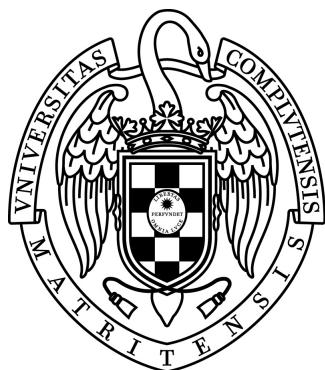

Monitorización de señales biomédicas en sistemas Android



TRABAJO DE FIN DE GRADO

Mario Michiels Toquero
Cristian Pinto Lozano

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

Junio 2017

Monitorización de señales biomédicas en sistemas Android

*Memoria que presentan para optar al título de Graduados en
Ingeniería del Software*

**Mario Michiels Toquero
Cristian Pinto Lozano**

*Dirigida por el Doctor
Joaquín Recas Piorno*

**Departamento de Arquitectura de Computadores y
Automática
Facultad de Informática
Universidad Complutense de Madrid**

Junio 2017

A nuestros padres

*Any fool can criticize, condemn and complain
- and most fools do. But it takes character
and self-control to be understanding and forgiving.
Dale Carnegie (1888 - 1955)*

Agradecimientos

A todos los que la presente vieran y entendieren.

Inicio de las Leyes Orgánicas. Juan Carlos I

Groucho Marx decía que encontraba a la televisión muy educativa porque cada vez que alguien la encendía, él se iba a otra habitación a leer un libro. Utilizando un esquema similar, nosotros queremos agradecer al Word de Microsoft el habernos forzado a utilizar L^AT_EX. Cualquiera que haya intentado escribir un documento de más de 150 páginas con esta aplicación entenderá a qué nos referimos. Y lo decimos porque nuestra andadura con L^AT_EX comenzó, precisamente, después de escribir un documento de algo más de 200 páginas. Una vez terminado decidimos que nunca más pasariamos por ahí. Y entonces caímos en L^AT_EX.

Es muy posible que hubiéramos llegado al mismo sitio de todas formas, ya que en el mundo académico a la hora de escribir artículos y contribuciones a congresos lo más extendido es L^AT_EX. Sin embargo, también es cierto que cuando intentas escribir un documento grande en L^AT_EX por tu cuenta y riesgo sin un enlace del tipo “*Author instructions*”, se hace cuesta arriba, pues uno no sabe por donde empezar.

Y ahí es donde debemos agradecer tanto a Pablo Gervás como a Miguel Palomino su ayuda. El primero nos ofreció el código fuente de una programación docente que había hecho unos años atrás y que nos sirvió de inspiración (por ejemplo, el fichero *guionado.tex* de TEXIS tiene una estructura casi exacta a la suya e incluso puede que el nombre sea el mismo). El segundo nos dejó husmear en el código fuente de su propia tesis donde, además de otras cosas más interesantes pero menos curiosas, descubrimos que aún hay gente que escribe los acentos españoles con el \'{\i}.

No podemos tampoco olvidar a los numerosos autores de los libros y tutoriales de L^AT_EX que no sólo permiten descargar esos manuales sin coste adicional, sino que también dejan disponible el código fuente. Estamos pensando en Tobias Oetiker, Hubert Partl, Irene Hyna y Elisabeth Schlegl, autores del famoso “The Not So Short Introduction to L^AT_EX 2_{\varepsilon}” y en Tomás

Bautista, autor de la traducción al español. De ellos es, entre otras muchas cosas, el entorno `example` utilizado en algunos momentos en este manual.

También estamos en deuda con Joaquín Ataz López, autor del libro “Creación de ficheros L^AT_EX con GNU Emacs”. Gracias a él dejamos de lado a WinEdt y a Kile, los editores que por entonces utilizábamos en entornos Windows y Linux respectivamente, y nos pasamos a emacs. El tiempo de escritura que nos ahorramos por no mover las manos del teclado para desplazar el cursor o por no tener que escribir `\emph` una y otra vez se lo debemos a él; nuestro ocio y vida social se lo agradecen.

Por último, gracias a toda esa gente creadora de manuales, tutoriales, documentación de paquetes o respuestas en foros que hemos utilizado y seguiremos utilizando en nuestro quehacer como usuarios de L^AT_EX. Sabéis un montón.

Y para terminar, a Donald Knuth, Leslie Lamport y todos los que hacen y han hecho posible que hoy puedas estar leyendo estas líneas.

Resumen

...

...

Abstract

...
...
...

Índice

Agradecimientos	IX
Resumen	XI
Abstract	XIII
1. Introducción	1
1.1. Motivación	1
1.2. Estructura de capítulos	2
2. Antecedentes	3
2.1. Electrocardiograma	3
2.1.1. Estructura de un ECG	4
2.1.2. Funcionamiento interno del corazón	5
2.1.3. Obtención del ECG	6
2.1.4. Utilidades del ECG	6
2.2. Tecnologías	6
2.2.1. BeagleBone Black	7
2.2.2. Bus SPI	9
2.2.3. Chip ADS1198	11
2.2.4. Analizador lógico Saleae	13
3. Desarrollo	15
3.1. Proyecto inicial	15
3.1.1. Objetivos	16
3.1.2. Alternativas	17
3.2. Programación en tiempo real	19
3.2.1. Decisiones de diseño	19
3.3. Configuración hardware	20
3.3.1. Device Tree	21
3.3.2. Device Tree Overlay	22
3.3.3. Conexiones físicas	24

3.4. Ecosistema software	26
3.4.1. Inicialización ADS	28
3.4.2. Comunicación SPI	32
3.4.3. Guardado de muestras en memoria	35
3.4.4. Lectura y transmisión de muestras	37
4. Aplicación Android	39
4.1. Envío de datos	39
4.2. Recepción de datos	41
4.3. Mejoras desarrolladas	42
4.4. Renovación visual	43
4.5. Ecosistema de streaming	46
4.5.1. Características del servidor	46
4.5.2. Características de la base de datos	46
4.5.3. Interpretación de datos	47
Bibliografía	49
Lista de acrónimos	50

Índice de figuras

2.1. Estructura habitual de la señal ECG durante un ciclo cardiaco.	4
2.2. Estructura anatómica del corazón humano	5
2.3. Vista general de una BeagleBone Black	7
2.4. Especificación hardware BeagleBone Black	9
2.5. Estándar de comunicación SPI	10
2.6. Configuraciones posibles de fase y polaridad en el estándar SPI	11
2.7. Diagrama de funcionamiento interno ADS1198	12
2.8. Software multiplataforma proporcionado por Saleae	14
3.1. Ejemplo de salida al ejecutar comando <code>top</code> en una terminal linux	17
3.2. Diagrama de conexiones físicas disponibles en una BBB	20
3.3. Ejemplo simple de un fichero <code>.dts</code>	22
3.4. Device Tree Overlay del proyecto para la BBB	23
3.5. Restricciones en la asignación de modos a pines relevantes para el proyecto	24
3.6. Conexiones físicas necesarias para el proyecto	26
3.7. Bancos de memoria del PRU	27
3.8. Inicialización ADS en el código original en C	29
3.9. Función SDATAC en código ensamblador	29
3.10. Función SET痈ADS痈SAMPLE痈RATE en código ensamblador	30
3.11. Función SET痈INTERNAL痈REFERENCE en código ensamblador	30
3.12. Función SEND痈RDATAAC en código ensamblador	31
3.13. Comunicación SPI en código ensamblador	32
3.14. Función SPICLK痈LOOP en código ensamblador (primera parte)	34
3.15. Función SPICLK痈LOOP en código ensamblador (segunda parte)	34
3.16. Función STORE痈RAM en código ensamblador (primera parte)	36
3.17. Función STORE痈RAM en código ensamblador (segunda parte)	36

3.18. Función principal del programa <code>mem2file.c</code> desarrollado en C	38
4.1. Dongle Bluetooth 4.0 CSR utilizado en la BBB	40
4.2. Programa desarrollado en Python <code>BTConnection.py</code>	41
4.3. Pantalla principal de la aplicación Android original	42
4.4. Fragmento del código que ejecuta el hilo encargado de la recepción de datos	43
4.5. Pantallas principales de la aplicación Android	45
4.6. Fragmento del código que ejecuta el servidor desarrollado en <code>Node.js</code>	47
4.7. Simulador de ECG utilizado durante el desarrollo del proyecto	48
4.8. Interpretación de la señal por parte de la aplicación Android	48

Índice de Tablas

3.1. Conexiones físicas necesarias para la alimentación del chip ADS1198	25
3.2. Conexiones físicas de carácter general GPIO	25
3.3. Conexiones físicas para posibilitar la conexión SPI	25
3.4. Conexiones físicas del analizador lógico Saleae	25

Capítulo 1

Introducción

*Un comienzo no desaparece nunca,
ni siquiera con un final.*

Harry Mulisch

Motivación

La sociedad actual en la que nos encontramos está completamente informatizada podríamos decir. Es posible encontrar software en todo tipo de lugares en los que jamás hubiesemos pensado hace décadas que hubiese sido posible, como por ejemplo, neveras en las que una vez acabado un cierto tipo de refrigerio, es el propio electrodoméstico el encargado de comprarlo por nosotros, relojes con los que podemos conectarnos a internet y comunicarnos, eliminando la necesidad de llevar un teléfono encima, e incluso zapatillas que se encargan de pedir nuestra comida favorita a domicilio con tan solo pulsar un pequeño botón.

Es imposible enumerar la cantidad de aplicaciones que el software podría tener, al menos en la presente memoria, y todo esto sin olvidar los futuros usos que adquirirá. Podríamos decir que la industria del Software está en pleno auge, es más, lleva en pleno auge desde hace décadas, y no parece que vaya a decaer. Con tantos posibles sectores en los que especializarse, parece complicado elegir uno en el que sumergirse.

Sin embargo, para los autores de la presente memoria siempre tuvo cierto atractivo el desarrollo de software para dispositivos empotrados que utilizan software libre. Fue entonces cuando se nos dio la posibilidad de trabajar en el presente proyecto, utilizando este tipo de dispositivos y además íntimamente enfocado y relacionado con el cuidado de la salud. La mezcla de ambos componentes nos fascinó a primera vista.

El desarrollo de software para dispositivos empotrados se encuentra en pleno crecimiento desde hace ya varios años, y cada vez son más populares las

comunidades que dan soporte a los desarrolladores de los mismos, facilitando así el proceso de creación del software. Asimismo también se encuentra en uno de sus mejores momentos todo tipo de *gadget* capaz de medir o monitorizar ciertas señales, como pueden ser los actuales relojes inteligentes, o las pulseras de actividad tan frecuentemente vistas. Estos dispositivos son capaces de monitorizar multitud de señales procedentes del cuerpo humano, y todo esto en un reducido espacio físico fácilmente portable.

La monitorización de ciertas señales biomédicas puede ser un factor fundamental a la hora de detectar problemas de salud de forma precoz. La presente memoria trata de ilustrar el proceso gracias al cual es posible monitorizar este tipo de señales utilizando un hardware de bajo coste y portable, y un software libre, adecuado y preciso, cuya unión pueda facilitar la práctica de este tipo de procesos en todos los contextos en los que pudiera ser necesario.

Estructura de capítulos

Capítulo 2

Antecedentes

*Cada día sabemos más
y entendemos menos.*

Albert Einstein

RESUMEN: Para la realización del presente proyecto ha sido necesario realizar una investigación que cubre desde aspectos médicos hasta aspectos técnicos propios de una ingeniería compleja. En este capítulo se expone un breve resumen de cada antecedente investigado.

Electrocardiograma

Un electrocardiograma (más popularmente conocido como ECG) es un proceso por el cual se registran las actividades eléctricas que emite el corazón durante un tiempo determinado.

El registro de esta actividad cardiaca es posible gracias a las diferencias de potencial existentes producidas por la contractilidad del corazón. El estudio de la información ilustrada por un ECG puede ser de gran utilidad a la hora de detectar multitud de enfermedades cardiovasculares, así como prevenir problemas cardíacos de forma precoz.

La ventaja de un ECG frente a otros métodos de medición para comprobar el estado del corazón, es que el ECG aporta mucha más información que los métodos más habituales como pueden ser simplemente medir el pulso o utilizar un estetoscopio.

Entre la información extra que puede obtenerse recurriendo al ECG, cabe destacar desde la medición continua durante un tiempo prolongado (días incluso si se utiliza un ECG portátil) hasta la obtención del movimiento de

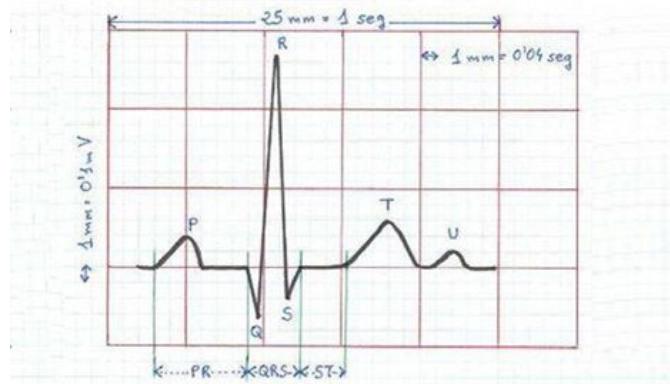


Figura 2.1: Estructura habitual de la señal ECG durante un ciclo cardiaco.

los músculos del corazón (los electrodos conectados al paciente registran esos movimientos en mV), lo cual permite saber cuando entra la sangre, cuando sale, cuanto dura cada movimiento, etc. Sin esa información sería imposible detectar ciertos tipos de arritmias y/o otras alteraciones del corazón.

Estructura de un ECG

La estructura habitual de un ECG [Fig 2.1] habitualmente está formada por un conjunto de ondas y complejos determinados:

- Onda P
- Onda Q
- Complejo QRS
- Onda T
- Onda U

Habitualmente este tipo de señales se encuentran dentro de un rango determinado de amplitudes, que generalmente abarca desde los $0.5mV$ hasta los $5mV$, existiendo además una componente continua causada por el contacto existente entre los electrodos y la piel.

Mayormente este tipo de señales suele ilustrarse en los libros de forma muy clara y reconocible, aunque no siempre es posible disponer de un entorno con las características propicias para eliminar toda existencia de ruido en la señal. Generalmente el ruido que se recoge al analizar este tipo de señales es despreciado, aunque en ciertas ocasiones puede llegar a ser tan difuso, que nos impida reconocer hasta las pautas más características de una señal ECG.

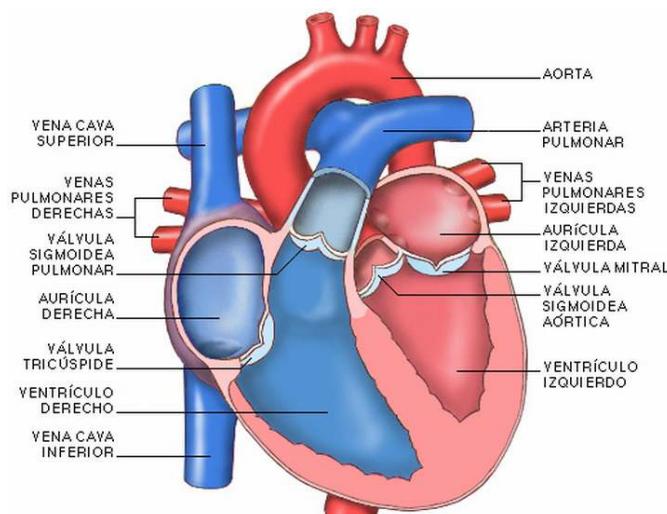


Figura 2.2: Estructura anatómica del corazón humano

Puede presentarse ruido en la señal simplemente debido a la luz tanto natural como artificial que incida indirectamente en los electrodos, así como debido a la corriente que reciben los dispositivos que nos permiten llevar a cabo la medición de la señal.

Funcionamiento interno del corazón

El corazón está constituido por cuatro cámaras diferenciadas: dos aurículas (izquierda y derecha) y dos ventrículos (izquierdo y derecho) como puede apreciarse en la [Fig 2.2]

Su funcionamiento sigue siempre un mismo ciclo que se repite una y otra vez: la aurícula derecha recibe la sangre venosa del cuerpo a través de la vena cava superior y la envía al ventrículo derecho. Para que dicha sangre pueda oxigenarse, el ventrículo derecho la envía a través de la arteria pulmonar a los pulmones, retornando al corazón, a través de la vena pulmonar, a la aurícula izquierda. Para finalizar con el ciclo, la sangre pasa de dicha aurícula al ventrículo izquierdo, el cual la distribuye por todo el cuerpo gracias a la arteria aorta, para volver posteriormente a la aurícula derecha y así cerrar el ciclo.

Para que este ciclo periódico funcione de manera correcta, síncrona y con total ausencia de errores, el corazón dispone de un sistema de conducción eléctrica constituido por fibras de músculo cardíaco cuya finalidad es la transmisión de impulsos eléctricos. El motivo por el cual no tenemos ningún tipo de control sobre los latidos del corazón es porque este sistema es autoexcitable.

Obtención del ECG

El proceso físico a través del cual se realiza el procedimiento de registro y se obtiene el ECG consiste en la medición de la actividad eléctrica del corazón entre distintos puntos corporales, ya que, mientras que el corazón pasa por los estados de polarización y despolarización, el cuerpo en su conjunto se comporta como un volumen conductor, propagando la corriente eléctrica.

El equipo de registro consta de una serie de electrodos, que se conectan a la piel del paciente y a un equipo de registro. Estos electrodos se colocan en unas posiciones predeterminadas y universales, lo que permite obtener el registro del sistema de la forma más precisa y exacta posible.

Utilidades del ECG

Tras la obtención de un ECG completo, se tendrá un registro completo acerca del tamaño, la cadencia, y la naturaleza de los impulsos eléctricos emitidos por el corazón. Gracias a esta información, es posible definir tanto el ritmo como la frecuencia cardiaca. Este tipo de información puede ser crucial a la hora de detectar ciertos problemas cardíacos. A continuación se detallan posibles utilidades que puede tener la obtención de un ECG:

- Obtener información sobre el estado físico del corazón.
- Detectar problemas cardíacos de forma precoz, o alteraciones electrolíticas de potasio, sodio, calcio, magnesio, u otros elementos químicos.
- Determinar si el corazón funciona correctamente, en caso de no encontrar ninguna anomalía.
- Adquirir información acerca de la repercusión miocárdica de diversas enfermedades cardíacas.
- Detección de anormalidades conductivas.
- Indicar bloqueos coronarios arteriales.

Tecnologías

Para llevar a cabo el correcto desarrollo del trabajo que se expone en la presente memoria, ha sido necesaria la investigación y el desarrollo de diversas tecnologías que se expondrán a continuación, así como el estudio y la asimilación de tecnologías ya existentes con el fin de adaptarlas y modificarlas para el proyecto desarrollado.

Como módulo de procesamiento se ha utilizado una placa BeagleBone Black (puede apreciarse una visión general de esta placa en la [Fig 2.3]), aprovechando la gran comunidad de desarrolladores existentes gracias a la

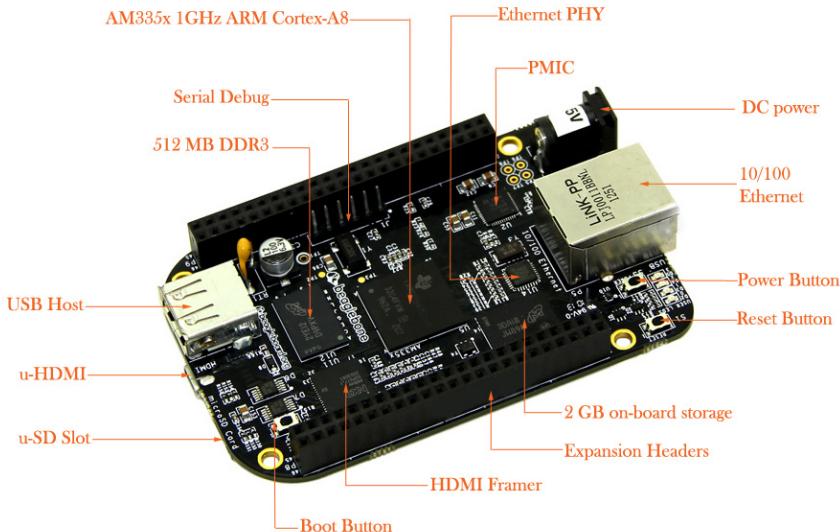


Figura 2.3: Vista general de una BeagleBone Black

cual fue posible resolver los problemas que fueron surgiendo durante el desarrollo, y aprovechando también el bajo costo de la placa así como la multitud de posibilidades que ofrece. En cuanto al tratamiento y captura de señales se ha utilizado el chip ADS1198 de Texas Instruments, que permite unas velocidades de captura suficientemente elevadas para este tipo de desarrollos además de tener un coste asequible. Con motivos de depuración durante el tratamiento de las distintas señales que se han obtenido, se ha utilizado Saleae Logic, un analizador lógico USB que cuenta con 8 canales para cada posible señal. Como método de sincronización entre los dispositivos que lo necesitan se ha recurrido a una interfaz SPI.

BeagleBone Black

La BeagleBone es una plataforma compacta, de bajo coste y operada por software libre Linux que puede ser usada para complejas aplicaciones en las que intervenga software de alto nivel y circuitos electrónicos de bajo nivel (Molloy, Enero, 2014). Nostros concretamente nos hemos decantado por trabajar con la última versión de la plataforma BeagleBone, es decir, BeagleBone Black (BBB), ya que era la placa que más flexibilidad aportaba al presente proyecto. Las características de la BBB son las siguientes:

- Es muy potente, ya que contiene un procesador que puede realizar hasta 2 billones de instrucciones por segundo (2000 MIPS).
- Tiene un coste bastante asequible, ya que su precio oscila entre los 45\$

y los 55\$.

- Es compatible con muchos estandares de interfaces para dispositivos electronicos (SPI entre otros).
- Es muy eficiente, ya que requiere únicamente entre 1 y 2.3 W, según este inactivo, o funcionando a máxima potencia.
- Es fácilmente ampliable a través de otro tipo de placas de expansión opcionales y de dispositivos USB.
- Tiene una gran comunidad de desarrolladores innovadores y entusiastas que respaldan y dan soporte a los usuarios de la plataforma.
- Es hardware libre, y es compatible con multitud de herramientas y de aplicaciones de software libre.

La plataforma BeagleBone lleva el sistema operativo Linux, lo que significa que es posible usar todo tipo de librerías de software libre, y aplicaciones. La posibilidad de usar software libre, brinda también la oportunidad de conectar esta plataforma con todo tipo de perifericos como pueden ser cámaras USB, teclados, adaptadores Wi-Fi ...

Versiones BeagleBoneBlack

La plataforma BeagleBone y sus placas computadoras han evolucionado progresivamente con el paso de los años, tanto en prestaciones, como en reducción del coste final. A continuación se muestra el progreso de la plataforma en orden histórico.

- **(2008) BeagleBoard (125\$)** La original placa de desarrollo de hardware libre basada en ARM que tiene soporte para video HD. Tiene un Procesador ARM A8 a 720MHz pero no tiene conexión Ethernet.
- **(2010) BeagleBoard xM (149\$)** Similar a la BeagleBoard, solo que tiene un procesador ARM AM37x a 1Ghz, 512MB de memoria, cuatro puertos USB, y soporte para Ethernet. Es una placa muy popular por su núcleo C64+TMDSP para procesamiento de señales digitales.
- **(2011) BeagleBone (89\$)** Más compacta que su antecesora la BeagleBoard. Tiene un procesador a 720Mhz y 256MB de memoria, soporta Ethernet y conexiones de salida de bajo nivel, pero no tiene soporte nativo de video.
- **(2013) BeagleBone Black (45\$-55\$)** Esta placa mejora la BeagleBone con un procesador a 1GHz, 512MB de memoria DDR3, conexión Ethernet, almacenamiento eMMC y soporte para conexión de video HDMI.

	Feature
Processor	Sitara AM3358BZCZ100
Graphics Engine	1GHz, 2000 MIPS
SDRAM Memory	SGX530 3D, 20M Polygons/S
Onboard Flash	512MB DDR3L 800MHZ
PMIC	4GB, 8bit Embedded MMC
Debug Support	TPS65217C PMIC regulator and one additional LDO.
Power Source	Optional Onboard 20-pin CTI JTAG, Serial Header
PCB	miniUSB USB or DC Jack
Indicators	5VDC External Via Expansion Header
HS USB 2.0 Client Port	3.4" x 2.1"
HS USB 2.0 Host Port	6 layers
Serial Port	1-Power, 2-Ethernet, 4-User Controllable LEDs
Ethernet	Access to USB0, Client mode via miniUSB
SD/MMC Connector	Access to USB1, Type A Socket, 500mA LS/FS/HS
User Input	UART0 access via 6 pin 3.3V TTL Header. Header is populated
Video Out	10/100, RJ45
Audio	microSD , 3.3V
Expansion Connectors	Reset Button
	Boot Button
	Power Button
Weight	16b HDMI, 1280x1024 (MAX)
Power	1024x768,1280x720,1440x900 ,1920x1080@24Hz w/EDID Support
	Via HDMI Interface, Stereo
	Power 5V, 3.3V , VDD_ADC(1.8V)
	3.3V I/O on all signals
	McASP0, SPI1, I2C, GPIO(69 max), LCD, GPMC, MMC1, MMC2, 7 AIN(1.8V MAX), 4 Timers, 4 Serial Ports, CAN0, EHRPWM(0,2), XDMA Interrupt, Power button, Expansion Board ID (Up to 4 can be stacked)
	1.4 oz (39.68 grams)
	Refer to Section 6.1.7

Figura 2.4: Especificación hardware BeagleBone Black

La especificación completa de las características de la BBB se muestran con más detalle en la [Fig. 2.4] (elinux.org, 2017)

Bus SPI

El bus SPI (del inglés Serial Peripheral Interface) es un estándar de comunicación síncrono, rápido y bidireccional que permite comunicar dispositivos como la BBB con otros dispositivos en una distancia corta (Molloy, Enero, 2014). Es posible que la comunicación sea bidireccional, ya que tanto la transmisión como la recepción de datos se realizan en buses separados.

En la transferencia de datos se implican distintos dispositivos que desempeñan diferentes funciones. Existe un dispositivo principal, más conocido como maestro, y otro dispositivo denominado esclavo (como mínimo debe

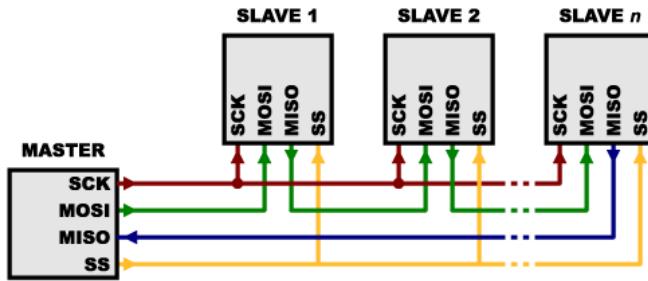


Figura 2.5: Estándar de comunicación SPI

existir uno). El maestro es responsable de enviar las señales de reloj y de seleccionar al esclavo activo en cada instante de la comunicación.

Las diferentes líneas existentes en este estándar de comunicación son las siguientes [Fig 2.5]:

- **MISO** (Master Input Slave Output): Bus de salida de datos de los dispositivos esclavos y de entrada al maestro.
- **MOSI** (Master Output Slave Input): Bus de salida de datos del maestro y entrada a los esclavos.
- **SCK** (Clock o señal de reloj): Pulso de reloj generado por el maestro.
- **SS/Select** (Chip Select): Bus de salida del maestro y entrada a los esclavos, encargado de seleccionar el dispositivo esclavo activo en la comunicación.

Con el fin de procesar una lectura desde un dispositivo esclavo, el maestro debe realizar una escritura en el bus, forzando de esta forma la generación de una señal del reloj que tendrá como consecuencia la escritura de datos por parte del dispositivo deseado.

El dispositivo maestro debe conocer las características de cada esclavo implicado en la comunicación, entre las cuales, es posible destacar las siguientes:

- **Frecuencia máxima de transferencia:** La velocidad de comunicación con cada dispositivo estará limitada por este valor, haciendo imposible enviar o recibir datos a mayor velocidad.
- **Polaridad (CPOL):** Determina la polaridad del reloj [Fig 2.6].
- **Fase (CPHA):** Determina el flanco de reloj en el que se desencadenará la escritura [Fig 2.6].

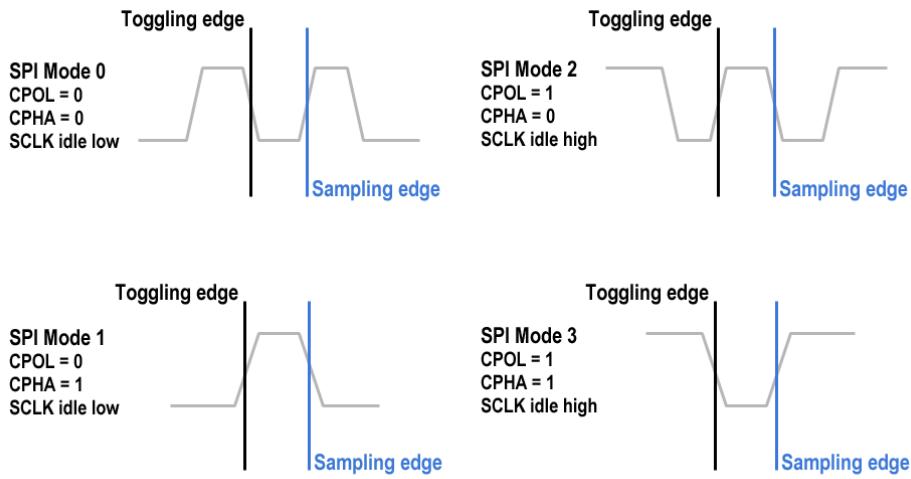


Figura 2.6: Configuraciones posibles de fase y polaridad en el estándar SPI

Chip ADS1198

El ADS1198 es un chip de Texas Instruments de bajo consumo para mediciones de señales ECG, que se caracteriza por tener ocho canales de 16 bits cada uno, una frecuencia de muestreo de hasta 8KHz, un amplificado de ganancia programable, referencia interna, y un oscilador integrado. (Instruments, 2017) Cada canal consta de un multiplexor que permite la lectura desde ocho entradas diferentes, siendo las más relevantes las entradas de temperatura, electrodos, y señal de test generada internamente.

El chip consta de una interfaz SPI para permitir la comunicación con otros dispositivos. Además de las líneas típicas de SPI se proporciona una línea adicional, **DRDY** (Data ready), que indica cuando se tienen nuevos datos válidos preparados para enviar. La lectura de datos se realiza siempre para los 8 canales, devolviendo adicionalmente una cabecera con información de la configuración del chip.

El chip permite obtener la alimentación de manera unipolar o bipolar:

- **Unipolar:** La alimentación unipolar se realiza mediante una entrada de 5V y otra de 3V.
- **Bipolar:** El modo bipolar requiere entradas desde -2.5V hasta 2.5V.

La alimentación que se ha utilizado en este proyecto ha sido unipolar, por ser más fácilmente adaptable a los pines que proporciona la BBB.

Internamente el ADS1198 cuenta con 25 registros que permiten al usuario configurar todas las características programables del chip [Fig 2.7]. Gran

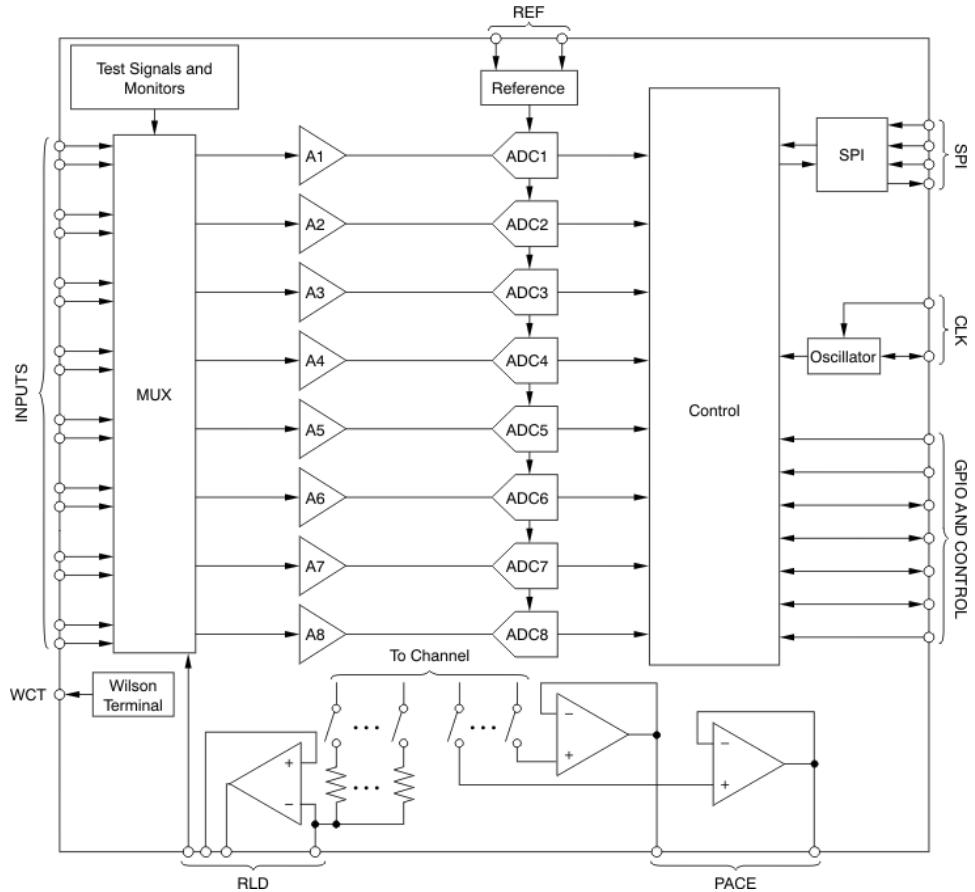


Figura 2.7: Diagrama de funcionamiento interno ADS1198

parte de la configuración permitida está relacionada con aspectos propios de la captura de señales, como las ganancias, el uso de un oscilador interno, o el voltaje de referencia utilizado para la captura. El resto de configuraciones posibles pueden variar la frecuencia de captura, las entradas de los canales o modificar las señales de test internas en amplitud y frecuencia.

Para el propósito del presente proyecto, se ha utilizado un kit de desarrollo, en el que se incluye el chip integrado en una placa que permite configurar la forma de alimentación o tener acceso a las entradas de datos de forma más cómoda y sencilla.

Características técnicas

Dado su elevado rendimiento, alto nivel de integración y bajo consumo, el ADS1198 permite el desarrollo de instrumentación médica de prestaciones elevadas, tamaño reducido y bajo coste. Entre las características técnicas

más destacables, podemos encontrar las siguientes:

- 8 canales de alta resolución.
- Bajo consumo: 0.55mW/canal.
- Frecuencia de muestreo: desde 125Hz hasta 8KHz.
- Ganancia programable.
- Alimentación Unipolar o Bipolar.
- Oscilador y referencia internos.
- Señales de test integradas.
- Comunicación mediante interfaz SPI.
- Rango de temperatura operativo: 0 °C a 70 °C.

Analizador lógico Saleae

Se ha utilizado un analizador lógico, en este caso Saleae, con motivos de depuración durante el desarrollo del proyecto. Gracias al uso de este dispositivo ha sido posible detectar fallos muy concretos en períodos de tiempo relativamente pequeños, ya que de otra forma, en caso de no haberlo utilizado hubiese sido muy complicado el desarrollo, o al menos, hubiese llevado muchísimo más tiempo del esperado.

La idea de funcionamiento de este dispositivo es muy sencilla. Cuenta con 8 canales que actúan como analizadores lógicos, por lo que basta conectar cada analizador al canal que se desee analizar.

Una vez conectado cada canal, entra en juego el software que proporciona el propio Saleae, que en este caso es multiplataforma¹. Tras descargarlo e instalarlo bastará con conectar el analizador lógico por usb y configurarlo para comenzar a analizar cada una de las señales.

Entre sus principales características, cabe destacar las siguientes:

- Flexible, software fácil de usar.
- Grandes buffers de muestras.
- Muy portable, conexión USB disponible.
- Analiza 24 protocolos distintos de comunicación.
- Decodificación de protocolos personalizada.

¹Software disponible en <https://www.saleae.com/downloads>

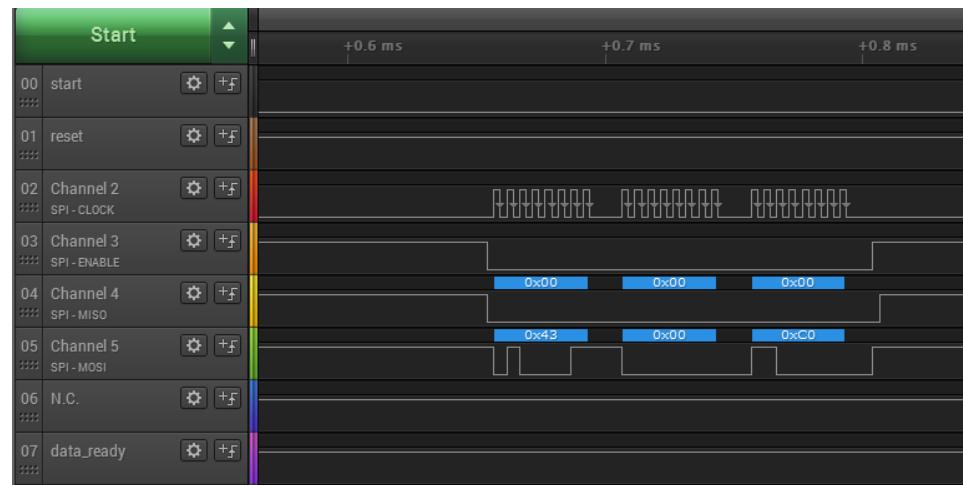


Figura 2.8: Software multiplataforma proporcionado por Saleae

- Formatos de exportación de datos: CSV, Binario, VCD y Matlab.
- Marcadores de medidas, señaladores y temporizadores.

Capítulo 3

Desarrollo

*Quisiera vivir para estudiar,
no estudiar para vivir.*

Sir Francis Bacon

RESUMEN: Para la correcta funcionalidad del proyecto, ha sido necesario configurar el hardware de forma precisa, además de desarrollar un software muy concreto y diseñado a medida para el contexto en el que trabajamos. En este capítulo se exponen los factores fundamentales de ambos ámbitos con cierto nivel de profundidad.

Proyecto inicial

Tras haber realizado la elección del tema en el que se basaría nuestro trabajo de fin de grado, así como tras la elección del director que dirigiría dicho trabajo, estabamos ansiosos de empezar lo antes posible e introducirnos en la materia. Nos pareció apasionante la idea de poder desarrollar software para dispositivos empotrados, y aun más, sabiendo que el campo con el que estaría íntimamente relacionado sería la medicina. Tras los sabios consejos que nos transmitió Joaquín sobre que placa concreta usar, de que hardware podríamos disponer para la realización del proyecto, así como el software que se nos podría facilitar, estabamos decididos a comenzar. La placa hardware sobre la que desarrollaríamos finalmente se trataba de una BeagleBone Black.

Inicialmente, Joaquín nos proporcionó un pequeño proyecto sobre el que poder comenzar a trabajar y sentar las bases de un proyecto mucho mayor. Este proyecto además podía funcionar correctamente con diversos chips de Texas Instruments, concretamente nosotros escogimos el chip ADS1198. Este

chip generaba simulaciones de muestras de forma continua. El código proporcionado era relativamente sencillo y fácil de entender. Se trataba de un programa realizado en C capaz de leer en espacio de usuario del dispositivo (en nuestro caso la BBB). Es decir, era capaz de recibir y tratar toda la información recibida de forma continua por el chip ADS1198. Hasta aquí todo es relativamente sencillo, gracias al protocolo SPI y al programa de usuario en C era posible gestionar toda la información recibida de las muestras simuladas y realizar el tratamiento pertinente. El único problema hasta aquí es que realmente no era posible gestionar toda la información, si no más bien, casi toda la información.

Objetivos

En ciertos contextos, la pérdida de cierto número de muestras puede ser despreciable y no requerir mayor atención. Sin embargo, en este contexto concreto, al tratarse de procesamiento de señales en tiempo real, la pérdida de un cierto número de muestras, por pequeña que sea, puede ser decisiva a la hora de interpretar la información. El programa inicial escrito en C registraba una pérdida de muestras, que se podía determinar fácilmente hallando la diferencia entre el número de muestras generadas y el número de muestras leidas en el espacio de usuario.

El conjunto total de muestras que se perdía durante la transmisión de información era variable según distintas condiciones y factores externos que influyesen en el momento de la recopilación de información. Generalmente, si únicamente se ejecutaba el programa de usuario sin tener otras aplicaciones que consumiesen muchos recursos tanto de memoria como de cálculo computacional en la BBB, la suma total de muestras perdidas podía encontrarse entre el 1 y el 2 % del total de muestras generadas. Sin embargo, si mientras se ejecutaba el programa en C, se ejecutaba algún otro programa que necesitase algún tipo de recurso para ser ejecutado, el número de muestras perdidas por la lectura en espacio de usuario podía dispararse.

Observamos que al ejecutar incluso ciertos comandos de consola, podría llegar a darse la situación anteriormente nombrada. Suponiendo que la ejecución del programa se realiza sobre un sistema operativo anfitrión con alguna distribución linux ejecutándose (en nuestro caso se trataba de una distribución Debian especial para la placa BBB), si tratábamos de ejecutar un comando como `top`¹ (que muestra las tareas que estén ejecutándose en la máquina anfitriona en tiempo real y actualizadolas en intervalos cortos de tiempo como puede apreciarse en la [Fig 3.1]), el número de muestras perdidas aumentaba de forma exponencial. Es decir, para que las pérdidas realmente no fuesen significativas, el dispositivo sobre el que se ejecutase el programa, no debería estar ejecutando nada adicional, o al menos no muchas

¹Más información sobre el comando `top` disponible en <https://linux.die.net/man/1/top>

top - 10:59:20 up 2 days, 34 min, 1 user, load average: 0,84, 0,64, 0,58													
Tasks: 223 total, 2 running, 221 sleeping, 0 stopped, 0 zombie													
%Cpu(s): 4,9 us, 2,3 sy, 0,0 ni, 92,9 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st													
KiB Mem : 5534180 total, 138844 free, 3423732 used, 1971604 buff/cache													
KiB Swap: 9705464 total, 9705464 free, 0 used. 1605796 avail Mem													
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND		
32618	cristian	20	0	1252848	458816	129536	R	11,9	8,3	18:07.28	chrome		
1456	cristian	20	0	582004	13500	9968	S	5,6	0,2	40:36.96	pulseaudio		
32657	cristian	20	0	407872	41732	21632	S	5,3	0,8	7:05.62	chrome		
31706	cristian	20	0	1600488	247408	107088	S	2,0	4,5	17:21.52	chrome		
1417	cristian	20	0	1613004	288636	76352	S	1,7	5,2	69:10.86	compiz		
774	message+	20	0	44520	5416	3468	S	0,3	0,1	0:45.32	dbus-daemon		
967	root	20	0	417964	82484	52828	S	0,3	1,5	76:39.37	Xorg		
3101	root	20	0	0	0	0	S	0,3	0,0	0:00.86	kworker/3:0		
3330	root	20	0	0	0	0	S	0,3	0,0	0:00.18	kworker/0:2		
3404	root	20	0	43020	3944	3292	R	0,3	0,1	0:00.31	top		
1	root	20	0	185416	6048	3976	S	0,0	0,1	0:07.06	systemd		
2	root	20	0	0	0	0	S	0,0	0,0	0:00.05	kthreadd		
3	root	20	0	0	0	0	S	0,0	0,0	0:04.33	ksoftirqd/0		
7	root	20	0	0	0	0	S	0,0	0,0	1:53.40	rcu_sched		
8	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_bh		
9	root	rt	0	0	0	0	S	0,0	0,0	0:00.07	migration/0		
10	root	0	-20	0	0	0	S	0,0	0,0	0:00.00	lru-add-dr+		

Figura 3.1: Ejemplo de salida al ejecutar comando top en una terminal linux

más aplicaciones. Siendo conscientes de la baja probabilidad de que esta situación se diese en un entorno real, sabíamos que algo teníamos que hacer para evitar esa pérdida de muestras en cualquier situación. Esta pérdida era básicamente producida por interrupciones que lanzaba el propio sistema operativo anfitrión cuando fuese preciso, otorgando la CPU a otros procesos que no fuesen el programa que recogía las muestras durante cierto tiempo, suficiente para significar una pérdida representativa.

Sabiendo que el sistema operativo no tenía por qué garantizarnos la CPU para el programa de usuario durante la mayor parte del tiempo que este se encuentra en ejecución, tuvimos que investigar la mejor forma de poder solucionar este problema.

Alternativas

Teniendo en mente el problema anteriormente nombrado, era momento de buscar alternativas para solventarlo. La solución más inmediata que pasó por nuestra cabeza, fue la de sustituir la distribución Debian que utilizábamos en la BBB por una distribución que estuviese pensada para trabajar con análisis y tratamientos de datos en tiempo real. De esta forma, el sistema operativo no interrumpiría el proceso de análisis de muestras otorgando la CPU a otras tareas, y por consiguiente, presumiblemente no se produciría la pérdida de ninguna muestra. Sin embargo esta solución parecía poco sostenible, ya que el hecho de recurrir a una distribución tan específica podría no ser accesible para todo tipo de usuarios y dispositivos, sin olvidar la dependencia que se generaría hacia ese sistema operativo concreto, impidiendo con la más

absoluta de las certezas que funcionase éxitosamente en otros tipo de sistemas operativos (ya que por decirlo de algún modo, sería como un traje a medida). Sin embargo, si se utilizase una distribución genérica, como el Debian que usabamos en aquel momento, no habría problemas, ya que se trata de una distribución de propósito general a la que cualquier usuario podría recurrir de manera relativamente sencilla.

Una vez estábamos decididos a mantener el sistema operativo, la solución tenía que encontrarse explorando otras posibles vías. Fue entonces, cuando se nos presentó la posibilidad de usar dos microprocesadores que tenía la BBB, denominadas PRUs (programable real-time unit). Estos dos microprocesadores con los que contábamos, podían ser programados específicamente para tratar procesos de análisis y recopilación de muestras en tiempo real. De esta forma, aunque el sistema operativo decidiese ceder el uso de la CPU a otro proceso, las PRUs podrían seguir tratando los procesos en tiempo real en segundo plano, sin necesidad de requerir la CPU principal, por lo que la pérdida de muestras se reduciría totalmente gracias a estos pequeños procesadores que incorpora la placa. Se trata de dos procesadores de alta frecuencia (200-MHz) de arquitectura de 32 bits que ofrecen la posibilidad a los desarrolladores de tratar con operaciones en tiempo real.

A continuación se enumeran los motivos por los que escogimos recurrir a las PRUs:

- Tienen acceso a los pins, así como a la memoria interna de la BBB y a los periféricos del principal procesador principal que incorpora.
- Están diseñados para proveer software específico para periféricos como parte del sistema PRU-ICSS (Programmable Real-time Unit Industrial Control SubSystem).
- Son capaces de implementar soluciones relativamente simples a problemas complejos.
- Consta de un gran ancho de banda para comunicarse con la CPU principal y sus controladores, por lo que es improbable que se produzca el fenómeno conocido como cuello de botella.²
- Existen multitud de recursos y proyectos en los que se utilizan estos procesadores, que pueden ser consultados de forma libre y gratuita en la red.

²Se denomina cuello de botella a la situación que se presenta cuando en un proceso productivo, una fase de producción es más lenta que otras, lo que ralentiza el proceso de producción global.

Programación en tiempo real

Abordar la decisión de trabajar con las PRUs no es una tarea trivial. Requiere una gran labor de investigación previa, así como una sólida base en ciertos lenguajes de programación. Además, han sido necesarias tomar ciertas decisiones que han resultado cruciales para el correcto avance y desarrollo del proyecto, como ha podido ser en qué lenguaje programar las PRUs, o la configuración hardware para trabajar en el desarrollo.

Decisiones de diseño

Cuando comenzó el desarrollo de este proyecto, existían dos posibles alternativas a escoger a la hora de elegir lenguaje con el que trabajar en las PRUs:

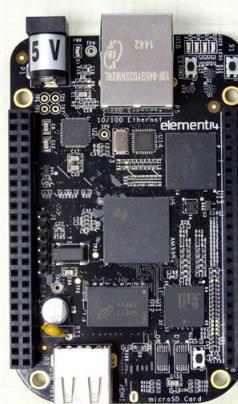
- **Posibilidad de programar en C:** Texas instruments introdujó en una de las últimas versiones de Code Composer Studio (Es un entorno de desarrollo integrado que soporta microcontroladores de Texas Instruments y otro tipo de procesadores embotados)³ la posibilidad de programar en C para programar las PRU. Las herramientas que proporciona Texas Instruments para ello se denominan CGT (Code Generation Tools) y son relativamente sencillas de utilizar.
- **Posibilidad de programar en ensamblador:** Hay dos tipos de ensamblador disponibles para la PRU, pasm y clpru.
 - Pasm es el ensamblador original para la PRU. Este ensamblador soporta una unidad de traducción simple y se monta directamente a una imagen binaria (u otros formatos compatibles).
 - Clpru es actualmente una herramienta de compilación completa que incluye ensamblador para la programación de la PRU. Soporta multitud de unidades de traducción y se monta directamente sobre ficheros objetos, los cuales deben ser enlazados al ejecutable final.⁴

Ante las alternativas que se nos plantearon, la decisión final fue la de programar en ensamblador, concretamente en pasm, ya que resultó ser una de las opciones más interesantes para este proyecto (aunque somos conscientes de que en la actualidad no es frecuente encontrarse con una situación en la que se requiera programar en ensamblador). Se optó por esta decisión de diseño, ya que investigar cual de las posibilidades era la que más flexibilidad podía aportar al contexto en el que se situaba el proyecto, pasm resultó

³Más información sobre este entorno de desarrollo en <http://www.ti.com/tool/ccstudio>

⁴Cabe destacar que tanto pasm como clpru soportan prácticamente la misma sintaxis.

Beaglebone Black Pinout Diagram



P9			P8		
Function	Physical Pins	Function	Function	Physical Pins	Function
DGND	1	2	DGND	1	2
VDD 3.3 V	3	4	VDD 3.3 V	3	4
VDD 5V	5	6	VDD 5V	5	6
SYS 5V	7	8	SYS 5V	7	8
PWR_BUT	9	10	SYS_RESET	9	10
UART4_RXD	11	12	GPIO_60	11	12
UART4_TXD	13	14	EHRPWM1A	13	14
GPIO_48	15	16	EHRPWM1B	15	16
SPIO_CSO	17	18	SPIO_D1	17	18
I2C2_SCL	19	20	I2C_SDA	19	20
SPIO_DO	21	22	SPIO_SCLK	21	22
GPIO_49	23	24	UART1_RXD	23	24
GPIO_117	25	26	UART1_TXD	25	26
GPIO_115	27	28	SP11_CSO	27	28
SP11_DO	29	30	GPIO_112	29	30
SP11_SCLK	31	32	VDD_ADC	31	32
AIN4	33	34	GND_ADC	33	34
AIN6	35	36	AIN5	35	36
AIN2	37	38	AIN3	37	38
AIN0	39	40	AIN1	39	40
GPIO_20	41	42	ECAPWMO	41	42
DGND	43	44	DGND	43	44
DGND	45	46	DGND	45	46

LEGEND

Power, Ground, Reset	1.8 Volt Analog Inputs
Digital Pins	Shared I2C Bus
PWM Output	Reconfigurable Digital

Figura 3.2: Diagrama de conexiones físicas disponibles en una BBB

aportar un mayor control y una valiosa posibilidad de optimizar el código final.

Configuración hardware

Tal y como indicábamos en la sección 2.2.1, la BBB es un dispositivo que aporta una gran flexibilidad a los desarrolladores, gracias a la facilidad que supone ampliar su funcionalidad con multitud de tipos de placas de expansión opcionales.

Cada uno de los pines físicos que incorpora la placa tiene una función determinada como puede apreciarse en la [Fig 3.2]. La BBB dispone de dos cabeceras completas (conocidas como P8 y P9) con multitud de pines disponibles, que permiten realizar conexiones físicas mediante cableado. La leyenda de la [Fig 3.2] muestra las funciones, o mejor dicho, las posibles funciones de los distintos pines:

- Para empezar, se han destacado en color rojo los pines de 5, 3 y 1.8 Volts, así como los pines de tierra (DGND). Hay que tener en cuenta que VDD_ADC es un pin de 1.8 V que se usa para proporcionar una

referencia para las funciones de lectura analógica.

- Los pines de propósito general de entrada y salida (GPIO) están destacados en color verde. Cabe destacar que algunos de estos pines se pueden usar para comunicación serie (UART)⁵.
- Si se desea simular una salida analógica comprendida entre 0 y 3.3 V, se pueden usar los pines PWM destacados en morado.
- Los pines destacados en color azul pueden ser utilizados como entradas analógicas⁶.
- Los pines en color naranja clarito pueden ser usados para I2C⁵.
- Los pines destacados en color naranja oscuro son fundamentalmente usados para aplicaciones de pantalla LCD.

Device Tree

Un Device Tree (DT) es una descripción del hardware de un sistema. Debería incorporar el nombre de la CPU, la configuración de memoria, y cualquier periférico (interno y externo). Un DT no debería ser usado para describir software, a pesar de que listar los modulos hardware pueda causar que estos se carguen. Es preciso recordar que los DTs son neutros en lo que se refiere al sistema operativo, es decir, no deberían incluir nada específico de Linux por ejemplo.

Un DT representa la configuración hardware como si de una jerarquía de nodos se tratase. Cada nodo puede contener propiedades y subnodos. Las propiedades se denominan arrays de bytes, los cuales pueden contener strings, numeros (big-endian), secuencias arbitrarias de bytes, y cualquier combinación de estos. Por analogia con un sistema de ficheros, los nodos son directorios, y las propiedades son ficheros.

Los DT generalmente se encuentran en un formato textual conocido como Device Tree Source (DTS) y son almacenados en ficheros con extensión .dts. La sintaxis DTS es como la de C, con llaves para agrupar código y punto y coma para concluir cada línea. En la [Fig 3.3] puede apreciarse un ejemplo muy simple de este tipo de ficheros.

Cabe destacar que los DTS requieren de punto y coma tras el cierre de una llave. El formato del binario compilado es denominado Flattened Device Tree (DFT) o Device Tree Blob (DTB), y se almacena en ficheros con extensión .dtb.

⁵Más información sobre UART e I2C disponible en <https://geekytheory.com/puertos-y-buses-1-i2c-y-uart>

⁶Estas entradas analógicas toleran voltajes comprendidos entre los 0 y los 1.8 Voltios, no soportan voltajes superiores a los 1.8 V

```
/dts-v1;
/include/ "common.dtsci";

/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];
        cousin: child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
            my-cousin = <&cousin>;
        };
    };
};

};


```

Figura 3.3: Ejemplo simple de un fichero .dts

Device Tree Overlay

Un SoC (System-on-Chip moderno) moderno es un dispositivo muy complejo; un DT completo podría suponer cientos de líneas de código. Situar un SoC en una placa junto a otros componentes solo hace que las cosas sean aun más complejas. Para que sea relativamente manejable, especialmente si hay dispositivos relaciones que comparten componentes, tiene sentido diferenciar los elementos comunes en ficheros con extensión `.dtsci`, para que sean incluidos desde los ficheros `.dts` que los requieren.

Pero cuando una placa como la BBB es compatible con accesorios expandidos, el problema puede ser aun más complejo. En última instancia, cada configuración posible requiere un DT para ser descrita, pero una vez que se tenga en cuenta el hardware de base y las expansiones que requieren el uso de unos pines GPIO determinados que pueden ser compartidos por otras configuraciones diferentes, el número de combinaciones posibles comienza a multiplicarse rápidamente.

Lo que se necesita es una forma de describir estos componentes opcionales utilizando un DT parcial, y luego poder construir un árbol completo tomando un DT como base y añadiendo una serie de elementos opcionales. Estos elementos opcionales son denominados *overlays*. Al DT que se forma siguiendo este proceso se le denomina Device Tree Overlay (DTO).

Para nuestro proyecto, fue necesaria la creación y personalización de un DTO; la base sobre la empezamos esta creación fue la que se utiliza en el

```

/dts-v1/;
/plugin/;

{
    compatible = "ti,beaglebone", "ti,beaglebone-black";
    part-number = "EBB-PRU-ADC2";
    version = "00A0";
    /* This overlay uses the following resources */
    exclusive-use =
        "P9.24", "P9.25", "P9.27", "P9.28", "P9.29", "P9.30", "P9.31", "P8.46", "pru0", "pru1";
    fragment@0 {
        target = <&am33xx_pinmux>;
        __overlay__ {
            pru_pru_pins: pinmux_pru_pru_pins { // The PRU pin modes
                pinctrl-single,pins = <
                    0x184 0x2e // DATA_READY P9_24 pr1_pru0_pru_r31_16, MODE6 | INPUT | DIS 00101110=0x2e
                    0x1ac 0xd // START P9_25 pr1_pru0_pru_r30_7, MODE5 | OUTPUT | DIS 00001101=0xd
                    0x190 0xd // RESET P9_31 pr1_pru0_pru_r30_0, MODE5 | OUTPUT | DIS 00001101=0xd
                    0x1a4 0xd // CS P9_27 pr1_pru0_pru_r30_5, MODE5 | OUTPUT | DIS 00001101=0xd
                    0x19c 0x2e // MISO P9_28 pr1_pru0_pru_r31_3, MODE6 | INPUT | DIS 00101110=0x2e
                    0x194 0xd // MOSI P9_29 pr1_pru0_pru_r30_1, MODE5 | OUTPUT | DIS 00001101=0xd
                    0x198 0xd // CLK P9_30 pr1_pru0_pru_r30_2, MODE5 | OUTPUT | DIS 00001101=0xd
                    // This is for PRU1, the sample clock -- debug only
                    0x0a4 0xd // SAMP P8_46 pr1_pru1_pru_r30_1, MODE5 | OUTPUT | DIS 00001101=0xd
                >;
            };
        };
    };
    fragment@1 { // Enable the PRUSS
        target = <&pruss>;
        __overlay__ {
            status = "okay";
            pinctrl-names = "default";
            pinctrl-0 = <&pru_pru_pins>;
        };
    };
};

pinctrl@1;

```

Figura 3.4: Device Tree Overlay del proyecto para la BBB

capítulo 13 de Exploring BBB (Molloy, Enero, 2014). A partir de ahí se realizaron diversas modificaciones y expansiones para obtener un DTO hecho a medida acorde con las necesidades del proyecto. El resultado final del DTO que se obtuvo finalmente y que es utilizado actualmente en la BBB se ilustra en la [Fig 3.4]. El código final fue comentado de modo que pudiese ser interpretado y comprendido de forma sencilla. Como puede apreciarse en la imagen, el primer paso es indicar cuales van a ser los pines que se van a utilizar de forma exclusiva. A continuación se le asigna un modo a cada uno de los pines anteriormente indicado. De esta forma, el modo que se asigna, la señal que va a ir asociada a ese pin concreto y otra información complementaria puede ser encontrada en el código comentado. Cabe destacar que existen ciertas restricciones a la hora de asignar cierto modo a un pin determinado, ya que no a todos los pines se les puede asignar cualquier modo. En nuestro caso, solo determinados pines de la BBB tienen acceso a determinados modos que permiten interacciones tanto con la PRU0, como con la PRU1. Por ejemplo, el modo `pr1_pru0_pru_r31_7` solo puede ser asignado al pin `P9_25` (PRU0), y el modo `pr1_pru1_pru_r31_16` solo puede asignarse al pin `P9_26` (PRU1). Los dos ejemplos anteriormente comentados con puestos en práctica en nuestro DTO, aunque la lista de restricciones abarca muchos más modos y pines. Esta información puede encontrarse más detallada en la [Fig 3.5]. Cabe destacar que este tipo de ficheros `.dts` sigue siempre una misma estructura en lo que al código respecta, aunque las posibles combinaciones

P9_11	28	0x870/070	30	UART4_RXD	gpio0[30]	uart4_rxd_mux2
P9_12	30	0x878/078	60	GPIO1_28	gpio1[28]	mcasp0_aclk_r_mux3
P9_13	29	0x874/074	31	UART4_TXD	gpio0[31]	uart4_txd_mux2
P9_14	18	0x848/048	50	EHRPWM1A	gpio1[18]	ehrpwm1A_mux1
P9_15	16	0x840/040	48	GPIO1_16	gpio1[16]	ehrpwm1_tripzone_input
P9_16	19	0x84c/04c	51	EHRPWM1B	gpio1[19]	ehrpwm1B_mux1
P9_17	87	0x95c/15c	5	I2C1_SCL	gpio0[5]	
P9_18	86	0x958/158	4	I2C1_SDA	gpio0[4]	
P9_19	95	0x97c/17c	13	I2C2_SCL	gpio0[13]	pr1_uart0_rts_n
P9_20	94	0x978/178	12	I2C2_SDA	gpio0[12]	pr1_uart0_cts_n
P9_21	85	0x954/154	3	UART2_RXD	gpio0[3]	EMU3_mux1
P9_22	84	0x950/150	2	UART2_RXD	gpio0[2]	EMU2_mux1
P9_23	17	0x844/044	49	GPIO1_17	gpio1[17]	ehrpwm0_sync0
P9_24	97	0x984/184	15	UART1_RXD	gpio0[15]	pr1_pru0_pru_r31_16
P9_25	107	0x9ac/1ac	117	GPIO3_21	gpio3[21]	pr1_pru0_pru_r31_7
P9_26	96	0x980/180	14	UART1_RXD	gpio0[14]	pr1_pru1_pru_r31_16
P9_27	105	0x9a4/1a4	115	GPIO3_19	gpio3[19]	pr1_pru0_pru_r31_5
P9_28	103	0x99c/19c	113	SPI1_CS0	gpio3[17]	pr1_pru0_pru_r31_3
P9_29	101	0x994/194	111	SPI1_D0	gpio3[15]	pr1_pru0_pru_r31_1
P9_30	102	0x998/198	112	SPI1_D1	gpio3[16]	pr1_pru0_pru_r31_2
P9_31	100	0x990/190	110	SPI1_SCLK	gpio3[14]	pr1_pru0_pru_r31_0

Figura 3.5: Restricciones en la asignación de modos a pines relevantes para el proyecto

de configuraciones hardware son prácticamente infinitas.

Conecciones físicas

Una vez diseñada la configuración hardware, era momento de realizar las conexiones físicas mediante cableado. Entre los objetivos que han de satisfacer estas conexiones físicas, cabe destacar los siguientes:

- Alimentar el ADS1198, ya que se alimenta a través de la BBB.
- Posibilitar la conexión SPI entre ambos dispositivos (ADS1198 y BBB).
- Garantizar el uso de pines con fines GPIO (para cada una de las distintas señales).
- Conectar el analizador lógico Saleae con cada una de las señales que se desee depurar.

Tabla 3.1: Conexiones físicas necesarias para la alimentación del chip ADS1198

BBB Pin	ADS1198 Pin	Uso de la conexión
P9 Pin 1	J4 Pin 5	GND
P9 Pin 3	J4 Pin 9	3.3V
P9 Pin 5	J4 Pin 10	5V

Tabla 3.2: Conexiones físicas de carácter general GPIO

BBB Pin	ADS1198 Pin	Señal asociada a la conexión
P9 Pin 23	J3 Pin 15	DATA_READY
P9 Pin 25	J3 Pin 8	RESET
P9 Pin 27	J3 Pin 14	START

Tabla 3.3: Conexiones físicas para posibilitar la conexión SPI

BBB Pin	ADS1198 Pin	Señal asociada a la conexión
P9 Pin 29	J3 Pin 13	SPI_OUT
P9 Pin 30	J3 Pin 11	SPI_IN
P9 Pin 31	J3 Pin 3	SPI_SCLK
P9 Pin 28	J3 Pin 1	SPI_CS0

Tabla 3.4: Conexiones físicas del analizador lógico Saleae

Canal	ADS1198 Pin	Señal asociada a la conexión
CH0	J3 Pin 15	DATA_READY
CH1	J3 Pin 3	CLK
CH2	J3 Pin 1	CHIP_SELECT
CH3	J3 Pin 13	DATA_OUT
CH4	J3 Pin 11	DATA_IN
CH5	J3 Pin 8	RESET
CH6	J3 Pin 14	START
CH7	J4 Pin 5	GND

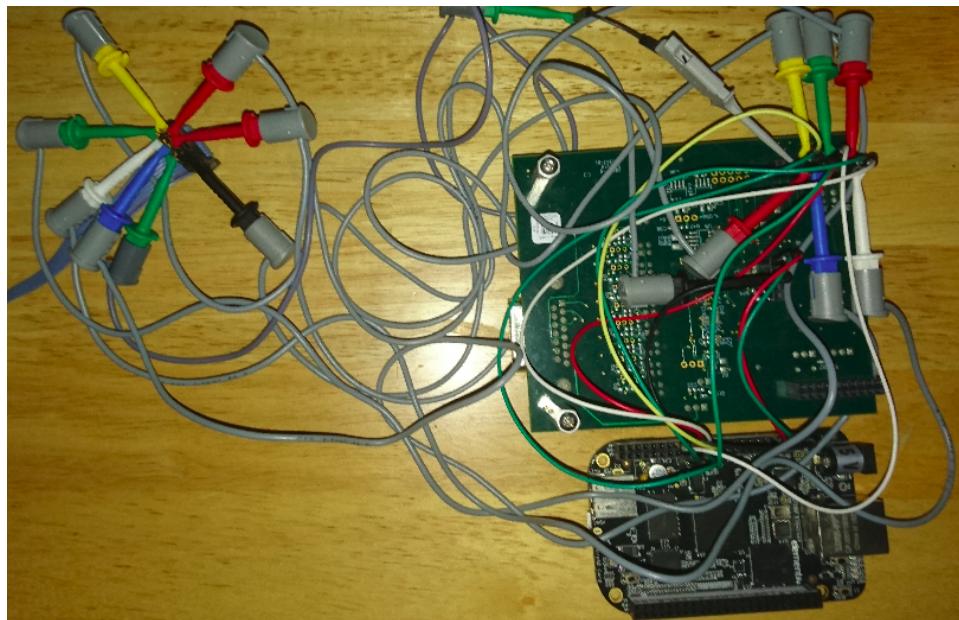


Figura 3.6: Conexiones físicas necesarias para el proyecto

Una vez se han realizado las conexiones indicadas en las tablas 3.1, 3.2, 3.3 y 3.4, puede considerarse que la configuración hardware ha concluido con éxito. En nuestro caso, el resultado de completar todas las conexiones físicas puede apreciarse en la [Fig 3.6].

Ecosistema software

El propósito de esta sección no es más que proporcionar al lector una idea general e intuitiva del modo de funcionamiento del ecosistema software, así como de sus relaciones internas, de forma que sea posible entender el contexto y la motivación de cada una de las acciones que se realizan en el software. La forma en la que se relaciona el ecosistema es bastante sencilla de entender, al menos, si no nos centramos en los complejos detalles técnicos referentes a la implementación de cada uno de los componentes.

El software más destacado que se ejecuta en la BBB es el siguiente:

- Programa principal desarrollado en C que se ejecuta en espacio de usuario denominado `medcape.c`.
- Programa desarrollado en código ensamblador, del que se hablará con más detenimiento más adelante denominado `PRUADC.p`.
- Programa desarrollado en C, denominado `mem2file.c` y encargado de la escritura de datos de memoria a un fichero externo.

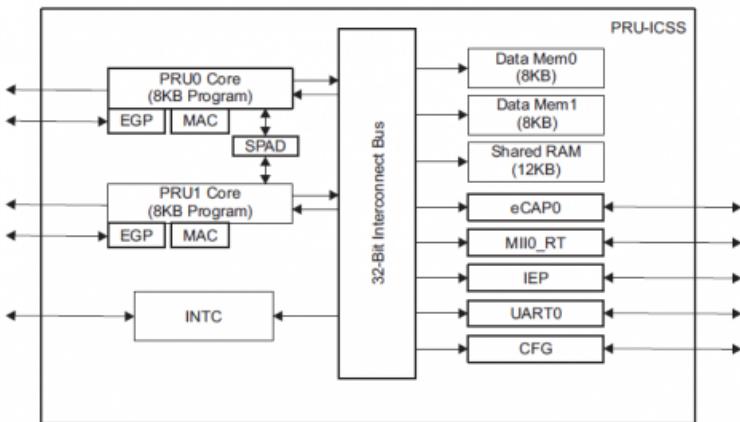


Figura 3.7: Bancos de memoria del PRU

- Programa muy sencillo desarrollado en Python, y denominado `BTConnection.py` y encargado de la recepción y envío de información a otros dispositivos.

La relación de este ecosistema es bastante simple. El programa principal `medcape.c`, el cual está programado para ejecutarse según enciende la BBB, es el encargado de realizar las inicializaciones pertinentes para que funcione el intercambio de información, de mapear ciertas regiones de memoria, de ejecutar el binario principal⁷, así como de capturar y generar interrupciones hacia el código que ejecuta el PRU. De forma paralela a la ejecución del programa en C, se ejecuta el programa en ensamblador ya que se da la orden de ejecutar el binario desde el propio programa en C.

El programa en C consta de un hilo que es el encargado de realizar las interacciones pertinentes con el código que se ejecuta en el PRU. Este hilo entra en modo de espera indefinido hasta que reciba una interrupción por parte del código ejecutándose en el PRU indicando que las muestras están listas y guardadas en el banco de memoria del PRU (Se ilustra más información sobre estos bancos de memoria en la [Fig. 3.7]).

Cuando la recopilación del paquete de muestras está lista y almacenada en el banco de memoria del PRU, se manda una interrupción desde el código ensamblador al programa C que se encontraba en espera de capturar un evento concreto. La ejecución continua con una llamada al programa en C `mem2file.c`, encargado de acceder a las regiones concretas del banco de memoria del PRU en la que el código ensamblador ha guardado las muestras, y guardarlas en un fichero (generalmente con extensión `.data`). Además de guardarse en este fichero, el paquete de muestras es enviado por una tubería de la que ahora hablaremos.

⁷Código ensamblador compilado que ejecuta el PRU, concretamente el fichero `PRUADC.p`

Es en este momento cuando entra en juego el programa desarrollado en Python, que accede a la información de la tubería, la cual permite la sincronización y comunicación con otros procesos. El programa Python es capaz de enviar esta información a otros dispositivos, en nuestro caso concreto, se envía a un dispositivo móvil con sistema operativo Android como veremos con más detalle más adelante.

Una vez es asimilado el modo de funcionamiento del ecosistema, aunque sólo sea de forma intuitiva, es posible entrar en detalles técnicos más concretos y complejos, ya que es posible comprender el contexto del ecosistema software.

Inicialización ADS

Una vez configurado correctamente el DTO (tal y como se comenta en la sección 3.3.2), y realizadas las conexiones físicas pertinentes entre la BBB y el ADS1198, es necesario inicializar el ADS. Inicializar el ADS por software es fundamental. Con inicialización nos referimos a realizar la correcta configuración de una serie de parámetros que permita que ambos dispositivos se puedan comunicar correctamente ya que de no configurarlo con los parámetros adecuados no conseguiríamos establecer ningún tipo de conexión entre ambos, es decir, sería como si ni siquiera estuviesen conectados físicamente. Cabe destacar que la inicialización se realiza en código ensamblador, y a diferencia del lenguaje C, no existen bibliotecas que faciliten al desarrollador una capa superior de abstracción para realizar la tarea.

En la Fig 3.8] y se ilustra el proceso de inicialización del ADS en en el código del proyecto original. Es un código relativamente sencillo y completamente funcional. Los pasos que se siguen en el código original, desde resetear el ciclo, hasta la lectura de datos en modo continuo, son replicados en código ensamblador. El objetivo es conseguir la misma correcta funcionalidad de la que gozaba el código original.

```

printf("Reset cycle\n");
if ( ads_reset() == -1 ) {
    printf("can't reset ADS\n");
    return -1;
}

printf("Stop Read Data Continuously mode (just in case)\n");
if ( ads_sdatac() == -1 ) {
    printf("can't stop read data continuously\n");
    return -1;
}
sleep(1);

printf("Set sample rate %d\n", tmp_srate);
if ( ads_set_rate(sample_rate) == -1 ) {
    printf("can't set sample rate!!\n");
    return -1;
}
sleep(1);

printf("Enable internal reference\n");
if ( ads_enable_intref() == -1 ) {
    printf("Error enabling internal reference!!\n");
    return -1;
}

printf("START: start continuous reading\n");
if ( ads_start() == -1 ) {
    printf("can't start conversion\n");
    return -1;
}

printf("RDATAC: read data continuously\n");
if ( ads_command(RDATAC) == -1 ) {
    printf("can't read data continuously\n");
    return -1;
}

```

Figura 3.8: Inicialización ADS en el código original en C

```

//-----SDATAC-----
SEND_SDATAC:
    LBB0    r2, r1, 0, 4    // Load sdatac command=0x11
    //We only want 8 bits, but 1 register=32 bits and we can only take 1 bit per clock edge,
    //so we do lsl(logical shift left) to move our LSB to MSB positions(i.e. first 8 positions)
    SUB    r13, r14, 8
    CALL   TRUNCATE
//-----
    CLR    r30.t5          // set the CS line low (active low)
    MOV    r4, 8             // going to write/read 8 bits (1 byte)
    CALL   SPICLK_LOOP      // repeat call the SPICLK procedure until all 8 bits written/read
    SET    r30.t5          // pull the CS line high (end of sample)
//-----SLEEP 2 SECONDS-----
MOV    r12, DELAYCOUNT
CALL   DELAY_FUNCTION

```

Figura 3.9: Función SDATAC en código ensamblador

```

//-----SET痈SAMPLE RATE      (i.e. data_ready rate)-----
SET痈SAMPLE RATE:
//1st bit=====
MOV r2, 0x41
//We only want 8 bits, but 1 register=32 bits and we can only take 1 bit per clock edge,
//so we do lsl(logical shift left) to move our LSB to MSB positions(i.e. first 8 positions)
SUB r13, r14, 8
CALL TRUNCATE
//-----
CLR r30.t5          // set the CS line low (active low)
MOV r4, 8            // going to write/read 8 bits (1 byte)
CALL SPICLK_LOOP    // repeat call the SPICLK procedure until all 8 bits written/read
MOV r12, 1000         //Número aleatorio para hacer un sleep de un poco de tiempo
CALL DELAY_FUNCTION
//2nd bit=====
MOV r2, 0x00000000
MOV r4, 8            // going to write/read 8 bits (1 byte)
CALL SPICLK_LOOP    // repeat call the SPICLK procedure until all 8 bits written/read
MOV r12, 1000         //Número aleatorio para hacer un sleep de un poco de tiempo
CALL DELAY_FUNCTION
//3rd bit=====
//Default is 0x04 (500Hz) (ADS1198)
//0x00 is 8kHz (Maximum speed supported by ADS1198)
//0x06 is 125Hz (Minimum speed supported by ADS1198)
//Each increased number is half the speed of the previous number
MOV r2, 0x06
//We only want 8 bits, but 1 register=32 bits and we can only take 1 bit per clock edge,
//so we do lsl(logical shift left) to move our LSB to MSB positions(i.e. first 8 positions)
SUB r13, r14, 8
CALL TRUNCATE
MOV r4, 8            // going to write/read 8 bits (1 byte)
CALL SPICLK_LOOP    // repeat call the SPICLK procedure until all 8 bits written/read
MOV r12, 1000         //Número aleatorio para hacer un sleep de un poco de tiempo
CALL DELAY_FUNCTION
SET r30.t5           // pull the CS line high (end of sample)
SET r30.t1 //MOSI is active in original hardware SPI
//-----SLEEP 2 SECONDS-----
MOV r12, DELAYCOUNT
CALL DELAY_FUNCTION

```

Figura 3.10: Función SET痈SAMPLE RATE en código ensamblador

```

//-----SET痈INTERNAL_REFERENCE_ADS(i.e. data_ready rate)-----
SET痈INTERNAL_REFERENCE_ADS:
//1st bit=====
//LBBO r2, r1, 8, 12    // Load sample rate speed
MOV r2, 0x43
//We only want 8 bits, but 1 register=32 bits and we can only take 1 bit per clock edge,
//so we do lsl(logical shift left) to move our LSB to MSB positions(i.e. first 8 positions)
SUB r13, r14, 8
CALL TRUNCATE
CLR r30.t5          // set the CS line low (active low)
MOV r4, 8            // going to write/read 8 bits (1 byte)
CALL SPICLK_LOOP    // repeat call the SPICLK procedure until all 8 bits written/read
MOV r12, 1000         //Número aleatorio para hacer un sleep de un poco de tiempo
CALL DELAY_FUNCTION
//2nd bit=====
MOV r2, 0x00000000
MOV r4, 8            // going to write/read 8 bits (1 byte)
CALL SPICLK_LOOP    // repeat call the SPICLK procedure until all 8 bits written/read
MOV r12, 1000         //Número aleatorio para hacer un sleep de un poco de tiempo
CALL DELAY_FUNCTION
//3rd bit=====
MOV r2, 0xC0
SUB r13, r14, 8
//We only want 8 bits, but 1 register=32 bits and we can only take 1 bit per clock edge,
//so we do lsl(logical shift left) to move our LSB to MSB positions(i.e. first 8 positions)
CALL TRUNCATE
MOV r4, 8            // going to write/read 8 bits (1 byte)
CALL SPICLK_LOOP    // repeat call the SPICLK procedure until all 8 bits written/read
MOV r12, 1000         //Número aleatorio para hacer un sleep de un poco de tiempo
CALL DELAY_FUNCTION
SET r30.t5           // pull the CS line high (end of sample)
SET r30.t1 //MOSI is active in original hardware SPI
//-----SLEEP 2 SECONDS-----
MOV r12, DELAYCOUNT
CALL DELAY_FUNCTION

```

Figura 3.11: Función SET痈INTERNAL_REFERENCE en código ensamblador

```
//-----RDATAC-----  
SEND_RDATAC:  
    LBBO    r2, r1, 4, 8      // Load rdatac command  
    SUB    r13, r14, 8  
    //We only want 8 bits, but 1 register=32 bits and we can only take 1 bit per clock edge,  
    //so we do lsl(logical shift left) to move our LSB to MSB positions(i.e. first 8 positions)  
    CALL    TRUNCATE  
  
    CLR     r30.t5          // set the CS line low (active low)  
    MOV     r4, 8            // going to write/read 8 bits (1 byte)  
    CALL    SPICLK_LOOP      // repeat call the SPICLK procedure until all 8 bits written/read  
  
    SET     r30.t5          // pull the CS line high (end of sample)  
    MOV    r16, 0             //Counter of bits received  
    // Need to wait at this point until it is ready to take a sample  
POLLING_DATA_READY_HIGH:  
    QBBS    POLLING_DATA_READY_HIGH, r31.t16  
    //Auxilliary register, to store 32 bits: it's stored in a register and then set to 0 every 32 bits  
    MOV    r21, 0x00000000  
    MOV    r22, 0  
    MOV    r26, 1
```

Figura 3.12: Función SEND_RDATAC en código ensamblador

```

GET_SAMPLE: //Receive 1 sample (18 bytes)
    CLR    r30.t5          // set the CS line low (active low)
    // Para evitar coger el 1er byte (FF) que nos llega de la muestra
    MOV    r12, 200 //Número aleatorio para hacer un sleep de un poco de tiempo
    CALL  DELAY_FUNCTION

    //-----.
    //Number of bytes/sample -- 19 Bytes/Sample (basandose en el SPI original por hardware)
    MOV r15, 21

GET_BYTE:
    MOV    r4, 8             // going to write/read 1 byte (8 bits)
    MOV    r2, 0x00000000      //What we want to write (i.e 0)
    CALL  SPICLK_LOOP        // repeat call the SPICLK procedure until all 8 bits written/read
    MOV    r12, 800 //Número aleatorio para hacer un sleep de un poco de tiempo
    CALL  DELAY_FUNCTION
    SET    r30.t1 //MOSI: 1 (before start taking the sample)
    MOV    r12, 800 //Número aleatorio para hacer un sleep de un poco de tiempo
    CALL  DELAY_FUNCTION
    CLR    r30.t1 //MOSI: 0 (before start taking the sample)
    SUB    r15, r15, 1         // decrement loop counter

    QBNE  GET_BYTE, r15, 0   // repeat loop unless zero
    SET    r30.t5          // pull the CS line high (end of sample)
    SET    r30.t1 //MOSI: 1 (before start taking the sample)
    LSR    r3, r3, 1           // SPICLK shifts left too many times left, shift right once
    LSR    r17, r17, 1           // SPICLK shifts left too many times left, shift right once
    LSR    r18, r18, 1           // SPICLK shifts left too many times left, shift right once
    LSR    r19, r19, 1           // SPICLK shifts left too many times left, shift right once
    LSR    r20, r20, 1           // SPICLK shifts left too many times left, shift right once
    LSR    r23, r23, 1           // SPICLK shifts left too many times left, shift right once
    LSR    r27, r27, 1           // SPICLK shifts left too many times left, shift right once

```

Figura 3.13: Comunicación SPI en código ensamblador

En las anteriores figuras se ilustra un código en ensamblador funcional, utilizado y probado en el presente proyecto, que imita el funcionamiento del código original de inicialización del ADS en C.

Comunicación SPI

La comunicación SPI funciona tal y como se explica en la sección 2.2.2. El proceso de comunicación se realiza en ensamblador también. Comienza dando el valor activo bajo a la señal Chip Select como puede apreciarse en la figura [Fig 3.13]. A continuación se lee o se escribe un número específico de bits, de forma que se llama a la función SPICLK_LOOP tantas veces como bits se quieran leer o escribir. Cuando ha concluido este proceso, se da el valor activo alto a la señal Chip Select. De esta forma se indica que ha finalizado una muestra concreta.

Cabe destacar que cada registro en ensamblador tiene una capacidad de 4 bytes. Debido al tamaño de cada muestra que se recoge, es necesario almacenar esta información en 7 registros. En la función SPICLK_LOOP se utilizan los registros r16 como contador para saber cuantos bits se han recibido, y r20 como registro auxiliar. Considerando que el tamaño de cada muestra es de 18 bytes, cada una es almacenada de la siguiente forma:

- **Registro r3:** Primer registro (3 bytes)
- **Registro r17:** Segundo registro (3 bytes)
- **Registro r18:** Tercer registro (3 bytes)

- **Registro r19:** Cuarto registro (3 bytes)
- **Registro r20:** Quinto registro (3 bytes)
- **Registro r23:** Sexto registro (3 bytes)
- **Registro r27:** Séptimo registro (3 bytes)

Esta información se refleja en forma de código en las figuras [Fig 3.14] y [Fig 3.15]. En estas figuras se muestra el código de la función SPICLK_LOOP escrito en código ensamblador, en la cual se refleja el proceso de iteración sobre cada uno de los bits concretos, así como el uso de todos los registros anteriormente descritos.

```

SPICLK_LOOP: //LOOP through the X bits (read and write)
    SPI_CLK_BIT:
        MOV     r0, r11 // time for clock low -- assuming clock low before cycle
        CLKLOW:
            SUB    r0, r0, 1 // decrement the counter by 1 and loop (next line)
            QBNE  CLKLOW, r0, 0 // check if the count is still low
            // So this way we take every bit we want
            QBBC  DATALOW, r2,t31 //QBC = Quick branch if bit clear (if bit=0)
            SET   r30.t1 //The bit that we want is 1, so we send a 1
            QBA   DATACONTD //QBA = Quick branch always
        DATALOW:
            CLR   r30.t1 //The bit that we want is 0, so we send a 0
        DATACONTD:
            SET   r30.t2 // set the clock high
            MOV   r0, r11 // time for clock high
        CLKHIGH:
            SUB    r0, r0, 1 // decrement the counter by 1 and loop (next line)
            QBNE  CLKHIGH, r0, 0 // check the count
            LSL   r2, r2, 1 //Perform logical shift left to take the next bit we want to write
            // clock goes low now -- read the response on MISO
            CLR   r30.t2 // set the clock low
            //----Store 1 received bit in a register (r21)----
            QBBC  DATAINLOW, r31.t3 //Take the MISO bit we receive from ADS
            OR    r21, r21, 0x000001 //bit received=1, so we set the LSB to 1
            DATAINLOW: //bit received=0, so we have nothing to do
            LSL   r21, r21, 1 //because we had already set all bits of the register to 0
            -----
            QBNE END_TAKING_MISO, r26, 1 //Flag para detectar que empieza la toma de datos
            QBLE STORE_1ST_REGISTER, r22, 23
            QBA BREAK_IF //r22 > 23
            STORE_1ST_REGISTER: //Entra aquí si 23 <= r22
            QBNE STORE_2ND_REGISTER, r22, 23 //23 < r22
            MOV   r3, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
            MOV   r21, 0x000000
            =====
            QBLE STORE_2ND_REGISTER, r22, (23*2)+1
            QBA BREAK_IF //r22 > 23
            STORE_2ND_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
            QBNE STORE_3RD_REGISTER, r22, (23*2)+2 //23 < r22
            MOV   r18, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
            MOV   r21, 0x000000
            =====
            QBLE STORE_3RD_REGISTER, r22, (23*3)+2
            QBA BREAK_IF //r22 > 23
            STORE_3RD_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
            QBNE STORE_4TH_REGISTER, r22, (23*3)+2 //23 < r22
            MOV   r18, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
            MOV   r21, 0x000000
            =====
            QBLE STORE_4TH_REGISTER, r22, (23*4)+3
            QBA BREAK_IF //r22 > 23
            STORE_4TH_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
            QBNE STORE_5TH_REGISTER, r22, (23*4)+3 //23 < r22
            MOV   r19, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
            MOV   r21, 0x000000
            =====
            QBLE STORE_5TH_REGISTER, r22, (23*5)+4
            QBA BREAK_IF //r22 > 23
            STORE_5TH_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
            QBNE STORE_6TH_REGISTER, r22, (23*5)+4 //23 < r22
            MOV   r20, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
            MOV   r21, 0x000000
            =====
            QBLE STORE_6TH_REGISTER, r22, (23*6)+5
            QBA BREAK_IF //r22 > 23
            STORE_6TH_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
            QBNE STORE_7TH_REGISTER, r22, (23*6)+5 //23 < r22
            MOV   r23, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
            MOV   r21, 0x000000
            =====
            QBLE STORE_7TH_REGISTER, r22, (23*7)+6
            QBA BREAK_IF //r22 > 23
            STORE_7TH_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
            QBNE BREAK_IF, r22, (23*7)+6 //23 < r22
            MOV   r27, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
            MOV   r21, 0x000000
            =====
            BREAK_IF:
                ADD r22, r22, 1 //Entra aquí si r22 < 23
        END_TAKING_MISO:
            SUB   r4, r4, 1 // count down through the bits
            QBNE  SPICLK_LOOP, r4, 0
    RET

```

Figura 3.14: Función SPICLK_LOOP en código ensamblador (primera parte)

```

QBNE STORE_3RD_REGISTER, r22, (23*3)+2
    QBA BREAK_IF //r22 > 23
    STORE_3RD_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
        QBNE STORE_4TH_REGISTER, r22, (23*3)+2 //23 < r22
        MOV   r18, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
        MOV   r21, 0x000000
    =====
    QBLE STORE_4TH_REGISTER, r22, (23*4)+3
    QBA BREAK_IF //r22 > 23
    STORE_4TH_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
        QBNE STORE_5TH_REGISTER, r22, (23*4)+3 //23 < r22
        MOV   r19, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
        MOV   r21, 0x000000
    =====
    QBLE STORE_5TH_REGISTER, r22, (23*5)+4
    QBA BREAK_IF //r22 > 23
    STORE_5TH_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
        QBNE STORE_6TH_REGISTER, r22, (23*5)+4 //23 < r22
        MOV   r20, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
        MOV   r21, 0x000000
    =====
    QBLE STORE_6TH_REGISTER, r22, (23*6)+5
    QBA BREAK_IF //r22 > 23
    STORE_6TH_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
        QBNE STORE_7TH_REGISTER, r22, (23*6)+5 //23 < r22
        MOV   r23, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
        MOV   r21, 0x000000
    =====
    QBLE STORE_7TH_REGISTER, r22, (23*7)+6
    QBA BREAK_IF //r22 > 23
    STORE_7TH_REGISTER: //Entra aquí si r22 >= 23      23 <= r22
        QBNE BREAK_IF, r22, (23*7)+6 //23 < r22
        MOV   r27, r21 //If we wan to check how many bits have been 1, we put here: MOV r3, 23
        MOV   r21, 0x000000
    =====
    BREAK_IF:
        ADD r22, r22, 1 //Entra aquí si r22 < 23
END_TAKING_MISO:
    SUB   r4, r4, 1 // count down through the bits
    QBNE  SPICLK_LOOP, r4, 0
RET

```

Figura 3.15: Función SPICLK_LOOP en código ensamblador (segunda parte)

Guardado de muestras en memoria

Según se van procesando las muestras, es necesario almacenarlas en algún sitio, para luego poder acceder a ellas desde el programa en C, del que se habla en la sección 3.4.4 (que a su vez vuelve a transmitir la información a otras vías). Es en este caso cuando entran en juego los bancos de memoria del PRU, en la que se almacenan de forma temporal todas las muestras que se recogen.

Este proceso de guardado de muestras en memoria se realiza en ensamblador. Para guardar la información de una muestra completa, se almacena la información almacenada en cada uno de los registros indicados en la sección 3.4.2. Es decir, una vez todos estos registros están llenos con la información de las muestras, el siguiente paso es el almacenamiento en memoria de cada uno de estos registros, que una vez guardados en posiciones de memoria consecutivas, reflejan el contenido de una muestra completa.

Una vez se guardan las muestras, se lanza una interrupción del PRU hacia el programa principal en C, el cual se encuentra en ejecución y a la espera de una interrupción por parte del código ensamblador. Una vez capturada esta interrupción, la ejecución del código en C puede reanudarse (tal y como se explica en la sección 3.4).

Todo este proceso se refleja en el código ensamblador, concretamente en la función STORE _ DATA. El contenido completo de esta función se muestra en las figuras [Fig 3.16] y [Fig 3.17].

```

STORE_DATA:
    SUB    r9, r9, 21      // store the sample value in memory
    //Reset Counter of bits written(They need to be clear for next sample to take) into register of their comment:
    MOV    r22, 0 //r3
    MOV    r26, 1 //r20
    //-----
    MOV    r5, r3
    CALL REVERSE_ENDIANNESS
    //-----Store in RAM the whole sample (18 bytes)-----
    SBB0   r1, r8, 0, 2    //last register only needs 2 bytes (It has byte 16 and byte 17)
    ADD    r8, r8, 2 // shifting RAM addres by 2 bytes (1 register = 3bytes, but this one only needs 2 bytes)
    MOV    r12, 50 //Número aleatorio para hacer un sleep de un poco de tiempo
    CALL DELAY_FUNCTION
    MOV    r5, r17
    CALL REVERSE_ENDIANNESS
    //-----
    SBB0   r1, r8, 0, 3
    ADD    r8, r8, 3
    MOV    r12, 50 //Número aleatorio para hacer un sleep de un poco de tiempo
    CALL DELAY_FUNCTION
    MOV    r5, r18
    CALL REVERSE_ENDIANNESS
    //-----
    SBB0   r1, r8, 0, 3
    ADD    r8, r8, 3
    MOV    r12, 50 //Número aleatorio para hacer un sleep de un poco de tiempo
    CALL DELAY_FUNCTION
    MOV    r5, r19
    CALL REVERSE_ENDIANNESS
    //-----
    SBB0   r1, r8, 0, 3
    ADD    r8, r8, 3
    MOV    r12, 50 //Número aleatorio para hacer un sleep de un poco de tiempo
    CALL DELAY_FUNCTION
    MOV    r5, r20
    CALL REVERSE_ENDIANNESS
    //-----
    SBB0   r1, r8, 0, 3    // store the value r3 in memory (It has byte 0, byte 1, byte 2, and byte 3)
    ADD    r8, r8, 3 // shifting RAM addres by 3 bytes (1 register = 3bytes)
    MOV    r12, 50 //Número aleatorio para hacer un sleep de un poco de tiempo
    CALL DELAY_FUNCTION
    MOV    r5, r23
    CALL REVERSE_ENDIANNESS

```

Figura 3.16: Función STORE_RAM en código ensamblador (primera parte)

```

SBB0   r1, r8, 0, 3    // store the value r3 in memory (It has byte 0, byte 1, byte 2, and byte 3)
ADD    r8, r8, 3 // shifting RAM addres by 4 bytes (1 register = 4bytes)
MOV    r12, 50 //Número aleatorio para hacer un sleep de un poco de tiempo
CALL DELAY_FUNCTION
MOV    r5, r27
CALL REVERSE_ENDIANNESS
//-----
SBB0   r1, r8, 0, 1    // store the value r3 in memory (It has byte 0, byte 1, byte 2, and byte 3)
ADD    r8, r8, 1 // shifting RAM addres by 4 bytes (1 register = 4bytes)
MOV    r12, 50 //Número aleatorio para hacer un sleep de un poco de tiempo
CALL DELAY_FUNCTION
ADD r6, r6, 1
//Comment this section to store everything without limit in RAM(without storing->removin->storing->removing)
// it'll cause kernel exceptions when full
QBNM CONTINUE_THIS_LOOP_RAM, r24, SIZE_CHUNK_RAM-1
//generate an interrupt to notifyicate a new chunk of samples is ready
// in RAM to be given to host program (In C)
MOV r1, 0XF
SBB0   r1, r8, 0, 1    // store the value r3 in memory (It has byte 0, byte 1, byte 2, and byte 3)
ADD    r8, r8, 1 // shifting RAM addres by 4 bytes (1 register = 4bytes)
//-----
MOV r8, r25 //Reset ram direction to points to initial cell of RAM memory(r25)
MOV r24, 0
MOV R31.b0, PRU0_R31_VEC_VALID | PRU_EVTOUT_1 //Notifyicate ARM for a sample ready
//WAIT FOR ARM TO READ:
WBS r31, #30
QBA END_PROCESS_RAM_BUFFER
CONTINUE_THIS_LOOP_RAM:
ADD r24, r24, 1
END_PROCESS_RAM_BUFFER:
// clear registers to receive the response from the ADS
MOV    r3, 0x00000000
MOV    r17, 0x00000000
MOV    r18, 0x00000000
MOV    r19, 0x00000000
MOV    r20, 0x00000000
MOV    r23, 0x00000000
MOV    r27, 0x00000000
//QBGE END, r9, 0    //Have taken the full set of samples. Comment if we want while(1)
//It's QBGE( r9 < 0 ) and not QBEQ (r0 == 0), just in case we receive more samples than we expect
POLLING_DATA_READY_LOW: //We have already taken the sample so we've plenty of time here...
QBBC  POLLING_DATA_READY_LOW, r31.t16
QBA   POLLING_DATA_READY_HIGH

```

Figura 3.17: Función STORE_RAM en código ensamblador (segunda parte)

Lectura y transmisión de muestras

Como ya se adelantaba en la sección 3.4.3, se ha desarrollado un programa en C encargado de leer la información de las muestras almacenadas en el banco de memoria del PRU, para posteriormente guardarla en un fichero (con extensión `.data`)⁸. Este programa en C está desarrollado a raíz de un ejemplo que se muestra en el capítulo 13 del libro Exploring BeagleBone (Molloy, Enero, 2014) y tanto ampliado como adaptado para su correcto funcionamiento en el presente proyecto.

Inicialmente, este programa crea el fichero en el que se guardarán los datos de las muestras. Accede al espacio de memoria del banco del PRU en la que se almacenan las muestras⁹. Es necesario asegurar que cada grupo de muestras contiene la información correcta y precisa, y para ello tras cada grupo de muestras se escriben dos bytes concretos, como si de un sello se tratase a modo de asegurar que los datos recibidos no sean corruptos o haya pérdidas. Concretamente tras el espacio exacto que requiere un grupo de muestras, se escriben a continuación los bytes FF.

El programa `mem2file.c` se asegura que contengan este identificador en el lugar exacto en el que corresponde. En caso de no encontrarlo, significa que no se ha escrito esta información aun, por lo que el programa sigue en bucle de espera hasta que la información se escriba. Sin embargo, si se encuentran estos dos bytes en el lugar específico tras cada grupo de muestras, es el propio programa el encargado de sustituir este sello por otros bytes distintos, a modo a su vez de indicador de que la muestra ha sido leída exitosamente. Casi como si de otro sello (para confirmar la lectura) se tratase, se sustituyen los antiguos bytes por 00.

Posteriormente se abre la tubería de Python (brevemente comentada en la sección 3.4)¹⁰, y trata de enviarse la información por la misma en caso de haberse abierto correctamente. En la [Fig 3.18] se ilustra el proceso completo tras haberlo introducido brevemente.

⁸Este fichero binario está limitado por software a un tamaño máximo de 50MB

⁹Las muestras se almacenan por grupos de 100

¹⁰La capacidad máxima de esta tubería es de 4KB

```

int mem2file_main(int number_chunk, int samples_taken) {
    target = addr;
    int i = 0;
    stat(filename, &st);
    int size_file = st.st_size;
    if (size_file >= (size_packet*samples_per_chunk)*50000) {
        if ((fd_output = freopen(filename, "w+b", fd_output)) == NULL ) {
            perror("Error al abrir el fichero de datos");
        }
    }
    while (((uint8_t *)pru1_data0)[(size_packet*samples_per_chunk)] != 0xFF) {
        printf("Waiting for mem shared to refresh \n");
        fflush(stdout);
        printf("Mem flag incorrect [%d] is: %x \n", (size_packet*samples_per_chunk),
        ((uint8_t *)pru1_data0)[(size_packet*samples_per_chunk)]);
        fflush(stdout);
        printf("\n");
    }
    printf("Mem flag correct [%d] is: %x \n", (size_packet*samples_per_chunk),
    ((uint8_t *)pru1_data0)[(size_packet*samples_per_chunk)]);
    fwrite(pru1_data0, size_packet, samples_per_chunk, fd_output);
    ((uint8_t *)pru1_data0)[(size_packet*samples_per_chunk)] = 0x00;
    // Try to open the Python pipe
    if (pipePython.fid < 1) {
        pipePython.fid = open(pipePython.name, O_WRONLY | O_NONBLOCK);
        //printf("Pipe open\n");
    }
    // Send data if the Python pipe is open
    if (pipePython.fid > 0) {
        int writeSuccess = write(pipePython.fid, pru1_data0, size_packet*samples_per_chunk);
        if (writeSuccess < 0) {
            close(pipePython.fid);
            pipePython.fid = -1;
        } else if (writeSuccess > PIPE_BUF) {
            printf("Warning!! Write in FIFO is not atomic\n");
        }
    }
    return 0;
}

```

Figura 3.18: Función principal del programa `mem2file.c` desarrollado en C

Capítulo 4

Aplicación Android

*Lo que sabemos es una gota de agua;
lo que ignoramos es el océano.*

Isaac Newton

RESUMEN: Aquí va el resumen del capítulo 4.

Envío de datos

Como ya se adelantaba en la sección 3.4 en el envío de datos interviene el programa desarrollado en Python `BTConnection.py`¹. El programa original ha sido ampliado y adaptado a las necesidades que exigía el presente Proyecto. Este programa lee datos de una tubería² en la que se introducen los paquetes de muestras que estén preparados para ser procesados. Las muestras son leídas una por una de la tubería y son enviadas por bluetooth vía socket a la aplicación Android.

La comunicación por socket no es más que el mecanismo por el cual estos dos programas pueden intercambiar flujos de datos de manera ordenada y fiable³.

Para que las muestras sean correctamente leídas es necesario que la BBB disponga de conexión bluetooth, algo que de lo que de forma nativa no dispone. Para ello ha sido necesario utilizar un periférico que le aportase esta característica a la placa. En concreto, el periférico que hemos utilizado se denomina Bluetooth CSR 4.0 Dongle, y puede apreciarse de forma visual en la [Fig 4.1].

¹El autor del programa original es José Manuel Bote

²Más información disponible en [https://es.wikipedia.org/wiki/Tuber%C3%ADa_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Tuber%C3%ADa_(inform%C3%A1tica))

³Más información al respecto disponible en https://es.wikipedia.org/wiki/Socket_de_Internet



Figura 4.1: Dongle Bluetooth 4.0 CSR utilizado en la BBB

Ha sido necesario configurar y emparejar este periférico con la BBB para que esta pudiese disponer de conexión Bluetooth. La configuración se ha realizado mediante la intrucción de comandos en el terminal Debian de la BBB mediante el protocolo `ssh`, ya que la comunicación con la BBB no puede realizarse mediante interfaz visual.

Cabe destacar que para que el envío de información de cada una de las muestras mediante sockets concluya correctamente, es necesario que previamente el dispositivo Android que ejecuta la aplicación y la BBB (mediante el dongle Bluetooth) se encuentren emparejados.

Una vez satisfecho el requisito de emparejamiento, las acciones que realiza el programa Python son relativamente sencillas. En primer lugar, realiza una espera una conexión por socket por parte de la aplicación Android. Hasta que no se acepte la conexión en el dispositivo móvil, la aplicación Python permanece bloqueada a la espera.

Tras aceptar la conexión, se accede a la tubería que contiene la información de los paquetes de muestras. La información se envía por socket (muestra a muestra) a la aplicación Android. Si no es posible realizar este envío, se lanzará una excepción por la que volverá a quedarse el programa a la espera de una conexión vía socket por parte de la aplicación Android.

Se puede apreciar de forma visual el contenido más importante del programa Python en la [Fig 4.2].

```

while True:

    # Wait for an incoming connection
    print "Waiting for a new connection on channel", serverSocketChannel
    clientSocket, clientAddress = serverSocket.accept()
    print "Accepted connection from ", clientAddress
    time.sleep(1)

    # Send the data received from pipe to connected device
    try:

        pipeFile = open(pipeName, "rb")
        print pipeName, "opened"
        #outputFile = open(outputFileName, "wb")
        #print outputFileName, "opened"

        while True:

            try:
                while True:
                    chunk = pipeFile.read(chunkSize)
                    if len(chunk) <= 0:
                        break
                    else:
                        clientSocket.send(chunk)
            except:
                break # Device is disconnected

    except IOError:
        print "Pipe cannot be established"

    print "Device disconnected"
    clientSocket.close()

```

Figura 4.2: Programa desarrollado en Python BTConnection.py

Recepción de datos

Frente al envío de los datos comentado en la sección 4.1, la recepción de los mismos se realiza desde la aplicación Android. El autor de la aplicación original es José Manuel Bote. Sobre la aplicación original se han realizado modificaciones y ampliaciones adaptándola así a las necesidades de nuestro proyecto. El lenguaje de programación utilizado para desarrollar la aplicación es Java.

La aplicación original estaba diseñada para comunicarse por Bluetooth con una BBB para que le transmitiese datos, y estos fuesen representados. La propia aplicación ofrece la posibilidad de activar el Bluetooth del terminal desde el que se usa, si es que tiene. Tras esto, tratará de emparejarse con el otro dispositivo, y esperar la transmisión de información. El propósito fundamental que tiene es representar en forma de onda la información que recibe por Bluetooth. Aunque su propósito sea relativamente simple, internamente presenta cierto nivel de complejidad, puede suponer todo un reto técnico

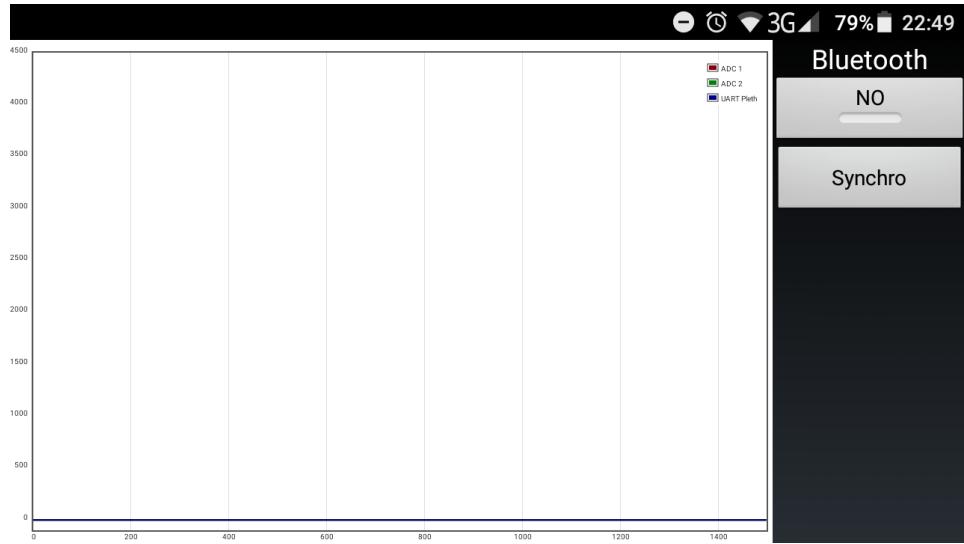


Figura 4.3: Pantalla principal de la aplicación Android original

modificarla y ampliarla, sobre todo para programadores poco avezados en la materia.

La aplicación original tiene una gráfica en la que se representan las ondas recibidas, pudiendo representar hasta un máximo de 2 canales distintos. En la [Fig 4.3] puede apreciarse la pantalla principal de la aplicación original ejecutándose en un dispositivo móvil.

La recepción de los datos por socket de la aplicación Python se realiza desde un hilo que se ejecuta indefinidamente en la aplicación Android. Este hilo se bloquea en espera hasta que haya datos disponibles en la tubería Python. Cuando hay datos de muestras disponibles, estos son analizados e interpretados por la aplicación para posteriormente ser representados de forma gráfica. Este proceso de espera a la disponibilidad de los datos y tratamiento de los mismos (sin incluir la representación gráfica) puede apreciarse de forma visual en la [Fig 4.4].

Mejoras desarrolladas

La realización de pequeñas mejoras y adaptaciones sobre la aplicación original era totalmente necesaria para garantizar el correcto funcionamiento de la misma, teniendo en cuenta que el contexto del proyecto requería una configuración diferente a la original. Una vez configurada, y realizadas las modificaciones, la aplicación funcionaba correctamente.

Sin embargo consideramos que todavía podían hacerse más mejoras e implementación de nuevas funcionalidades. Entre estas posibles mejoras se a

```

void beginListen() {
    stopWorker = false;
    workerThread = new Thread(new Runnable() {
        @Override
        public void run() {
            while (!Thread.currentThread().isInterrupted() && !stopWorker) {
                if (BTConnection.receiveDataIsAvailable()) {
                    byte[] recvBuff = BTConnection.receiveData();
                    int bytesRead = BTConnection.getBytesRead();
                    for (int i = 0; i < bytesRead; i++) {
                        ringBuffer.add(recvBuff[i]);
                    }
                    try {
                        if (bytesRead > 0) {
                            rawOutputStream.write(recvBuff, 0, bytesRead);
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    });
}

```

Figura 4.4: Fragmento del código que ejecuta el hilo encargado de la recepción de datos

realizar, planteamos las siguientes:

- Renovación total de la interfaz gráfica de la aplicación, dotandola de un diseño vistoso y amigable.
- Despliegue de servidores que establezcan conexión con la aplicación vía internet utilizando sockets específicos.
- Separación de la aplicación en dos posibles versiones, una para pacientes y otra para doctores.
- Retransmisión en tiempo real (*streaming*) de ECG de pacientes a doctores.

Sabíamos que la realización de todas estas ampliaciones supondría un reto técnico a nuestras capacidades como informáticos, sobre todo si queríamos implementar éxitosamente la opción de realizar una retransmisión de información en tiempo real de pacientes a doctores. El despliegue de un servidor que actuase como intermediario entre ambos era totalmente necesario para garantizar la comunicación entre los dos extremos. La conexión a este servidor se realizaría por ambas partes mediante una conexión a internet.

Renovación visual

La renovación visual de la aplicación ha supuesto un cambio muy significativo respecto a la apariencia de la misma. En lugar de acceder directamente a la pantalla principal, que nos muestra la gráfica y la opción de sincronizar mediante Bluetooth, según se abre la nueva aplicación, se accede a una

pantalla de identificación de usuario, tal y como puede apreciarse en la [Fig 4.5a].

Una vez el usuario correspondiente se ha identificado, accede a un pequeño menu con 3 opciones disponibles como puede observarse en la [Fig4.5b]. La primera opción permite la recepción de información mediante Bluetooth con la BBB ([Fig4.5d]). La segunda permite buscar un paciente para ver en tiempo real su ECG en tiempo real ([Fig4.5c]), y la última nos proporciona información de nuestro perfil de usuario.

Como puede apreciarse por las imágenes, la renovación visual ha llevado la implementación de diversas mejoras en la aplicación, y a vez ha adquirido una interfaz más amigable de cara a la experiencia final de usuario.

Cabe destacar que la aplicación facilita su uso tanto en modo horizontal, como en modo vertical (como se aprecia en la [Fig 4.5]). Sin embargo, la versión original solo facilitaba su uso en modo horizontal. La ampliación de posibilidades de uso facilita al usuario flexibilidad a la hora de utilizar la app.

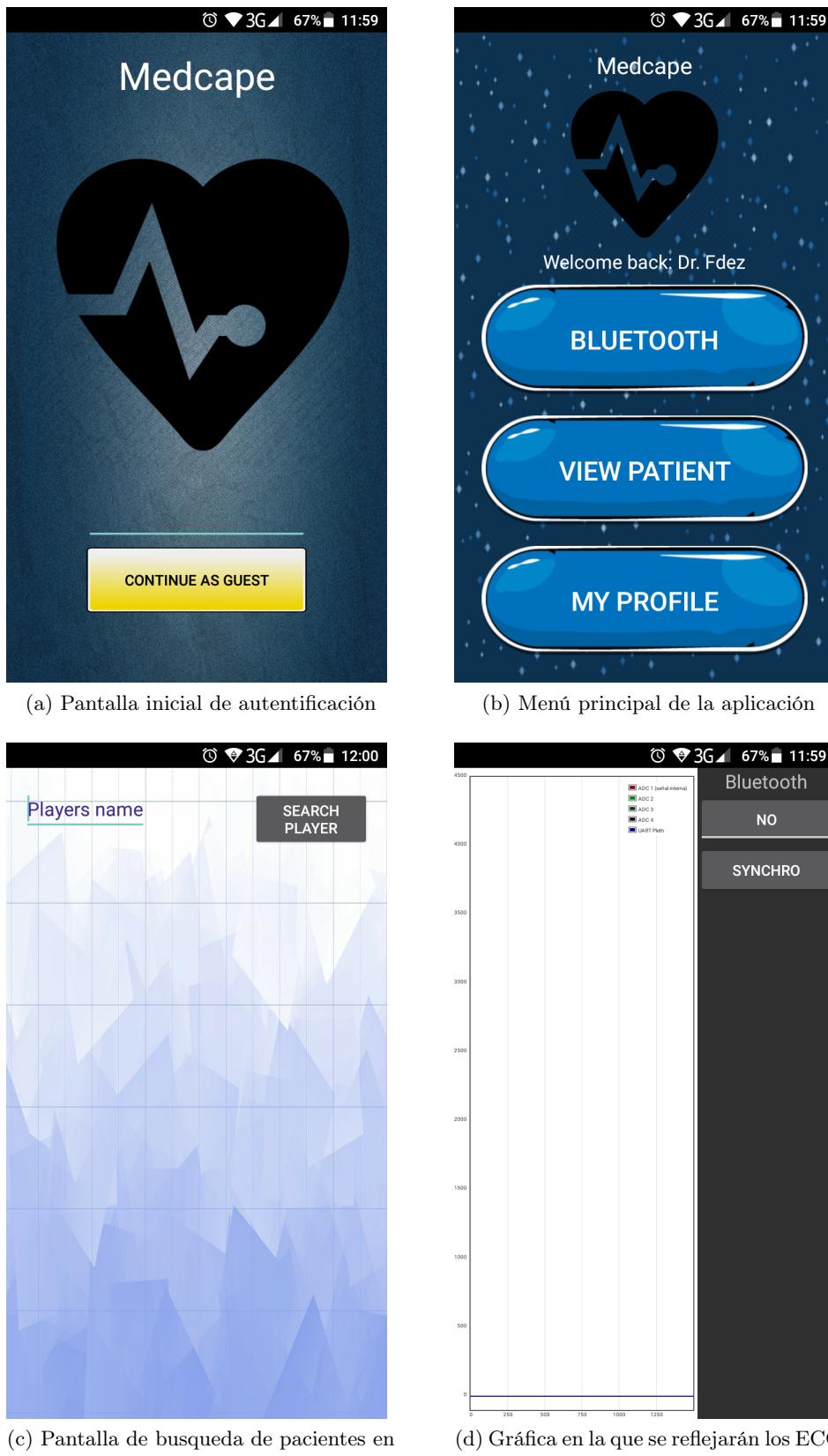


Figura 4.5: Pantallas principales de la aplicación Android

Ecosistema de streaming

Tal y como adelantábamos en la sección 4.4, ofrecer un servicio de *streaming* en tiempo real desde la aplicación Android no era un reto sencillo.

Para poder comunicar éxitosamente en tiempo real la aplicación que utilizase un paciente con la que utilizase un doctor, era necesario introducir una infraestructura de comunicación que permitiese la comunicación entre ambas partes. Esta infraestructura sería finalmente un servidor que atendiese y sirviese peticiones realizadas desde la propia aplicación.

Características del servidor

El servidor que utilizamos para realizar la comunicación por ambos extremos fue un ordenador personal, ya que no contabamos en ese momento con otros medios que facilitasen dicha opción.

Este servidor está desarrollado en `Node.js`, un lenguaje muy similar a Javascript, solo que a diferencia de Javascript, está pensado para ser utilizado en la parte del servidor en lugar de la del cliente. El servidor es capaz de atender peticiones a través de internet mediante sockets, y de servir peticiones de la misma forma. Es decir, las dos aplicaciones (tanto la del paciente como la del doctor) como el servidor deben disponer de conexión a internet para que la comunicación en tiempo real funcione.

La función del servidor es recibir paquetes de `bytes` desde la aplicación de un paciente, y despacharlos de la misma forma a la aplicación de un doctor, donde estos pueden ser representados de forma simultánea (o casi) en ambos dispositivos. Es decir la información se recibe en crudo, y se envía en crudo, sin ningún tratamiento por parte del servidor.

El servidor puede ser accedido desde cualquier lugar del planeta, gracias a que la ip local del servidor ha sido montada sobre un servidor de dns⁴.

En la [Fig 4.6] puede apreciarse uno de los fragmentos de código más significativos del servidor.

Características de la base de datos

Para el almacenamiento de usuarios se ha recurrido a una base de datos NoSQL, en este caso concreto el sistema de base de datos utilizado ha sido `MongoDB`.

La base de datos se hace necesaria en el momento en el que se quiere tener un almacen persistente de información. En nuestro caso, la utilizamos para el almacenaje de usuarios, ya que por ejemplo, si un doctor quiere buscar a un paciente determinado para ver su ECG en tiempo real, es necesario realizar

⁴Concretamente nosotros hemos recurrido a no-ip. Más información sobre este servicio gratuito en su página web oficial <https://www.noip.com/>

```

socket.on('request transmission server', function(data){
    var patient_socket_id = users[data.patient_name];
    console.log('request transmission server');
    console.log("Doctor: ", socket.id);
    console.log("Patient: ", data.patient_name, " with id ",
    patient_socket_id);
    var data_to_client = {};
    data_to_client["doctor_socket_id"] = socket.id;
    data_to_client["patient_socket_id"] = patient_socket_id;

    socket.broadcast.to(patient_socket_id).emit('request transmission client',
    {doctor_socket_id: socket.id});
});

socket.on('emit transmission server', function(data){
    var doctor_socket_id = data.doctor_socket_id;
    socket.broadcast.to(doctor_socket_id).emit('receive transmission client',
    {transmission: data.transmission});
});

```

Figura 4.6: Fragmento del código que ejecuta el servidor desarrollado en `Node.js`

una consulta en la base de datos para saber si este paciente existe. Una vez tenemos la certeza de que existe, y que está emitiendo datos, es posible que el doctor los reciba en tiempo real.

Interpretación de datos

Los datos finales son interpretados y representados por la aplicación Android. En principio, los datos podrían ser retransmitidos desde pacientes reales utilizando ciertos sensores que permitan generar un ECG a partir de sus señales cardíacas. Sin embargo, este proyecto ha sido desarrollado utilizando un simulador de ECG, que en principio, y bajo las circunstancias propicias, debería aportar la misma información que un paciente real. El simulador de pacientes utilizado durante el desarrollo y las pruebas del proyecto puede apreciarse en la [Fig 4.7].

La información generada por el simulador, tras ser interpretada y tratada llega a la aplicación Android, la cual es la encargada de representarla, e incluso de retransmitirla en tiempo real. La representación de información por parte de la aplicación Android, y con ausencia de errores, debería ser como muestra la [Fig 4.8]. De la misma forma, un doctor que utilice la aplicación debería ser capaz de representar de la misma forma en la que le llega al paciente la señal, y todo esto, en tiempo real.



Figura 4.7: Simulador de ECG utilizado durante el desarrollo del proyecto



Figura 4.8: Interpretación de la señal por parte de la aplicación Android

Bibliografía

*Y así, del mucho leer y del poco dormir,
se le secó el celebro de manera que vino
a perder el juicio.*

Miguel de Cervantes Saavedra

ELINUX.ORG. *BeagleBone Black*. Versión electrónica, 2017. Disponible en <http://elinux.org/Beagleboard:BeagleBoneBlack> (último acceso, Mayo, 2017).

INSTRUMENTS, T. *ADS1198*. Versión electrónica, 2017. Disponible en <http://www.ti.com/product/ADS1198> (último acceso, Mayo, 2017).

MOLLOY, D. *Exploring BeagleBone*. Versión electrónica, Enero, 2014. Disponible en <http://exploringbeaglebone.com/> (último acceso, Mayo, 2017).

Lista de acrónimos

*-¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.*

*Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes*

*-Buena está – dijo Sancho –; firmela vuestra merced.
–No es menester firmarla – dijo Don Quijote–,
sino solamente poner mi rúbrica.*

*Primera parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes*

