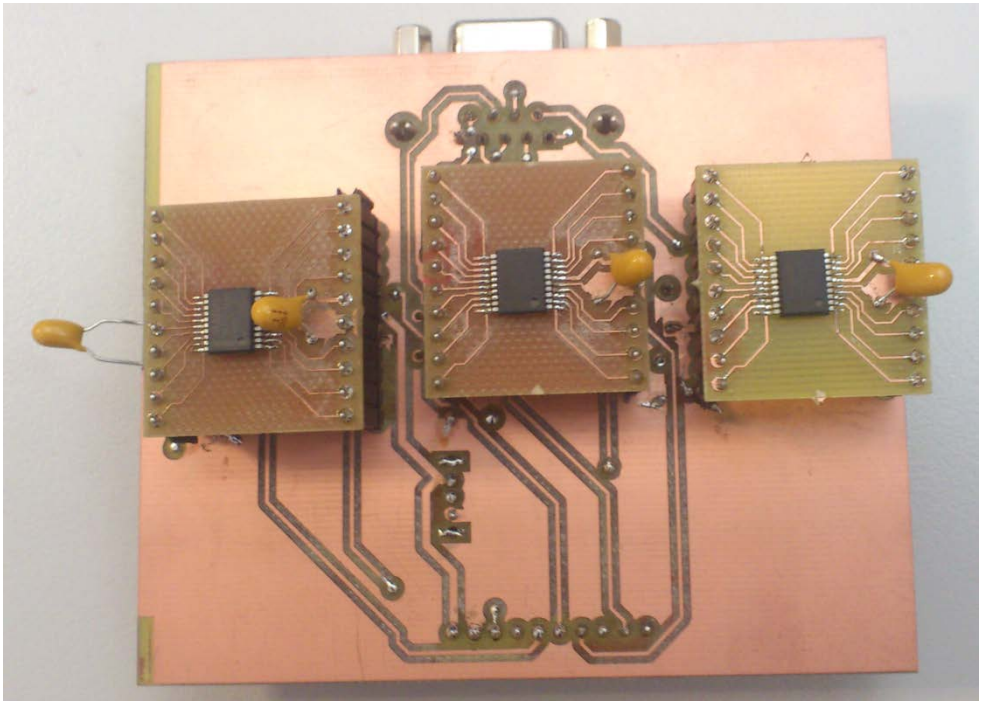


Figura 4.42: Vista superior de la placa para conexión del dispositivo de usuario con el PC.

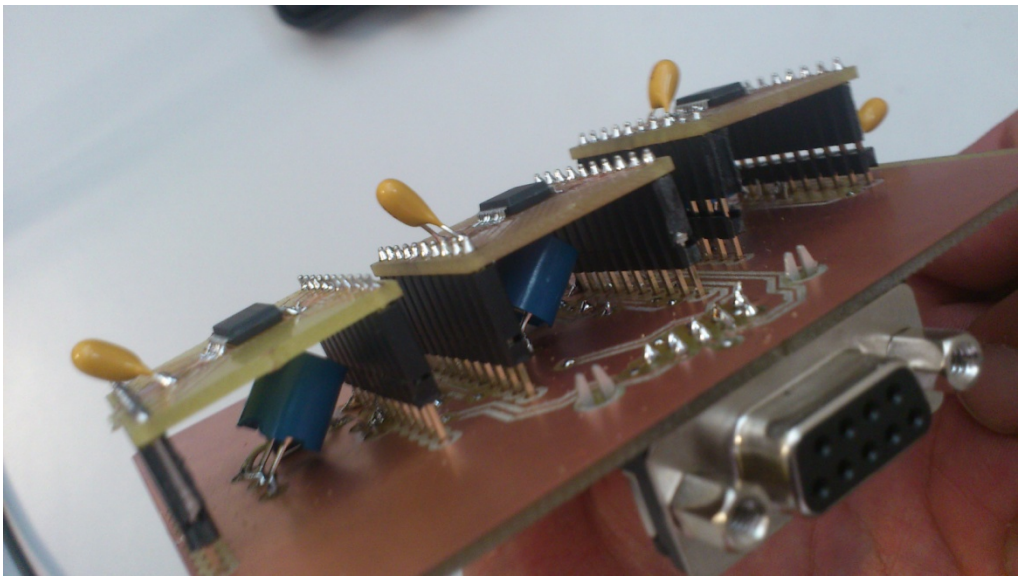


Figura 4.43: Vista lateral de la placa para conexión del dispositivo de usuario con el PC.

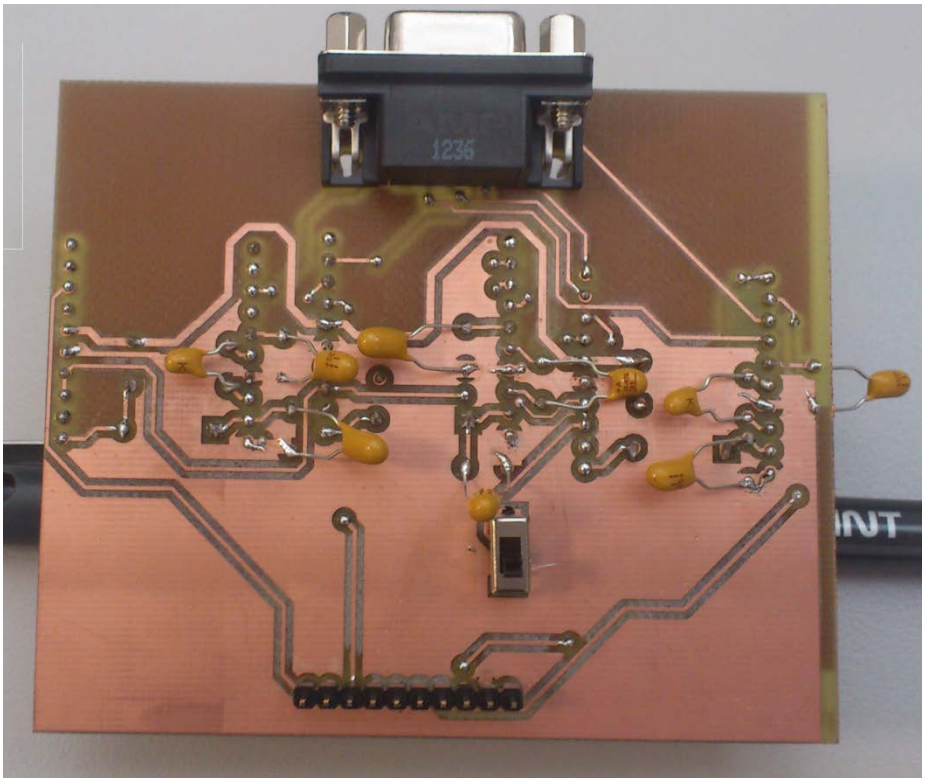


Figura 4.44: Vista inferior de la placa para conexión del dispositivo de usuario con el PC.

Por último, hablaremos del montaje de la placa superior del dispositivo final de usuario. Dicho PCB fue desarrollado y montado por parte de una empresa externa a la UAM. Para el montaje de los elementos SMD y LGA sobre este PCB, se creó una pantalla metálica construida a medida, para depositar una pasta de estaño líquida sobre las zonas de contacto del PCB previo al montaje de los componentes. Sin este procedimiento no podríamos realizar la soldadura de aquellos pads que queden cubiertos por el propio encapsulado.

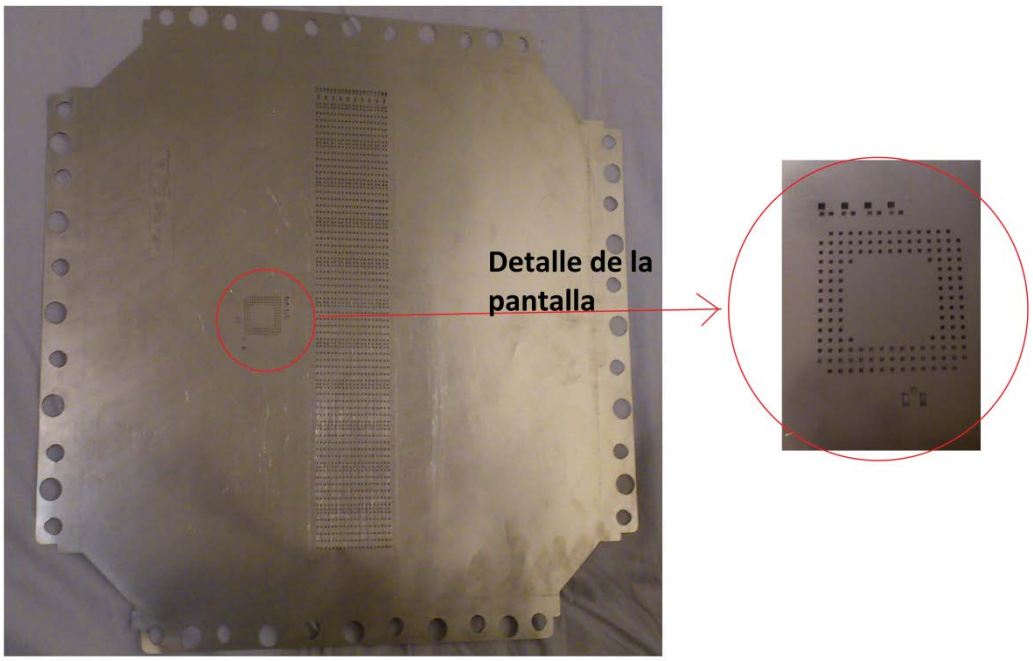


Figura 4.45: Pantalla desarrollada para deposición de la pasta sobre el PCB.

Una vez distribuida esta pasta sobre los puntos de contacto del PCB se procede al emplazamiento de componentes mediante una máquina. Esta misma máquina pasa la placa montada a través de un horno. Este horno ha de configurarse previamente siguiendo una curva de temperatura, en este caso, indicada por el fabricante Telit en su manual hardware. Una vez sale del horno, se deja enfriar un rato. El último paso consistiría en soldar manualmente todos aquellos componentes de tipo THD, tales como el diodo Zener del circuito de protección de alimentación, los leds, etc. Finalmente, de forma opcional, puede procederse a aplicar manualmente una capa de tropicalizado para proteger este PCB. Procedemos ahora a mostrar varias figuras del montaje de esta placa y del dispositivo final completo.

Figura 4.46: Ambas caras de la placa superior una vez montada.

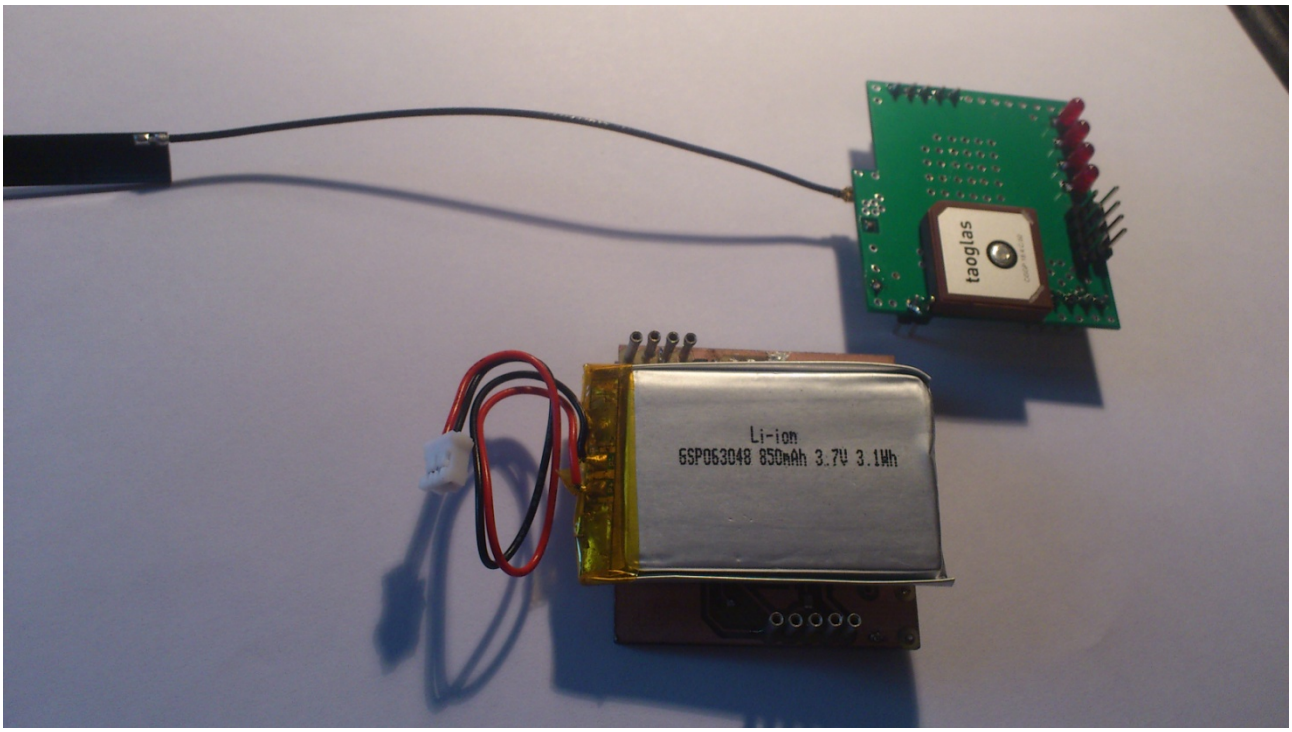


Figura 4.47: Partes del dispositivo final de usuario.

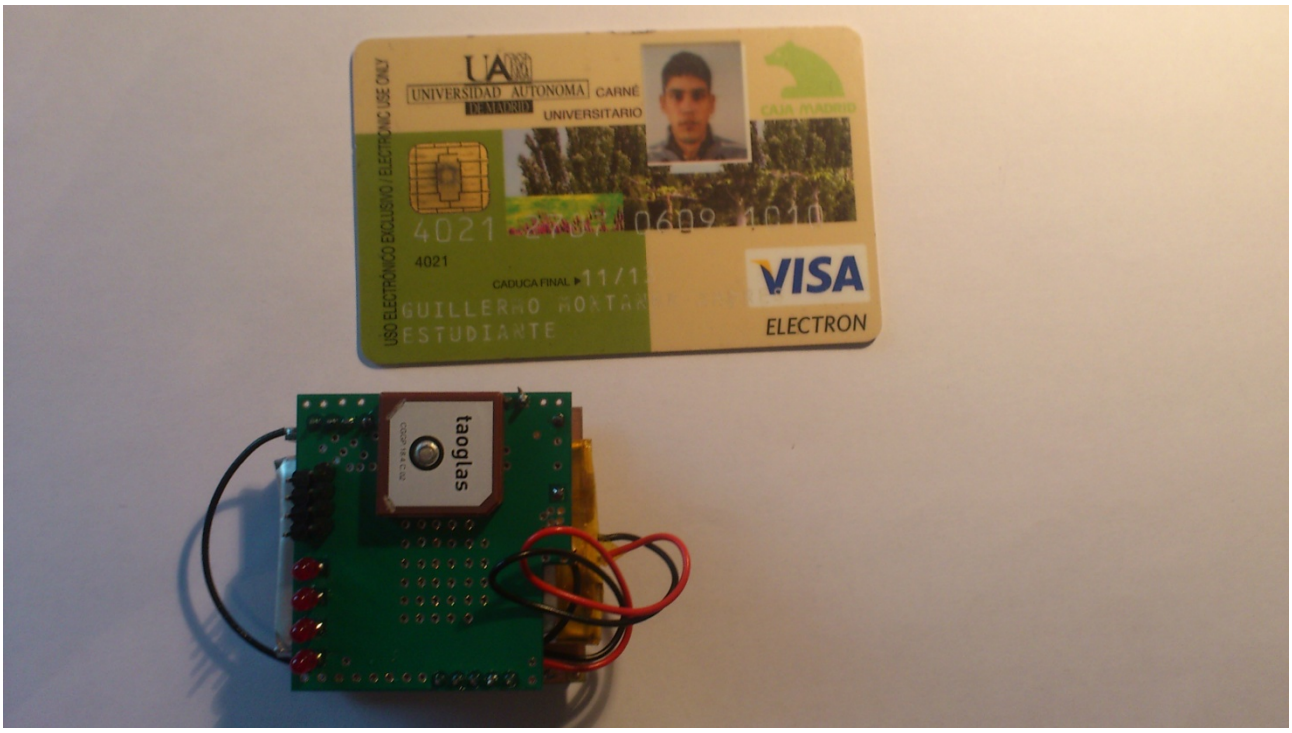


Figura 4.48: Vista superior del dispositivo final de usuario.

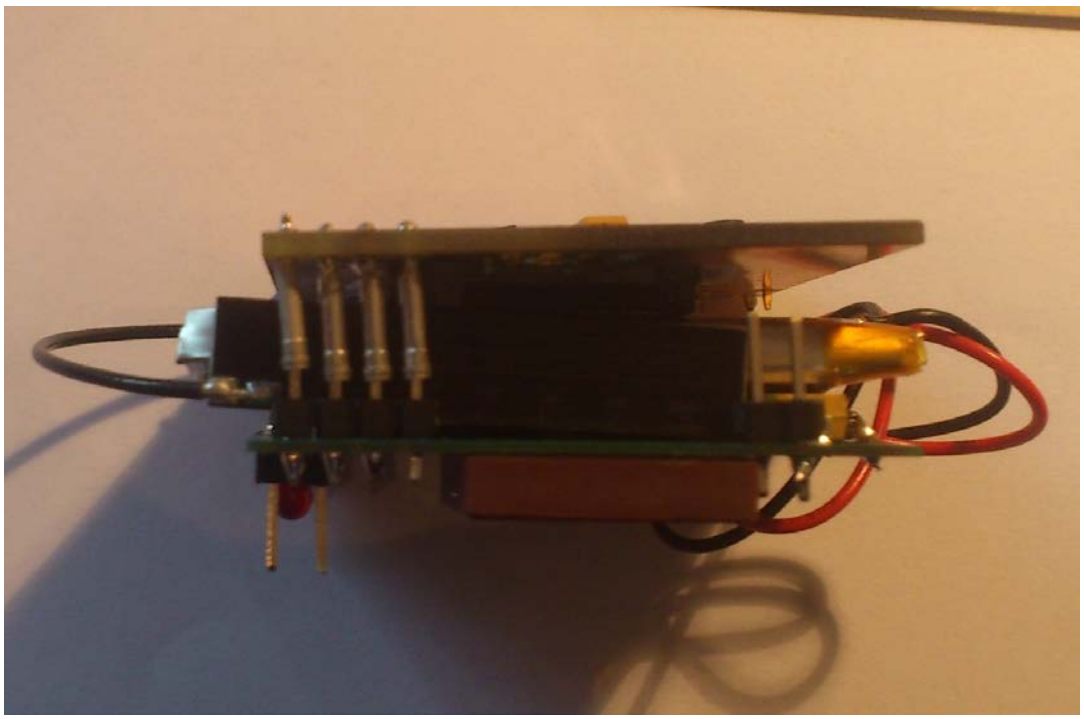


Figura 4.49: Vista lateral del dispositivo final de usuario.

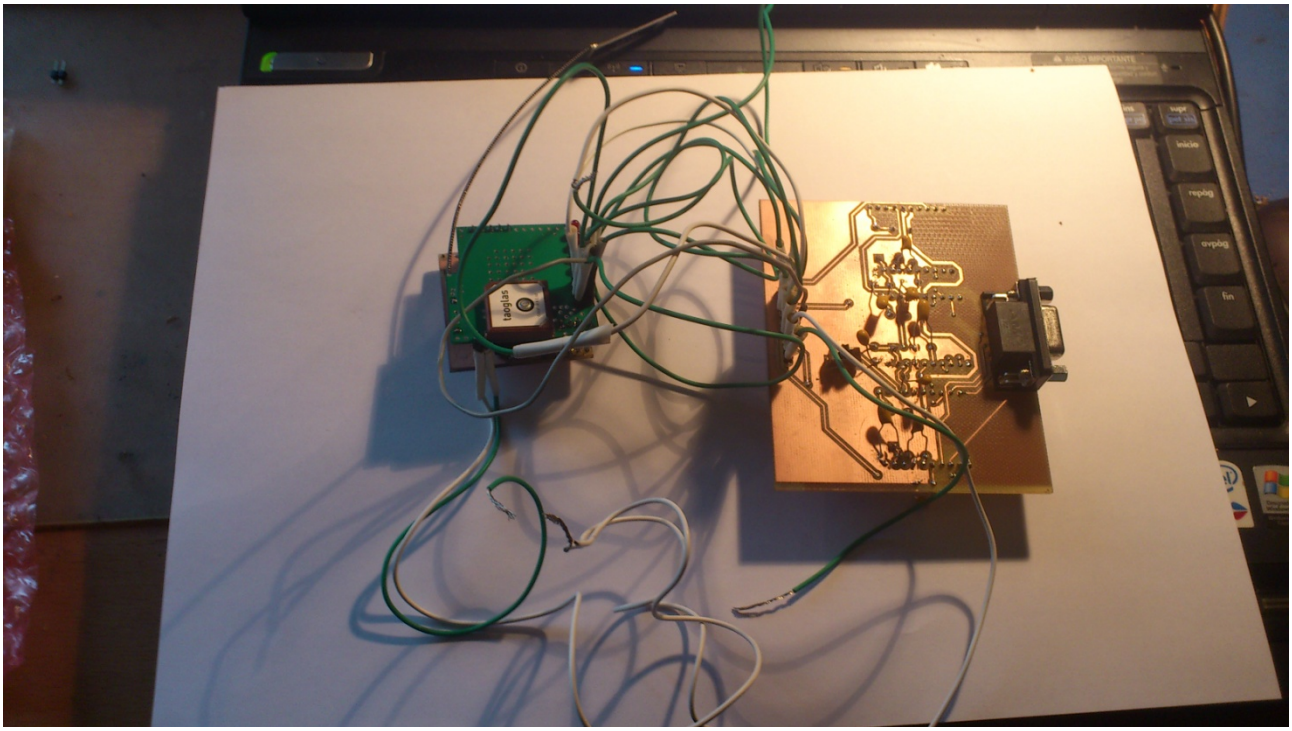


Figura 4.50: Conexionado del dispositivo final de usuario con la placa de comunicaciones con el PC.

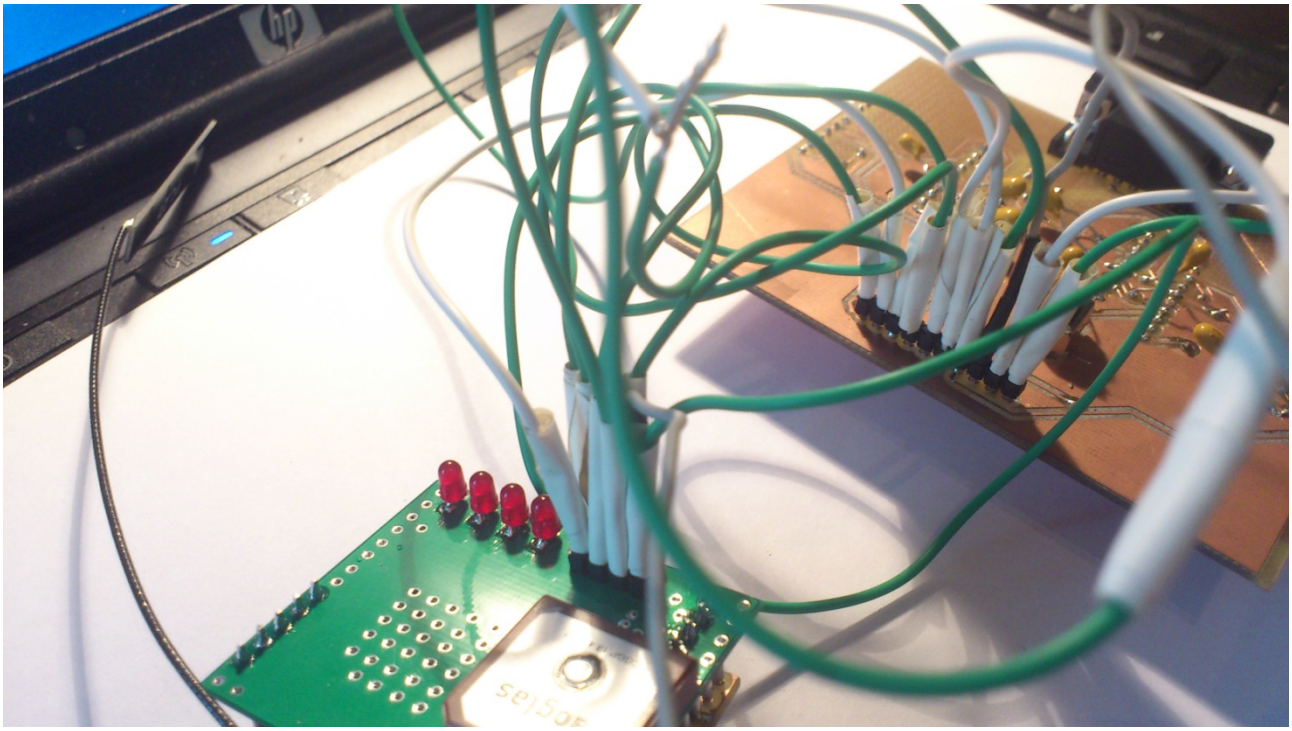


Figura 4.51: Detalle del conexionado del dispositivo final de usuario con la placa PC Com.

4.6 Programación del dispositivo

En este apartado nos encargaremos de terminar el desarrollo del programa final que correrá en nuestro dispositivo. Tenemos aún que desarrollar el código de gestión y manejo del receptor GNSS.

Para comunicar el dispositivo de usuario con el PC, debemos, en primer lugar, montar la placa superior de usuario sobre la inferior. Lo siguiente será conectar las líneas de comunicación con sus homólogas de la placa de comunicación serial. Ponemos el interruptor deslizante de la placa de comunicación serial en la posición de OFF. Ahora, pasamos a interconectar el cable de comunicación serial RS232 con el ordenador y la placa. Hecho esto, procederemos a alimentar al dispositivo de usuario con 3‘7 voltios, mientras que la otra placa será alimentada con 1‘8 voltios. Debemos tener muy en cuenta que ambas placas deben tener la misma referencia de GND, por lo que cortocircuitaremos las líneas de GND de las fuentes empleadas.

Una vez tengamos todas las conexiones del dispositivo, procederemos a encender, en primer lugar, el dispositivo de usuario final, accionando el botón de aviso. Una vez encendido, procedemos a deslizar el interruptor de encendido de la placa de comunicación serial. Llegados a este punto ya podremos abrir el software Hyperterminal de nuestro PC y actuar de la misma manera que hicimos con el EVK2 para comunicarnos con el GE910-GNSS.

Para añadir la funcionalidad GNSS al programa desarrollado en el apartado de desarrollo software, simplemente debemos programar una función que consiga los valores de posicionamiento y HDOP arrojados por el receptor GNSS embebido en el GE910. Esta función será llamada desde el último bucle del cuerpo del programa principal, el cual supervisa el tiempo que toma conseguir el posicionamiento GNSS y el valor HDOP del mismo. A continuación se muestra el código de la función desarrolla:

```
def getGPS():
    global HDOP
    global Coordinates
    a=-1
    while a==-1:
        a=MDM.send('AT$GPSACP\r\n',1)
    pprint('AT$GPSACP\n')
    c=-1
    while c==-1: #Tiempo
        c=MDM.readbyte()
    d=MDM.read()
    pprint(d)
    while 'ERROR' in d: #Tiempo
        a=-1
        while a==-1:
            a=MDM.send('AT#GPSACP\r\n',1)
        pprint('AT#GPSACP\n')
        c=-1
        while c==-1:
            c=MDM.readbyte()
        d=MDM.read()
    e=d.split(',',44)
    if len(e[1])>=2:
        Coordinates=e[1]+e[2]
        HDOP=float(e[3])
    else:
        HDOP=101
```

Figura 4.52: Código de la función encargada de conseguir el posicionamiento y su HDOP.

Podemos apreciar que, para que el programa principal vea modificadas las variables `Coordinates` y `HDOP`, en esta función han de definirse como variables globales. Lo primero que se hace dentro de esta función es proceder al envío del comando `AT$GPSACP`. Al ejecutar este comando, el módulo procede a responder con la información de la última frase NMEA de tipo GGA en un formato propio (y el texto `OK` si todo fue bien, o `ERROR` en caso contrario). Lo siguiente que hay que hacer es proceder a examinar la respuesta del módulo a dicho comando. En caso de encontrar la respuesta `ERROR` procedemos a repetir la emisión del comando. Una vez no encontremos el texto `ERROR`, procedemos a separar la información de la sentencia GGA en una lista; para ello aprovechamos que la información viene separada por el signo ‘,‘ y empleamos el método `split()` de la clase string de Python. Al hacer uso del método `split()`, en la lista devuelta las coordenadas ocuparán el primer y segundo lugar dentro de esta lista, de tal modo, que si la longitud de estos campos no es superior a una unidad (posible espacio en blanco) será debido a que el receptor GNSS aún no ha obtenido un posicionamiento. En caso de no tener un posicionamiento, haremos que el valor de HDOP siga siendo `101` y terminamos con la ejecución. Si se tuviese un posicionamiento, el valor del HDOP aparecería como tercer elemento de la lista devuelta por el método `split()`.

Ahora lo único que habría que hacer sería añadir al cuerpo del programa mostrado en el aparatado de desarrollo software de esta memoria, la llamada a la función `getGPS()` y las líneas encargadas del envío del mensaje de texto con el posicionamiento. Como ya teníamos esto previsto, simplemente debemos localizar en el cuerpo del programa principal los espacios destinados a tal fin, como pudimos observar en el apartado de desarrollo software.

4.7 Pruebas Finales

Con el dispositivo final ya desarrollado, pasaremos a realizar las pruebas pertinentes, para comprobar sus características a la hora de dar un aviso de emergencia. Esto nos permitirá realizar, también, una comparativa con el prototipo, contrastando los resultados las pruebas sobre ambos dispositivos.

4.6.1 Pruebas sobre el receptor GPS

De la misma forma que hicimos con el prototipo de este proyecto, rellenaremos en una tabla de Excel los valores HDOP arrojados por el GE910-GNSS frente al error aproximado de este posicionamiento. Esta tabla será procesada por Matlab, de la misma manera que hicimos con el prototipo del proyecto.

Cabe destacar que todas las pruebas realizadas sobre el receptor GNSS del GE910-GNSS fueron realizadas con el cielo cubierto a más del 50% (terrazas, ventanas), obteniendo unos resultados excepcionales frente a los obtenidos incluso en buenas condiciones por el prototipo del proyecto. A continuación se muestra una gráfica con los resultados del HDOP frente al error obtenido en los diferentes experimentos.

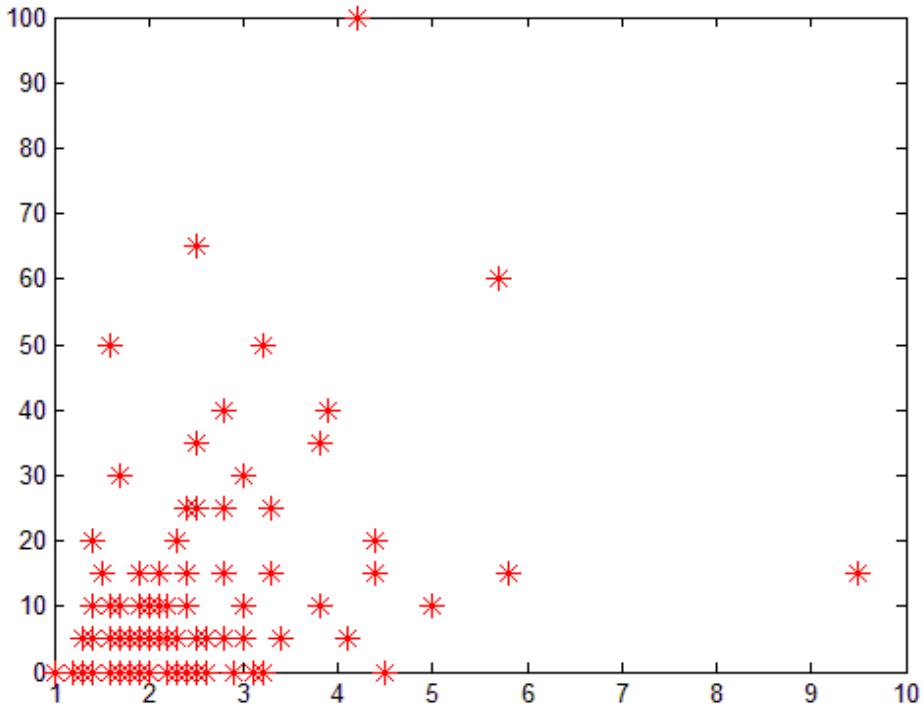


Figura 4.53: Gráfica I con los valores del error cometido frente al HDOP.

Se puede apreciar que con el dispositivo final, trabajando incluso en escenarios medianamente agresivos (dispositivo frente a una ventana cerrada, dispositivo situado en una terraza) el máximo error obtenido en un posicionamiento es de 100 metros, y ocurre en un sólo experimento. Más del 70% de los posicionamientos tienen un error inferior a los 30 metros, independientemente del valor del HDOP obtenido. Mostramos a continuación una imagen de la gráfica anterior, pero ampliando la región de mayor interés.

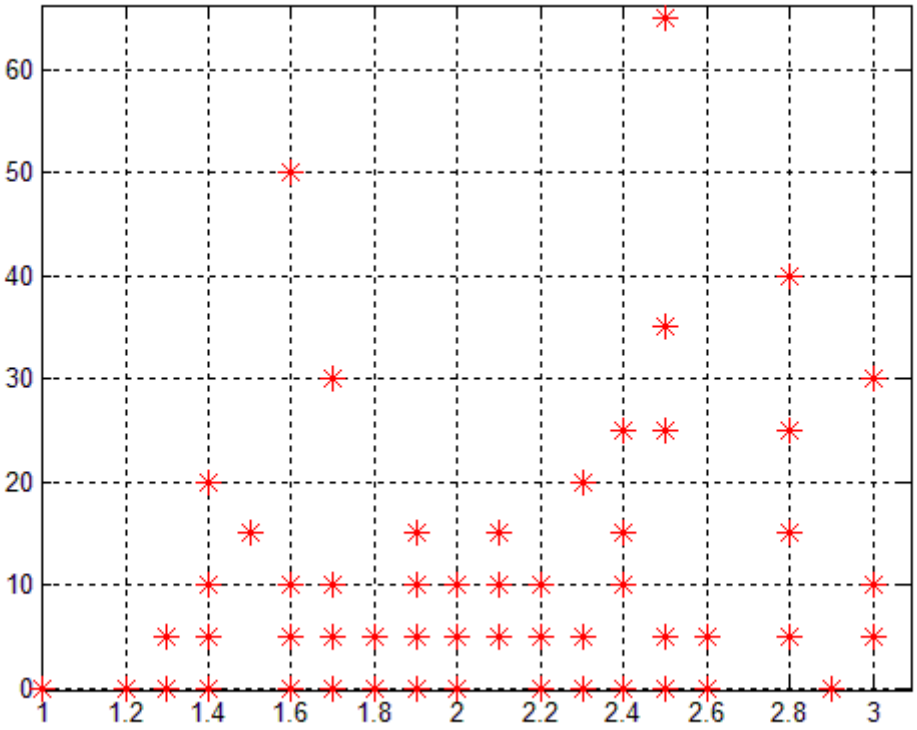


Figura 4.54: Gráfica II con los valores del error cometido frente al HDOP.

En la gráfica ampliada, podemos observar que escogiendo un umbral del HDOP igual a 3 unidades, más del 92% de los resultados tienen un error inferior a los 30 metros. Si escogemos un valor del HDOP igual a 2.5 unidades, para las pruebas realizadas, todos los posicionamientos tendrían un error inferior a los 50 metros.

No obstante, estos resultados hacen que este prototipo no deba usarse en aplicaciones que impliquen avisos de emergencia, ya que la precisión del receptor GPS es bastante dudosa.

4.6. Pruebas de tiempo sobre el dispositivo final

De la misma manera que hicimos con el prototipo del proyecto, dividiremos el aviso de emergencia en diferentes tareas. Estas tareas son: registro de red, obtención de la información de celda, obtención de la información de celdas vecinas, envío de un mensaje de texto y obtención del posicionamiento GNSS. Efectuaremos pruebas sobre dos escenarios distintos, urbano y rural, puesto que para realizar las pruebas de tiempo necesitamos monitorizar el puerto serie del dispositivo (las pruebas en entorno rural con mala cobertura no podrán realizarse).

En aquellas secciones en que se encuentren diferencias de tiempo de unos escenarios a otros se comentarán. En cambio, si no se encuentran diferencias apreciables, los resultados de las pruebas serán mostrados de forma conjunta sin hacer distinciones.

Tiempo necesario para la ejecución del script

En esta primera sección hablaremos del tiempo que tarda el módulo desde que es encendido hasta que se comienza a ejecutar el script que carguemos en él. Para que el módulo ejecute automáticamente un script desde su encendido empleamos el comando AT#STARTMODESCR=1. Como ya vimos en el apartado **4.3.1** (Programación del módulo GE910 mediante el EVK2) el script comenzará a ejecutarse vencido un tiempo de 10 segundos desde su encendido. A la hora de dar el dispositivo al usuario reduciremos el tiempo de guarda lo máximo posible, empleando el comando AT#STARTMODESCR=2 (el script se ejecutará en cada encendido independientemente de que se envíen comandos AT). La razón de emplear el tiempo de guarda es para no perder el control sobre el módulo, en aquellos casos que no queramos ejecutar el script cargado. Cabe destacar que la carga del script representa unos **16 segundos** de retardo, tiempo que podría reducirse drásticamente en caso de cargar scripts precompilados. Debido a que en este proyecto no fuimos capaces de poder ejecutar scripts precompilados, dejamos la reducción de este factor de tiempo para trabajos futuros. En caso de ejecutar un script precompilado, el tiempo previo a su ejecución se reduce simplemente a leer dicho archivo de la memoria del intérprete de Python, por lo que el tiempo se vería reducido a casi **0 segundos**.

Tiempo necesario para el registro de red

En esta segunda sección detallamos el tiempo de conexión, el cual definiremos como el tiempo que tarda en registrarse el dispositivo desde que se inicia la ejecución del programa. Para calcular el tiempo que tarda el dispositivo en registrarse a la red desde que éste es encendido, habrá que sumar al tiempo hallado en esta sección el tiempo de guarda antes de comenzar a ejecutar el script más el tiempo de compilación del mismo (en caso de no haber cargado un script precompilado).

Mostraremos a continuación una tabla con los tiempos de registro en escenarios urbano y rural, puesto que se han encontado diferencias en el valor del tiempo de registro entre ambos casos.

Tabla 4.1: Tiempo de registro de red del dispositivo final

Tipo de escenario	Tiempo Medio (s.)	Desviación (s.)	Tiempo máximo (s.)	Tiempo mínimo (s.)
Urbano	11.41	1.2737	13.27	9.21
Rural	10.04	0.9991	11.25	9.19

A pesar de que las diferencias son mínimas, podemos hablar de que el escenario urbano requiere de algo más de un segundo extra en media de tiempo de conexión frente al escenario rural.

Tiempo necesario para obtener la información de celda

Este tiempo será el empleado en el envío del comando AT#MONI por primera vez y la obtención de una respuesta correcta al dicho comando. Mostramos una tabla con una fila única debido a que no hemos encontrado diferencias significativas entre ambos escenarios

Tabla 4.2: Tiempo de búsqueda de información de celda del dispositivo final

Tiempo Medio (s.)	Desviación (s.)	Tiempo máximo (s.)	Tiempo mínimo (s.)
0.0467	0.0157	0.08	0.03

Tiempo necesario para hacer un 'escaneo' de red en busca de celdas vecinas:

Este tiempo será el empleado en realizar el escaneo completo de red en busca de información de celdas vecinas. El tiempo se medirá como el comprendido entre que se emite (por primera vez) el comando AT#CSURV hasta que se obtiene una respuesta correcta al mismo. Como ya vimos en el prototipo, existirán diferencias sustanciales entre los diferentes escenarios debido al número de estaciones base disponibles en cada uno de ellos (a menor número de estaciones base menor tiempo emplea el dispositivo en realizar el 'escaneo').

Tabla 4.3: Tiempo de búsqueda de información de celdas vecinas del dispositivo final

Tipo de escenario	Tiempo Medio (s.)	Desviación (s.)	Tiempo máximo (s.)	Tiempo mínimo (s.)
Urbano	198.73	58.58	422.97	165.43
Rural	59.54	1.87	63.76	56.32

Como podemos apreciar en la tabla superior, el tiempo medio necesario para obtener la información de celdas vecinas en escenarios urbanos llega a ser más del triple que el empleado en escenarios rurales. Además los escenarios urbanos presentan mayor variabilidad de tiempos entre experimientos, como puede deducirse del valor de la desviación típica.

Tiempo necesario para el envío de un sms

Este tiempo será el empleado desde que queramos enviar un SMS, hasta que el módulo nos responda con la sentencia OK (tiempo comprendido entre la primera emisión del comando AT+CMGS y la respuesta OK a dicho comando). Las diferencias de tiempo entre escenarios no son lo suficientemente significativas, a excepción del caso del mensaje de texto con la información de las celdas vecinas en entornos urbanos. Esto se debe a que en ocasiones esta información ocupa dos mensajes de texto, y por tanto el tiempo de respuesta al comando AT+CMGS se ve aumentado drásticamente (a más del doble que en el caso de mensaje único).

Tabla 4.4: Tiempo de envío de mensaje del dispositivo final

Tipo de mensaje	Tiempo Medio (s.)	Desviación (s.)	Tiempo máximo (s.)	Tiempo mínimo (s.)
Mensaje información de celdas vecinas en entorno urbano	12.02	4.53	24.03	6.77
Resto de SMS	3.69	1.87	18.23	2.88

Los tiempos máximos obtenidos en esta tabla se corresponden con el envío fallido de SMS, lo que supuso un reajuste en el valor del timeout de mensaje de texto del programa final. De hecho estos mensajes conseguían enviarse, por lo que el receptor de los mensajes los recibía duplicados. A partir de dicho reajuste no se "pierden" mensajes de texto, y los tiempos máximos de envío no volvieron a alcanzarse. De esta manera, conseguiremos reducir aún más los valores del tiempo de envío de mensaje con respecto a los de la tabla (excepto, posiblemente, los valores mínimos).

Tiempo necesario para obtener un posicionamiento GPS

Como ya podemos deducir, este tiempo va a ser el que mayor variabilidad va a tener dentro de las pruebas realizadas. En los entornos donde se realizaron las pruebas en mejores condiciones, se obtuvieron los posicionamientos en torno a los 60 segundos desde la ejecución del script. Es por ello, que el tiempo de espera a obtener un posicionamiento GNSS se ha aumentado a 70 segundos desde que se manda el mensaje con la información de la celda de registro. En los mejores casos pudo obtenerse un posicionamiento a los 30 segundos (cold start).

Cronogramas de ejecución de la aplicación

Como hicimos en el capítulo del prototipo, procederemos a mostrar mediante cronogramas varias situaciones típicas de ejecución del aviso de emergencia, valiéndonos de los tiempos antes hallados. Para una mayor visualización los cálculos se realizan para entornos rurales (así evitamos largos trozos de tiempo de búsqueda de información de celdas vecinas). Mostraremos cronogramas para las mismas situaciones que las mostradas con el prototipo, de cara a la futura comparativa de esta memoria (en estos cronogramas suponemos la ejecución automática incondicional de un script precompilado).

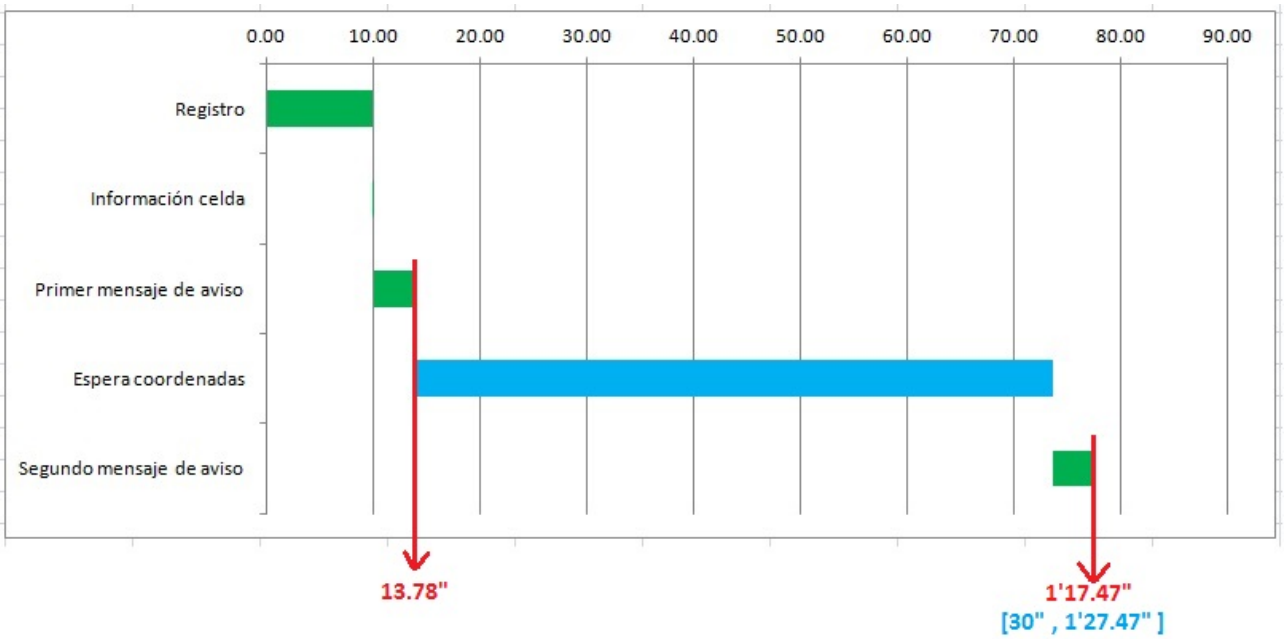


Figura 4.55: Cronograma de una correcta ejecución de la aplicación del dispositivo final.

Como podemos observar en el cronograma arriba mostrado, el tiempo de envío del primer mensaje se produce algo por encima del segundo 13. El segundo mensaje de aviso podría estar disponible desde el segundo 30 hasta 1’ y 27.47” (evitando rebasar el timeout).

Figura 4.56: Cronograma con timeout GNSS del dispositivo final.

En el cronograma superior podemos observar el caso en que se rebasa el timeout GNSS. En este caso se pasa a buscar la información de celdas vecinas, una vez se mande el SMS con dicha información (en torno a los 2’ 27”) se pasará a observar de nuevo la salida del receptor GNSS. El mensaje de aviso con el posicionamiento puede no llegar a enviarse, o tardar como poco 2’ 3.7” en ser enviado.

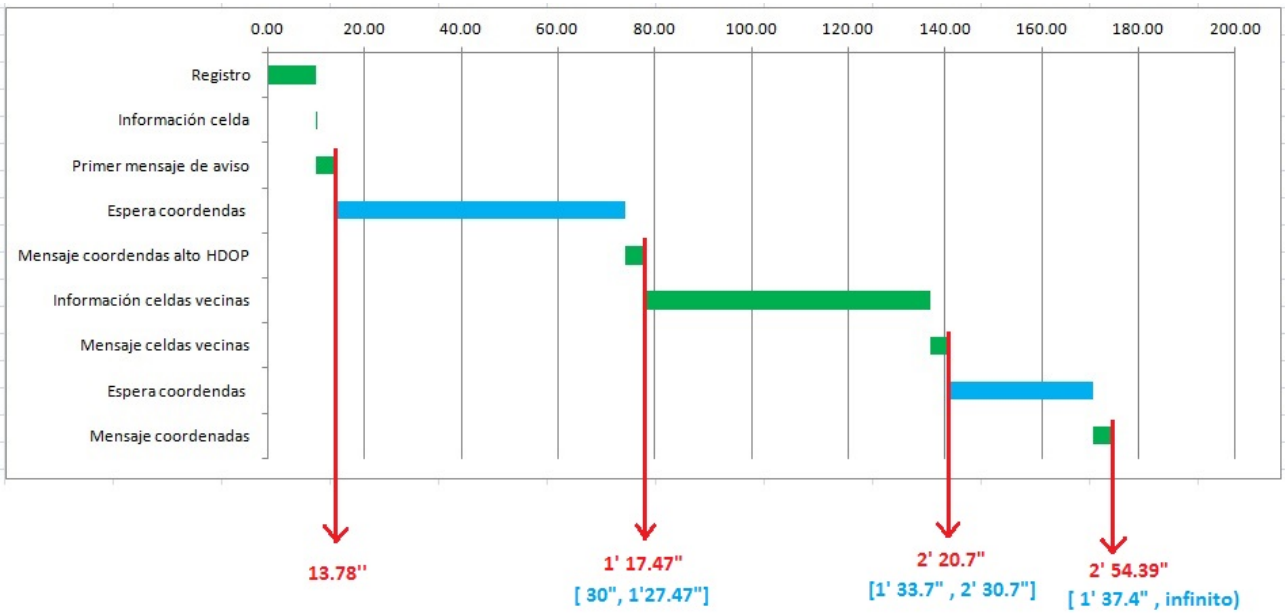


Figura 4.57: Cronograma con fallo precisión GNSS del dispositivo final.

En el cronograma superior puede apreciarse el caso en que se obtiene un posicionamiento GNSS con HDOP elevado.

4.6.3 Pruebas sobre el consumo

Como pudimos observar en el apartado de **Aproximación al consumo eléctrico del prototipo** de este proyecto, la fuente puede ser un buen medio para visualizar la corriente media que requiere el dispositivo en las distintas fases que puede atravesar el aviso de emergencia.

Para realizar una estimación de la corriente media del dispositivo de emergencia, mantendremos a éste realizando avisos de emergencia, y anotaremos para cada fase que atraviese el aviso distintas muestras de los valores de corriente arrojados por la fuente. Como pudimos observar en el apartado homólogo del capítulo del prototipo, el consumo de corriente puede aumentar al realizar una operación hasta más allá del tiempo de respuesta de dicha operación. Esto no va a suponer un problema a la hora de realizar la estimación, puesto que este aumento de corriente será contabilizado en las primeras muestras tomadas sobre el consumo de siguiente fase de la aplicación. Mediante Matlab hallaremos el valor medio de las muestras tomadas de la corriente de cada tarea que realice la aplicación. Para hallar el consumo total de cada fase multiplicaremos el valor medio de la corriente por el tiempo medio de ejecución del peor caso (generalmente urbano).

Tabla 4.5: Consumo de corriente del dispositivo final

Tipo Evento	Corriente media (mA.)	Consumo medio (mAh.)
Registro Red	130.8	0.4360
Registrado	130.8	-
Información de celda	132.5	0.0007
Mensaje	151.4	0.5046
Información celdas vecinas	159.1	5.7010

4.6.4 Comparativa y conclusiones de las pruebas del dispositivo final y del prototipo

Desde que se redactaron las pruebas sobre la relación del HDOP y el error en el posicionamiento GNSS del dispositivo final, se puede observar que este receptor resulta mucho más preciso que el del prototipo. Mientras que el GPS del prototipo ha llegado a arrojar errores de más de un kilómetro, el GNSS del dispositivo final no dio nunca un error superior a los 100 metros. Para visualizar esto mejor, decidimos lanzar una prueba, tanto con el prototipo como con el dispositivo final del proyecto desde la ventana del despacho C-220 (Despacho de Bazil Taha Ahmed). A continuación se muestra un mapa con los resultados de dicha prueba. La cruz roja representa el lugar desde donde se dio el aviso de emergencia. Los marcadores verdes representan la salida del receptor GNSS del dispositivo final de un mismo aviso de emergencia. Por otro lado, los marcadores rojos representan la salida del receptor GPS del prototipo en dos ejecuciones diferentes del aviso de emergencia (desde este despacho no se consigue un segundo posicionamiento con el prototipo). Esta imagen representa lo que sería una salida típica de ambos receptores en una situación con un mal despejamiento de la antena GNSS.

Figura 4.58: Mapa con posicionamientos logrados desde el despacho C-220 con ambos dispositivos.

En la imagen superior puede apreciarse una situación típica en la que, mientras el error del GNSS del dispositivo final se mantiene en torno a varias decenas de metros para valores del HDOP superiores o iguales a 2’5, el error del GPS llega a ser de casi 400 metros en el peor caso (en el mejor caso el error supera los 300 metros).

Respecto a la sensibilidad de los receptores, con el dispositivo final hemos llegado a detectar estaciones base que ofrecían un nivel de potencia de señal de -104 dBm, mientras que con el prototipo se han llegado a detectar estaciones con un nivel de -108 dBm. Por otro lado, cabe destacar, que el prototipo siempre recibe mejor calidad de señal que el dispositivo final (se han observado diferencias de hasta 20 dB con ambos dispositivos bajo el mismo escenario). Esto es debido a varias razones: en primer lugar, la antena del prototipo resulta tener unas características

mucho mejores en términos de ganancia y VSWR que la del dispositivo final (a costa de tener un tamaño mucho mayor); en segundo lugar, es posible que el diseño de la pista de 50 ohmios de conexión del módulo con la antena esté mejor realizada (mejor tratamiento de interferencias, haber realizado un test y tunning tras el montaje, etc.) en el prototipo; y por último, es posible también que el montaje del conector de RF del prototipo resulte mejor que el del dispositivo final.

Con respecto a los tiempos de ejecución de cada una de las fases que puede atravesar el aviso de emergencia, veremos una tabla con los valores de tiempo medios del prototipo así como del dispositivo final en los escenarios rural y urbano.

Tabla 4.6: Comparativa de tiempos del prototipo y del dispositivo final

Situación	Dispositivo Final	Prototipo
Tiempo Registro Urbano	11.41	24.453
Tiempo Registro Rural	10.04	20.676
Tiempo información celda servidora	0.047	0.230
Tiempo SMS	3.69	3.87
Tiempo SMS CSURV	12.02	3.87
Tiempo CSURV Urbano	198.73	129.09
Tiempo CSURV Rural	59.54	57.39

Podemos comprobar que el dispositivo final obtiene unos resultados mejores que el prototipo, a excepción del tiempo de búsqueda de información de celdas vecinas y el envío del mensaje con dicha información en escenarios urbanos. Respecto al aumento del dispositivo final en el tiempo de mensaje de información de celdas vecinas, éste se debe a que el módulo GE910-GNSS tarda más tiempo en devolver la respuesta ‘OK’ que el GE865-QUAD. Esto mismo sucede con el tiempo de búsqueda de información de celdas vecinas.

Es fácil observar mediante los cronogramas que el tiempo en que se mandará el primer mensaje será inferior en el dispositivo final.

En cuanto al consumo de corriente, ambos dispositivos tienen consumos parecidos, siendo algo superior el consumo del dispositivo final, ya que el módulo GE910-GNSS resulta consumir bastante más que el empleado en el prototipo (a pesar de que en el prototipo haya un receptor GPS externo y un microcontrolador, los cuales han de ser alimentados). En cuanto a las baterías empleadas en ambos dispositivos, el dispositivo final, al poder trabajar con 3’7 voltios, puede ser alimentado mediante una única batería de Li-ION, frente a las dos conectadas en serie del prototipo.

Como ya podemos deducir, el dispositivo final resulta bastante más apropiado que el prototipo debido principalmente a la precisión del receptor GNSS. También resulta más adecuado en términos de espacio, y, en general, en los tiempos de aviso.

Capítulo 5
Conclusiones y Trabajo Futuro

5.1 Conclusiones

A pesar de que el sistema GNSS sea el que mayores prestaciones ofrece en términos de localización, hemos visto que la influencia del entrono en los resultados obtenidos es determinante en su precisión. Es por ello, que nunca un sistema de localización de carácter sanitario debe basarse únicamente en éste servicio de localización.

Como hemos podido observar, tanto en el prototipo como en el dispositivo final de este proyecto, el tiempo que se tarda en dar el aviso de emergencia está ligado a numerosos factores. Los factores más influyentes sobre el aviso de emergencia son la cobertura de la red celular (GSM) y el despejamiento acimutal de la antena del receptor GNSS. Sin cobertura celular resultaría imposible dar el aviso de emergencia, y sin cobertura GNSS no se podrán disponer de una localización exacta en un plazo de tiempo razonable (de hecho es posible que no obtengamos posicionamiento GNSS alguno por un tiempo indefinido).

El empleo del comando encargado de buscar la información de las celdas vecinas implica, en la mayoría de casos, una enorme inversión de tiempo. Deberíamos plantearnos la utilidad de este método, pues como vimos en el apartado de posicionamiento mediante telefonía celular, los métodos basados en la triangulación por potencia resultan no ser muy precisos.

A pesar de que algunos avisos fallen en su posicionamiento GNSS, se pueden aprovechar estos fallos para crear una red de “puntos negros” en los cuales puedan, por ejemplo, estimarse mapas de potencia a partir de la información de las celdas vecinas. De esta manera, posteriores avisos en esos emplazamientos podrían atenderse como si se dispusiese de una localización precisa.

5.2 Trabajo Futuro

En este apartado hablaremos de ciertos aspectos de nuestro proyecto que puedan ser mejorados, así como de nuevas funcionalidades posibles.

Por un lado, sería necesario realizar el modelo final de usuario de este dispositivo, ya encerrado dentro de una carcasa adecuada. Por otro lado, el tamaño de este dispositivo podría aún reducirse un tanto, con un montaje más complicado de las antenas y el empleo de una batería más reducida.

Sería muy recomendable conseguir un módulo que no sólo pudiese funcionar en las bandas del servicio GSM, sino, que además permitiese el trabajo en las bandas de la tercera generación (UMTS). De esta forma se vería aumentado en muchos casos el número de BTS al alcance.

La incorporación de una llamada telefónica a la central de alertas podría resultar útil en muchos casos, ya sea tras finalizar la rutina del programa de este proyecto o mientras ésta esté corriendo (el servicio de mensajes es accesible incluso cuando hay llamadas en curso en el terminal que desea acceder). Situaciones en las que la central de atención reciba avisos de emergencia con el paciente consciente y un posicionamiento GNSS no disponible (o con un alto DOP), podrían ser subsanadas en gran medida con esta mejora. En este caso sería el paciente el que indique al servicio de emergencias dónde se encuentra. Para ello, por ejemplo empleando el mismo GE910-GNSS, podríamos emplear los pads de audio (de hecho este módulo permite la posibilidad de elegir entre audio analógico o digital) y conectar un micrófono y un auricular de la forma que recomienda el fabricante. No obstante, este nuevo diseño implicaría un aumento tanto del tamaño del dispositivo como de su complejidad.

También resultaría interesante el uso de un único dispositivo tanto para estar en la calle como en el propio domicilio. Un ejemplo sería hacer que este dispositivo buscase si el otro dispositivo de emergencia que funciona a través del teléfono fijo está al alcance. Para ello sería necesario incorporar a nuestro dispositivo un transmisor o receptor para la detección y comunicación del dispositivo interconectado a la red fija. En caso de estar al alcance, se supondrá que el individuo se encuentra en su domicilio, y se activaría el aviso mediante el servicio de tele-asistencia convencional.

Evaluar de forma rigurosa si es necesario llevar a cabo el desarrollo de un control, manejo y detección exhaustivos de excepciones, para evitar posibles fallos del software encargado de dar el aviso de emergencia. La API de Python cuenta con una clase dedicada exclusivamente a las excepciones, pero su manejo quedó fuera del alcance de este proyecto.

Referencias

[1] **David Abelardo García Álvarez**. *Sistema GNSS.* Enero 2008.

[2] **Jorge R. Rey**. *El sistema de posicionamiento global GPS.* Mayo 2006.

[3] **Luis García-Asenjo Villamayor**. *Aplicación de la programación orientada a objetos al tratamiento de datos GPS. Modelización, desarrollo software y resultados.* Diciembre 2003.

[4] **Diego Antolín Cañada**. *Desarrollo de un sistema de transmisión de datos a larga distancia para WSN mediante un módulo GSM/GPRS.* Junio 2011.

[5] **Emiliano Trevisani, Andrea Vitaletti** *Cell ID location technique, limits and benefits: an experimental study* Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA) 2004.

[6] **Jérôme Soubielle, Inbar Fijalkow, Patrick Duvaut, Alain Bibaut** *GPS Positioning in a Multipath Environment* IEEE transactions on signal processing, Vol. 50, No. 1, pags. 141-150, Enero 2002.

[7] **Eduardo Huerta, Aldo Mangiaterra, Gustavo Noguera**. *GPS. Posicionamiento Satelital.* UNR Editora, 2005.

[8] **José Manuel Huidobro Moya**. *Comunicaciones Móviles. Sistemas GSM, UMTS y LTE.* RAMA EDITORIAL, 2012.

[9] **M. Hernando, F. Pérez-Fontán**. *Introduction to Mobile Communications Engineering,* Artech House 1999.

[10] Evaluation Kits y módulos GSM, GPS y de la compañía Telit:
http://www.telit.com/en/products/gsm-gprs.php

[11] Módulos GSM, GPS y Evaluation Kits de la compañía U-Blox:
http://www.u-blox.com/en/embedded-gps-and-gsm-products.html

[12] Página de la compañía Arduino, microcontroladores montados sobre módulos interfaz:
http://www.arduino.cc/es/

[13] Microchip, tecnología electrónica y microcontroladores:

http://www.microchip.com/

ANEXOS

A - Creación de componentes en Orcad

En cada proyecto de Orcad Capture, deberemos crear aquellos componentes que no se encuentren disponibles en las librerías incorporadas al software. Para crear un componente, podemos crearlo a partir de uno existente, para lo cual, emplazaríamos éste sobre el esquemático, pulsando sobre él con el botón derecho del ratón, y escogiendo la opción `Edit Part`. A continuación se nos abrirá una ventana con dicho componente, en la cual podremos, entre otras cosas y de manera sencilla, cambiar la forma del componente, añadir o eliminar todos los pines que queramos y definir el nombre y número de éstos. Para nuestro diseño hemos necesitado crear un componente para el módulo GE910, la ranura de inserción de la tarjeta SIM, el conector SMA de la antena GSM, y el conector de la alimentación. Mostramos a continuación una imagen del módulo GE910-GNSS.

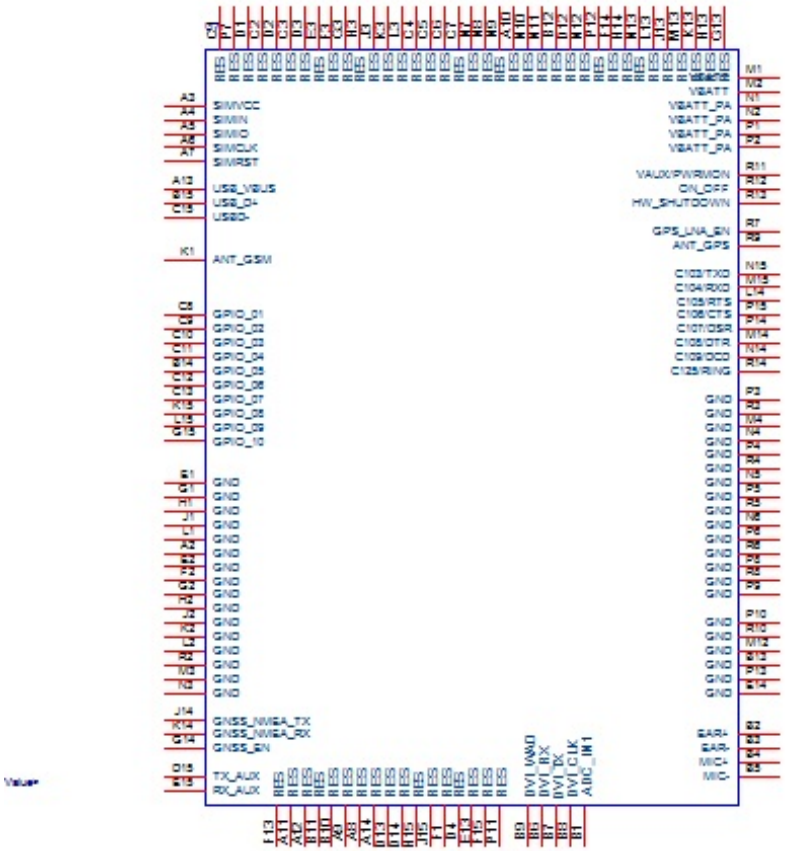


Figura A.1: Creación del componente GE-910 en Orcad Capture.

B - Creación de footprints desde Orcad Layout

Ahora una vez desarrollados todos los componentes necesarios para crear los esquemáticos, pasaremos a crear todos los footprints que no encontremos entre las librerías de footprints de Orcad Layout. Para ver los componentes de estas librerías, ejecutamos el programa Orcad Layout, y pinchamos con el ratón sobre `Tools`, dentro del menú de opciones, escogiendo la pestaña de `Library Manager`. Desde la ventana que se abre ahora, podremos ver y navegar por las librerías de footprints, pudiendo ver todos aquellos disponibles. También podremos crear librerías y footprints desde esta ventana. Para saber si los footprints se ajustan al tamaño requerido, haremos mediciones con la herramienta `Dimension` accesible en el desplegable `Tools` del menú de opciones. Para realizar estas mediciones empleando los milímetros como unidad de medida,

previamente seleccionaremos dentro del desplegable `Options` del menú de opciones, la opción `System Settings`. Hecho esto se nos abrirá una ventana, la cual aparece a continuación.

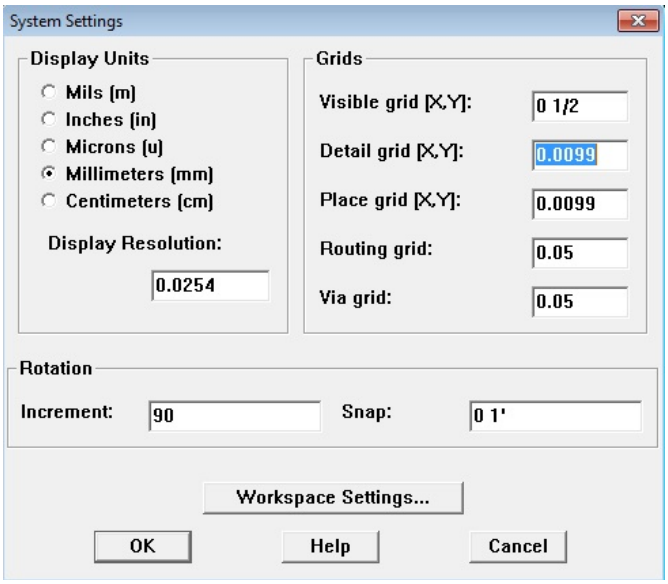


Figura B.1: Ventana de configuración de sistema dentro de Library Manager de Layout.

En esta ventana aparecen las unidades de presentación del programa, los mallados de ruteo (mínima desviación que puede sufrir una pista), de vía (resolución de fondo sobre el cual se puede colocar el centro de una vía), de posicionamiento (resolución sobre la ventana del Library Manager para el posicionamiento de un footprint) y el mallado visible. Lo ajustamos a nuestras necesidades, como podemos observar más arriba. De esta manera, podremos realizar medidas y verificar si un candidato de footprint se ajusta o no a las dimensiones del componente asociado.

En la siguiente imagen se muestra un ejemplo del uso de la herramienta dimensión para verificar si un footprint disponible en una de las librerías se ajusta o no a las medidas especificadas en las datasheets de uno de los componentes.

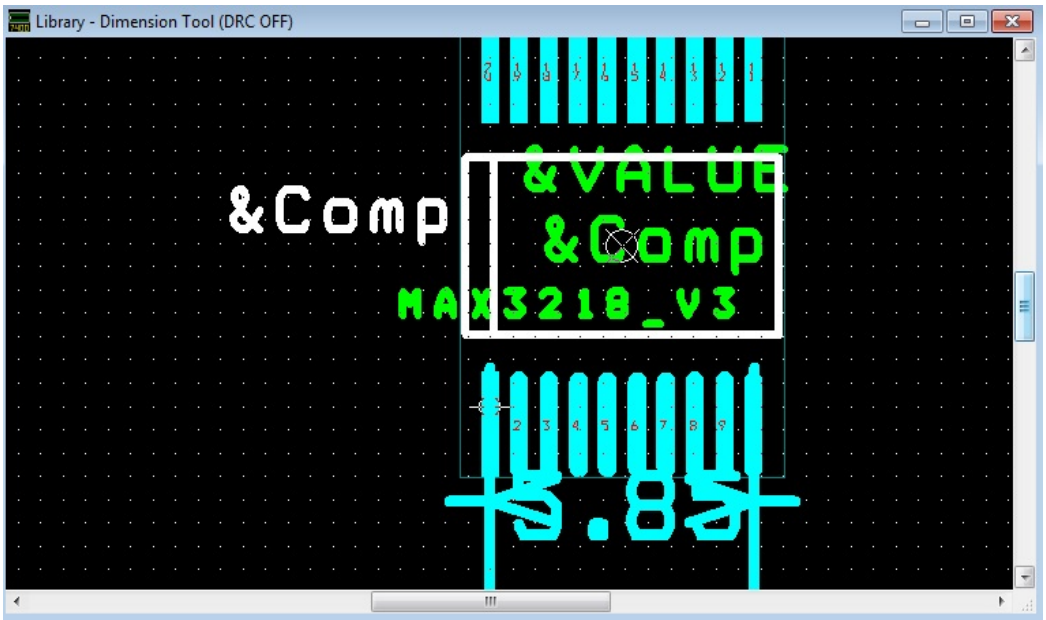


Figura B.2: Midiendo la distancia entre pines extremos.

En la imagen superior podemos apreciar como medimos 5’85 milímetros de separación entre

el pin 1 y el pin 10 de un posible candidato a footprint del chip MAX3218. De hecho, este footprint resulta bastante adecuado para el chip MAX3218, no obstante hubo que modificarlo para que la fresadora de la escuela pudiese construirlo adecuadamente, puesto que los huecos sin cobre entre pines eran menores que el tamaño mínimo de hueco que ésta aceptaba (0.2 mm.).

Por simplicidad, se ha desarrollado una librería con todos los footprints necesarios para el diseño, incluyendo copias de otras librerías. Para crear un nuevo footprint, no hay más que pulsar sobre el botón `Create New Footprint` del panel izquierdo del Library Manager, o modificar uno existente con las utilidades del panel de herramientas disponibles a tal efecto. Para crear un nuevo footprint, debemos también, en ocasiones, crear el pad que éste tendrá. Para la creación de nuevos pads, debemos seleccionar en el desplegable `Tool` (del menú de herramientas) escogiendo `Padstack` y eligiendo `Select From Spreadsheet`. Ahora nos aparecerá una ventana con una tabla que contendrá todos los pads definidos en las librerías, junto con la representación que tendrían en cada una de las capas de un PCB. En el cuadro que aparece pinchamos con el botón derecho fuera de la tabla y seleccionamos la opción `New`. De esta manera se nos abrirá la ventana de creación de nuevos pads, la cual es mostrada en la página siguiente.

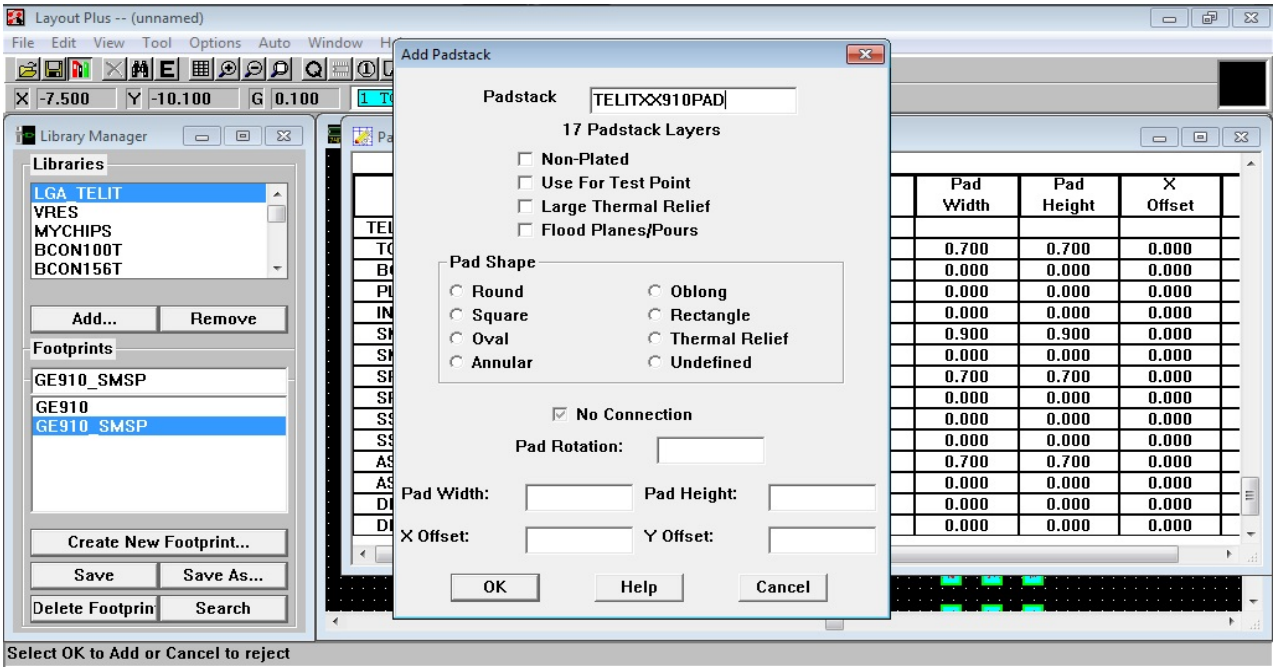


Figura B.3: Ventana para la creación de un nuevo pad.

De ahora, en adelante, enseñaremos el proceso de creación completo con el ejemplo del desarrollo del módulo GE910-GNSS. Como podemos ver en la imagen superior, nombraremos al nuevo pad como 'TELITXX910PAD', y daremos con el ratón a `OK`. En la tabla anterior se habrá añadido un nuevo pad con el nombre definido. En esta tabla, pinchando con el botón derecho sobre cada una de las capas en las que nos interese definir el pad, especificaremos las dimensiones y la forma del pad. Debemos tener en cuenta, que los pads del GE910 aparecerán en una única cara del PCB (al igual que los pads de los chips SMD, BGA y en contraposición a los pads 'through-hole'), por lo que sólo tendrá sentido definirlas en las siguientes capas (para el resto de capas, diremos que el pad es `undefined)::`

1.- TOP: esta es la cara superior del pcb.
2.- SMTOP: máscara de soldadura de la cara superior.
3.- SPTOP: capa superior correspondiente a la pasta de soldadura.
4.- ASYTOP: capa virtual superior de montaje (assembly).

En todas estas capas diremos que el pad es de forma cuadrada con una anchura de 0.7 mm, excepto en la capa SMTOP, en la cual indicaremos que la anchura es de 0.9 mm para indicar que la máscara de soldadura (si ésta se emplease) estuviese a una distancia mínima de 0.1 mm de los bordes del pad, cumpliendo así una de las recomendaciones del fabricante como puede verse, en la siguiente imagen:

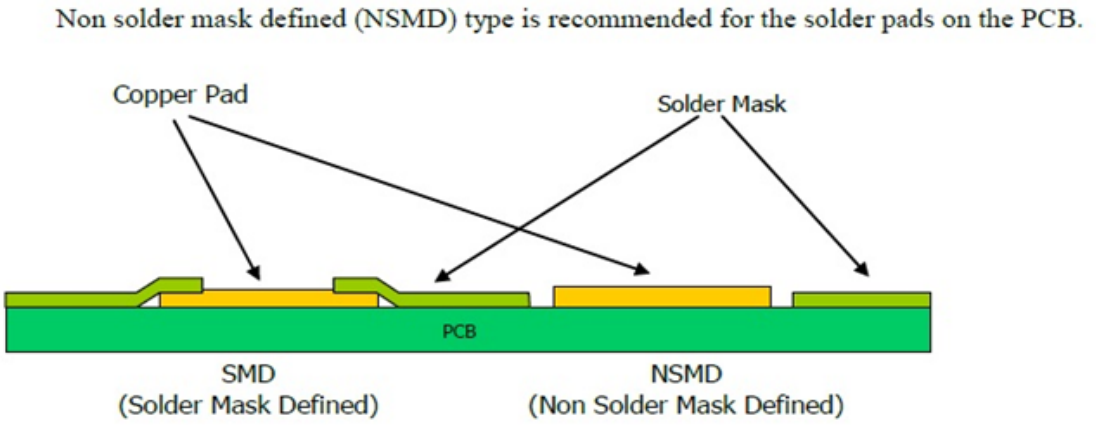


Figura B.4: Recomendaciones del fabricante en cuanto al uso de máscara de soldadura.

Hecho esto pinchamos con el botón derecho sobre el pad ya definido en la tabla y seleccionamos la opción de `Save to Library`, creando una nueva librería si queremos, e indicando donde lo guardaremos.

El siguiente paso para crear el nuevo footprint será emplear la utilidad `Pin tool` del panel de herramientas, para ir emplazando los pads en la posición adecuada y nombrándolos de la misma manera que vemos en el manual hardware de usuario del GE910. En las siguientes imágenes podemos observar el modelo a seguir para la definición del footprint, proporcionado por el fabricante.

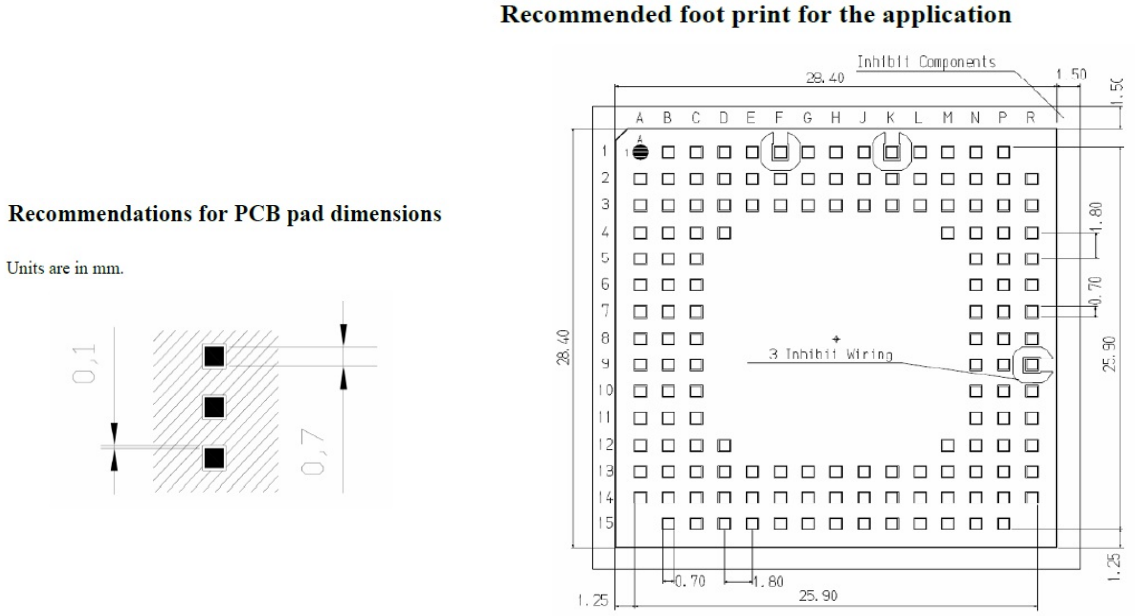


Figura B.5: Esquemas proporcionados por Telit para la creación del footprint del GE910.

Posicionados todos los pads, procedemos a comprobar que las dimensiones se ajustan a las especificadas por el fabricante. Para ello, haciendo 'zoom in', nos aseguramos que todos los pads se encuentran alineados en filas y columnas, y que la distancia entre los pads más extremos es de 25.9 mm.

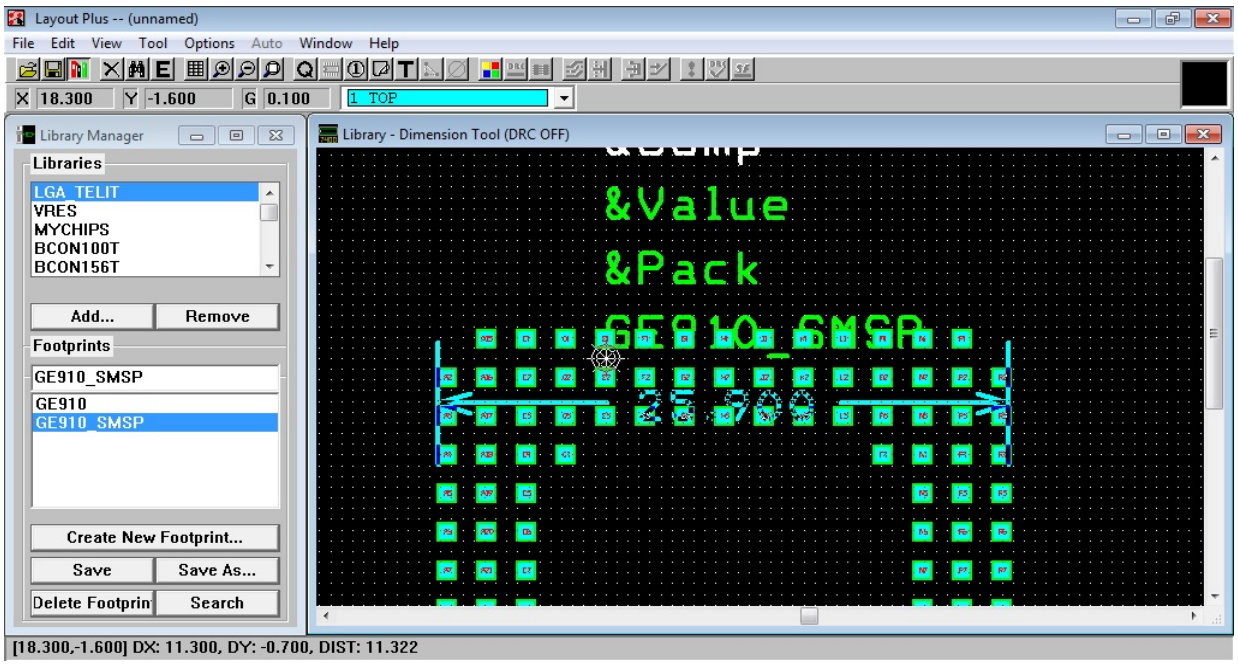


Figura B.6: Comprobación de la distancia entre pads extremos del GE910.

En la imagen superior puede apreciarse que la separación entre pads extremos es la correcta. Continuaremos con nuestra tarea, guardando siempre, cada pocos pasos, el nuevo componente en nuestra librería.

Lo siguiente será definir el borde del chip, para ello seleccionamos sobre `tool` escogiendo `obstacle` y de nuevo eligiendo `select tool`. Definimos una anchura de 0.254 mm para la línea de borde e indicamos que será un obstáculo de tipo 'place outline' (líneas exteriores de posicionado o borde del componente, empleadas para mantener una distancia entre los componentes). Ahora, pulsando con el ratón, dibujamos esta línea de borde. El resultado final puede verse en la siguiente imagen:

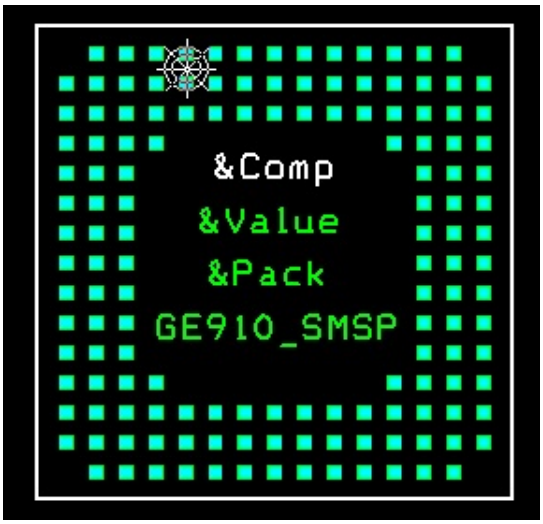


Figura B.7: Resultado final del desarrollo del footprint del GE910.

Repetiremos los mismos pasos descritos con el resto de componentes para los cuales no tenemos un footprint equivalente en las librerías del programa Orcad Layout. Será necesario crear un footprint para cada uno de los siguientes componentes:

a) **La ranura de inserción de la tarjeta SIM:** la versión adquirida de este componente no tiene ninguna documentación asociada, por lo que este footprint se realizó tomando medidas directamente sobre el componente. Además debemos tener especial cuidado con el mapeo de los pines de este componente si no queremos cometer un fallo en nuestro diseño que imposibilite la comunicación del GE910 con la tarjeta SIM. Para ello, debemos tener claro cuál es la disposición de los pads de una tarjeta SIM estándar. Una vez conozcamos dicha disposición, tendremos en cuenta cómo se inserta esta tarjeta dentro de la ranura, para asociar cada pin de la ranura con un pad de la tarjeta SIM. Finalmente, analizaremos la disposición de las conexiones en el esquemático de este circuito, para asociar cada señal con un número de pin. Esta será la numeración de los pines que aplicaremos al footprint.

   A continuación se muestra el footprint desarrollado para este componente.

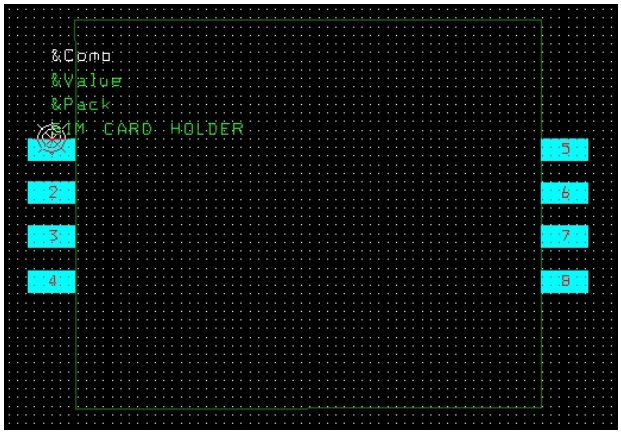


Figura B.8: Footprint de la ranura de inserción de la tarjeta SIM.

b) **El conector RF de la antena GSM**. Este conector es de tipo coaxial en miniatura, denominado UFL, y se emplea en aquellos diseños donde el tamaño resulta un punto crucial (este conector ocupa apenas 3 mm^2). Los conectores para soldado al PCB son los macho, y disponen de tres pads (uno de RF y dos de tierra).

c) **La antena GNSS:** la cual posee dos pads, uno encargado de llevar la señal de RF y otro que se interconectará a masa. En cuanto a la forma de estos pads, el de RF consiste en un pin que se soldará en la cara opuesta de donde sea montada la antena; mientras que el pad de tierra tendrá una forma de anillo en torno al pin de RF y se conectará en la cara donde sea montada la antena. Inicialmente desarrollamos primeramente un footprint con dos pines, uno encargado de la señal de RF y el otro de masa. El pin de RF es definido como un pin circular cuyo diámetro es el mismo que el de una pista de 50 ohmios (ver más adelante en el apartado de la placa superior), cuya conexión se realiza sobre la cara inferior, y posee un taladro de . El pad de masa se define como un pad SMD (sin tala-

dros) dispuesto en la cara superior. Para emular la forma de anillo de este pad se decide crear un objeto de tipo `copper area` interconectado al pin de masa, creando por tanto un 'anillo rectangular'. Para concluir este componente, definimos un objeto de tipo `Anti-copper` en torno al pin de RF, para evitar que el plano de masa pueda caer dentro de la zona de este pin. A continuación mostramos una imagen del footprint desarrollado para este componente.

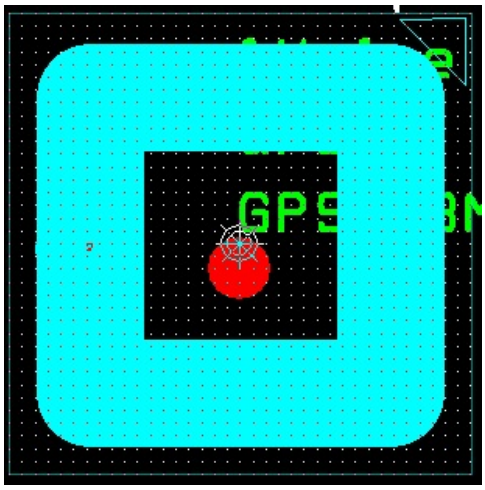


Figura B.9: Footprint de la antena GNSS.

Cabe destacar que la conexión del pad de tierra de este componente (color azul verdoso en el dibujo superior) se realiza con un adhesivo que éste lleva incorporado, por lo que no necesita ser soldado.

d) **Los diferentes encapsulados SMD (tipos 0603 y 0805) de resistencias y condensadores cerámicos:** para la creación de estos footprints dibujamos dos pads cuya separación sea la misma que la del componente, pero con un área de contacto mucho mayor. De esta forma, a la hora de soldar el componente, el cobre sobresaldrá por debajo del pad del componente, simplificando el proceso de soldado de estos minúsculos componentes (el 0603 apenas llegar a medir dos milímetros de largo).

Figura B.10: Footprint encapsulado SMD 0603.

e) **Los encapsulados SMD de los condensadores electrolíticos de Tántalo:** seguimos el mismo proceso que el descrito para el desarrollo de los encapsulados SMD de resistencias y condensadores cerámicos.

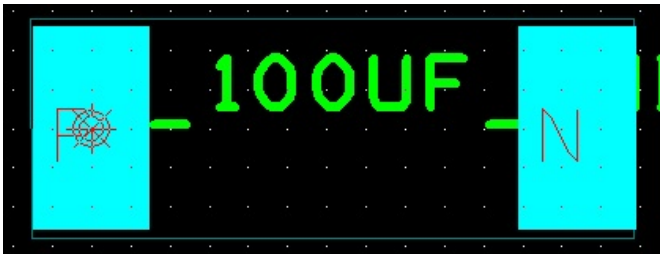


Figura B.11: Resultado final del desarrollo del footprint del condensador de 100 uF.

f) **Los componentes pasivos tales como los condensadores cerámicos y de Tántalo, así como la bobina de 15 microhenrios y los LEDs, en versiones THD radial.** Estas footprints constan de dos pads tipo THD con una separación suficiente para montar los dispositivos que representan sin forzar sus patas en caso de soldarse 'pegados' al PCB. La línea de borde dejará encerrada toda el área que pueda ocupar el componente. A continuación mostramos imágenes del LED, el condensador THD de Tántalo y la bobina de 15 uH. (de izquierda a derecha, respectivamente).

Figura B.12: Footprints de los diferentes componentes THD radiales.

g) **El diodo Schottky para el circuito de comunicación con el PC:** componente que nos exige el chip MAX3218 al ser usado con un bajo voltaje de alimentación. Para este componente THD de tipo axial (una pata de conexión en cada extremo) situamos dos pads THD lo suficientemente alejados como para que a la hora de soldarlo queden alejados de los extremos del dispositivo. De esta forma no forzaremos mecánicamente el componente y nos resultará más fácil de soldar, al tener un mayor despejamiento.

h) **Los diferentes conectores de paso 2'54:** desarrollados, simplemente, colocando pines con la separación adecuada y rodeándolos con un objeto de tipo borde de componente lo suficientemente grande. A continuación mostramos una imagen del footprint del conector de 10 pines.

Figura B.13: Footprint del conector de 10 pines de paso 2'54 mm.

i) **El interruptor de encendido de la placa para comunicación con el PC:** este componente posee cinco pines, dos de los cuales (situado uno en cada extremo) sirven únicamente como fijación mecánica. El pin nº 2 se cortocircuitará con el pin 1 ó 3 según se posicione el elemento deslizante del interruptor.

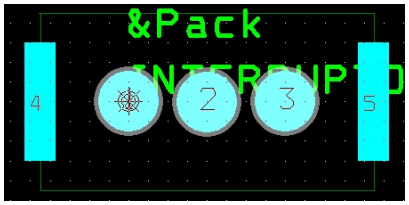


Figura B.14: Footprint interruptor deslizante.

j) **El pulsador de aviso de emergencia (ó pulsador de encendido):** el cual dispone de cuatro pines, cortocircuitados dos a dos (pines 1 y 3 por un lado, y 2 y 4, por otro).

Figura B.15: Footprint del pulsador.

k) **Las vías:** en nuestro diseño emplearemos unos remaches de 0’4mm de diámetro interior, cuyo diámetro exterior es de 0’6 mm. Para definirlas simplemente debemos crear un nuevo pad para ellas. Para ello definimos un pad THD redondo de diámetro 1’2 mm con unos taladros de 0’7 mm de diámetro (si definimos los taladros de 0’6 mm. tendremos que forzar el PCB o el remache para que éste entre). Guardamos dicho pad bajo el nombre de VIA2.

Finalmente, para una verificación final de que todos los footprints desarrollados se ajustaban al tamaño requerido, imprimimos, a través de un mismo proyecto de Orcad Layout, una muestra de cada elemento desarrollado. El programa Orcad Layout es capaz de generar archivos de impresión con la imagen del proyecto (imagen del PCB) a escala 1:1. También podremos imprimir el tamaño de los taladros de aquellos pines THD. De este modo, podremos posicionar los componentes sobre un papel y ver si el footprint asociado se ajusta de la forma deseada.

C - Datasheets y manuales empleados en el desarrollo HW del proyecto

Chip MAX3218 de Maxim

19-0380; Rev 0; 3/95

MAXIM

1µA Supply Current, 1.8V to 4.25V-Powered RS-232 Transceiver with AutoShutdown™

MAX3218

General Description

The MAX3218 RS-232 transceiver is intended for battery-powered EIA/TIA-232E and V.28/V.24 communications interfaces that need two drivers and two receivers with minimum power consumption from a single low-voltage supply. It provides a wide +1.8V to +4.25V operating voltage range while maintaining true RS-232 and EIA/TIA-562 voltage levels. The MAX3218 runs from two alkaline, NiCd, or NiMH cells without any form of voltage regulator. A guaranteed 120kbps data rate provides compatibility with popular software for communicating with personal computers.

Supply current is reduced to 1 µA with Maxim's new AutoShutdown™ feature. When the MAX3218 does not sense a valid signal level on the receiver inputs, the on-board power-supply and drivers shut down. This occurs if the RS-232 cable is disconnected or if the transmitters of the connected peripheral are turned off. The system turns on again when a valid level is applied to either RS-232 receiver input. As a result, the system saves power without changes to the existing software. Additionally, the MAX3218 can be forced into or out of shutdown, under logic control.

While shut down, all receivers can remain active or can be disabled under logic control, permitting a system incorporating the CMOS MAX3218 to monitor external devices while in low-power shutdown. Three-state drivers are provided on both receiver outputs so that multiple receivers, generally of different interface standards, can be on the same bus. The MAX3218 is available in 20-pin DIP and SSOP packages.

Applications

Battery-Powered Equipment
Subnotebook Computers
PDAs
Hand-Held Equipment
Peripherals
Cellular Phones

™ AutoShutdown is a trademark of Maxim Integrated Products.

Features

BETTER THAN BIPOLAR!

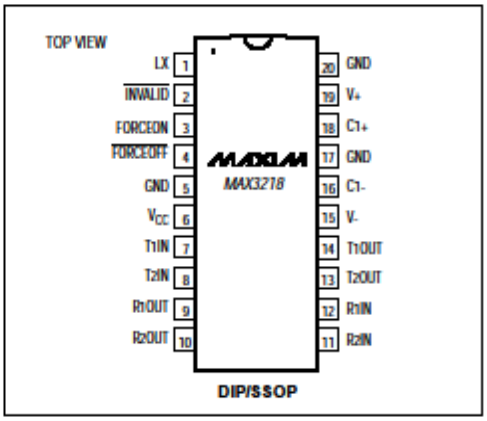
- **1µA Supply Current Using AutoShutdown™**
- **Operates Directly from Two Alkaline, NiCd or NiMH Cells**
- **+1.8V to +4.25V Single-Supply Voltage Range**
- **120kbps Data Rate Guaranteed**
- **Low-Cost Surface-Mount Components**
- **Meets EIA/TIA-232E Specifications**
- **Three-State Receiver Outputs**
- **Flow-Through Pinout**
- **On-Board DC-DC Converters**
- **20-Pin SSOP and DIP Packages**

Ordering Information

PART	TEMP. RANGE	PIN-PACKAGE
MAX3218CPP	0°C to +70°C	20 Plastic DIP
MAX3218CAP	0°C to +70°C	20 SSOP
MAX3218C/D	0°C to +70°C	Dice†
MAX3218EPP	-40°C to +85°C	20 Plastic DIP
MAX3218EAP	-40°C to +85°C	20 SSOP

† Contact factory for dice specifications.

Pin Configuration



Call toll free 1-800-998-8800 for free samples or literature.

1µA Supply Current, 1.8V to 4.25V-Powered RS-232 Transceiver with AutoShutdown™

ABSOLUTE MAXIMUM RATINGS

Supply Voltages
- V_{CC} ..... -0.3V to +4.6V
- V+ ..... (V_{CC} - 0.3V) to +7.5V
- V- ..... +0.3V to -7.4V
- V_{CC} to V- ..... +12V
- LX ..... -0.3V to (1V + V+)

Input Voltages
- T_IN, FORCEON, $\overline{FORCEOFF}$ ..... -0.3V to +7V
- R_IN ..... ±25V

Output Voltages
- T_OUT ..... ±15V)
- R_OUT ..... -0.3V to (V_{CC} + 0.3V)

Short-Circuit Duration, R_OUT, T_OUT to GND ..... Continuous

Continuous Power Dissipation (T_A = +70°C)
- Plastic DIP (derate 11.11mW/°C above +70°C) 889mW
- SSOP (derate 8.00mW/°C above +70°C) 640mW

Operating Temperature Ranges
- MAX3218C_ P ..... 0°C to +70°C
- MAX3218E_ P ..... -40°C to +85°C

Storage Temperature Range ..... -65°C to +150°C

Lead Temperature (soldering, 10sec) ..... +300°C

Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated in the operational sections of the specifications is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

ELECTRICAL CHARACTERISTICS

(Circuit of Figure 1, V_{CC} = 1.8V to 4.25V, C1 = 0.47µF, C2 = C3 = C4 = 1µF, L1 = 15µH, $T_A = T_{MIN}$ to T_{MAX}, unless otherwise noted. Typical values are at V_{CC} = 3.0V, T_A = +25°C.)

PARAMETER	CONDITIONS		MIN	TYP	MAX	UNITS
DC CHARACTERISTICS						
Operating Voltage Range			1.8		4.25	V
Supply Current, AutoShutdown™	V_{CC} = 3.0V, T_A = +25°C, all R_IN open, FORCEON = GND, $\overline{FORCEOFF}$ = V_{CC}			1.0	10	µA
Supply Current, Shutdown	$\overline{FORCEOFF}$ = GND, T_A = +25°C, V_{CC} = 3.0V			1.0	10	µA
Supply Current, AutoShutdown™ Disabled	FORCEON = $\overline{FORCEOFF}$ = V_{CC} = 3.0V, no load			2.0	3.0	mA
LOGIC INPUTS AND RECEIVER OUTPUTS						
Input Logic Threshold Low	T_IN, FORCEON, $\overline{FORCEOFF}$				0.33 x V_{CC}	V
Input Logic Threshold High	T_IN, FORCEON, $\overline{FORCEOFF}$		0.67 x V_{CC}			V
Input Leakage Current	T_IN, FORCEON = $\overline{FORCEOFF}$ = 0V or V_{CC}			0.001	±1	µA
Output Voltage Low	R_OUT, I_{OUT} = 1.0mA				0.4	V
Output Voltage High	R_OUT, I_{OUT} = -0.4mA		V_{CC} - 0.25	V_{CC} - 0.08		V
Output Leakage Current	R_OUT, 0V ≤ R_OUT ≤ V_{CC}, FORCEON = $\overline{FORCEOFF}$ = 0V			0.05	±10	µA
AUTOSHUTDOWN (FORCEON = GND, $\overline{FORCEOFF}$ = V_{CC})						
Receiver Input Thresholds, Transmitters Enabled	Figure 4a	Positive threshold			2.8	V
		Negative threshold	-2.8			
Receiver Input Thresholds Transmitters Disabled	1µA supply current, Figure 4a		-0.3		0.3	V
$\overline{INVALID}$ Output Low Voltage	I_{OUT} = 1.0mA, -0.3V < R_IN < 0.3V				0.4	V
$\overline{INVALID}$ Output High Voltage	I_{OUT} = -0.4mA, |R_IN| > 2.8V		V_{CC} - 0.25			V

ELECTRICAL CHARACTERISTICS (continued)

(Circuit of Figure 1, V_{CC} = 1.8V to 4.25V, C1 = 0.47µF, C2 = C3 = C4 = 1µF, L1 = 15µH, $T_A = T_{MIN}$ to T_{MAX}, unless otherwise noted. Typical values are at V_{CC} = 3.0V, T_A = +25°C.)

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS
EIA/TIA-232E RECEIVER INPUTS					
Input Voltage Range		-25		+25	V
Input Threshold Low	V_{CC} = 2.0V to 4.25V	0.4			V
	V_{CC} = 1.8V to 4.25V	0.3			
Input Threshold High	V_{CC} = 1.8V to 4.25V			3.0	V
	V_{CC} = 1.8V to 3.6V			2.8	
Input Hysteresis			0.7		V
Input Resistance	-15V < R_IN < 15V	3	5	7	kΩ
EIA/TIA-232E TRANSMITTER OUTPUTS					
Output Voltage Swing	All transmitter outputs loaded with 3kΩ to ground	±5	±6		V
Output Resistance	V_{CC} = 0V, -2V < T_OUT < 2V	300			Ω
Output Short-Circuit Current			±24	±100	mA

TIMING CHARACTERISTICS

(Circuit of Figure 1, V_{CC} = 1.8V to 4.25V, C1 = 0.47µF, C2 = C3 = C4 = 1µF, L1 = 15µH, $T_A = T_{MIN}$ to T_{MAX}, unless otherwise noted. Typical values are at V_{CC} = 3.0V, T_A = +25°C.)

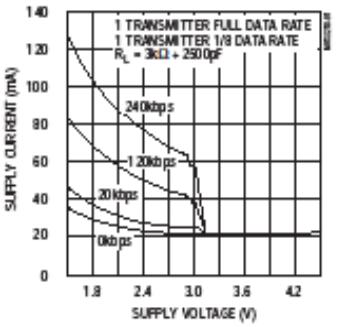
PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
Data Rate		2500pF ‖ 3kΩ load each transmitter, one transmitter switching, 150pF load each receiver	120	235		kbps
Receiver Output Enable Time	t_{ER}			90	300	ns
Receiver Output Disable Time	t_{DR}			200	500	ns
Transmitter Output Enable Time	t_{ET}			250	450	µs
Transmitter Output Disable Time	t_{DT}			500		ns
Receiver Propagation Delay	t_{PHLR}	150pF load		290	1000	ns
	t_{PLHR}	150pF load		260	1000	
Transmitter Propagation Delay	t_{PHLT}	2500pF ‖ 3kΩ load		1.9	2.7	µs
	t_{PLHT}	2500pF ‖ 3kΩ load		1.8	2.7	
Transition Region Slew Rate		T_A = +25°C, V_{CC} = 3.0V, R_L = 3kΩ to 7kΩ, C_L = 50pF to 2500pF, measured from +3V to -3V or -3V to +3V	3.0		30	V/µs
AUTOSHUTDOWN TIMING						
Receiver Threshold to Transmitters Enabled	t_{WU}	Figure 4b		250		µs
Receiver Positive or Negative Threshold to INVALID High	t_{INVH}	Figure 4b		1		µs
Receiver Positive or Negative Threshold to INVALID Low	t_{INVL}	Figure 4b		30		µs

1µA Supply Current, 1.8V to 4.25V-Powered RS-232 Transceiver with AutoShutdown™

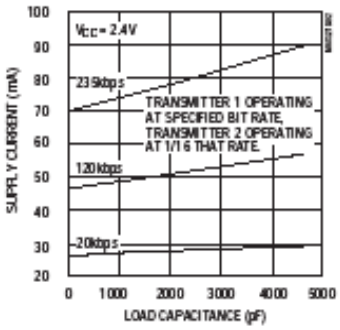
Typical Operating Characteristics

(Circuit of Figure 1, V_{CC} = 1.8V, all transmitter outputs loaded with 3kΩ, T_A = +25°C, unless otherwise noted.)

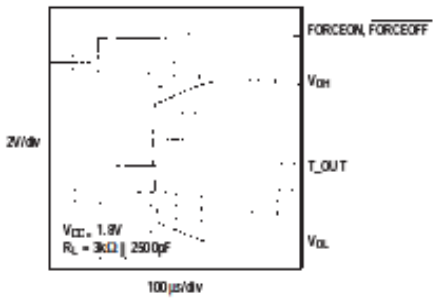
SUPPLY CURRENT vs. SUPPLY VOLTAGE



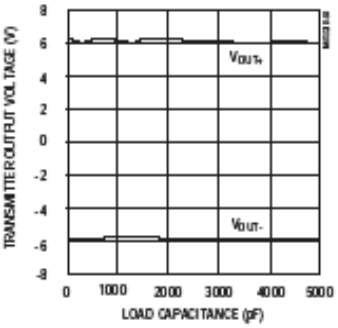
TRANSMITTING SUPPLY CURRENT vs. LOAD CAPACITANCE



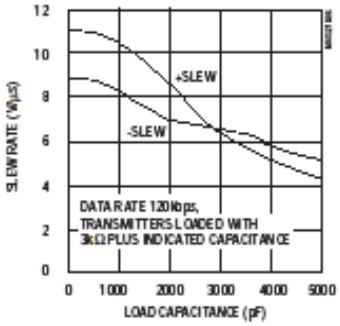
TIME TO EXIT SHUTDOWN (ONE TRANSMITTER HIGH, ONE TRANSMITTER LOW)



TRANSMITTER OUTPUT VOLTAGE vs. LOAD CAPACITANCE AT 120kbps



SLEW RATE vs. TRANSMITTER CAPACITANCE



1µA Supply Current, 1.8V to 4.25V-Powered RS-232 Transceiver with AutoShutdown™

Pin Description

PIN	NAME	FUNCTION
1	LX	Inductor/Diode Connection Point
2	$\overline{\text{INVALID}}$	Output of Invalid Signal Detector. Low if invalid RS-232 levels are present on all receiver inputs, otherwise high.
3	FORCEON	Drive high when $\overline{\text{FORCEOFF}}$ = high to override automatic circuitry, keeping transmitters on.
4	$\overline{\text{FORCEOFF}}$	Drive low to shut down transmitters and on-board power supply, overriding all automatic circuitry and FORCEON.
5, 17, 20	GND	Ground
6	VCC	Supply Voltage Input, 1.8V to 4.25V. Bypass to GND with at least 1µF.
7, 8	T1IN, T2IN	Transmitter Inputs
9, 10	R1OUT, R2OUT	Receiver Outputs
11, 12	R2IN, R1IN	Receiver Inputs
13, 14	T2OUT, T1OUT	Transmitter Outputs, swing between V+ and V-
15	V-	Negative Supply generated on-board
16, 18	C1-, C1+	Terminal for Charge-Pump Capacitor
19	V+	Positive Supply generated on-board

Detailed Description

The MAX3218 line driver/receiver is intended for battery-powered EIA/TIA-232 and V.28/V.24 communications interfaces that require two drivers and two receivers. The operating voltage extends from 1.8V to 4.25V, yet the device maintains true RS-232 and EIA/TIA-562 transmitter output voltage levels. This wide supply voltage range permits direct operation from a variety of batteries without the need for a voltage regulator. For example, the MAX3218 can be run directly from a single lithium cell or a pair of alkaline cells. It can also be run directly from two NiCd or NiMH cells from full-charge voltage down to the normal 0.9V/cell end-of-life point. The 4.25V maximum supply voltage allows the two rechargeable cells to be trickle- or fast-charged while driving the MAX3218.

The circuit comprises three sections: power supply, transmitters, and receivers. The power-supply section converts the supplied input voltage to 6.5V, providing the voltages necessary for the drivers to meet true RS-232 levels. External components are small and inexpensive.

The transmitters and receivers are guaranteed to operate at 120kbps data rates, providing compatibility with LapLink™ and other high-speed communications software.

The MAX3218 is equipped with Maxim's new proprietary AutoShutdown™ circuitry. This achieves a 1µA supply current by shutting down the device when the RS-232 cable is disconnected or when the connected peripheral transmitters are turned off. While shut down, both receivers can remain active or can be disabled under logic control. With this feature, the MAX3218 can be in low-power shutdown mode and still monitor activity on external devices. Three-state drivers are provided on both receiver outputs.

Three-state drivers on both receiver outputs are provided so that multiple receivers, generally of different interface standards, can be wire-ORed at the UART.

Switch-Mode Power Supply

The switch-mode power supply uses a single inductor with one diode and three small capacitors to generate ±6.5V from an input voltage in the 1.8V to 4.25V range.

Inductor Selection

Use a 15µH inductor with a saturation current rating of at least 350mA and less than 1 Ω resistance. Table 1 lists suppliers of inductors that meet the 15 µH/350mA/1Ω specifications.

™ LapLink is a trademark of Traveling Software, Inc.
AutoShutdown is a trademark of Maxim Integrated Products.

1µA Supply Current, 1.8V to 4.25V-Powered RS-232 Transceiver with AutoShutdown™

Table 1. Suggested Component Suppliers

MANUFACTURER	PART NUMBER	PHONE	FAX
Inductors			
Murata-Erie	LQH4N150K-TA	USA (814) 237-1431	USA (814) 238-0490
Sumida	CD43150	USA (708) 956-0666 Japan (03) 3607-5111	USA (708) 956-0702 Japan (03) 3607-5428
TDK	NLC453232T-150K	USA (708) 803-6100 Japan (03) 3278-5111	USA (708) 803-6296 Japan (03) 3278-5358
Diodes—Surface-Mount			
Allegro	TMPD6050LT	USA (508) 853-5000	USA (508) 853-7556
Central Semiconductor	CMPSH-3 (Schottky)	USA (516) 435-1110	USA (516) 435-1824
Motorola	MMBD6050LT1 (silicon)	USA (408) 749-0510	USA (408) 991-7420
Philips	PMBD6050 (silicon)	USA (401) 762-3800	USA (401) 767-4493
Diodes—Through-Hole			
Motorola	1N6050 (silicon), 1N5817 (Schottky)	USA (408) 749-0510	USA (408) 991-7420

Diode Selection

Key diode specifications are fast recovery time (<10ns), average current rating (>100mA), and peak current rating (>350mA). Inexpensive fast silicon diodes, such as the 1N6050, are generally recommended. More expensive Schottky diodes improve efficiency and give slightly better performance at very low V_{CC} voltages. Table 1 lists suppliers of both surface-mount and through-hole diodes. 1N914s are usually satisfactory, but specifications and performance vary widely with different manufacturers.

Capacitor Selection

Use capacitors with values at least as indicated in Figure 1. Capacitor C2 determines the ripple on V+, but not the absolute voltage. Capacitors C1 and C3 determine both the ripple and the absolute voltage of V-. Bypass V_{CC} to GND with at least 1 µF (C4) placed close to pins 5 and 6. If the V_{CC} line is not bypassed elsewhere (e.g., at the power supply), increase C4 to 4.7µF.

You may use ceramic or polarized capacitors in all locations. If you use polarized capacitors, tantalum types are preferred because of the high operating frequency of the power supplies (about 250kHz). If aluminum electrolytics are used, higher capacitance values may be required.

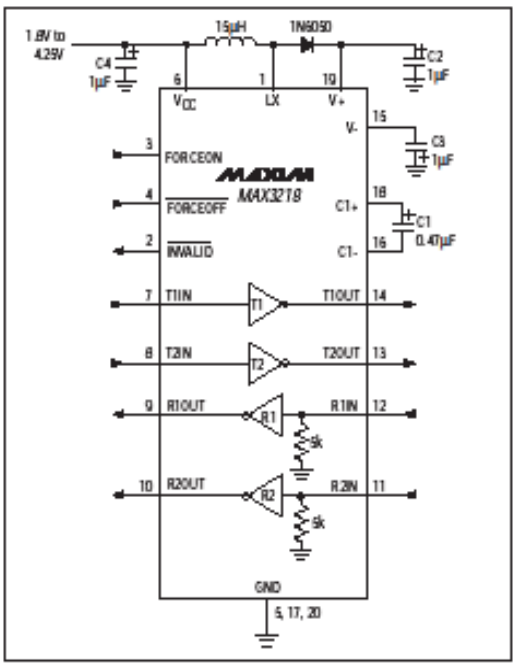


Figure 1. Typical Operating Circuit

1µA Supply Current, 1.8V to 4.25V-Powered RS-232 Transceiver with AutoShutdown™

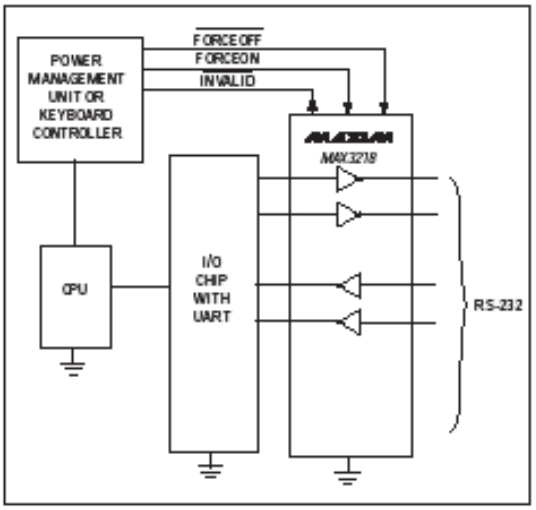


Figure 2. Interface Under Control of PMU

RS-232 Drivers

The two drivers are identical, and deliver EIA/TIA-232E and EIA/TIA-562 output voltage levels when V_{CC} is between 1.8V and 4.25V. One transmitter can drive up to 3kΩ in parallel with 2500pF at up to 120kbps. Connect unused drivers to either GND or V_{CC}. When FORCEOFF is driven low, or when AutoShutdown circuitry senses invalid voltage levels at **all** receiver inputs, the drivers are disabled and their outputs are forced into a high-impedance state. Driver inputs do not have internal pull-up resistors.

RS-232 Receivers

The two receivers are identical, and accept both EIA/TIA-232E and EIA/TIA-562 input signals. The CMOS receiver outputs are inverting and swing rail-to-rail. Receivers are disabled only when FORCEON and FORCEOFF inputs are low. (See Table 2.)

Table 2. Receiver Status

FORCEON	FORCEOFF	RECEIVER STATUS
X	H	Receiver Enabled
H	X	
L	L	Receiver Disabled

Shutdown

When FORCEOFF is low, power supplies are disabled and the transmitters are placed in a high-impedance state. Receiver operation is not affected by taking FORCEOFF low. Power consumption is dramatically reduced in shutdown mode. Supply current is minimized when the receiver inputs are static in any one of three states: floating (ground), GND, or V_{CC}.

AutoShutdown™

A 1µA supply current is achieved with Maxim's new AutoShutdown feature, which operates when FORCEON is low and FORCEOFF is high. When the MAX3218 senses no valid signal level on either receiver input for typically 30µs, the on-board power supply and drivers shut down, reducing supply current to 1µA. Internal 5kΩ resistors pull undriven receiver inputs to ground. This occurs if the RS-232 cable is disconnected or if the connected peripheral transmitters are turned off. The system turns on again when a valid level is applied to either RS-232 receiver input. As a result, the system saves power without changes to the existing BIOS or operating system. When using the AutoShutdown feature, INVALID is high when the device is on and low when the device is shut down. The INVALID output indicates the condition of the receiver inputs.

Table 3 summarizes the MAX3218 operating modes. FORCEON and FORCEOFF override the automatic circuitry and force the transceiver into its normal operating state or into its low-power standby state. When neither control is asserted, the IC selects between these states automatically based on receiver input levels. Figure 4 depicts valid and invalid RS-232 receiver levels. The

Table 3. AutoShutdown Logic

RS-232 SIGNAL PRESENT AT RECEIVER INPUT	FORCEOFF INPUT	FORCEON INPUT	INVALID OUTPUT	TRANSCEIVER STATUS
Yes	H	X	H	Normal Operation
No	H	H	L	Normal Operation (Forced On)
No	H	L	L	Shutdown (AutoShutdown)
Yes	L	X	H	Shutdown (Forced Off)
No	L	X	L	Shutdown (Forced Off)

TE555LP

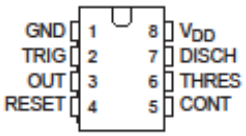
http://www.ti.com/lit/ds/symlink/tlc555.pdf

TLC555
LinCMOS™ TIMER

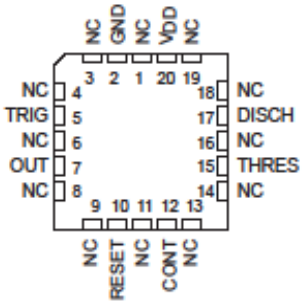
SLFS043F – SEPTEMBER 1983 – REVISED FEBRUARY 2005

- Very Low Power Consumption
  - 1 mW Typ at V_{DD} = 5 V
- Capable of Operation in Astable Mode
- CMOS Output Capable of Swinging Rail to Rail
- High Output-Current Capability
  - Sink 100 mA Typ
  - Source 10 mA Typ
- Output Fully Compatible With CMOS, TTL, and MOS
- Low Supply Current Reduces Spikes During Output Transitions
- Single-Supply Operation From 2 V to 15 V
- Functionally Interchangeable With the NE555; Has Same Pinout
- ESD Protection Exceeds 2000 V Per MIL-STD-883C, Method 3015.2
- Available in Q-Temp Automotive
  - High Reliability Automotive Applications
  - Configuration Control/Print Support
  - Qualification to Automotive Standards

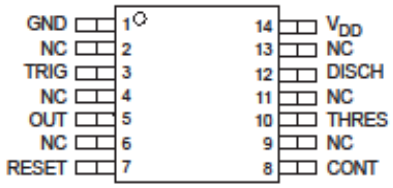
D, DB, JG, OR P PACKAGE
(TOP VIEW)



FK PACKAGE
(TOP VIEW)



PW PACKAGE
(TOP VIEW)



NC – No internal connection

description

The TLC555 is a monolithic timing circuit fabricated using the TI LinCMOS™ process. The timer is fully compatible with CMOS, TTL, and MOS logic and operates at frequencies up to 2 MHz. Because of its high input impedance, this device uses smaller timing capacitors than those used by the NE555. As a result, more accurate time delays and oscillations are possible. Power consumption is low across the full range of power supply voltage.

Like the NE555, the TLC555 has a trigger level equal to approximately one-third of the supply voltage and a threshold level equal to approximately two-thirds of the supply voltage. These levels can be altered by use of the control voltage terminal (CONT). When the trigger input (TRIG) falls below the trigger level, the flip-flop is set and the output goes high. If TRIG is above the trigger level and the threshold input (THRES) is above the threshold level, the flip-flop is reset and the output is low. The reset input (RESET) can override all other inputs and can be used to initiate a new timing cycle. If RESET is low, the flip-flop is reset and the output is low. Whenever the output is low, a low-impedance path is provided between the discharge terminal (DISCH) and GND. All unused inputs should be tied to an appropriate logic level to prevent false triggering.

While the CMOS output is capable of sinking over 100 mA and sourcing over 10 mA, the TLC555 exhibits greatly reduced supply-current spikes during output transitions. This minimizes the need for the large decoupling capacitors required by the NE555.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

description (continued)

The TLC555C is characterized for operation from 0°C to 70°C. The TLC555I is characterized for operation from −40°C to 85°C. The TLC555Q is characterized for operation over the automotive temperature range of −40°C to 125°C. The TLC555M is characterized for operation over the full military temperature range of −55°C to 125°C.

AVAILABLE OPTIONS†

PACKAGED DEVICES							
T_A	V_{DD} RANGE	SMALL OUTLINE (D)‡	SSOP (DB)‡	CHIP CARRIER (FK)	CERAMIC DIP (JG)	PLASTIC DIP (P)	TSSOP (PW)‡
0°C to 70°C	2 V to 15 V	TLC555CD	TLC555CDB	—	—	TLC555CP	TLC555CPW
−40°C to 85°C	3 V to 15 V	TLC555ID	—	—	—	TLC555IP	—
−40°C to 125°C	5 V to 15 V	TLC555QD	—	—	—	—	—
−55°C to 125°C	5 V to 15 V	TLC555MD	—	TLC555MFK	TLC555MJG	TLC555MP	—

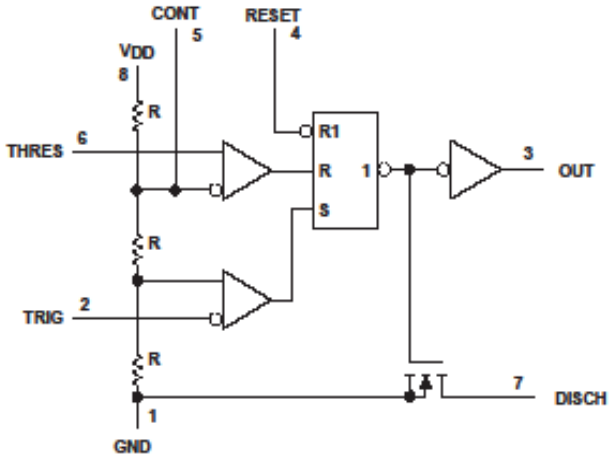
† For the most current package and ordering information, see the Package Option Addendum at the end of this document, or see the TI web site at www.ti.com.
‡ This package is available taped and reeled. Add the R suffix to device type (e.g., TLC555CDR).

FUNCTION TABLE

RESET VOLTAGE‡	TRIGGER VOLTAGE‡	THRESHOLD VOLTAGE‡	OUTPUT	DISCHARGE SWITCH
<MIN	Irrelevant	Irrelevant	L	On
>MAX	<MIN	Irrelevant	H	Off
>MAX	>MAX	>MAX	L	On
>MAX	>MAX	<MIN	As previously established	

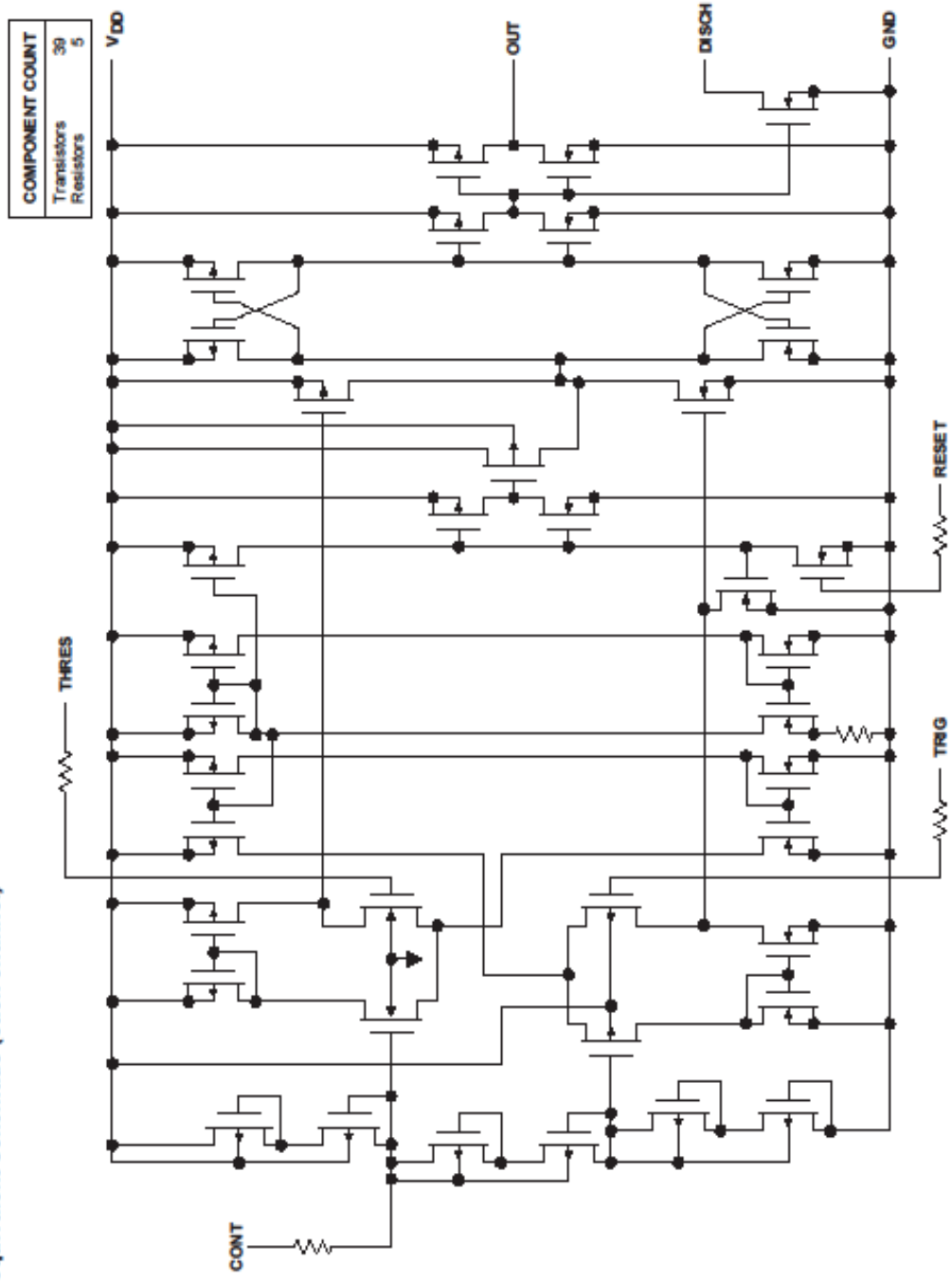
‡ For conditions shown as MIN or MAX, use the appropriate value specified under electrical characteristics.

functional block diagram



Pin numbers are for all packages except the FK package. RESET can override TRIG, which can override THRES.

equivalent schematic (each channel)



absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage, V_{DD} (see Note 1) . . . . . 18 V
Input voltage range, V_I (any input) . . . . . −0.3 to V_{DD}
Sink current, discharge or output . . . . . 150 mA
Source current, output, I_O . . . . . 15 mA
Continuous total power dissipation . . . . . See Dissipation Rating Table
Operating free-air temperature range, T_A: C-suffix . . . . . 0°C to 70°C
I-suffix . . . . . −40°C to 85°C
Q-suffix . . . . . −40°C to 125°C
M-suffix . . . . . −55°C to 125°C
Storage temperature range . . . . . −65°C to 150°C
Case temperature for 60 seconds: FK package . . . . . 260°C
Lead temperature 1,6 mm (1/16 inch) from case for 60 seconds: JG package . . . . . 300°C
Lead temperature 1,6 mm (1/16 inch) from case for 10 seconds: D, DB, P, or PW package . . . . . 260°C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTE 1: All voltage values are with respect to network GND.

DISSIPATION RATING TABLE

PACKAGE	$T_A \le 25°C$ POWER RATING	DERATING FACTOR ABOVE $T_A = 25°C$	$T_A = 70°C$ POWER RATING	$T_A = 85°C$ POWER RATING	$T_A = 125°C$ POWER RATING
D	725 mW	5.8 mW/°C	464 mW	377 mW	145 mW
DB	525 mW	4.2 mW/°C	336 mW	273 mW	105 mW
FK	1375 mW	11.0 mW/°C	880 mW	715 mW	275 mW
JG	1050 mW	8.4 mW/°C	672 mW	546 mW	210 mW
P	1000 mW	8.0 mW/°C	640 mW	520 mW	200 mW
PW	525 mW	4.2 mW/°C	336 mW	273 mW	105 mW

recommended operating conditions

		MIN	MAX	UNIT
Supply voltage, V_{DD}		2	15	V
Operating free-air temperature range, T_A	TLC555C	0	70	°C
	TLC555I	−40	85	
	TLC555Q	−40	125	
	TLC555M	−55	125	

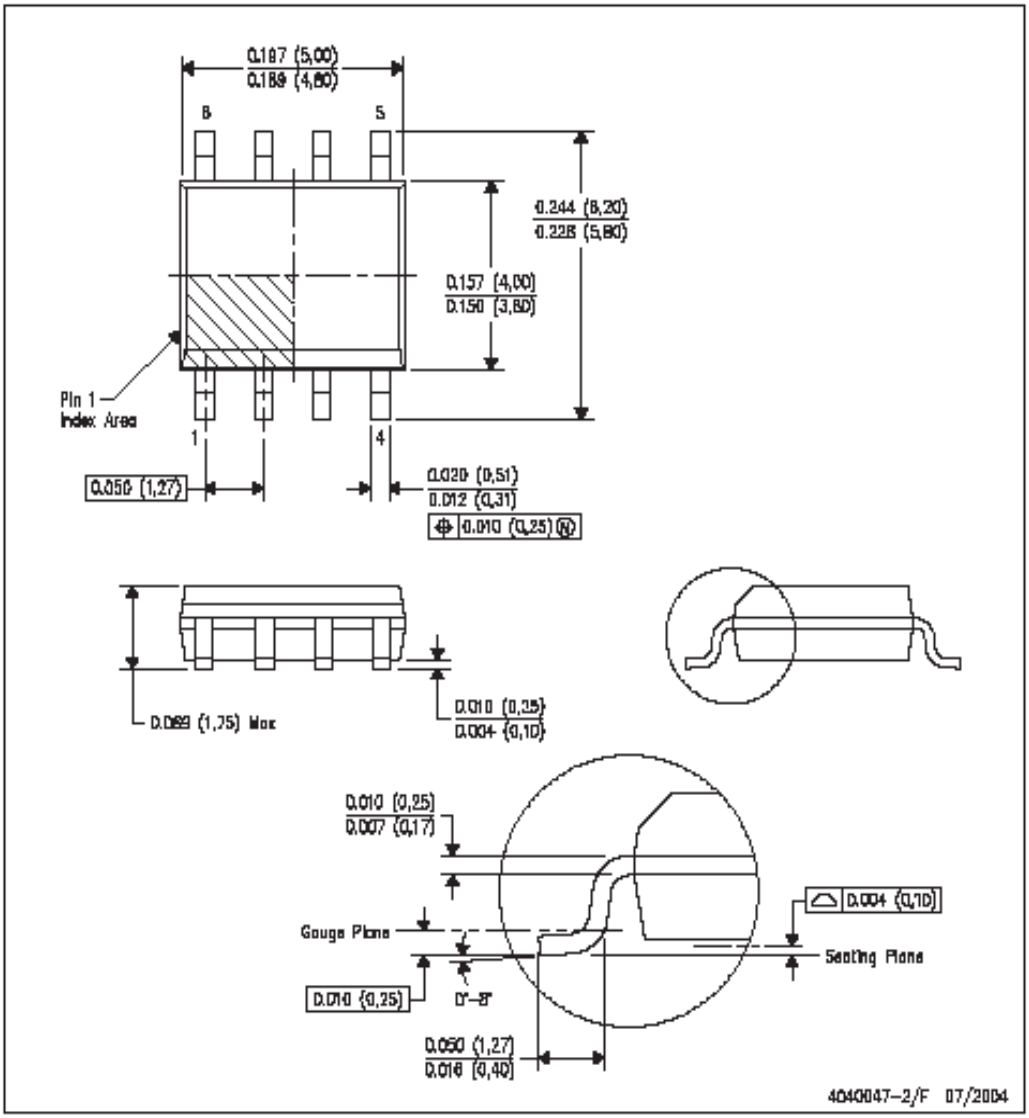
electrical characteristics at specified free-air temperature, V_{DD} = 2 V for TLC555C, V_{DD} = 3 V for TLC555I

PARAMETER		TEST CONDITIONS	T_A†	TLC555C			TLC555I			UNIT
				MIN	TYP	MAX	MIN	TYP	MAX	
V_{IT}	Threshold voltage		25°C	0.95	1.33	1.65	1.6		2.4	V
			Full range	0.85		1.75	1.5		2.5	
I_{IT}	Threshold current		25°C		10			10		pA
			MAX		75			150		
$V_{I(TRIG)}$	Trigger voltage		25°C	0.4	0.67	0.95	0.71	1	1.29	V
			Full range	0.3		1.05	0.61		1.39	
$I_{I(TRIG)}$	Trigger current		25°C		10			10		pA
			MAX		75			150		
$V_{I(RESET)}$	Reset voltage		25°C	0.4	1.1	1.5	0.4	1.1	1.5	V
			Full range	0.3		2	0.3		1.8	
$I_{I(RESET)}$	Reset current		25°C		10			10		pA
			MAX		75			150		
	Control voltage (open circuit) as a percentage of supply voltage		MAX		66.7%			66.7%		
	Discharge switch on-stage voltage	I_{OL} = 1 mA	25°C		0.03	0.2		0.03	0.2	V
			Full range			0.25			0.375	
	Discharge switch off-stage current		25°C		0.1			0.1		nA
			MAX		0.5			120		
V_{OH}	High-level output voltage	I_{OH} = −300 µA	25°C	1.5	1.9		2.5	2.85		V
			Full range	1.5			2.5			
V_{OL}	Low-level output voltage	I_{OL} = 1 mA	25°C		0.07	0.3		0.07	0.3	V
			Full range			0.35			0.4	
I_{DD}	Supply current	See Note 2	25°C			250			250	µA
			Full range			400			500	

† Full range is 0°C to 70°C for the TLC555C and −40°C to 85°C for the TLC555I. For conditions shown as MAX, use the appropriate value specified in the recommended operating conditions table.

NOTE 2: These values apply for the expected operating configurations in which THRES is connected directly to DISCH or to TRIG.

D (R–PDSO–G8) PLASTIC SMALL–OUTLINE PACKAGE



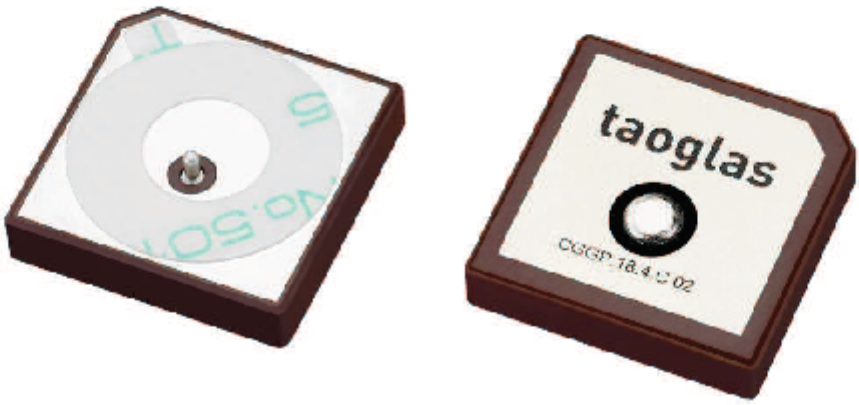
NOTES:
A. All linear dimensions are in inches (millimeters).
B. This drawing is subject to change without notice.
C. Body dimensions do not include mold flash or protrusion not to exceed 0.006 (0,15).
D. Falls within JEDEC MS–012 variation AA.

Antena GPS y Glonass
http://www.taoglas.com/images/product_images/original_images/CGGP.18.4.C.02.pdf

taoglas
antenna solutions

SPECIFICATION

Part No.	:	**CGGP.18.4.C.02**
Product Name	:	18mm Ceramic GPS/GLONASS Patch Antenna, 1575-1610MHz
Features	:	Wide-band Operation 18mm*18mm*4mm 3dBiC Peak Gain (on 70mm*70mm ground-plane) Pin type Automotive TS16949 Production and Quality Approved RoHS compliant



1. Introduction

This 18mm ceramic GPS/Glonass patch antenna, by means of a double resonance design, has unique wide-band operation over the whole operating bands of GPS and Glonass systems from 1575MHz to 1610MHz. The antenna achieves best in class right hand circularly polarized gain with good axial ratio characteristics mitigating multi-path effects which is vital to improved S/N carrier ratio and faster locking times on the device itself. It is mounted via pin and double-sided adhesive.

This antenna has been tuned for a centre position on a 70mm*70mm ground-plane. It is manufactured and tested in a TS16949 first tier automotive approved facility. For further optimization to customer specific device environments where positioning is off centre or on different ground-plane sizes, custom tuned patch antennas can be supplied. For more details please email sales@taoglas.com

2. Specification

ELECTRICAL	
Range of Receiving Frequency	GPS: 1575.42±1.023MHz GLONASS: 1602±5MHz
Center Frequency	1592MHz ± 3MHz
VSWR	2.4 max
Gain at Zenith	+3.0 dBic typ.
Polarization	RHCP
Impedance	50 ohms
MECHANICAL	
Ceramic Dimension	18mm x 18mm x 4mm
Pin Diameter	0.9mm
Pin Length	1.8mm
ENVIRONMENTAL	
Operation Temperature	-40°C to 85°C
Humidity	Non-condensing 65°C 95% RH

* Antenna properties were measurement with the antenna mounted on 70*70mm Ground Plane

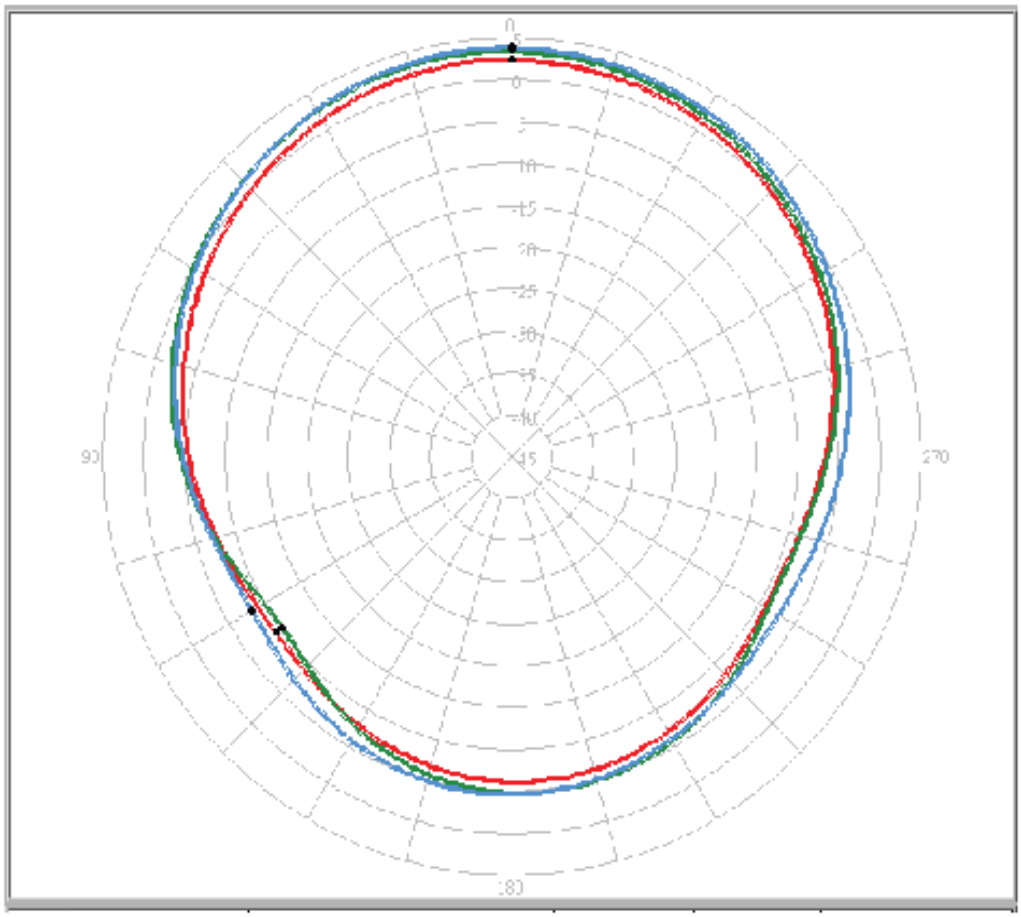
Taoglas Part # CGGPD.18.C

3. **Antenna S11 Properties** - 1592 MHz Typical Return Loss, SWR, Impedance, measured on the test fixture

XXX

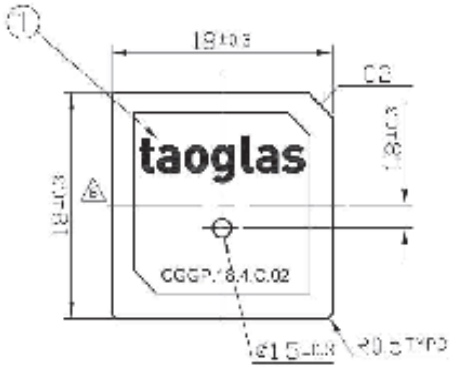
4. Antenna Radiation Properties

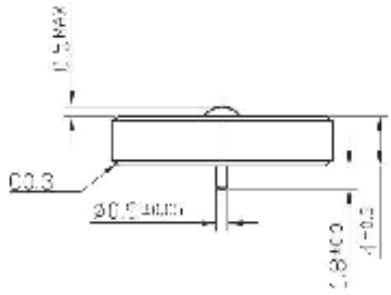
4.1. XZ – Plane Radiation

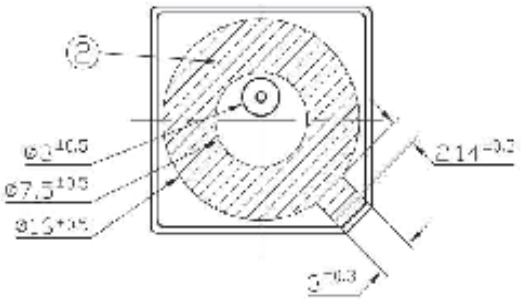


Item	XZ-plane
1575 MHz	2.49 dBi
1592 MHz	3.47 dBi
1608 MHz	3.85 dBi

5. Mechanical Drawing







	Name	Material	Finish	QTY	NOTES:
1	CGGP.18 Patch 18x18x4	Ceramic	Clear	1	1.Double sided adhesive area.
2	Double sided Adhesive	NITTO 5015	White Liner	1	

Antena GSM impresa en PCB

http://www.taoglas.com/images/product_images/original_images/PC27.07.0100A%20Quad-Band%20GSM%20PCB%20Antenna%20290110.pdf

taoglas
antenna solutions

SPECIFICATION

Part No.	:	PC27.07.0100A
Product Name	:	TheStripe™ 850/900/1800/1900MHz GSM PCB Antenna w/100mm IPEX 1.13mm diameter MHF II connector
Feature	:	34mm*7mm*0.8mm Compatible with Hirose U.FL

I. Introduction

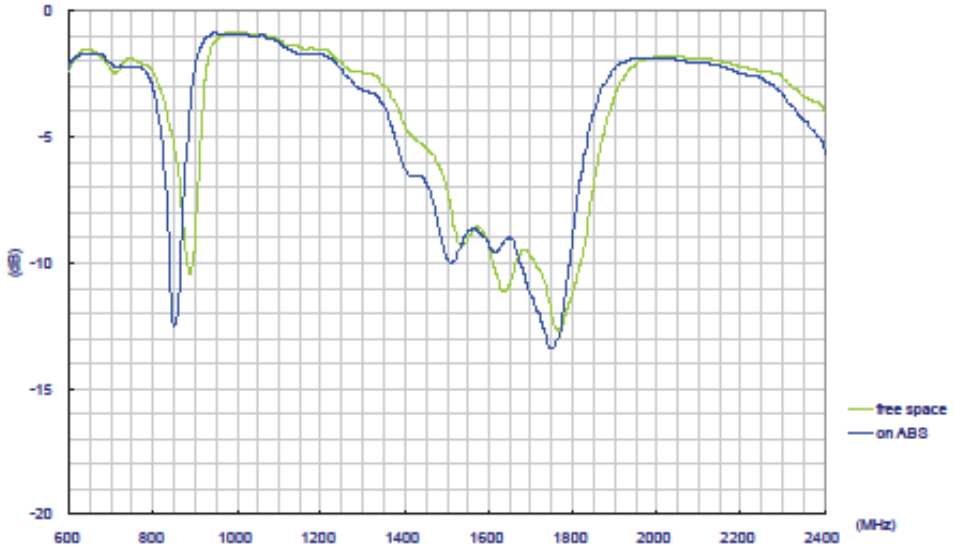
This miniaturized low profile PCB antenna is based on smart TheStripe™ antenna technology. It consists of a PCB antenna and 1.13mm co-axial cable with Ipex MHFII (Hirose U.FL) connector.

II. Specification

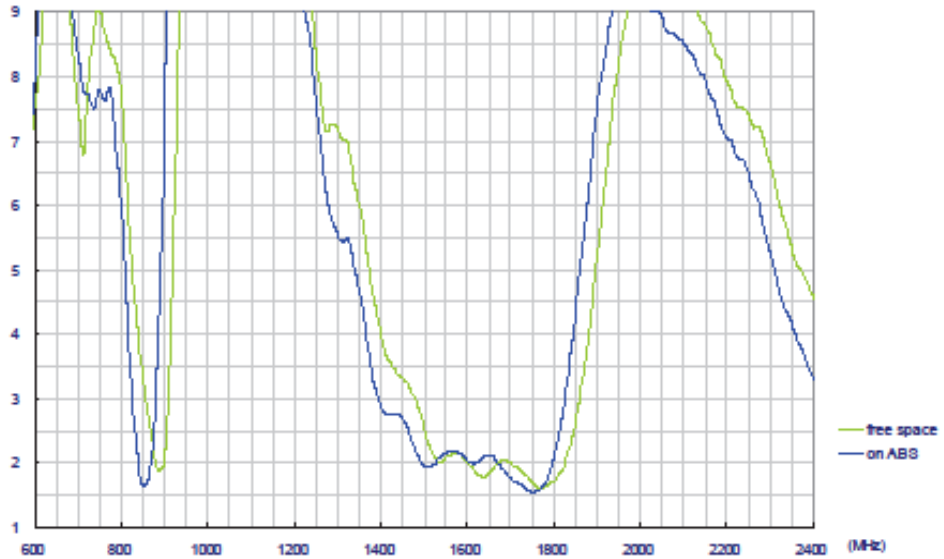
ELECTRICAL				
Frequency (MHz)	824 ~ 896	880~960	1710~1880	1850~1990
Free Space Peak Gain	-4.1dBi	-13dBi	0.6dBi	-0.4dBi
Free Space Efficiency	13%	8%	51%	11%
Impedance	50Ω			
Polarization	Linear			
Radiation Pattern	Omni			
MECHANICAL				
Antenna Dimensions	34 x 7 x 0.8mm			
Cable	1.13 co-axial			
Connector	Ipex MHF II			
ENVIRONMENTAL				
Operation Temperature	-40°C to 85°C			
Humidity	Non-condensing 65°C 95% RH			

III. S11 Property

Return Loss

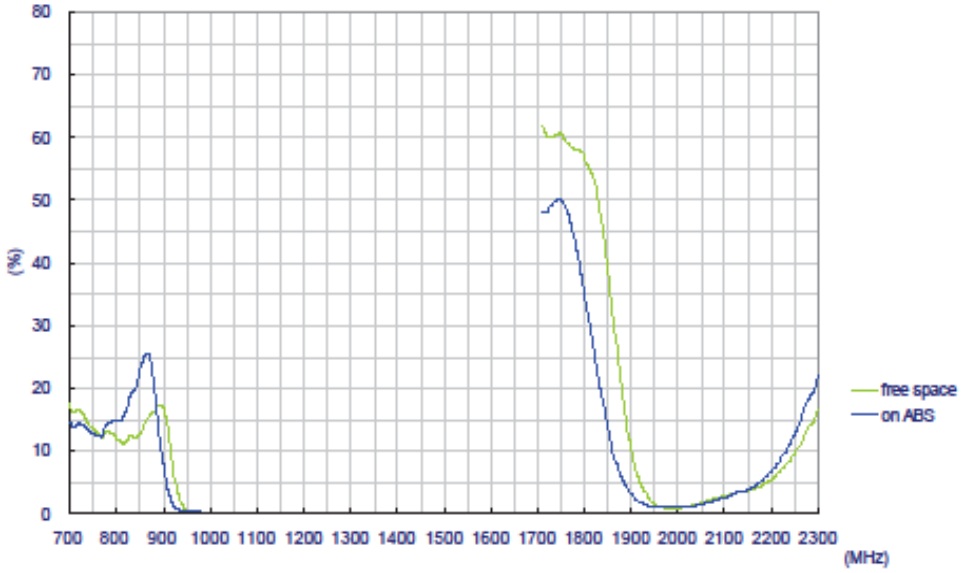


VSWR

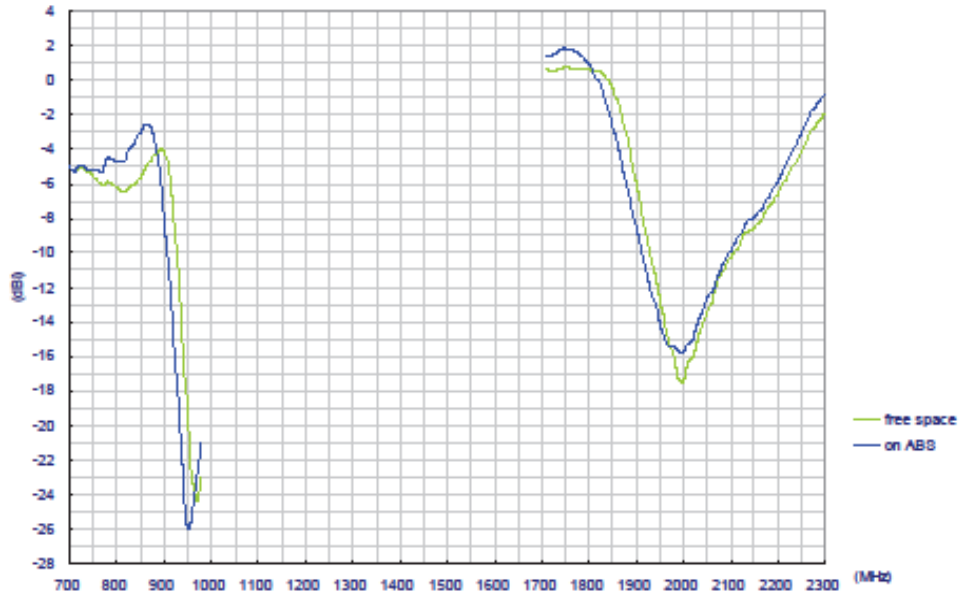


IV. 3D Radiation Property

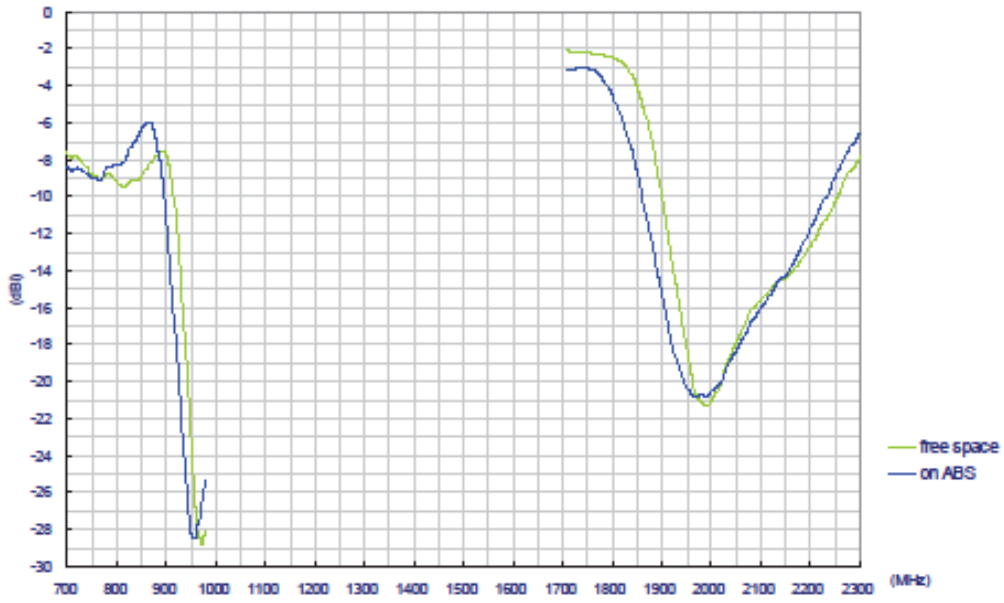
IV.1. Efficiency



IV.2. Peak Gain



IV.3. Average Gain



Módulo GE864

http://www.telit.com/telit/Pulsar/en_US.Store.display.1002./ge865-quad

Documentos más importantes:
Telit_Ge865_Hardware_User Guide_r12
Telit_Modules_Software_User_Guide_r15
Telit_AT_Commands_Reference_Guide_r20
Telit_Easy_Scan_User_Guide_r1
Telit_Easy_Script_Python_r17

A continuación se muestran algunas páginas del manual hardware del GE864.

GE865 Hardware User Guide

1vv0300799 Rev.16 - 2013-04-22

Making machines talk.

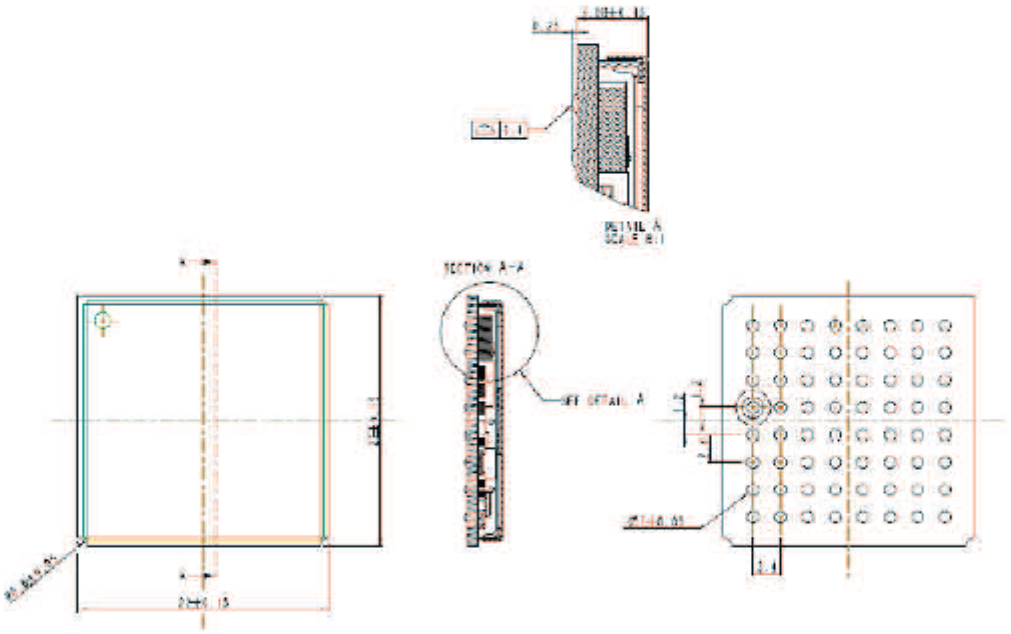
Contents

XL

3. GE865 Mechanical Dimensions

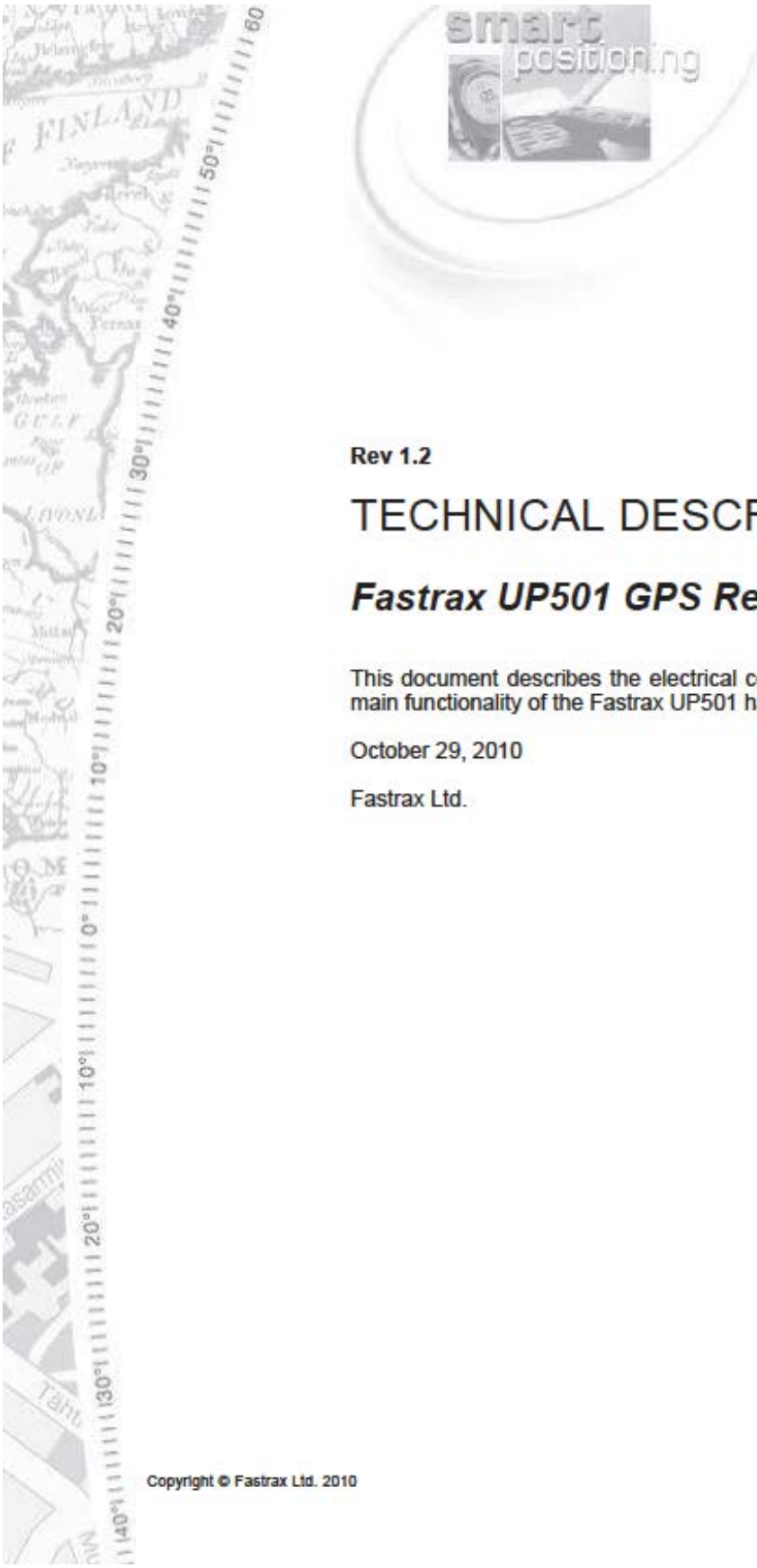
The GE865-QUAD overall dimensions are:

- **Length:** **22 mm**
- **Width:** **22 mm**
- **Thickness:** **3.0 mm**
- **Weight:** **3,2 g**



Receptor GPS Fastrax

https://www.sparkfun.com/products/retired/10702



CONTENTS

2. SPECIFICATIONS

2.1 General

Table 2 General Specifications for UP501.

Receiver	GPS L1 C/A-code, SPS
Channels	66 acquisition and 22 tracking
Update rate	1 Hz default (fix rate configurable up to 10Hz)
Acquisition Sensitivity (Cold start)	-148 dBm (1)
Re-acquisition Sensitivity	-158 dBm (1)
Navigation Sensitivity	-165 dBm (1)
Supply voltage, VDD	+3.0 V...+4.2 V (+3.0V...+5.5V for UP501H)
Back up supply voltage, VDD_B	+2.0 V...+4.2 V (+2.0V...+5.5V for UP501H)
Power consumption, VDD	75 mW typical @ 3.0 V (2) (Typ. 115mW@3.0V in satellite search phase)
Power consumption, VDD_B	15 uW typical @ 3.0 V (during battery backup state).
Operating temperature range	-40 ºC...+85 ºC (4)
Serial port protocol	Port 0: NMEA
Serial data format	8 bits, no parity, 1 stop bit
Serial data speed (default)	NMEA: 9600 baud
CMOS I/O signal levels (3)	V_{IL}: -0.3V...0.8V, V_{IH}: 2.0V...3.6V, V_{OL}:-0.3V...0.4V, V_{OH}: 2.4V...3.2V
I/O sink/source capability	+/- 2 mA max.
PPS output	+/- 50 ns (RMS) accuracy

Note (1): measured by conducted measurement from GPS simulator.

Note (2): Navigation with good signals, max. 12 satellites in view.

Note (3): Fastrax UP501R UART signals are RS232 compatible.

Note (4): UP501 backup battery operating range is -10 ºC...+60 ºC.

3. OPERATION

3.1 Operating modes

After power up the receiver boots from the internal flash memory for normal operation. Modes of operation:

- Tracking/navigating mode
- Low power tracking/navigating mode
- Backup mode

3.1.1 Tracking/Navigating mode

In tracking/navigating mode the Fastrax UP501 receiver module will search for additional satellites and collects almanac data. Once the receiver has collected almanac data (this takes about 12 minutes from Cold Start), it will enter Low Power Tracking mode. The VDD power consumption in table 1 is measured in Low Power Tracking/Navigating mode.

3.1.2 Low Power Tracking/Navigating mode

In Low power tracking/navigating mode the receiver continues normal navigation but does not collect further Almanacs data. Therefore the current consumption is reduced to level of <75 mW (<85 mW for UP501R with default UART baud rate).

3.1.3 Backup mode

When the operating voltage VDD is removed from the Fastrax UP501, the module enters Backup mode. In this mode, the module is keeping time by the RTC oscillator. Also, satellite ephemeris data is stored in battery backup RAM in order to get fast TTFF when VDD is connected again. Any user configuration settings are also valid as long as the backup supply VDD_B is active. When the VDD_B is powered off, the configuration is reset to factory configuration on next power up.

4. CONNECTIVITY

4.1 Connection assignments

The I/O connections are available on the 6-pin, 2.54mm pitch pin-header pads.

Table 4 Connections

Contact	Signal name	I/O	Signal description
1	RXD	I	UART Port 0 async. input. Internal pull high resistor 75kΩ.
2	TXD	O	UART Port 0 async. output.
3	GND	-	Ground
4	VDD	I	Main power supply
5	VDD_B	I	Backup supply
6	PPS	O	Pulse per second output.

Notes:

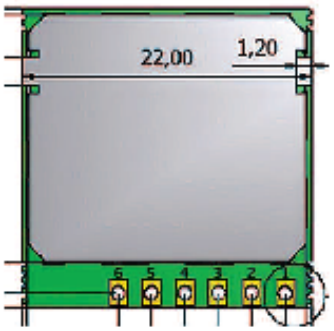


Figure 1. Pin numbering in the Fastrax UP501 module.

4.2 Power supply

The Fastrax UP501 module (including all variants, except UP501B) requires two separate power supplies: VDD_B for non-volatile back

Módulo GE910

Toda la información acerca de este módulo se encuentra en la página del fabricante, fácilmente accesible desde el siguiente enlace:

http://www.telit.com/telit/Pulsar/en_US.Store.display.1099./ge910-quad

Los principales manuales seguidos para el desarrollo del dispositivo final son los siguientes:

Telit_GE910_Hardware_User_Guide_r14
Telit_Modules_Software_User_Guide_r15
Telit_AT_Commands_Reference_Guide_r20
Telit_EVK2_User_Guide_r18
Telit_Easy_Script_Python_2.7_r4
Telit_GE910_Family_Ports_Arrangements_r2
Telit_SIM/USIM_Toolkit_Application_Note_r4
Telit_SIM_Integration_Design_Guide_Application_Note_r10

A continuación se muestran las primeras páginas del manual más importante del chip GE910: la guía Hardware de usuario. En este manual están presentes las indicaciones a seguir para: la alimentación del módulo, el encendido, la creación el footprint del chip, el conexionado de las antenas, etc.

GE910 Hardware User Guide

1vv0300962 Rev.13 – 2013-11-05

Making machines talk.

L

Contents

1.4. Document Organization

This document contains the following chapters:

Chapter 1: "Introduction" provides a scope for this document, target audience, contact and support information, and text conventions.

Chapter 2: "Overview" provides an overview of the document.

Chapter 3: "GE910 Module Connections" deals with the pin out configuration and layout.

Chapter 4: "Hardware Commands" How to operate on the module via hardware.

Chapter 5: "Power supply" Power supply requirements and general design rules.

Chapter 6: "GSM Radio" The antenna connection and board layout design are the most important parts in the full product design.

Chapter 7: "GNSS Receiver" This section describes the GNSS receiver.

Chapter 8: "Logic Level specifications" Specific values adopted in the implementation of logic levels for this module.

Chapter 9: "USB Port" Describes the USB port and the interface between the module and OEM hardware.

Chapter 10: "Serial ports" Describes the UART ports present and the interface between the module and OEM hardware.

Chapter 11: "Audio Section overview" Refers to the audio blocks of the Base Band Chip of the GE910 Telit Modules.

Chapter 12: "General Purpose I/O" How the general purpose I/O pads can be configured.

Chapter 13: "Mounting the GE910 on the application board" Recommendations and specifics on how to mount the module on the user's board.

Chapter 14: "Packing system" Packaging Information.

Chapter 15: "Conformity Assessment Issues" Information related to the Conformity Assessments.

Chapter 16: "Safety Recommendations" Information related to the Safety topics.

Chapter 17: "Document History"

D - Programa completo del prototipo

Librería

```
/*
GSM_Shield.cpp - Libary for using the GSM and GPS-Modul on the GSM/GPRS/GPS-Shield Rev.6
Included Functions
Version:     1.3.2
Date:        16.03.2012
Company:     antrax Datentechnik GmbH
Uses with:   Arduino Duemilanove (ATmega328) and
             Arduino UNO (ATmega328)
*/

#if defined(ARDUINO) && ARDUINO >= 100
  // Choose Arduino.h for IDE 1.0
  #include "Arduino.h"
#else
  // Choose WProgram.h if IDE is older than 1.0
  #include "WProgram.h"
#endif
#include "SIEMERG_GSM_GPS_Shield.h"
#include <SPI.h>

// Adresses of the SCI16IS750 registers
#define THR       0x00 << 3
#define RHR       0x00 << 3
#define FCR       0x02 << 3
#define LCR       0x03 << 3
#define LSR       0x05 << 3
#define SPR       0x07 << 3
#define IODIR     0x0A << 3
#define IOSTATE   0x0B << 3
#define IOCTL     0x0E << 3

// special registers
#define DLL       0x00 << 3
#define DLM       0x01 << 3
#define EFR       0x02 << 3

// SPI
#define CS        10
#define MOSI      11
#define MISO      12
#define SCK       13

// GSM
#define RXD       0
#define TXD       1
#define RING      2
#define CTS       3
#define DTR       4
#define RTS       5
#define DCD       6

int readAnswer = 0;
int state = 0;
int inByte = 0;
int GSM_OK = 0;
int sim_error = 0;
int reg_OK = 0;
char temp_in[10];
// Arduino Duemilanove (ATmega328) & Arduino UNO (ATmega328)
GSM::GSM(int powerON_pin, int baud)
{
 pinMode(powerON_pin, OUTPUT);                        // Pin 7
  _powerON_pin = powerON_pin;
  _baud = baud;
  pinMode(_powerON_pin, OUTPUT);
  pinMode(RXD, INPUT);
  pinMode(TXD, OUTPUT);
```

```
  pinMode(RING, INPUT);
  pinMode(CTS, INPUT);
  pinMode(DTR, OUTPUT);
  pinMode(RTS, OUTPUT);
  pinMode(DCD, INPUT);
}

void GSM::initializeGSM(char pwd[20])
{
  Serial.begin(_baud);                              // 9600 Baud
  pinMode(_powerON_pin, OUTPUT);                    // Pin 7 as Output
  digitalWrite(_powerON_pin, HIGH);                // enable Power GSM-Modul

  delay(3000);
  state = 0;
  readAnswer=0;

  do
    if(readAnswer == 0)
    {
      if(state == 0)
      {
        inByte=0;
        Serial.print("AT\r");                 // send AT  (wake up
        readAnswer = 1;
      }

      if(state == 1)
      {
        inByte = 0;
        Serial.print("ATE0\r");                // disable Echo
        readAnswer = 1;
      }

      if(state == 2)
      {
        inByte = 0;
        Serial.print("AT+IPR=9600\r");         // set Baudrate
        readAnswer = 1;
      }

      if(state == 3)
      {
        inByte = 0;
        Serial.print("AT#SIMDET=1\r");         // set SIM Detection mode  (SIM)
        readAnswer = 1;
      }

      if(state == 4)
      {
        sim_error = 0;
        inByte = 0;
        delay(1000);
       Serial.print("AT+CPIN?\r");              // pin need?   (SIM)
        readAnswer = 1;
      }

      if(state == 5)
      {
        inByte = 0;
        Serial.print("AT+CPIN=");              // enter pin   (SIM)
        Serial.print(pwd);
        Serial.print("\r");
        readAnswer = 1;
      }

      if(state == 6)
      {
        inByte = 0;
        reg_OK = 0;
        delay(2000);
```

```
        Serial.print("AT+CREG?\r");         // Network Registration Report
        readAnswer = 1;
      }

      if(state == 7)
      {
        inByte = 0;
        Serial.print("AT+CMGF=1\r");        // use text-format for sms
        readAnswer = 1;
      }
    }
    else
    {
      readGSMAnswer();
    }
  while(state <= 7);
}

void GSM::setTime(){
       lastTime=millis();
}

void GSM::setExternalTime(){
       externalTime=millis();
}

void GSM::VaciarPuertoSerie (){
  //Serial.println("Vaciando puerto serie");
  while(Serial.available()>0){
    Serial.read();
    delay(10);
  }
  while(Serial.read()!=-1);
   //Serial.println("puerto serie vacio");
}

unsigned int GSM::potencia(byte factor){
  byte i;
  unsigned int resultado=1;
  for (i=0; i<factor; i++){
    resultado=resultado*10;
  }
  return resultado;
}

unsigned int GSM::convHexaDec(char * Hexa, unsigned int tam){
  byte i;
  unsigned int Dec=0;
  unsigned int factor=1;
   for (i=0; (int)i<tam-1; i++){
    factor=factor*16;
   }
    for (i=0; i<tam; i++){
          if ((int)Hexa[i]>=48 && (int)Hexa[i]<=57){
            Dec=Dec+(factor*((int)Hexa[i]-48));
            factor=factor/16;
          }
          else if((int)Hexa[i]==97){
             Dec=Dec+(factor*(10));
             factor=factor/16;
          }
          else if((int)Hexa[i]==98){
            Dec=Dec+(factor*(11));
             factor=factor/16;
          }
          else if((int)Hexa[i]==99){
            Dec=Dec+(factor*(12));
             factor=factor/16;
          }
          else if((int)Hexa[i]==100){
            Dec=Dec+(factor*(13));
```

```
            factor=factor/16;
          }
          else if((int)Hexa[i]==101){
            Dec=Dec+(factor*(14));
             factor=factor/16;
          }
          else if((int)Hexa[i]==102){
            Dec=Dec+(factor*(15));
             factor=factor/16;
          }
          else if((int)Hexa[i]==65){
             Dec=Dec+(factor*(10));
             factor=factor/16;
          }
          else if((int)Hexa[i]==66){
            Dec=Dec+(factor*(11));
             factor=factor/16;
          }
          else if((int)Hexa[i]==67){
            Dec=Dec+(factor*(12));
             factor=factor/16;
          }
          else if((int)Hexa[i]==68){
            Dec=Dec+(factor*(13));
             factor=factor/16;
          }
          else if((int)Hexa[i]==69){
            Dec=Dec+(factor*(14));
             factor=factor/16;
          }
          else {
            Dec=Dec+(factor*(15));
             factor=factor/16;
          }

    }
    return Dec;
}



void GSM::VaciarPuertoSerieCsurv (){
  //Serial.println("Vaciando puerto serie");
  do{
    if(Serial.available()!=0){
      buffer[0]=buffer[1];
      buffer[1]=buffer[2];
      buffer[2]=buffer[3];
      buffer[3]=buffer[4];
      buffer[4]=(char)Serial.read();
    }
  }while( (buffer[3]!='O' || buffer[4]!='K') && (buffer[0]!='E' || buffer[1]!='R' ||
buffer[2]!='R' || buffer[3]!='O' || buffer[4]!='R'));

}

int GSM::getInfoCell(){
  byte i, j, cont;
  byte serialByte;
  unsigned int intermedio;
  char interm[6]="Vacio";

  buffer[0]='E'; buffer[1]='m';buffer[2]='p'; buffer[3]='t';buffer[4]='y';
  buffer[5]='E'; buffer[6]='m';buffer[7]='p'; buffer[8]='t';buffer[9]='y';
  buffer[10]='E'; buffer[11]='m';buffer[12]='p'; buffer[13]='t';buffer[14]='y';
  buffer[15]='E'; buffer[16]='m';buffer[17]='p'; buffer[18]='t';buffer[19]='y';
  buffer[20]='E'; buffer[21]='m';buffer[22]='p'; buffer[23]='t';buffer[24]='y';

  if(Serial.available()>0){
    VaciarPuertoSerie();
    delay(200);
```

```
  }
  Serial.println("AT#MONI\r");
  setTime();
    //delay(1000);
  while(Serial.available()<5 ){
        if((millis()-lastTime)>30000) {   //VIGILAR TIMEOUT 20 segundos
          Serial.println("MONI TIMEOUT");
          return 0;
      }
  }
  cont=0;
  do{
      if(Serial.available()>0){
          cont++;
          serialByte=Serial.read();
          buffer[0]=buffer[1];          buffer[1]=buffer[2];
buffer[2]=buffer[3];          buffer[3]=buffer[4];          buffer[4]=buffer[5];
          buffer[5]=buffer[6];          buffer[6]=buffer[7];
buffer[7]=buffer[8];          buffer[8]=buffer[9];          buffer[9]=buffer[10];
          buffer[10]=buffer[11];        buffer[11]=buffer[12];
buffer[12]=buffer[13];        buffer[13]=buffer[14];        buffer[14]=buffer[15];
          buffer[15]=buffer[16];        buffer[16]=buffer[17];
buffer[17]=buffer[18];        buffer[18]=buffer[19];        buffer[19]=buffer[20];
          buffer[20]=buffer[21];        buffer[21]=buffer[22];
buffer[22]=buffer[23];        buffer[23]=buffer[24];        buffer[24]=(char)serialByte;


          Serial.print(buffer[24]);

      }
      if((millis()-lastTime)>90000){
        Serial.println("MONI TIMEOUT 2");
          return 0;
      }
      else if (buffer[0]=='M' && buffer[1]=='O' && buffer[2]=='N' && buffer[3]=='I' &&
buffer[4]==':'){
        if (buffer[6]=='C' && buffer[7]=='c' && buffer[8]==':'){
          intermedio=0;
          ServingCell.NombreDisponible=false;
          for (i=9; i<19; i++){
            if (buffer[i]==' ' && buffer[i+1]=='N' && buffer[i+2]=='c' &&
buffer[i+3]==':'){
              for (j=(i-9); j>0; j--){
                intermedio=intermedio+(pow(10,j-1)*buffer[(i-j)+1]);
                Serial.print("Intermedio = ");
                Serial.println(intermedio);
              }
              ServingCell.Mcc=intermedio;
//Obtenemos MCC
              i=19;
            }

          }
        }
        else{
          ServingCell.NombreDisponible=true;
          for (i=6; i<19; i++){
            if (buffer[i]==' ' && buffer[i+1]=='B' && buffer[i+2]=='S' &&
buffer[i+3]=='I' && buffer[i+4]=='C' && buffer[i+5]==':'){
                //ServingCell.Nombre[i-6]='\0';
                i=19;
            }
            else{
              ServingCell.Nombre[i-6]=buffer[i];
//Obtenemos Nombre

            }
          }
        }
      }
      else if (buffer[0]=='L' && buffer[1]=='A' && buffer[2]=='C' && buffer[3]==':'){
```

```
        j=0;
           for(i=4; i<20; i++){
              if(buffer[i]==' ' && buffer[i+1]=='I' && buffer[i+2]=='d' &&
buffer[i+3]==':'){
                i=20;
              }
              else{
                interm[i-4]=buffer[i];
                j++;
              }

           }
           ServingCell.Lac=convHexaDec(interm,j);
//Obtenemos LAC

      }
      else if (buffer[0]=='I' && buffer[1]=='d' && buffer[2]==':' ){

          j=0;
           for(i=3; i<20; i++){
              if(buffer[i]==' ' && buffer[i+1]=='A' && buffer[i+2]=='R' &&
buffer[i+3]=='F' && (buffer[i+4]=='C' || buffer[i+4]=='N')){
                i=20;
              }
              else{
                interm[i-3]=buffer[i];
                j++;
              }

           }
           ServingCell.CellId=convHexaDec(interm,j);
//Obtenemos CellId

      }


      // ¿ Vigilancia de no pasarse de tiempo ?
    }while ((buffer[23]!='O' || buffer[24]!='K') && (buffer[20]!='E' || buffer[21]!='R'
|| buffer[22]!='R' || buffer[23]!='O' || buffer[24]!='R'));
    if (buffer[20]=='E' && buffer[21]=='R' && buffer[22]=='R' && buffer[23]=='O' &&
buffer[24]=='R')
      return 0;
    else{
      for(i=0; i<21; i++){
        if (buffer[i]=='P' && buffer[i+1]=='W' && buffer[i+2]=='R' && buffer[i+3]==':'){
            ServingCell.Pwr=10*(((int)buffer[i+5])-48)+(((int)buffer[i+6])-48);
            i=21;            //Obtenemos Pwr
        }
      }
      for(i=0; i<22; i++){
        if (buffer[i]=='T' && buffer[i+1]=='A' && buffer[i+2]==':' ){
          ServingCell.Ta=((int)buffer[i+3])-48;
          i=22;          //Obtenemos Ta
        }
      }
      return 1;
    }
}

int GSM::getInfoAllCells(){
  byte i,j;
  byte serialByte;
  byte lectura=0;
  byte numLecturas=0;
  unsigned int intermedio;

 if(Serial.available()>0){
    VaciarPuertoSerie();
    delay(200);
  }
  Serial.println("AT#CSURV\r");
```

```
  setTime();
  while(Serial.available()<25 ){
        if((millis()-lastTime)>15000) {   //VIGILAR TIMEOUT 15 segundos
          Serial.println("CSURV TIMEOUT");
          return 0;
      }
  }
  for(i=0;i<25;i++){
      Serial.read();
  }

  serialByte=' ';
  buffer[0]='E'; buffer[1]='m';buffer[2]='p'; buffer[3]='t';buffer[4]='y';
  buffer[5]='E'; buffer[6]='m';buffer[7]='p'; buffer[8]='t';buffer[9]='y';
  buffer[10]='E'; buffer[11]='m';buffer[12]='p'; buffer[13]='t';buffer[14]='y';
  buffer[15]='E'; buffer[16]='m';buffer[17]='p'; buffer[18]='t';buffer[19]='y';
  buffer[20]='E'; buffer[21]='m';buffer[22]='p'; buffer[23]='t';buffer[24]='y';


  setTime();
  do{
      if(Serial.available()>0){

          serialByte=Serial.read();
          buffer[0]=buffer[1];          buffer[1]=buffer[2];
buffer[2]=buffer[3];          buffer[3]=buffer[4];          buffer[4]=buffer[5];
          buffer[5]=buffer[6];          buffer[6]=buffer[7];
buffer[7]=buffer[8];          buffer[8]=buffer[9];          buffer[9]=buffer[10];
          buffer[10]=buffer[11];        buffer[11]=buffer[12];
buffer[12]=buffer[13];        buffer[13]=buffer[14];        buffer[14]=buffer[15];
          buffer[15]=buffer[16];        buffer[16]=buffer[17];
buffer[17]=buffer[18];        buffer[18]=buffer[19];        buffer[19]=buffer[20];
          buffer[20]=buffer[21];        buffer[21]=buffer[22];
buffer[22]=buffer[23];        buffer[23]=buffer[24];        buffer[24]=(char)serialByte;
          //Serial.print(buffer[24]);
      }
      if ( millis()-lastTime > 300000){   //290 segundos de timeout para el comando
haciendo un fullscan, ponemos 200 para timeout de lectura
        Serial.println("CSURV TIMEOUT 2");
        return 0;
      }

      else if (buffer[0]=='r' && buffer[1]=='x' && buffer[2]=='L' && buffer[3]=='e' &&
buffer[4]=='v' && buffer[5]==':'){
        for(i=9; i<13; i++){
          if(buffer[i]=='b' && buffer[i+1]=='e' && buffer[i+2]=='r' && buffer[i+3]==':'){
//Caso en que estamos ante una celda con los datos a buscar disponibles
//
            lectura=1;
          }
        }
        if (lectura==1 && buffer[6]==' '){
          j=0;
          intermedio=0;
          while(buffer[j+8]!=' '){
            j++;
          }
          for(int i=j; i>0; i--){
            intermedio=intermedio+(int)(potencia(i-1)*((int)buffer[8+j-i]-48));
          }
          CellArray[numLecturas].Pwr=intermedio;
        }

      }
      else if (lectura==1 && buffer[0]=='m' && buffer[1]=='c' && buffer[2]=='c' &&
buffer[3]==':'){
        CellArray[numLecturas].Mcc=(100*((int)buffer[5]-48))+(10*((int)buffer[6]-
48))+(1*((int)buffer[7]-48));
        lectura=2;
      }
      else if (lectura==2 && buffer[0]=='m' && buffer[1]=='n' && buffer[2]=='c' &&
```

```
buffer[3]==':' &&  buffer[4]==' ' && buffer[5]=='0' && buffer[6]=='1'){
        CellArray[numLecturas].Mnc=1;
        lectura=3;
      }
      else if (lectura==3 && buffer[0]=='l' && buffer[1]=='a' && buffer[2]=='c' &&
buffer[3]==':'){
          j=0;
          intermedio=0;
          while(buffer[j+5]!=' '){
            j++;
          }
          for(int i=j; i>0; i--){
            intermedio=intermedio+(int)(potencia(i-1)*((int)buffer[5+j-i]-48));
          }
          CellArray[numLecturas].Lac=intermedio;
          lectura=4;
      }
      else if (lectura==4 && buffer[0]=='c' && buffer[1]=='e' && buffer[2]=='l' &&
buffer[3]=='l' && buffer[4]=='I' && buffer[5]=='d' && buffer[6]==':'){
          j=0;
          intermedio=0;
          serialByte=0;
          while(buffer[j+8]!=' '){
            if(buffer[j+8]>=48 && buffer[j+8]<58){
              serialByte++;
            }
            j++;
          }

          for(int i=j; i>0; i--){
            if((int)buffer[8+j-i]>=48 && (int)buffer[8+j-i]<58){
              intermedio=intermedio+(int)(potencia(serialByte-1)*((int)buffer[8+j-i]-
48));
              serialByte--;

            }
          }
          CellArray[numLecturas].CellId=intermedio;
          lectura=0;
          numLecturas++;
      }

    }while ((buffer[23]!='O' || buffer[24]!='K') && (buffer[20]!='E' || buffer[21]!='R'
|| buffer[22]!='R' || buffer[23]!='O' || buffer[24]!='R') && (numLecturas < 30) &&
((CellArray[numLecturas].Pwr-CellArray[0].Pwr)<=75));

    VaciarPuertoSerie();

   /* for(i=0; i<numLecturas; i++){
      Serial.print("Antena numero "); Serial.print(i);
      Serial.print(" --> [Mcc: "); Serial.print(CellArray[i].Mcc);
      Serial.print("  Mnc: "); Serial.print(CellArray[i].Mnc);
      Serial.print("  Pwr: -"); Serial.print(CellArray[i].Pwr);
      Serial.print("dBm  Lac: "); Serial.print(CellArray[i].Lac);
      Serial.print("  Cell Id: "); Serial.print(CellArray[i].CellId);
      Serial.println("]");
    }*/
    if (buffer[20]=='E' && buffer[21]=='R' && buffer[22]=='R' && buffer[23]=='O' &&
buffer[24]=='R')
      return 0;
    else if (buffer[23]=='O' && buffer[24]=='K' )
      return numLecturas;
    else{
      VaciarPuertoSerieCsurv();
      return numLecturas;
    }
}


int GSM::MandarSMS(byte tipoSms, byte numAnts, float _DOP, char _coordinates[40])
{
```

```
 byte serialByte, caractCont, antCont;


 caractCont=0;
 Serial.print("AT+CMGS=");         // send Message
 Serial.print("0034633191768");
 Serial.print(",129\r");
 serialByte='A';
 setTime();
 do{
   if(Serial.available()){
      serialByte=Serial.read();
   }
 if(millis()-lastTime > 20000){
      Serial.println("CMGS Timeout 1");
      return 0;
    }
 }while((char)serialByte!='>');


//Leemos la palabra OK o ERROR
 if(tipoSms==1){
       Serial.println("Primer mensaje de Aviso ");
       Serial.println("Informacion Antena GSM: ");
       if (ServingCell.NombreDisponible==true){
         Serial.print("Nombre:");
         Serial.print(ServingCell.Nombre);
       }
       else{
         Serial.print("MCC:");
         Serial.print(ServingCell.Mcc);
         Serial.print("; MNC:");
         Serial.print(ServingCell.Mnc);
       }
       Serial.print("; LAC:");
       Serial.print(ServingCell.Lac, HEX);
       Serial.print("; CellId:");
       Serial.print(ServingCell.CellId, HEX);
       Serial.print("; Pwr:-");
       Serial.print(ServingCell.Pwr);
       Serial.print("dBm; Ta:");
       Serial.print(ServingCell.Ta);        // Message-Text
       Serial.write(26);      // CTRL-Z
 }
 else if(tipoSms==2){
    if (numAnts==1){
       Serial.print("POSIBLE FALLO EN LA PRECISION DE LA APROXIMACION-->");
    }
    Serial.print("Coordenadas del Sujeto: ");

    Serial.print(_coordinates);
    Serial.print("; Valor DOP:");

    Serial.print(_DOP);
       // Message-Text
    Serial.write(26); // CTRL-Z
 }
 else{
   antCont=0;
     while(antCont < numAnts){
       Serial.print(CellArray[antCont].Lac, HEX);
       Serial.print(",");
       Serial.print(CellArray[antCont].CellId, HEX);
       Serial.print(",");
       Serial.print(CellArray[antCont].Pwr);
       Serial.print("; ");
       antCont++;
     }
   Serial.write(26);
  }
```

```
  buffer[0]='E'; buffer[1]='m'; buffer[2]='p'; buffer[3]='t'; buffer[4]='y';
  setTime();
  do{
      if(Serial.available()>0){
          serialByte=Serial.read();
          buffer[0]=buffer[1];
          buffer[1]=buffer[2];
          buffer[2]=buffer[3];
          buffer[3]=buffer[4];
          buffer[4]=(char)serialByte;
          Serial.print(buffer[4]);
       }
      //  Vigilancia de no pasarse de tiempo
      if ( (millis()-lastTime)>30000){
         Serial.print("CMGS Timeout 2");
         return 0;
      }
    }while ((buffer[3]!='O' || buffer[4]!='K') && (buffer[0]!='E' || buffer[1]!='R' ||
buffer[2]!='R' || buffer[3]!='O' || buffer[4]!='R'));
   if ( buffer[0]=='E' && buffer[1]=='R' && buffer[2]=='R' && buffer[3]=='O' &&
buffer[4]=='R' ) {
      Serial.print("CMGS Error 1");
      return 0;
   }
  return 1;
}



void GSM::readGSMAnswer()
{
  if(readAnswer == 1)
  {
    inByte = Serial.read();

    if((char(inByte) != ' '))
    {
      //FIFO GSM-Answer
      temp_in[0] = temp_in[1];
      temp_in[1] = temp_in[2];
      temp_in[2] = temp_in[3];
      temp_in[3] = temp_in[4];
      temp_in[4] = temp_in[5];
      temp_in[5] = temp_in[6];
      temp_in[6] = temp_in[7];
      temp_in[7] = temp_in[8];
      temp_in[8] = temp_in[9];
      temp_in[9] = char(inByte);
      //Serial.print(char(inByte));
    }

    delay(50);

    if((temp_in[8] == 'O')  && (temp_in[9] == 'K') // If answer is OK!
    {
      GSM_OK = 1;
      Serial.print(" ");
      Serial.print(temp_in[8]);
      Serial.println(temp_in[9]);
    }

    else if(temp_in[9] == '>')    // If answer is >
    {
      GSM_OK = 1;
    }

    else if((temp_in[5] == 'R')  && (temp_in[6] == 'E')  && (temp_in[7] == 'A') &&
(temp_in[8] == 'D')  && (temp_in[9] == 'Y'))   // if SIM is ready no pin is needed
    {
      state = state + 1;
    }
```

```
    else if((temp_in[4] == 'S') && (temp_in[5] == 'I')  && (temp_in[6] == 'M')  &&
(temp_in[7] == 'P') && (temp_in[8] == 'U')  && (temp_in[9] == 'K'))
// if the PUK is needed
    {
      Serial.println("Please enter PUK and new PIN --> PUK,PIN");
      GSM_OK = 0;
      delay(5000);
      state = 8;      //end of Programm
    }

    else if((temp_in[5] == 'E')  && (temp_in[6] == 'R')  && (temp_in[7] == 'R') &&
(temp_in[8] == 'O')  && (temp_in[9] == 'R'))
    {
      sim_error = 1;

    }

    else if((temp_in[7] == '0') && (temp_in[8] == ',')  && (temp_in[9] == '0'))
    {
      reg_OK = 0;
      state = 0;
      readAnswer = 0;
    }

    else if((temp_in[7] == '0') && (temp_in[8] == ',')  && (temp_in[9] == '1'))
    {
      reg_OK = 1;

    }

    else if((temp_in[7] == '0') && (temp_in[8] == ',')  && (temp_in[9] == '5'))
    {
      reg_OK = 1;

    }


    if(int(Serial.available()) <= 1)
    {
      if((sim_error == 1) && ((state == 4) || (state == 5)))
      {
        GSM_OK = 0;

        Serial.println("ERROR, Attention: Please Check your SIM-PIN and restart the
program!");
        delay(5000);

        readAnswer = 0;

        while(1);
      }

      if((reg_OK == 0) && (state == 6))
      {
        GSM_OK = 0;

        readAnswer = 0;
      }

      if(GSM_OK == 1)
      {
        readAnswer = 0;

        state = state + 1;  // go to next state in current function

        GSM_OK = 0;
        delay(500);
      }
    }
  }
```

```
}

GPS::GPS(int baud) // With Arduino Duemilanove (ATmega328) & Arduino UNO (ATmega328)
{
  _baud = baud;
}

char GPS::initializeGPS()
{
  char test_data = 0;
  char clr_register = 0;

  // set pin's for SPI
  pinMode(MOSI, OUTPUT);
  pinMode(MISO, INPUT);
  pinMode(SCK,OUTPUT);
  pinMode(CS,OUTPUT);
  digitalWrite(CS,HIGH);

  SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR1)|(1<<SPR0);  // Initalize the SPI-Interface
  clr_register=SPSR;         // read register to clear them
  clr_register=SPDR;
  delay(10);

  Serial.begin(_baud);

  WriteByte_SPI_CHIP(LCR, 0x80);   // set Bit 7 so configure baudrate
  WriteByte_SPI_CHIP(DLL, 0x18);   // 0x18 = 9600 with Xtal = 3.6864MHz
  WriteByte_SPI_CHIP(DLM, 0x00);   // 0x00 = 9600 with Xtal = 3.6864MHz

  WriteByte_SPI_CHIP(LCR, 0xBF);   // configure uart
  WriteByte_SPI_CHIP(EFR, 0x10);   // activate enhanced registers
  WriteByte_SPI_CHIP(LCR, 0x03);   // Uart: 8,1,0
  WriteByte_SPI_CHIP(FCR, 0x06);   // Reset FIFO registers
  WriteByte_SPI_CHIP(FCR, 0x01);   // enable FIFO Mode

  // configure GPIO-Ports
  WriteByte_SPI_CHIP(IOCTL, 0x01);    // set as GPIO's
  WriteByte_SPI_CHIP(IODIR, 0x04);    // set the GPIO directions
  WriteByte_SPI_CHIP(IOSTATE, 0x00);  // set default GPIO state


//Cold Restart  $PMTK104*37<CR><LF>
  resetGPS();

  // Check functionality
  WriteByte_SPI_CHIP(SPR, 'A');       // write an a to register and read it
  test_data = ReadByte_SPI_CHIP(SPR);
    if(test_data == 'A')
    return 1;
  else
    return 0;

}



void GPS::getGPS()
{
  byte gps_data_buffer[20];
  char in_data;
  char no_gps_message[20] = "no valid gps data\r\n";
  int high_Byte;
  int i,j,k;
  int GPGGA;
  int Position;

  GPGGA = 0;
  i = 0;

  do
```

```
  {
    if(ReadByte_SPI_CHIP(LSR) & 0x01)
    {
      in_data = ReadByte_SPI_CHIP(RHR);

      // FIFO-System to buffer incomming GPS-Data
      gps_data_buffer[0] = gps_data_buffer[1];
      gps_data_buffer[1] = gps_data_buffer[2];
      gps_data_buffer[2] = gps_data_buffer[3];
      gps_data_buffer[3] = gps_data_buffer[4];
      gps_data_buffer[4] = gps_data_buffer[5];
      gps_data_buffer[5] = gps_data_buffer[6];
      gps_data_buffer[6] = gps_data_buffer[7];
      gps_data_buffer[7] = gps_data_buffer[8];
      gps_data_buffer[8] = gps_data_buffer[9];
      gps_data_buffer[9] = gps_data_buffer[10];
      gps_data_buffer[10] = gps_data_buffer[11];
      gps_data_buffer[11] = gps_data_buffer[12];
      gps_data_buffer[12] = gps_data_buffer[13];
      gps_data_buffer[13] = gps_data_buffer[14];
      gps_data_buffer[14] = gps_data_buffer[15];
      gps_data_buffer[15] = gps_data_buffer[16];
      gps_data_buffer[16] = gps_data_buffer[17];
      gps_data_buffer[17] = gps_data_buffer[18];
      gps_data_buffer[18] = in_data;



      if((gps_data_buffer[0] == '$') && (gps_data_buffer[1] == 'G') &&
(gps_data_buffer[2] == 'P') && (gps_data_buffer[3] == 'G') && (gps_data_buffer[4] ==
'G')&& (gps_data_buffer[5] == 'A'))
      {
          GPGGA = 1;
      }

      if((GPGGA == 1) && (i < 80))
      {
        if( (gps_data_buffer[0] == 0x0D))// every answer of the GPS-Modul ends with an
cr=0x0D
        {
          i = 80;
          GPGGA = 0;
         //Serial.println("");
        }
        else
        {
          gps_data[i] = gps_data_buffer[0];  // write Buffer into public variable
         //Serial.print(gps_data[i]);
          i++;
        }
      }
    }
  }while(i<80) ;

  // filter gps data

  if(int(gps_data[18]) == 44)
  {
    j = 0;
    for(i = 0; i < 20; i++)
    {
      coordinates[j] = no_gps_message[i];  // no gps data available at present!
      j++;
    }
    HDOP=100;
  }
  else
  {
    j = 0;                                  // format latitude
    for(i = 18; i < 29 ; i++)
    {
```

```
      if(gps_data[i] != ',')
      {
        latitude[j] = gps_data[i];
        j++;
      }

      if(j==2)
      {
       latitude[j] = ' ';
       j++;
      }
    }

    j = 0;
    for(i = 31; i < 42 ; i++)
    {                                          // format longitude
      if(gps_data[i] != ',')
      {
        longitude[j] = gps_data[i];
        j++;
      }

      if(j==2)
      {
        longitude[j] = ' ';
        j++;
      }
    }

      j=0;                                      // format longitude
      for(i=0; i<80; i++){
       if (int(gps_data[i]) == 44)
              j++;
       if (j==8){
              k=i+1;
              i=80;
       }
      }
      if(gps_data[k+1]==46){
       HDOP=(((float)gps_data[k])-48)+((0.10)*(((float)gps_data[k+2])-
48))+((0.01)*(((float)gps_data[k+3])-48));
        i=80;
      }
      else{
        HDOP=(10*(((float)gps_data[k])-48))+(((float)gps_data[k+1])-
48)+((0.10)*(((float)gps_data[k+3])-48))+((0.01)*(((float)gps_data[k+4])-48));
       i=80;
      }


    for(i = 0; i < 40; i++)                // clear coordinates
      coordinates[i] = ' ';

    j = 0;
    for(i = 0; i < 11; i++)                // write gps data to coordinates
    {
      coordinates[j] = latitude[i];
      j++;
    }

    coordinates[j] = ',';
    j++;

    coordinates[j] = ' ';
    j++;

    for(i = 0; i < 11; i++)
    {
      coordinates[j] = longitude[i];
      j++;
    }
```

```
    coordinates[j++] = '\r';
    coordinates[j++] = '\n';
    coordinates[j++] = '\0';
  }
}

void GPS::resetGPS(){
       //Cold Restart  $PMTK104*37<CR><LF>
       WriteByte_SPI_CHIP(THR, '$');
       WriteByte_SPI_CHIP(THR, 'P');
       WriteByte_SPI_CHIP(THR, 'M');
       WriteByte_SPI_CHIP(THR, 'T');
       WriteByte_SPI_CHIP(THR, 'K');
       WriteByte_SPI_CHIP(THR, '1');
       WriteByte_SPI_CHIP(THR, '0');
       WriteByte_SPI_CHIP(THR, '4');
       WriteByte_SPI_CHIP(THR, '*');
       WriteByte_SPI_CHIP(THR, '3');
       WriteByte_SPI_CHIP(THR, '7');
       WriteByte_SPI_CHIP(THR, 0x0D);
       WriteByte_SPI_CHIP(THR, 0x0A);


}



void GPS::habilitarSBAS(){
       //Habilitar SBAS  $PMTK313,1*2E<CR><LF>
       WriteByte_SPI_CHIP(THR, '$');
       WriteByte_SPI_CHIP(THR, 'P');
       WriteByte_SPI_CHIP(THR, 'M');
       WriteByte_SPI_CHIP(THR, 'T');
       WriteByte_SPI_CHIP(THR, 'K');
       WriteByte_SPI_CHIP(THR, '3');
       WriteByte_SPI_CHIP(THR, '1');
       WriteByte_SPI_CHIP(THR, '3');
       WriteByte_SPI_CHIP(THR, ',');
       WriteByte_SPI_CHIP(THR, '1');
       WriteByte_SPI_CHIP(THR, '*');
       WriteByte_SPI_CHIP(THR, '2');
       WriteByte_SPI_CHIP(THR, 'E');
       WriteByte_SPI_CHIP(THR, 0x0D);
       WriteByte_SPI_CHIP(THR, 0x0A);
}


char GPS::checkS1()
{
  int value;
  value = (ReadByte_SPI_CHIP(IOSTATE) & 0x01);       // read S1 button state
  return value;
}

char GPS::checkS2()
{
  int value;
  value = (ReadByte_SPI_CHIP(IOSTATE) & 0x02);  // read S2 button state
  return value;
}

void GPS::setLED(char state)
{
  if(state == 0)
    WriteByte_SPI_CHIP(IOSTATE, 0x00);         // turn off LED
  else
    WriteByte_SPI_CHIP(IOSTATE, 0x07);         // turn on LED
}

void GPS::WriteByte_SPI_CHIP(char adress, char data)
```

```
{
  char databuffer[2];
  int i;

  databuffer[0] = adress;
  databuffer[1] = data;

  Select_SPI(CS);                                 // pull CS-line low

  for(i = 0; i < 2; i++)
    SPI.transfer(databuffer[i]);                  // write data

  Release_SPI(CS);                                // release CS-Line
}

char GPS::ReadByte_SPI_CHIP(char adress)
{
  char incomming_data;

  adress = (adress | 0x80);

  Select_SPI(CS);

  SPI.transfer(adress);
  incomming_data = SPI.transfer(0xFF);

  Release_SPI(CS);

  return incomming_data;
}

void GPS::Select_SPI(char cs_pin)
{
  digitalWrite(cs_pin, LOW);
}

void GPS::Release_SPI(char cs_pin)
{
  digitalWrite(cs_pin, HIGH);
}
```

Programa principal

```
#include <SIEMERG_GSM_GPS_Shield.h>

/*
Using_GSM_and_GPS.pde - Example for using the GSM and GPS-Modul Rev.6
Included Functions
Version:     1.3.2
Date:        16.03.2012
Company:     antrax Datentechnik GmbH
Uses with:   Arduino Duemilanove (ATmega328) and
             Arduino UNO (ATmega328)
*/
//#include <avr/sleep.h>
#if defined(ARDUINO) && ARDUINO >= 100
  // Choose Arduino.h for IDE 1.0
  #include "Arduino.h"
#else
  // Choose WProgram.h if IDE is older than 1.0
  #include "WProgram.h"
#endif

#include <SPI.h>


GSM gsm(7,9600);                                                              //
(power_pin, baudrate)
GPS gps(9600);                                                                //
(baudrate)
float DOP, last_DOP;
byte error_gps=0;
byte num_antenas=0;
long time;
void(* resetFunc) (void) = 0;

void setup()
{

  if(gps.initializeGPS())
    Serial.println("GPS ON");
  else
    resetFunc();
  gps.habilitarSBAS();
  delay(200);
  gsm.initializeGSM("3815");
  time=millis();
  Serial.print("t conex = ");
  Serial.println(time);

  // Enter your SIM-Pin if it's required (Example: "1234") --> the quote signs are
important
  //delay(200);
  gsm.setExternalTime();
  while(gsm.getInfoCell()==0){
     //PRIMER SMS
     if ((millis()-gsm.externalTime)>40000){
       Serial.println("Fallo tras varias llamadas a MONI");
       resetFunc();
     }
  }
  Serial.print("t info = ");
  Serial.println(millis());

 gsm.setExternalTime();
 while(!gsm.MandarSMS(1,0,0,"")){
            delay(500);
            if ((millis()-gsm.externalTime)>40000){
               Serial.println("Fallo tras varias llamadas a CMGS para primer sms aviso");
               resetFunc();
     }
```

```
          }
  time=millis();
  Serial.print("t sms1 = ");
  Serial.println(millis());

  gps.getGPS();
  gsm.setExternalTime();
  while(gps.coordinates[0] == 'n')
  {
    gps.getGPS();
    if (error_gps==0){
      delay(20);
      gps.setLED(1);
      delay(20);
      gps.setLED(0);
    }
    if ((millis()-gsm.externalTime)>90000 && error_gps==0){     //  El GPS tarda mucho
          Serial.println("GPS timeout");   //aqui vamos a mandar a conseguir toda la info
de cell
          error_gps=1;
          Serial.println("");
          Serial.print("t Inicial = ");
          Serial.println(millis());
          delay(750);
         do{
           num_antenas=gsm.getInfoAllCells();
           delay(500);
          }while(!num_antenas);
          time=millis();
          Serial.print("t final = ");
          Serial.println(time);
          gsm.setExternalTime();
          while(!gsm.MandarSMS(3,num_antenas,0,"")){
            delay(500);
            if ((millis()-gsm.externalTime)>40000){
               Serial.println("Fallo tras varias llamadas a CMGS xa mandar info cells
tras gps timeout");
               resetFunc();
             }
          }
          time=millis();
          Serial.print("t sms2 = ");
          Serial.println(time);
    }
  }
  last_DOP=134; //Así si se obtiene por primera vez DOP=99.99 se enviará el SMS
(99.99/0.75 = 133.32)
}



void loop()
{


  DOP=gps.HDOP;
  Serial.print("t_salida = ");
  Serial.print(millis());
  Serial.print("--> ");
  Serial.print(gps.coordinates);
  Serial.print("--> DOP = ");
  Serial.println(DOP);
  gps.getGPS();
  if (DOP>2.5 && DOP < 0.5*last_DOP){
     // Aqui se manda un sms advirtiendo alta DOP con las coords obtenidas y se buscan la
info de todas las celdas
     //
      last_DOP=DOP;
       gsm.setExternalTime();
       while(!gsm.MandarSMS(2,1,DOP,gps.coordinates)){
            delay(500);
```

```
            if ((millis()-gsm.externalTime)>40000){
               Serial.println("Fallo tras varias llamadas a CMGS gps alto dop");
               resetFunc();
             }
        }
        time=millis();
          Serial.print("t sms3 = ");
          Serial.println(millis());
     if(error_gps==0){
         error_gps=2;
         Serial.print("t Inicial = ");
         Serial.println(millis());
         delay(100);
         do{
           num_antenas=gsm.getInfoAllCells();
           delay(500);
          }while(!num_antenas);
         gsm.setExternalTime();
       time=millis();
          Serial.print("t final = ");
          Serial.println(millis());
         while(!gsm.MandarSMS(3,num_antenas,0,"")){
            delay(500);
            if ((millis()-gsm.externalTime)>40000){
               Serial.println("Fallo tras varias llamadas a CMGS sms tras alto dop");
               resetFunc();
             }
         }
         time=millis();
          Serial.print("t sms4 = ");
          Serial.println(millis());

     }

  }
  else if (DOP<2.5){
  // LLegados a este punto todo ha ido ok mandar sms con las coordenadas mandar también
el DOP sin advertencia
     Serial.print("salida a las ");
     Serial.print(millis());
     Serial.print("--> ");
     Serial.print(gps.coordinates);
     Serial.print("--> DOP = ");
     Serial.println(DOP);
     gsm.setExternalTime();
      while(!gsm.MandarSMS(2,0,DOP,gps.coordinates)){
            delay(500);
            if ((millis()-gsm.externalTime)>40000){
               Serial.println("Fallo tras varias llamadas a CMGS sms final");
               resetFunc();
             }
         }
      gps.setLED(1);
      Serial.print("Tiempo Final del programa completo --> ");
      Serial.print(millis());
      Serial.println(" segundos; ¡TODO OK!");
      while(1);
  }
}
```

E - Programa completo del dispositivo final

```
import MDM
import SER
import time
import GPIO

def resetGPS():
    a=-1
    while a==-1:
        a=MDM.send('AT$GPSR=1\r\n',1)
        pprint('AT$GPSR=1\r\n')

def getGPS():
    global HDOP
    global Coordinates
    global tiempo
    a=-1
    while a==-1:
        a=MDM.send('AT$GPSACP\r\n',1)
    #pprint('AT$GPSACP\n')
    setTime()
    c=-1
    while c==-1: #Tiempo
        if time.clock()- tiempo > 8:
            HDOP=101
            return
        c=MDM.readbyte()
    d=MDM.read()
    setTime()
    while 'ERROR' in d: #Tiempo
        a=-1
        while a==-1:
            a=MDM.send('AT#GPSACP\r\n',1)
        pprint('AT#GPSACP\n')
        c=-1
        while c==-1:
            if time.clock()- tiempo > 15:
                HDOP=101
                return
            c=MDM.readbyte()
        d=MDM.read()
    e=d.split(',',44)
    if len(e[1])>=2:
        #ppprint(d)
        Coordinates=e[1]+' '+e[2]
        HDOP=float(e[3])
    else:
        HDOP=101


def restart_program():
    a=-1
    while a==-1:
        pprint('AT#REBOOT\n')
        a=MDM.send('AT#REBOOT\r\n',1)

def pprint(cadena):
    SER.send(str(cadena))
    SER.sendbyte(0x0a)
    SER.sendbyte(0x0d)

def ppprint(cadena):
    SER.send(str(cadena))

def setTime():
    global tiempo
    tiempo=time.clock()

def limpiar (cadena):
```

```
    caden=bytearray(cadena)
    j=0
    for i in range(0, len(caden)):
        if caden[i:i+1]!='\n' and caden[i:i+1]!='\r' :
            caden[j]=cadena[i]
            j=j+1
    return caden[0:j]



def respInicGSM():
    global estado
    global fallosRegistro
    global tiempo
    primerByte=-1
    k=0
    setTime()
    while primerByte==-1:
        primerByte=MDM.readbyte()
        if time.clock()-tiempo > 10:
            fallosRegistro=fallosRegistro+1
            pprint('timeout_respInic')
            return
    resp=MDM.read()
    if 'OK' in resp:
        pprint('OK')
        if estado==1 or estado==2 or estado==4:
            estado=estado+1
        elif '0,1' in resp:
            pprint('0,1')
            if estado==3:
                estado=4
        elif '0,5' in resp:
            pprint('0,5')
            if estado==3:
                estado=4
        elif '0,0' in resp and estado==3:
            pprint('0,0')
            estado=1
    elif 'ERROR' in resp and estado!=2:
        pprint('ERROR')
        fallosRegistro=fallosRegistro+1
    elif 'ERROR' in resp and estado==2:
        pprint('ERROR')
        estado=0


def inic_gsm():
    global estado
    global fallosRegistro
    a=-1
    while(estado<5):
        if fallosRegistro > 5:
            estado=1
            pprint('reboot')
            restart_program()
        if estado==1:
            while a==-1:
                a=MDM.send('AT\r\n',1)
                pprint('AT\n')
            a=-1
            time.sleep(0.005)
            respInicGSM()
        elif estado==2:
            while a==-1:
                a=MDM.send('AT+CPIN=3913\r\n',1)
                pprint('AT+CPIN=3913\r\n')
            a=-1
            time.sleep(0.005)
            respInicGSM()
        elif estado==3:
```

```
            while a==-1:
                a=MDM.send('AT+CREG?\r\n',1)
                pprint('AT+CREG?\n')
            a=-1
            time.sleep(2)
            respInicGSM()
        elif estado==4:
            while a==-1:
                a=MDM.send('AT+CMGF=1\r\n',1)
                pprint('AT+CMGF=1\n')
            a=-1
            time.sleep(1)
            respInicGSM()
        elif estado==5:
            return estado
        else:
            return 0
    return estado

def obtenerInfoCell():
    global tiempo
    a=-1
    while a==-1: #Tiempo
        a=MDM.send('AT#MONI\r\n',1)
        pprint('AT#MONI\r\n')
    primerByte=-1
    setTime()
    while primerByte==-1:
        primerByte=MDM.readbyte()
        if time.clock()-tiempo > 20:
            pprint('timeout moni')
            return ()
    pprint('INFO:')
    info=MDM.read()
    if 'ERROR' in info:
        pprint('error moni')
        return ()
    pprint(info)
    inferior=info.find(' LAC:')+5
    superior=info.find(' Id:')
    LAC=info[inferior:superior]
    CELLID=info[info.find(' Id:')+4:info.find(' ARFCN:')]
    PWR=info[info.find(' PWR:')+5:info.find(' TA:')]
    TA=info[info.find(' TA:')+4:info.find(' TA:')+5]
    if '#MONI: Cc:' in info:
        MCC=info[info.find(' Cc:')+4:info.find(' Cc:')+7]
        MNC=info[info.find('Nc:')+3:info.find('Nc:')+5]
        informacion=(MCC, MNC, LAC, CELLID, PWR, TA)
    else:
        nombre=info[info.find('#MONI: ')+7:info.find('BSIC:')]
        informacion=(nombre, LAC, CELLID, PWR, TA)
    return informacion



def mandarSms(cadena):
    global tiempo
    envioOk=-1
    while envioOk==-1:#Tiempo
        envioOk=MDM.send('AT+CMGS=0034633191768,129\r\n',1)
        pprint('AT+CMGS=0034609381404,129\r\n')
    respuesta=''
    setTime()
    while len(respuesta)==0: #FEO
        respuesta=MDM.read()
        if time.clock()-tiempo > 10:
            pprint('timeout1 en cmgs')
            return -1
    if '>' in respuesta:
        pprint('INFO:')
        envioOk=-1
```

```
        while envioOk==-1:
            envioOk=MDM.send(str(cadena),1)
            pprint(cadena)
        envioOk=-1
        while envioOk==-1:
            envioOk=MDM.sendbyte(0x1a, 1)
        respuesta=''
        setTime()
        while len(respuesta)==0:
            respuesta=MDM.read()
            if time.clock()-tiempo > 20:
                pprint('timeout2 en cmgs')
                return -1
        if 'OK' in respuesta:
            pprint('Resp Envio OK')
            return 1
        else:
            pprint('Resp Envio ERROR')
            return 0
    else:
        pprint('CMGS da error')
        return 0



def obtenerInfoAllCells():
    global tiempo
       #         1         2         3         4         5         6         7
    s='012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789'
    CellList=[]
    buff=bytearray(s)
    a=-1
    while a==-1: #Tiempo
        a=MDM.send('AT#CSURV\r\n',1)
        pprint('AT#CSURV\r\n')
    setTime()
    c=-1
    while c==-1:
        c=MDM.readbyte()
        if time.clock() - tiempo > 20:
            pprint('Csurv timeout 1')
            return 0
    setTime()
    while buff[53:79]!='Network survey started ...':
        z=MDM.readbyte()
        if time.clock() - tiempo > 30:
            pprint('Csurv timeout 2')
            return 0
        if z!=-1:
            buff[0],buff[1],buff[2],buff[3],buff[4] =
buff[1],buff[2],buff[3],buff[4],buff[5]
            buff[5],buff[6],buff[7],buff[8],buff[9] =
buff[6],buff[7],buff[8],buff[9],buff[10]
            buff[10],buff[11],buff[12],buff[13],buff[14] =
buff[11],buff[12],buff[13],buff[14],buff[15]
            buff[15],buff[16],buff[17],buff[18],buff[19] =
buff[16],buff[17],buff[18],buff[19],buff[20]
            buff[20],buff[21],buff[22],buff[23],buff[24] =
buff[21],buff[22],buff[23],buff[24],buff[25]
            buff[25],buff[26],buff[27],buff[28],buff[29] =
buff[26],buff[27],buff[28],buff[29],buff[30]
            buff[30],buff[31],buff[32],buff[33],buff[34] =
buff[31],buff[32],buff[33],buff[34],buff[35]
            buff[35],buff[36],buff[37],buff[38],buff[39] =
buff[36],buff[37],buff[38],buff[39],buff[40]
            buff[40],buff[41],buff[42],buff[43],buff[44] =
buff[41],buff[42],buff[43],buff[44],buff[45]
            buff[45],buff[46],buff[47],buff[48],buff[49] =
buff[46],buff[47],buff[48],buff[49],buff[50]
            buff[50],buff[51],buff[52],buff[53],buff[54] =
buff[51],buff[52],buff[53],buff[54],buff[55]
```

```
            buff[55],buff[56],buff[57],buff[58],buff[59] =
buff[56],buff[57],buff[58],buff[59],buff[60]
            buff[60],buff[61],buff[62],buff[63],buff[64] =
buff[61],buff[62],buff[63],buff[64],buff[65]
            buff[65],buff[66],buff[67],buff[68],buff[69] =
buff[66],buff[67],buff[68],buff[69],buff[70]
            buff[70],buff[71],buff[72],buff[73],buff[74] =
buff[71],buff[72],buff[73],buff[74],buff[75]
            buff[75],buff[76],buff[77],buff[78],buff[79] =
buff[76],buff[77],buff[78],buff[79],z
            #ppprint(buff[78:79])
    setTime()
    while buff[59:79]!='Network survey ended':
        z=MDM.readbyte()
        if time.clock() - tiempo > 220:
            pprint('Csurv timeout3')
            return 0
        if z!=-1:
            buff[0],buff[1],buff[2],buff[3],buff[4] =
buff[1],buff[2],buff[3],buff[4],buff[5]
            buff[5],buff[6],buff[7],buff[8],buff[9] =
buff[6],buff[7],buff[8],buff[9],buff[10]
            buff[10],buff[11],buff[12],buff[13],buff[14] =
buff[11],buff[12],buff[13],buff[14],buff[15]
            buff[15],buff[16],buff[17],buff[18],buff[19] =
buff[16],buff[17],buff[18],buff[19],buff[20]
            buff[20],buff[21],buff[22],buff[23],buff[24] =
buff[21],buff[22],buff[23],buff[24],buff[25]
            buff[25],buff[26],buff[27],buff[28],buff[29] =
buff[26],buff[27],buff[28],buff[29],buff[30]
            buff[30],buff[31],buff[32],buff[33],buff[34] =
buff[31],buff[32],buff[33],buff[34],buff[35]
            buff[35],buff[36],buff[37],buff[38],buff[39] =
buff[36],buff[37],buff[38],buff[39],buff[40]
            buff[40],buff[41],buff[42],buff[43],buff[44] =
buff[41],buff[42],buff[43],buff[44],buff[45]
            buff[45],buff[46],buff[47],buff[48],buff[49] =
buff[46],buff[47],buff[48],buff[49],buff[50]
            buff[50],buff[51],buff[52],buff[53],buff[54] =
buff[51],buff[52],buff[53],buff[54],buff[55]
            buff[55],buff[56],buff[57],buff[58],buff[59] =
buff[56],buff[57],buff[58],buff[59],buff[60]
            buff[60],buff[61],buff[62],buff[63],buff[64] =
buff[61],buff[62],buff[63],buff[64],buff[65]
            buff[65],buff[66],buff[67],buff[68],buff[69] =
buff[66],buff[67],buff[68],buff[69],buff[70]
            buff[70],buff[71],buff[72],buff[73],buff[74] =
buff[71],buff[72],buff[73],buff[74],buff[75]
            buff[75],buff[76],buff[77],buff[78],buff[79] =
buff[76],buff[77],buff[78],buff[79],z
           # ppprint(buff[78:79])
            if buff[74:79]=='ERROR':
                pprint('error')
                return 0
            elif buff[77:79]=='OK':
                pprint('fin anticipado')
                if len(CellList)==0:
                    pprint('no hay vecinos')
                    return 1
                return CellList
            elif buff[24:40]=='mcc: 214 mnc: 01':
                #pprint('Encontramos uno')
                inferior=buff.find(' lac: ')+6
                superior=buff.find(' cellId: ')
                LAC=buff[inferior:superior]
                inferior=buff.find(' cellId: ')+9
                superior=buff.find(' cellStatus')
                CELLID=buff[inferior:superior]
                inferior=buff.find('rxLev: ')+7
                superior=buff.find(' ber: ')
                PWR=buff[inferior:superior]
```

```
                #pprint((str(PWR),str(LAC),str(CELLID)))
                CellList.append((str(PWR),str(LAC),str(limpiar(str(CELLID)))))
    y=MDM.read()
    pprint (y)
    return CellList

pprint('Texe: '+str(time.clock()))
estado=1
tiempo=time.clock()
fallosRegistro=0
#Inicializacion receptor GNSS
resetGPS()
GPIO.setIOdir(9,1,0)
GPIO.setIOdir(8,1,0)
GPIO.setIOdir(10,1,0)
GPIO.setIOdir(5,1,0)
#time.sleep(0.5)
GPIO.setIOvalue(9,1)
SER.set_speed('115200')
res=inic_gsm()
pprint('Treg: '+str(time.clock()))
GPIO.setIOvalue(8,1)
info=()
fallos=-1
while len(info)==0:
    fallos=fallos+1
    if fallos==10:
        restart_program()
    info=obtenerInfoCell()   #con len(info) sabemos de que tipo es
pprint('Tinfo: '+str(time.clock()))
fallos=-1
while mandarSms(info)!=1:
    fallos=fallos+1
    if fallos==10:
        restart_program()
    time.sleep(1)
pprint('Tsms1: '+str(time.clock()))
GPIO.setIOvalue(10,1)
timeout = 0
fin_bucle=0
Coordinates=0
HDOP=101
lastHDOP=143
externalTime=time.clock()
while fin_bucle==0:
    #Conseguir coordenadas y HDOP
    getGPS()
    if HDOP >= 2.5 and HDOP < 100 and HDOP <= 0.7*lastHDOP:
        lastHDOP=HDOP
        fallos=-1
        pprint('Tcoord: '+str(time.clock()))
        while(mandarSms(Coordinates+'-->HDOP='+str(HDOP))!=1):
            fallos=fallos+1
            if fallos==10:
                restart_program()
            time.sleep(1)
        pprint('Tsmscoord: '+str(time.clock()))
        if timeout==0:
            t=0
            fallos=-1
            pprint('TinicCsurv: '+str(time.clock()))
            while t==0:
                fallos=fallos+1
                if fallos==10:
                    restart_program()
                time.sleep(10)
                t=obtenerInfoAllCells()
            pprint('TfinCsurv: '+str(time.clock()))
            while mandarSms(t)!=1:
                time.sleep(1)
            timeout=1
```

```
            pprint('Tsms_csurv: '+str(time.clock()))
    elif HDOP < 2.5:
        fallos=-1
        pprint('TcoordOK: '+str(time.clock()))
        while(mandarSms(Coordinates+'-->HDOP='+str(HDOP))!=1):
            fallos=fallos+1
            if fallos==10:
                restart_program()
            time.sleep(1)
        pprint('TsmsCoord: '+str(time.clock()))
        fin_bucle=1
    elif time.clock()-externalTime > 60 and timeout==0:
        timeout=1
        t=0
        fallos=-1
        pprint('TinicCsurv: '+str(time.clock()))
        while t==0:
            fallos=fallos+1
            if fallos==10:
                restart_program()
            time.sleep(10)
            t=obtenerInfoAllCells()
        pprint('TfinCsurv: '+str(time.clock()))
        fallos=-1
        while mandarSms(t)!=1:
            fallos=fallos+1
            if fallos==10:
                restart_program()
            time.sleep(1)
        #Fin del programa para GE910 sin GNSS
        pprint('Tsms_csurv: '+str(time.clock()))

    GPIO.setIOvalue(5,1)
    time.sleep(1)
    GPIO.setIOvalue(5,0)
    time.sleep(1)
GPIO.setIOvalue(5,1)
pprint('fin')
```

F - Presupuesto del Proyecto:

1) Ejecución Material

- Compra de ordenador personal ..... 1.000 €
- Licencias Software..... ..... 2.000 €
- Alquiler de impresora láser durante 6 meses.....50 €
- Compra componentes del proyecto .....3250 €
- Creación de PCB .....800 €
- Montaje PCB .....400 €
- Alquiler de máquina fresadora durante 15 días .....1500 €
- Medidas con osciloscopio durante 7 días .....70 €
- Fuente de Alimentación.....300 €
- Multímetro.....50 €
- Total de ejecución material.....9420 €

2) Gastos generales

- 16 % sobre Ejecución Material ..... 1507.2 €

3) Beneficio Industrial

- 6 % sobre Ejecución Material ..... 565.2 €

4) Honorarios Proyecto

- 2000 horas a 15 € / hora ..... 30000 €

5) Material fungible

- Gastos de impresión ..... 200 €
- Encuadernación ..... 5 €

6) Subtotal del presupuesto

- Subtotal Presupuesto ..... 41697.4 €

7) I.V.A. aplicable

- 21% Subtotal Presupuesto..... 8756.45 €

8) Total presupuesto

- Total Presupuesto ..... 50453.85 €

Madrid, Julio de 2014
El Ingeniero Jefe de Proyecto
Fdo.: Guillermo Montaner Andrés
Ingeniero de Telecomunicación

Pliego de condiciones

Este documento contiene las condiciones legales que guiarán la realización, en este proyecto, de un sistema de tele-emergencia para personas con necesidades especiales. En lo que sigue, se supondrá que el proyecto ha sido encargado por una empresa cliente a una empresa consultora con la finalidad de realizar dicho sistema. Dicha empresa ha debido desarrollar una línea de investigación con objeto de elaborar el proyecto. Esta línea de investigación, junto con el posterior desarrollo de los programas está amparada por las condiciones particulares del siguiente pliego.

Supuesto que la utilización industrial de los métodos recogidos en el presente proyecto ha sido decidida por parte de la empresa cliente o de otras, la obra a realizar se regulará por las siguientes:

Condiciones generales

1. La modalidad de contratación será el concurso. La adjudicación se hará, por tanto, a la proposición más favorable sin atender exclusivamente al valor económico, dependiendo de las mayores garantías ofrecidas. La empresa que somete el proyecto a concurso se reserva el derecho a declararlo desierto.

2. El montaje y mecanización completa de los equipos que intervengan será realizado totalmente por la empresa licitadora.

3. En la oferta, se hará constar el precio total por el que se compromete a realizar la obra y el tanto por ciento de baja que supone este precio en relación con un importe límite si este se hubiera fijado.

4. La obra se realizará bajo la dirección técnica de un Ingeniero Superior de Telecomunicación, auxiliado por el número de Ingenieros Técnicos y Programadores que se estime preciso para el desarrollo de la misma.

5. Aparte del Ingeniero Director, el contratista tendrá derecho a contratar al resto del personal, pudiendo ceder esta prerrogativa a favor del Ingeniero Director, quien no estará obligado a aceptarla.

6. El contratista tiene derecho a sacar copias a su costa de los planos, pliego de condiciones y presupuestos. El Ingeniero autor del proyecto autorizará con su firma las copias solicitadas por el contratista después de confrontarlas.

7. Se abonará al contratista la obra que realmente ejecute con sujeción al proyecto que sirvió de base para la contratación, a las modificaciones autorizadas por la superioridad o a las órdenes que con arreglo a sus facultades le hayan comunicado por escrito al Ingeniero Director de obras siempre que dicha obra se haya ajustado a los preceptos de los pliegos de condiciones, con arreglo a los cuales, se harán las modificaciones y la valoración de las diversas unidades sin que el importe total pueda exceder de los presupuestos aprobados. Por consiguiente, el número de unidades que se consignan en el proyecto o en el presupuesto, no podrá servirle de fundamento para entablar reclamaciones de ninguna clase, salvo en los casos de rescisión.

8. Tanto en las certificaciones de obras como en la liquidación final, se abonarán los trabajos realizados por el contratista a los precios de ejecución material que figuran en el presupuesto para cada unidad de la obra.

9. Si excepcionalmente se hubiera ejecutado algún trabajo que no se ajustase a las condiciones de la contrata pero que sin embargo es admisible a juicio del Ingeniero Director de obras, se dará conocimiento a la Dirección, proponiendo a la vez la rebaja de precios que el Ingeniero estime justa y si la Dirección resolviera aceptar la obra, quedará el contratista obligado a conformarse con la rebaja acordada.

10. Cuando se juzgue necesario emplear materiales o ejecutar obras que no figuren en el presupuesto de la contrata, se evaluará su importe a los precios asignados a otras obras o materiales análogos si los hubiere y cuando no, se discutirán entre el Ingeniero Director y el contratista, sometiéndolos a la aprobación de la Dirección. Los nuevos precios convenidos por uno u otro procedimiento, se sujetarán siempre al establecido en el punto anterior.

11. Cuando el contratista, con autorización del Ingeniero Director de obras, emplee materiales de calidad más elevada o de mayores dimensiones de lo estipulado en el proyecto, o sustituya una clase de fabricación por otra que tenga asignado mayor precio o ejecute con mayores dimensiones cualquier otra parte de las obras, o en general, introduzca en ellas cualquier modificación que sea beneficiosa a juicio del Ingeniero Director de obras, no tendrá derecho sin embargo, sino a lo que le correspondería si hubiera realizado la obra con estricta sujeción a lo proyectado y contratado.

12. Las cantidades calculadas para obras accesorias, aunque figuren por partida alzada en el presupuesto final (general), no serán abonadas sino a los precios de la contrata, según las condiciones de la misma y los proyectos particulares que para ellas se formen, o en su defecto, por lo que resulte de su medición final.

13. El contratista queda obligado a abonar al Ingeniero autor del proyecto y director de obras así como a los Ingenieros Técnicos, el importe de sus respectivos honorarios facultativos por formación del proyecto, dirección técnica y administración en su caso, con arreglo a las tarifas y honorarios vigentes.

14. Concluida la ejecución de la obra, será reconocida por el Ingeniero Director que a tal efecto designe la empresa.

15. La garantía definitiva será del 4% del presupuesto y la provisional del 2%.

16. La forma de pago será por certificaciones mensuales de la obra ejecutada, de acuerdo con los precios del presupuesto, deducida la baja si la hubiera.

17. La fecha de comienzo de las obras será a partir de los 15 días naturales del replanteo oficial de las mismas y la definitiva, al año de haber ejecutado la provisional, procediéndose si no existe reclamación alguna, a la reclamación de la fianza.

18. Si el contratista al efectuar el replanteo, observase algún error en el proyecto, deberá comunicarlo en el plazo de quince días al Ingeniero Director de obras, pues transcurrido ese plazo será responsable de la exactitud del proyecto.

19. El contratista está obligado a designar una persona responsable que se entenderá con el Ingeniero Director de obras, o con el delegado que éste designe, para todo relacionado con ella. Al ser el Ingeniero Director de obras el que interpreta el proyecto, el contratista deberá consultarle cualquier duda que surja en su realización.

20. Durante la realización de la obra, se girarán visitas de inspección por personal facultativo de la empresa cliente, para hacer las comprobaciones que se crean oportunas. Es obligación del contratista, la conservación de la obra ya ejecutada hasta la recepción de la misma, por lo que el deterioro parcial o total de ella, aunque sea por agentes atmosféricos u otras causas, deberá ser reparado o reconstruido por su cuenta.

21. El contratista, deberá realizar la obra en el plazo mencionado a partir de la fecha del contrato, incurriendo en multa, por retraso de la ejecución siempre que éste no sea debido a causas de fuerza mayor. A la terminación de la obra, se hará una recepción provisional previo reconocimiento y examen por la dirección técnica, el depositario de efectos, el interventor y el jefe de servicio o un representante, estampando su conformidad el contratista.

22. Hecha la recepción provisional, se certificará al contratista el resto de la obra, reservándose la administración el importe de los gastos de conservación de la misma hasta su recepción definitiva y la fianza durante el tiempo señalado como plazo de garantía. La recepción definitiva se hará en las mismas condiciones que la provisional, extendiéndose el acta correspondiente. El Director Técnico propondrá a la Junta Económica la devolución de la fianza al contratista de acuerdo con las condiciones económicas legales establecidas.

23. Las tarifas para la determinación de honorarios, reguladas por orden de la Presidencia del Gobierno el 19 de Octubre de 1961, se aplicarán sobre el denominado en la actualidad “Presupuesto de Ejecución de Contrata” y anteriormente llamado ”Presupuesto de Ejecución Material” que hoy designa otro concepto.

Condiciones particulares

La empresa consultora, que ha desarrollado el presente proyecto, lo entregará a la empresa cliente bajo las condiciones generales ya formuladas, debiendo añadirse las siguientes condiciones particulares:

1. La propiedad intelectual de los procesos descritos y analizados en el presente trabajo, pertenece por entero a la empresa consultora representada por el Ingeniero Director del Proyecto.

2. La empresa consultora se reserva el derecho a la utilización total o parcial de los resultados de la investigación realizada para desarrollar el siguiente proyecto, bien para su publicación o bien para su uso en trabajos o proyectos posteriores, para la misma empresa cliente o para otra.

3. Cualquier tipo de reproducción aparte de las reseñadas en las condiciones generales, bien sea para uso particular de la empresa cliente, o para cualquier otra aplicación, contará con autorización expresa y por escrito del Ingeniero Director del Proyecto, que actuará en representación de la empresa consultora.

4. En la autorización se ha de hacer constar la aplicación a que se destinan sus reproducciones así como su cantidad.

5. En todas las reproducciones se indicará su procedencia, explicitando el nombre del proyecto, nombre del Ingeniero Director y de la empresa consultora.

6. Si el proyecto pasa la etapa de desarrollo, cualquier modificación que se realice sobre él, deberá ser notificada al Ingeniero Director del Proyecto y a criterio de éste, la empresa consultora decidirá aceptar o no la modificación propuesta.

7. Si la modificación se acepta, la empresa consultora se hará responsable al mismo nivel que el proyecto inicial del que resulta el añadirla.

8. Si la modificación no es aceptada, por el contrario, la empresa consultora declinará toda responsabilidad que se derive de la aplicación o influencia de la misma.

9. Si la empresa cliente decide desarrollar industrialmente uno o varios productos en los que resulte parcial o totalmente aplicable el estudio de este proyecto, deberá comunicarlo a la empresa consultora.

10. La empresa consultora no se responsabiliza de los efectos laterales que se puedan producir en el momento en que se utilice la herramienta objeto del presente proyecto para la realización de otras aplicaciones.

11. La empresa consultora tendrá prioridad respecto a otras en la elaboración de los proyectos auxiliares que fuese necesario desarrollar para dicha aplicación industrial, siempre que no haga explícita renuncia a este hecho. En este caso, deberá autorizar expresamente los proyectos presentados por otros.

12. El Ingeniero Director del presente proyecto, será el responsable de la dirección de la aplicación industrial siempre que la empresa consultora lo estime oportuno. En caso contrario, la persona designada deberá contar con la autorización del mismo, quien delegará en él las responsabilidades que ostente.