Comunicació de dades a través de línies de potència mitjançant un sistema basat en BeagleBone
(Data communication through power lines using a system based on BeagleBone)

Estudis: Enginyeria de Telecomunicació
Autor: Josep Maria Ortega César
Director/a: Manuel Domínguez Pumar
Any: 2016
Data communication through power lines using a system based on BeagleBone
Table of Contents

Agraïments ........................................................................................................ 11
Resum del Projecte ............................................................................................. 12
Resumen del Proyecto ......................................................................................... 13
Abstract ............................................................................................................ 14

1 Introduction ..................................................................................................... 15
  1.1 Context ........................................................................................................ 15
  1.2 Scenarios of use .......................................................................................... 16
    1.2.1 Industrial scope: data reporting to a central unit ............................... 16
    1.2.2 Domestic scope: sensors with medium-power consumption ............. 16
  1.3 Objectives ................................................................................................... 17
  1.4 Challenges .................................................................................................. 18
  1.5 Document structure .................................................................................... 18
  1.6 Document styles ........................................................................................ 19

2 Background ...................................................................................................... 20
  2.1 PLC ............................................................................................................ 20
    2.1.1 Basic notions ......................................................................................... 20
    2.1.2 Standard protocols ............................................................................... 21
    2.1.3 CENELEC Bands ............................................................................... 23
    2.1.4 Free open software solutions ................................................................. 23
    2.1.5 Commercial development kits ............................................................... 24
  2.2 BeagleBone black ....................................................................................... 26
  2.3 Linux ........................................................................................................... 27
  2.4 PlcCape board ............................................................................................ 29
    2.4.1 Overview ............................................................................................... 29
    2.4.2 PlcCape-v1 .......................................................................................... 30
    2.4.3 PlcCape-v2 .......................................................................................... 31
    2.4.4 Software management ......................................................................... 32
  2.5 AFE031 ...................................................................................................... 33

3 Implementation ................................................................................................. 36
  3.1 Overview ..................................................................................................... 36
  3.2 Software development strategy ................................................................. 36
  3.3 Components developed ............................................................................ 38
  3.4 Drivers ........................................................................................................ 39
    3.4.1 Main target ........................................................................................... 39
    3.4.2 Basic notions ......................................................................................... 39
    3.4.3 Loadable vs built-in ............................................................................... 39
    3.4.4 Device Tree Overlays ......................................................................... 40
      3.4.4.1 Basic notions ............................................................................... 40
      3.4.4.2 SPI data pins direction ................................................................. 41
      3.4.4.3 Driver entry-point hooking ......................................................... 42
      3.4.4.4 DTO loading .............................................................................. 42
    3.4.5 Drivers loading notes .......................................................................... 43
    3.4.6 Drivers developed ............................................................................... 43
      3.4.6.1 spi-driver ...................................................................................... 44
3.4.6.1.1 Summary .................................................. 44
3.4.6.1.2 Requirements ........................................ 44
3.4.6.1.3 Testing means ........................................ 44
3.4.6.1.4 Auto Chip Select toggling ....................... 45
3.4.6.1.5 Ping Pong buffer with EDMA .................. 49
3.4.6.1.6 Analysis .................................................. 53
3.4.6.1.7 Effective DAC frequency ......................... 54
3.4.6.2 adc-driver ................................................ 58
3.4.6.2.1 Summary .............................................. 58
3.4.6.2.2 Requirements ........................................ 58
3.4.6.2.3 Patch .................................................... 58
3.4.6.2.4 Analysis ................................................ 61
3.5 Static libraries .................................................. 68
3.5.1 Overview .................................................... 68
3.5.2 Libraries developed ...................................... 68
  3.5.2.1 libplc-tools ............................................. 69
  3.5.2.2 libplc-gpio ............................................. 70
  3.5.2.3 libplc-adc .............................................. 71
  3.5.2.4 libplc-cape ............................................ 72
3.6 Plugins .......................................................... 73
3.6.1 Overview .................................................... 73
3.6.2 Categories ................................................... 73
  3.6.2.1 Encoders ................................................ 74
  3.6.2.2 Decoders ................................................. 76
  3.6.2.3 User Interface ......................................... 77
3.6.3 Plugins developed ........................................ 80
  3.6.3.1 encoder-wave ......................................... 81
  3.6.3.2 encoder-pwm ............................................ 83
  3.6.3.3 decoder-pwm ............................................ 85
  3.6.3.4 encoder-ook ............................................. 86
  3.6.3.5 decoder-ook ............................................. 87
  3.6.3.6 ui-ncurses .............................................. 90
  3.6.3.7 ui-console .............................................. 91
3.7 Applications .................................................... 92
3.7.1 Overview .................................................... 92
3.7.2 Applications execution .................................. 92
3.7.3 Applications developed ................................ 93
  3.7.3.1 plc-cape-demo-lib-usage .......................... 94
  3.7.3.2 plc-cape-autotest .................................... 95
  3.7.3.3 plc-cape-lab .......................................... 96
    3.7.3.3.1 Main concepts .................................... 96
    3.7.3.3.2 Features ........................................... 97
    3.7.3.3.3 Basic operation ................................... 97
    3.7.3.3.4 Command line parameters ..................... 102
3.8 Pending tasks .................................................. 102
3.9 Development with Octave .................................. 104
3.9.1 Post-analysis .............................................. 104
3.9.2 IIR coefficients .......................................... 104
8 Annex III. Getting started with the development environment ........................................ 153

8.1 Tree structure ............................................................................................................. 153
8.1.1 Source code .......................................................................................................... 153
8.1.2 Output warehouse .................................................................................................. 154
8.2 Basic equipment ........................................................................................................ 154
8.3 Checking the building environment .......................................................................... 155
8.4 Basic setup: compiling the ‘plc-cape’ framework ..................................................... 156
8.5 Advanced setup: out-of-tree building ....................................................................... 156
8.6 Benefits of the out-of-tree building .......................................................................... 158
8.7 Required packages: ncurses ..................................................................................... 159
8.8 Building hierarchy ..................................................................................................... 160
8.9 Building options ........................................................................................................ 161
8.10 makefile strategy on user-space components ....................................................... 161
8.11 makefile strategy on kernel-modules ...................................................................... 162
8.12 Doxygen .................................................................................................................... 166

9 Annex IV. Software design principles ........................................................................ 169

9.1 Object Oriented Programming .................................................................................. 169
9.2 Other source code guidelines ................................................................................... 170

10 Annex V. Source code files ....................................................................................... 173

10.1 Main C source code ................................................................................................ 173
10.2 Device Tree Overlays ............................................................................................ 173
    10.2.1 SPI ................................................................................................................... 173
    10.2.2 ADC ................................................................................................................ 174
10.3 Octave ..................................................................................................................... 175

11 Bibliography .............................................................................................................. 178

11.1 BeagleBoneBlack .................................................................................................... 178
11.2 Linux ....................................................................................................................... 178
11.3 GNU packages ........................................................................................................ 179
11.4 C Language ............................................................................................................. 179
11.5 PLC ........................................................................................................................ 179
11.6 SDR ........................................................................................................................ 180
11.7 AFE031 ................................................................................................................... 180
11.8 Tools ....................................................................................................................... 180
Figures

Figure 1. BeagleBone Black components .......................................................... 15
Figure 2. BBB main specifications .................................................................. 26
Figure 3. PlcCape board attached to a BeagleBone Black ................................. 29
Figure 4. Blocks diagram of the assembly BBB (blue) + PlcCape (green) ........... 29
Figure 5. PlcCape-v1 mounted board ............................................................... 30
Figure 6. PlcCape-v1 probing points ............................................................... 30
Figure 7. PlcCape-v2 mounted board ............................................................... 31
Figure 8. PlcCape-v2 probing points ............................................................... 31
Figure 9. AFE031 blocks and pins ................................................................. 33
Figure 10. Drivers tree .................................................................................... 43
Figure 11. Writing to the DAC register ........................................................... 45
Figure 12. Absence of CSN pulse with the standard SPI driver ....................... 45
Figure 13. SPI Single-Channel Mode behavior .............................................. 46
Figure 14. Automatic CSN pulse with the patched SPI driver ......................... 47
Figure 15. Real-time test of DAC samples transmission using a sawtooth wave .. 47
Figure 16. Test of sinusoid generation at 50kHz ............................................. 48
Figure 17. Typical effect in the DAC generation of a background operation ...... 48
Figure 18. Chip-Select timing control ............................................................. 54
Figure 19. SCLK (blue) and CSN (red) at SPI bitrate = 375kbps ..................... 54
Figure 20. SCLK (blue) and CSN (red) at SPI bitrate = 1500kbps .................. 55
Figure 21. SCLK (blue) and CSN (red) at SPI bitrate = 3000kbps .................. 55
Figure 22. SCLK (blue) and CSN (red) at SPI bitrate = 6000kbps ................. 56
Figure 23. Width of the CSN pulse measured at 3000 kbps ............................ 56
Figure 24. ADC main specifications ............................................................... 58
Figure 25. Sinusoid of 10kHz captured by the ADC and plotted with Octave .. 62
Figure 26. Zoom of the ADC captured data to the sample level ..................... 63
Figure 27. ADC capture of a sawtooth wave .................................................. 63
Figure 28. ADC capture of a triangular wave ................................................ 64
Figure 29. ADC capture of a sinusoid of 80kHz ............................................. 65
Figure 30. ADC capture of a sinusoid of 90kHz ............................................ 65
Figure 31. ADC capture of a sinusoid of 100kHz ......................................... 66
Figure 32. ADC capture of a sinusoid of 110kHz ........................................ 66
Figure 33. ADC capture of a sinusoid of 120kHz ........................................ 67
Figure 34. Libraries tree .............................................................................. 68
Figure 35. Plugins tree ............................................................................... 73
Figure 36. Example of PWM-encoded message .......................................... 83
Figure 37. Test of the OOK modulation in calibration mode (auto-loop) ......... 88
Figure 38. OOK captured data + IIR filtering result + Data decoded ............. 89
Figure 39. OOK encoding for the 'H' character = 72 = 0b01001000 ............... 89
Figure 40. OOK encoding for the 'i' character = 105 = 0b01101001 ............... 89
Figure 41. OOK 10kHz carrier detail ............................................................ 89
Figure 42. Main screen of the ‘plc-cape-lab’ when running with ui-console .... 91
Figure 43. Applications tree ....................................................................... 92
Figure 44. Main screen of the 'plc-cape-autotest' application ....................... 95
Figure 45. Main screen of the 'plc-cape-lab' application (running with 'ui-ncurses') 96
Figure 46. Settings corresponding to the profile 'TX+RX CAL 2kHz' ............. 100
Figure 47. Test kit ....................................................................................... 105
Figure 48. Calibration mode diagram .......................................................... 107
Figure 49. Testing environment on calibration mode.........................................................108
Figure 50. PlcCape-v2 patched with a wire....................................................................108
Figure 51. Testing tools in the desktop PC.......................................................................109
Figure 52. Communication test with plccape-lab in calibration mode...............................110
Figure 53. TX_F_OUT corresponding to 3 repetitions of the message transmitted..............111
Figure 54. Detail of the 110kHz carrier..............................................................................112
Figure 55. Data captured by the ADC of the BBB............................................................112
Figure 56. Data captured by the ADC: zoom to 2000 first samples.................................113
Figure 57. Data captured by the ADC: zoom to 100 first samples....................................113
Figure 58. Testing environment with PlcCape-v1.............................................................114
Figure 59. Communication test with PlcCape-v1 in calibration mode..............................114
Figure 60. TX_F_OUT of 3 messages iterations with PlcCape-v1........................................115
Figure 61. Identification of the characters composing the message.................................115
Figure 62. Detail of the 110kHz carrier generated by the PlcCape-v1...............................116
Figure 63. Testing environment to check the Power Amplifier of the PlcCape-v1..............117
Figure 64. Line-coupling auxiliary circuit.......................................................................117
Figure 65. Configuration settings to generate output through the Power Amplifier..........118
Figure 66. Signals at TX_F_OUT (CH1) and TXRX (CH2), 100ms/div..............................118
Figure 67. Signals at TX_F_OUT (CH1) and TXRX (CH2), 10us/div................................119
Figure 68. Configuration to probe the coupled-line.......................................................119
Figure 69. TX_F_OUT (CH1) and coupled-line (CH2), 100ms/div....................................120
Figure 70. TX_F_OUT (CH1) and coupled-line (CH2), 10us/div......................................120
Figure 71. Testing environment for communication over an unpowered line....................121
Figure 72. Testing tools in the desktop PC......................................................................122
Figure 73. Screen capture of the plc-cape-lab at transmission........................................123
Figure 74. Screen capture of the plc-cape-lab at reception.............................................123
Figure 75. Detail of the carrier signal in the TXRX pin.....................................................124
Figure 76. Probing point for the ADCIN signal...............................................................125
Figure 77. TXRX at PlcCape-v1 (CH1) and ADCIN at PlcCape-v2 (CH2).........................125
Figure 78. TXRX at PlcCape-v1 (CH1) and ADCIN at PlcCape-v2 (CH2), 100ms/div.........126
Figure 79. Correct reception of data when configuring RX_PGA2 to 16V/V (rx2:2)...........127
Figure 80. Detail of the ADCIN magnitude (CH2) when configuring RX_PGA2 to 16V/V....127
Figure 81. Testing environment in a 12VAC line............................................................129
Figure 82. Detail of the wiring in the testing environment................................................130
Figure 83. Capture of the 12VAC power rail (CH2) with no transmission (CH1).............131
Figure 84. Capture of the 12VAC power rail (CH2) with PWM transmission (CH1).........131
Figure 85. Detail of the 12VAC with superimposed data and without.............................132
Figure 86. Detail of the end of a character transmission..................................................132
Figure 87. Detail of the PWM carrier in the 12VAC.......................................................133
Figure 88. Correct reception of data...............................................................................133
Figure 89. TXRX at PlcCape-v1 (CH1) and ADCIN at PlcCape-v2 (CH2), 10ms/div..........134
Figure 90. TXRX at PlcCape-v1 (CH1) and ADCIN at PlcCape-v2 (CH2), 10us/div..........134
Figure 91. Data captured by the ADC plotted with Octave.............................................135
Figure 92. Detail of the data captured by the ADC over 100 samples................................135
Figure 93. Testing environment in a 240VAC line (the mains)........................................136
Figure 94. Typical IP configuration on Windows XP when a BBB is attached to USB......143
Figure 95. Main screen of the Putty application..............................................................144
Figure 96. BBB login message (via a remote console on Putty)........................................145
Figure 97. Version of the Linux Kernel of the BBB.......................................................145
Figure 98. Resource sharing dialog in Windows XP......................................................151
Data communication through power lines using a system based on BeagleBone

Figure 99. Source code tree..............................................................................................................153
Figure 100. Output tree....................................................................................................................154
Figure 101. Basic development environment................................................................................155
Figure 102. Main screen of the Doxywizard application................................................................166
Figure 103. Home page of the Doxygen documentation.................................................................167
Figure 104. Doxygen documentation generated from the comments in the source code.............168
Tables

Table 1. Summary of topics covered per chapter................................................................. 18
Table 2. Document styles.................................................................................................. 19
Table 3. PLC standards..................................................................................................... 22
Table 4. CENELEC bands................................................................................................. 23
Table 5. Benefits and Drawbacks of handling modulations by software.......................... 23
Table 6. Pros and cons of developing a framework in C language over other existing frameworks. 28
Table 7. Linux concepts applied in this project............................................................... 28
Table 8. AFE031 main blocks......................................................................................... 34
Table 9. AFE031 pins and signals................................................................................... 35
Table 10. Main benefits of a modular approach.............................................................. 37
Table 11. Components developed per category.............................................................. 38
Table 12. Basic commands for modules management.................................................... 40
Table 13. Static libraries developed................................................................................. 68
Table 14. Plugin categories............................................................................................. 73
Table 15. Plugins developed within the 'encoder' category............................................. 80
Table 16. Plugins developed within the 'decoder' category............................................ 80
Table 17. Plugins developed within the 'user interface' category................................. 80
Table 18. Type of waves generated by the 'encoder-wave' plugin.................................... 81
Table 19. Applications developed.................................................................................. 93
Table 20. Tests available in the 'plc-cape-autotest' application...................................... 95
Table 21. Main screen panels......................................................................................... 98
Table 22. Main menu options......................................................................................... 98
Table 23. Power supplies used on the tests..................................................................... 106
Table 24. Devices and tools used on the tests................................................................. 106
Table 25. Testing tools summary.................................................................................... 109
Table 26. Pinout of the AC-coupling auxiliary board..................................................... 117
Table 27. Linux commands used on setup...................................................................... 146
Table 28. Pros and cons of the out-of-tree building....................................................... 158
Table 29. Building hierarchy.......................................................................................... 160
Agraïments

Aquest projecte ha estat un gran repte que no hauria pogut assolir sense l'ajut inestimable de totes les persones que m'han envoltat durant aquest llarg període.

Primer vull donar les gràcies al Manel, professor de Sistemes Encastats, pels seus consells i per haver-me donat l'oportunitat de treballar en un projecte en l'àrea que més m’atrau: el desenvolupament software i electrònic en sistemes encastats.

També he d'agrair al Vicente, mestre de laboratori, la seva valuosa ajuda. Ell em va preparar les plaques i em va soldar amb una precisió sorprenent l'integrat principal.

En l’àmbit personal estic molt agraït a tota la meva família i amics pel seu suport, sobretot en els moments més difícils.

I finalment però més important de tot, no trobaré mai forma suficient d'agrair a la meva estimada dona Mireia i als meus estimats petits, Jana i Marc, de 8 i 3 anys, la seva paciència, els seus ànims, la seva ajuda i sobretot el seu sacrifici per haver-me regalat d'una forma tan altruista el preat temps que tant he necessitat per poder dur a terme aquest projecte.

Moltíssimes gràcies!
L'objectiu d'aquest projecte és dissenyar un sistema assequible però altament personalitzable per poder ser utilitzat com a eina de laboratori per experimentar amb la comunicació entre dispositius a través d'una línia de potència i, en particular, a través de la xarxa elèctrica de 240VAC, fent ús de la banda de baixes freqüències autoritzada (al voltant dels 100kHz).

L'eina hauria de servir per a poder avaluar el desplegament de la tecnologia PLC en escenaris diferents i així determinar la seva viabilitat. Per tant, ha de proporcionar una gran varietat d'opcions de configuració per a una màxima versatilitat, com per exemple, diferents esquemes de modulació, velocitats de transmissió configurables, nivells de tensió ajustables, etc.

La transmissió de dades a través de la xarxa elèctrica es pot resumir en aquests punts claus:

- En la part transmissora les dades s’han de modular per a no interferir amb altres senyalitzacions existents, com pot ser la telemetria intel·ligent emprada per les companyies elèctriques. L’ús de la banda disponible està regulat per organitzacions globals i és diferent segons la regió.

- En la part receptora el senyal s’ha de capturar amb una freqüència de mostreig suficient per poder desmodular-lo correctament i recuperar les dades enviades.

- En ambdós costats de la comunicació l’electrònica ha de gestionar l’acoblament entre el senyal modulat que porta les dades i el subministrament d’energia.

Aquest projecte consisteix en el desenvolupament d’un paquet complet de software encarregat de governar un sistema existent basat en la popular plataforma BeagleBone Black amb una placa annexa (cape) que ha estat desenvolupada explícitament per aquest projecte (denominada PlcCape). El cor d’aquesta placa és un bloc analògic que integra un DAC d’alta velocitat que es controla mitjançant un bus SPI. El DAC, juntament amb una etapa de potència integrada, generarà el senyal modulat que es transmetrà per la línia de potència.

Aquest projecte abasta les àrees següents:

- Modificacions del codi font dels drivers SPI i ADC que vénen de sèrie amb la versió del Linux de la BBB, amb l’objectiu d’assolir el rendiment en temps-real que es necessita.

- Creació de llibreries estàtiques reutilitzables que encapsulin l’accés de baix nivell al hardware.

- Disseny d’un sistema basat en extensions (plugins) per permetre una fàcil personalització i expansió del software mitjançant la incorporació d’algoritmes de modulació de tercers.

- Desenvolupament d’una aplicació dedicada per a ser utilitzada com una eina de laboratori que permeti provar amb diferents formes d’ona, freqüències, modulacions, filtres, guanys, etc.

- Execució de proves de tot el sistema en escenaris diferents per confirmar la viabilitat de la solució proposada.
Resumen del Proyecto

El objetivo de este proyecto es diseñar un sistema asequible pero a su vez altamente personalizable para ser usado como una herramienta para experimentar con la comunicación entre dispositivos a través de una línea de potencia y, en particular a través de la red de suministro de 240V AC, usando la banda de bajas frecuencias autorizada (sobre los 100kHz).

La herramienta debería servir para poder evaluar el despliegue de la tecnología PLC en diferentes escenarios y así determinar su viabilidad. Por tanto, debe proporcionar una gran variedad de opciones de configuración para una máxima versatilidad, como por ejemplo, diferentes esquemas de modulación, velocidades de transmisión configurables, niveles de tensión ajustables, etc.

La transmisión de datos por la red eléctrica se puede resumir en estos puntos clave:

- En la parte transmisora los datos se tienen que modular para no interferir con otras señalizaciones existentes, como puede ser la telemetría inteligente usada por las compañías eléctricas. El uso de la banda disponible está regulado por organizaciones globales y es diferente según la región.

- En la parte receptora la señal se tienen que capturar con una frecuencia de muestreo suficiente para ser capaces de demodularla correctamente y recuperar los datos enviados.

- En ambos lados de la comunicación la electrónica debe gestionar el acoplamiento entre la señal modulada que lleva los datos y el suministro de energía.

Este proyecto consiste en el desarrollo de un aplicativo o paquete completo de software encargado de gobernar un sistema existente basado en la popular plataforma BeagleBone Black con una placa anexa (cape) que ha sido desarrollada explícitamente para este proyecto (denominada PlcCape). El corazón de esta placa es un bloque analógico que integra un DAC de alta velocidad que se controla mediante un bus SPI. El DAC, junto con una etapa de potencia integrada, generará la señal modulada que se transmitirá por la línea de potencia.

Este proyecto abarca las siguientes áreas:

- Parcheado del código fuente de los drivers SPI y ADC que vienen de serie con la versión de Linux de la BeagleBone, con el objetivo de alcanzar el rendimiento a tiempo-real que se necesita.

- Creación de librerías estáticas reutilizables que encapsulen el acceso de bajo nivel al hardware.

- Diseño de un sistema basado en extensiones (plugins) para permitir una fácil personalización y expansión del aplicativo mediante la incorporación de algoritmos de modulación de terceros.

- Desarrollo de una aplicación dedicada para ser utilizada como una herramienta de laboratorio que permita probar con diferentes formas de onda, frecuencias, modulaciones, filtros, ganancias, etc.

- Ejecución de pruebas de todo el sistema en escenarios diferentes para confirmar la viabilidad de la solución propuesta.
Abstract

The aim of this project is to design an inexpensive but at the same time highly customizable system to be used as a tool to experiment with the communication between devices through a power line and, in particular, through the 240VAC mains, using the authorized low frequency band (around 100kHz).

The tool should allow the evaluation of the deployment of the PLC technology in different scenarios to determine its feasibility. Therefore, it must provide many customizable options for maximum versatility, like different modulations schemes, configurable transmission rates, adjustable voltage levels, etc.

The transmission of data over the mains must address these main challenges:

- At transmission, it requires the data to be modulated to don’t interfere with other existing signals, as the smart metering used by the energy providers. The usage of the available band is regulated by global organizations and it’s different depending on the region.
- At reception, it requires the signal to be captured at high enough sampling rates to be able to properly demodulate it and recover the transmitted data.
- In both sides, the electronics must be designed to manage the coupling between the modulated signal carrying the data and the AC supplied energy.

This project consists on the development of a full software framework to manage an existing system consisting on the popular BeagleBone Black platform plus an attached cape which has been explicitly developed for this project (called PlcCape). The core of that board is an Analog Front End which integrates a high speed DAC that must be handled by a SPI bus. The DAC in conjunction with an integrated Power Amplifier stage will generate the modulated signal that will be transmitted over the power line.

This project covers these main areas:

- Software patches over the off-the-self SPI and ADC drivers that come with the Linux version in the BeagleBone, with the aim of attaining the real-time performance required.
- Creation of static reusable libraries to encapsulate the low-level access to the underlying hardware.
- Design of a plugin-based system to allow an easy customization and expansion of the framework through the addition of third-party modulation algorithms.
- Development of a dedicated application to be used as a laboratory tool to test with different waves, frequencies, modulations, filters, gains, etc.
- Testing of the whole system in different scenarios to confirm the viability of the solution proposed.
1 Introduction

1.1 Context

Nowadays the communication between different devices is rapidly gaining ground. It is called Machine-To-Machine communication (or M2M\(^{35}\)). Moreover, in the Internet Of Things era (IoT) most machines may provide valuable information that can be easily monitored, processed and made available to users.

A wide variety of technologies are available for data exchange, being the wireless-based systems (WiFi, GSM, 3G, Bluetooth, ZigBee, NFC...) the most popular because the low cost and the comfort. However, there are still some scenarios where wired systems can have some benefits over wireless because can offer wider bandwidths for faster transmission rates, can be more protected against noise, they have less exposure to interferences, can use lower power for longer transmission distances, etc.

This project focuses on a well-known subject, the communication of data reusing the existing power line infrastructure (aka Power Line Communications or PLC\(^{34}\)), but with a specific target: to build an economical but highly configurable framework using the modern available tools at our disposal today.

For this goal, the BeagleBone Black platform\(^{14133}\) (BBB) has been selected as the key core.

The BeagleBone Black is an inexpensive mini computer but still having the most typical blocks of a full-featured desktop PC:

![BeagleBone Black components](image)

*Figure 1. BeagleBone Black components*
1.2 Scenarios of use

This project is focused on the transmission of short packets of sporadic data, which can cover several scenarios.

1.2.1 Industrial scope: data reporting to a central unit

An example of an interesting application of this technology is on industrial machines that need to report periodic status information to central servers:

- When compared with other wired technologies, an advantage of PLC is its low implementation cost because it can reuse the existing infrastructure instead of having to deploy new rewiring (e.g. for a dedicated Ethernet connection). When compared with wireless systems, PLC is not attenuated by thick walls, it doesn’t require repeaters, it is less affected by RF interferences.

- On the negative side, PLC in the industrial scope can be impacted by the high levels of noise generated by the high power machines operating in the factories. Moreover, the PLC devices could be forced to source high levels of current during the transmission of the data to avoid the attenuation due to the low-impedance of the high power machines. So, the viability of a PLC system on the industrial domain will probably depend on each specific situation: type and amount of machinery connected, distance to be covered, noise, quality and layout of the power supply network, reporting schedule (if it is acceptable or not to send data on idle periods), etc.

A typical use case could be a package machine (or similar) in a production line reporting sporadic data. For example, it could send a data packet, after a predetermined number of products or at a predefined time interval, to notify a server (in a different place in the warehouse) about the status of the production, the products discarded, the incidences occurred, etc.

1.2.2 Domestic scope: sensors with medium-power consumption

There are lots of sensors that are more suitable to be powered over the mains than to be battery-based. For example, the presence-detectors, which may send a packet of data to a server when people come into a room. Being usually located in places which are hard to reach, they would pose a maintenance problem if powered by batteries. In such a case, a PLC-based technology could be cheap enough to compete with Wifi or Bluetooth and may avoid the concerns around the deployment of “Access points” (repeaters, IP addresses, RF interferences, etc.).

Concerning costs, a wireless solution is unbeatable when a low power device is involved (a temperature sensor for example) because some cheap battery-based implementation can be used. But if a medium-power device is required (a light bulb or some rolling shutters, for example), the wireless solution needs to interact with the AC mains (through a relay, TRIAC or similar), and therefore it will have a similar cost to the PLC implementation when considering the overall system. In particular, the cost of the communication block for each technology would come from:

- wireless: IC to manage wireless protocol + antenna
- PLC: IC to manage PLC protocol + Analog Front End (AFE) to produce the modulated signal + line-coupling transformer

Other typical scenarios where the PLC technology can compete with Wi-Fi are those that require covering medium or long distances. For instance, to monitor information coming from presence-detectors or weather-stations in a community of neighbors (or in a large garden area): it could be more practical and convenient to
install PLC-based devices than a wireless based solution that will likely require additional repeaters (the transmission power that a wireless device can deliver is limited by law to properly manage the sharing of the publicly available band).

1.3 Objectives

This project has these main goals in mind:

- **create a highly customizable system** able to fit into different scenarios:
  - low-power AC lines (e.g. 12 VAC), high-power AC lines (e.g. 240 VAC), DC power (e.g. 5VDC), unpowered direct wire
  - short and long transmission distances
  - noise, interferences, data coming from other devices in the power supply network
  - negotiable transmission rates for increased robustness and reliability when required
- **propose an inexpensive system** to allow anyone to be introduced in the PLC domain
- **design a comfortable open free source framework** to simplify evolutions, maintenance and experimentation with (the unstoppable DIY trend)
- **open the functionality offered by the device** not only to PLC but to other areas relying on a DAC (Digital-to-Analog Converter) and/or an ADC (Analog-to-Digital Converter)
- **design a reusable and portable framework** to allow adapting the project to other similar platforms (e.g. Raspberry Pi) with no excessive effort
- **learn** about the programming model of a Linux-based device
- **make a clear and detailed documentation** of the whole development process just in case other developers can benefit somehow of the work done for this project

The system will be focused on the transmission of short packets of sporadic data (control commands, presence detection, temperature...). Therefore, the data transmission rate is not a determining factor. Robustness is one of the priorities.

Note that there are several standard protocols referring PLC. The target of this project is not to fit within one of those standards nor to compete with them but to give the user the possibility to define his own, which will probably be much simpler (and affordable) but featured enough for specific scenarios. Anyway, the proposed framework shall allow the development of pieces of custom software (plugins) for higher levels of agreement with existing standards. The hardware should be ready to meet them; the limiting factor could be the computational effort involved.

There are also many commercial products concerning PLC. The target of this project is not to vie with them but to provide a versatile tool for students or hobbyists willing to be introduced to this world in an open, and affordable way.
1.4 Challenges

Due to the chosen implementation, this project involves these main challenges:

- Understand how the BeagleBone Black works and the features offered by the core microcontroller, the SITARA AM3358 SoC
- Learn how to develop device drivers in Linux and tune/patch the ones for SPI and ADC to optimize performance for the PlcCape specific design
- Learn how to develop different types of components (libraries, plugins and applications) in Linux
- Learn how to use third party libraries, as ncurses for richer User Interfaces
- Use, as much as possible, free software tools for the development (gcc, make, Octave, Eclipse…) and for the documentation (LibreOffice, Doxygen)
- Make the application generic enough to be used for other purposes

1.5 Document structure

This document covers the following areas in five main chapters:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Topics covered</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction</strong></td>
<td>Basic notions, targets, challenges and a summary of the main topics covered by this project</td>
</tr>
<tr>
<td>Page 15</td>
<td></td>
</tr>
<tr>
<td><strong>Background</strong></td>
<td>Basic notions on Power Line Communications (PLC)</td>
</tr>
<tr>
<td>Page 20</td>
<td>A quick overview of some existing solutions</td>
</tr>
<tr>
<td></td>
<td>Some notions about the BeagleBone Black platform</td>
</tr>
<tr>
<td></td>
<td>Some notes about the Linux Operating System</td>
</tr>
<tr>
<td></td>
<td>Main features of the Analog Front End chip used, the AFE031</td>
</tr>
<tr>
<td></td>
<td>A brief explanation about the PlcCape boards used</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>Software development principles and strategy</td>
</tr>
<tr>
<td>Page 36</td>
<td>Description of the patches done on the SPI and ADC drivers</td>
</tr>
<tr>
<td></td>
<td>Summary of the components provided: libraries, plugins and applications</td>
</tr>
<tr>
<td></td>
<td>A detailed description of the plc-cape-lab application which is the main tool developed to interact with the final assembly (BBB + PlcCape)</td>
</tr>
<tr>
<td></td>
<td>Some notes about Octave use</td>
</tr>
<tr>
<td><strong>Analysis &amp; results</strong></td>
<td>Detailed description of a series of communication experiments, indicating the purpose, the testing environment, the results and the conclusions obtained for each one</td>
</tr>
<tr>
<td>Page 105</td>
<td></td>
</tr>
<tr>
<td><strong>Conclusions</strong></td>
<td>Final conclusions about the whole system feasibility after the tests</td>
</tr>
<tr>
<td>Page 138</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Summary of topics covered per chapter
To simplify the reading of this documentation I’ve focused on the specific details of the project. I’ve also mentioned some basic background notions but tried to avoid reproducing much information easily available elsewhere (I’ve indicated the URL links instead).

1.6 Document styles
For a simpler reading, the following styles have been used depending on the category of the documented entity:

<table>
<thead>
<tr>
<th>Category</th>
<th>Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td><strong>main.c</strong></td>
</tr>
<tr>
<td></td>
<td><code>#include &lt;stdio.h&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>int main(int argc, char *argv[])</code></td>
</tr>
<tr>
<td></td>
<td>`{</td>
</tr>
<tr>
<td></td>
<td>// Just a comment</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;Hello world\n&quot;);</td>
</tr>
<tr>
<td></td>
<td>return 0;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Other file content</td>
<td><strong>test.sh</strong></td>
</tr>
<tr>
<td></td>
<td><code>#!/bin/bash</code></td>
</tr>
<tr>
<td></td>
<td><code>@echo Hello World</code></td>
</tr>
<tr>
<td>Linux console commands</td>
<td><code># echo Hello world</code></td>
</tr>
<tr>
<td></td>
<td>Note: Linux command lines starting with ‘#’ indicate execution from the root user, the default way in the BBB; ‘$’ means execution from a normal user, the default and recommended way for commands executed in a desktop PC with Linux (whenever possible)</td>
</tr>
<tr>
<td>Windows XP console window</td>
<td><code>C:\&gt;echo Hello World</code></td>
</tr>
<tr>
<td>Tips</td>
<td>Tip: This might be something helpful…</td>
</tr>
<tr>
<td>Quotes, citations, fragments of existing documentation</td>
<td><code>[…]</code></td>
</tr>
<tr>
<td></td>
<td>This is a fragment of text from somewhere</td>
</tr>
<tr>
<td></td>
<td><code>[…]</code></td>
</tr>
<tr>
<td>Cross-links to other parts in the document</td>
<td>This is a link to this Style table</td>
</tr>
<tr>
<td>References to Bibliography</td>
<td>This is a reference to the first bibliographic reference[^1]</td>
</tr>
</tbody>
</table>

Table 2. Document styles
2 Background

2.1 PLC

2.1.1 Basic notions

Power Line communications is an old well-known technology used to transfer data reusing the same lines that are used to power the devices.

Contrary to the wireless technologies, which usually fit to well-known standards (WiFi, BT, ZigBee…), PLC is not so standardized. That’s because it’s difficult to have a standard that efficiently covers all the needs of all the different possible PLC scenarios, like:

- telemetry (smart-metering) for energy providers
- telemetry on industrial specific areas (e.g. in solar plants)
- monitoring of equipment in factories
- home automation where devices are controlled/supervised from a central unit (lights, shutters, alarms, weather sensors…)
- LAN over the power supply network at home
- Internet brought to clients by energy providers

Each application has its own particularities and challenges. Defining a protocol or a hardware that works well in all the situations could unnecessarily increase the cost of some basic solutions that may satisfy specific scenarios.

These are examples of different requirements according to the target field:

- in the industrial scope we can have hard and noisy environments because the action of the high power machines (motors, brute switching of high-loads, etc.)
- in the home automation area we can sporadically have some noise and interferences coming from some high-power appliances (hairdryers, microwaves, etc)

Depending on the environment and the requirements, the way to fight against the typical troubles. For example:

- A custom transmission protocol could wait the source of the trouble (e.g. the hairdryer) to be switched off (monitoring the line). A basic error detection system may be enough to reject the invalid data received and wait for the next packet
- Other applications may require FEC (Forward Error Correction) by the addition of an ECC (Error Correcting Code) prior to the transmission in order the receiver to be able to correct most of the wrong data captured
- And other applications can require requesting the transmitter to resend the packet if an error is detected. This would involve more complexity in both devices, the transmitter and the receiver, because they would have to implement both communication directions (TX and RX)
Depending on the power supply network we can also receive data from neighbors. If confidential information has to be managed, some kind of encryption should be considered.

Note that in most cases we will need to deal with low impedance loads which impose the transmitters to source relatively high currents to avoid signal attenuation.

### 2.1.2 Standard protocols

As already commented in 1.3 Objectives, the target of this project is not to develop good modulations algorithms nor complex protocol layers. Anyway some basic information about the existing standards is provided. For that and just for quick reference, I’ve reproduced the main points of the summarized information given in the always helpful Wikipedia:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Wikipedia Link</th>
</tr>
</thead>
</table>

**HomePlug**

HomePlug is the family name for various power line communications specifications under the HomePlug designation, with each offering unique performance capabilities and coexistence or compatibility with other HomePlug specifications.

Some HomePlug specifications target broadband applications such as in-home distribution of low data rate IPTV, gaming, and Internet content, while others focus on low-power, low throughput, and extended operating temperatures for applications such as smart power meters and in-home communications between electric systems and appliances.

**PRIME**

PRIME is a specification for narrow band powerline communication. Power-line communication uses power lines as transmission media.

PRIME is an acronym for “PoweRline Intelligent Metering Evolution”.

PRIME was conceived in 2007.

The PRIME physical layer is based on OFDM (Orthogonal Frequency Division Multiplexing) and Differential Phase Shift Keying (BPSK, DQPSK and D8PSK) as carrier modulation. To address averse power line channel properties, robustness mechanism convolutional encoding (optional), scrambling and interleaving are used. PRIME Specification v1.4 also introduces repetition coding [5] as additional robustness mechanism.

Originally, PRIME uses carrier frequencies (42 – 89 kHz) within the CENELEC A band and offers raw data rates between 5.4 kbit/s (Robust mode: DBPSK with convolutional encoding and repetition code) and 128.6 kbit/s (D8PSK). Since specification version 1.4, more frequency bands were introduced to utilize the higher frequencies (up to 471 kHz) in ARIB and FCC bands. Using the full FCC band, raw data rates are eight times as high as in CENELEC A band.

**G3-PLC**

G3-PLC is the low layer protocol to enable large scale infrastructure on the electrical grid. G3-PLC may operate on CENELEC A band (35 kHz to 91 kHz) or CENELEC B band (98 kHz to 122 kHz) in Europe, on ARIB band (155 kHz to 403 kHz) in Japan and on FCC (155 kHz to 487 kHz) for the US and the rest of the world. The technology used is OFDM sampled at 400 kHz with adaptative modulation and tone mapping. Error detection and correction is made by both a convolutional code and Reed-Solomon error correction. The required media access control is taken from IEEE 802.15.4, a radio standard.
In December 2011, G3 PLC technology was recognised as an international standard at ITU in Geneva where it is referenced as G.9903.[26][27] Narrowband orthogonal frequency division multiplexing power line communication transceivers for G3-PLC networks.

With the use of IPv6, both PRIME and G3 enable communication between meters, grid actuators as well as smart objects.

---

**X10**

[X10](https://en.wikipedia.org/wiki/X10_(industry_standard))

X10 is a protocol for communication among electronic devices used for home automation (domotics). It primarily uses power line wiring for signaling and control, where the signals involve brief radio frequency bursts representing digital information. A wireless radio based protocol transport is also defined.

---

**UPB**

[UPB](https://en.wikipedia.org/wiki/Universal_powerline_bus)

Universal powerline bus (UPB) is a proprietary software protocol for power line communication between devices used for home automation. Household electrical wiring is used to send digital data between UPB devices via pulse-position modulation.[1]

Communication can be peer to peer, with no central controller necessary.[2]

UPB addressing allows 250 devices per house and 250 houses per transformer, and switches can co-exist with other powerline carrier systems within the same house.[3]

While UPB may be more efficient than X10, it has far fewer products available on the market.[4]

---

**LonTalk**

[LonTalk](https://en.wikipedia.org/wiki/LonTalk)

LonTalk is a protocol optimized for control. Originally developed by Echelon Corporation for networking devices over media such as twisted pair, powerlines, fiber optics, and RF. It is popular for the automation of various functions in industrial control, home automation, transportation, and buildings systems such as lighting and HVAC; see Intelligent building, the protocol has now been adopted as an open international control networking standard in the ISO/IEC 14908[1] family of standards. Published through ISO/IEC JTC 1/SC 6, this standard specifies a multi-purpose control network protocol stack optimized for smart grid, smart building, and smart city applications.

---

*Table 3. PLC standards*
2.1.3 CENELEC Bands

CENELEC is the European Committee for Electrotechnical Standardization\textsuperscript{[38][39]}. This organization has regulated the way the low-frequency band can be used in power supply networks in the European Region:

<table>
<thead>
<tr>
<th>CENELEC Frequency Band</th>
<th>Frequency Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Band A</td>
<td>9 kHz to 95 kHz</td>
<td>Reserved for energy providers, mainly for telemetry</td>
</tr>
<tr>
<td>Band B</td>
<td>95 kHz to 125 kHz</td>
<td>Available for customers use. No protocol defined</td>
</tr>
<tr>
<td>Band C</td>
<td>125 kHz to 140 kHz</td>
<td>Available for customers use. Protocols must implement CSMA</td>
</tr>
<tr>
<td>Band D</td>
<td>140 kHz to 148.5 kHz</td>
<td>Available for customers use. No protocol defined</td>
</tr>
</tbody>
</table>

Table 4. CENELEC bands

For this project the CENELEC B band has been used.

2.1.4 Free open software solutions

One of the principles of this project is to do as much as possible by software in order any modulation algorithm can be added just by updating the source code, without requiring changes on the electronics. This approach has also some drawbacks:

<table>
<thead>
<tr>
<th>Benefits</th>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>New modulation schemes can be added just by updating the software</td>
</tr>
<tr>
<td>Adaptability</td>
<td>We have a mini laboratory tool that may help us to evaluate the feasibility of the PLC approach in different scenarios (if transmission speed is not important we can use slow encoding algorithms but with improved robustness and with reinforces tolerance to specific error conditions)</td>
</tr>
<tr>
<td>Reusability</td>
<td>By having full control of the DAC, we can afford applications other than just PLC, as we’ll see</td>
</tr>
<tr>
<td>Cost effectiveness</td>
<td>We can implement simple modulation schemes that can help cheap devices to report data with low-cost electronics</td>
</tr>
</tbody>
</table>

Table 5. Benefits and Drawbacks of handling modulations by software

That’s not a new idea. In fact the same concept is already put in place in some radio frequency systems. It is called Software Defined Radio\textsuperscript{[40]} (SDR). The main difference here is that, in our case, we are focused on the more affordable low-frequency band of the PLC technology. Note that experimenting with SDR usually requires expensive hardware due to the high frequencies involved.
An example of a free open software radio ecosystem is the GNU Radio\textsuperscript{41}. It is very powerful with lots of modulators, demodulators, filters, generators, already implemented.

It is possible that some algorithms implemented for the GNU Radio (in C++) can be adapted to our project in C (it’s something to evaluate in future).

In any case, other open free software packages being specialized in signal processing, are also available, as the IT\textsuperscript{+}\textsuperscript{32}.

As an example of the source code availability, find following some links to some information about the OFDM implementation in both ecosystems:

- \url{http://gnuradio.org/doc/doxygen/page_ofdm.html}
- \url{http://itpp.sourceforge.net/devel/ofdm_8h.html}

Finally, many contributions from the Community can also be found in GitHub or similar hosting services.

For example this project implements a QPSK modulator/demodulator in two versions, fixed point and floating-point:

- \url{https://github.com/gvaf/QPSK-Modem}

### 2.1.5 Commercial development kits

There are many proprietary products concerning PLC. For example, find below some developer kits for PLC from Texas Instruments (the manufacturer of the AFE031 chip used in this project):

<table>
<thead>
<tr>
<th>C2000 Power Line Modem Developer’s Kit - TMDSPLCKIT-V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>\url{<a href="http://www.ti.com/tool/tmdsplckit-v3%7D">http://www.ti.com/tool/tmdsplckit-v3}</a></td>
</tr>
</tbody>
</table>

The C2000 Power Line Modem Developer’s Kit enables easy development of software based Power Line Communication (PLC) modems. The kit includes two PLC modems based on the C2000 TMS320F28069 controlCARD and TI's advanced PLC analog front end AFE031. The included PLC SUITE software supports several communication techniques, including OFDM (PRIME/G3-PLC and 1901.2) and is SFSK-capable. The kit includes onboard USB JTAG emulation and Code Composer Studio.

PRICE = $599.00(USD)

<table>
<thead>
<tr>
<th>Power Line Communications (PLC) Lite™ Reference Design for Industrial Applications - TIDM-INDUSTRIAL-PLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>\url{<a href="http://www.ti.com/tool/tidm-industrial-plc%7D">http://www.ti.com/tool/tidm-industrial-plc}</a></td>
</tr>
</tbody>
</table>

This TI implementation of Power Line Communications (PLC) is a new option for adding communications to industrial equipment such as solar arrays. PLC Lite™ offers higher data throughput than earlier forms of PLC and greater robustness for transmission across noisy electric lines by using OFDM PHY technology. This design is based upon a C2000 controlCARD and an analog front end for PLC. The C2000 MCU executes the entire PLC Lite software stack. TIDM-INDUSTRIAL-PLC achieves up to 21 kbps data rates across AC or DC power lines compared to traditional SFSK PLC solutions (2-5 kbps) all while decreasing the overall BOM cost of a typical PLC modem.

PRICE = TMDSPLCKIT-V3 ($599) + TMDSCNCD28035 ($59)
Data communication through power lines using a system based on BeagleBone

Power Line Communications Kit for CENELEC Frequency Band - TMDSPLCKITV4-CEN
http://www.ti.com/tool/tmdsplckitv4-cen#0

This PLC kit is designed for the development of narrowband PLC systems across the CENELEC frequency bands (35 kHz-121kHz). The kit includes a pair of PLC modems each using Texas Instruments’ PLC docking station and SOMPLC-F28PLC83 System on Module (SOM). The SOMPLC-F28PLC83 SOM is a single-board design for PLC in the CENELEC A frequency band. This single hardware design supports several popular PLC industry standards including G3 and PRIME. TI develops and provides the complete PLC software stacks along with the SoM. The PLC docking station includes the AC mains line coupling circuitry and system power management.

PRICE = $499.00(USD)

These kits are usually sold along with proprietary software in the form of libraries or firmware (binaries), as it is mentioned in this page:

http://processors.wiki.ti.com/index.php/Smart_Grid_PLC_FAQ

…

Is the PLC software open source?
The software is an "open API" with libraries. We provide libraries and documentation on the contents of the libraries and how to use them.

[…]

2.2 BeagleBone black

The main features of the BeagleBone black are conveniently summarized in this picture\(^1\):

![BeagleBone Black diagram](http://cdn.arstechnica.net/wp-content/uploads/2013/04/beaglebone-specs.png)

**Figure 2. BBB main specifications**

These are the main characteristics that makes the BBB an ideal candidate for this project:

- low-cost full featured platform
- open source hardware
- popular and full-featured ARM Cortex-A8 processor from Texas Instruments
- popular Debian Wheezy 7.8 distribution installed on the off-the-shelf device
- small device (the size of a credit card)
- wide support from the Community

There are also lots of good sources of official documentation. Find following some literature that I consider has been specially relevant for the development of this project:


Introductory user’s manual explaining the main features, describing how to connect the BBB to a PC, etc.

---

1 Figure obtained from [http://cdn.arstechnica.net/wp-content/uploads/2013/04/beaglebone-specs.png](http://cdn.arstechnica.net/wp-content/uploads/2013/04/beaglebone-specs.png)
Technical Reference Manual for the AM335x Sitara™ Processors. It describes in detail the registers of the processor to fully configure the SPI block, the ADC, the DMA subsystem, etc. It’s compulsory reading for developing drivers

Some timing and electronics characteristics of the AM335x Sitara™ Processors.

More references are available in the Bibliography.

2.3 Linux

The Linux Operating System\(^\text{[15]}\) (in conjunction with the Debian GNU distribution) meets the targets for this project because:

- It’s free and open source software with code easily available in the web\(^\text{[16]}\)
- It has been ported to many architectures (x86, ARM, MIPS, SPARC, m68k…)
- It’s present in many embedded platforms. Thus, we could be able to easily switch to other more powerful platforms (if required) with minor software changes
- Many open source packages are available from third-party contributors. For instance, the ncurses package, used in this project to provide a user-friendly User Interface
- It’s fully configurable to take advantage of the underlying hardware
- It’s an Operating System that is gaining ground each day (the irruption of Android, which is based on the Linux kernel, is an example)
- Wide support from the Community

The software framework for this project has been developed in C. There are other simpler frameworks that can be used to program the BBB like the BoneScript (which is an easy to learn JavaScript library encapsulating main BBB functionality) but the use of the C language has important benefits in this project:

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing speed</td>
<td>In the C language we are close to the hardware, with minimum overhead. So we can take the maximum advantage of the underlying electronics. Other alternatives as Java are good for portability (because the same “binaries” can be executed on different platforms) but at expenses of a speed reduction</td>
</tr>
<tr>
<td>Customization possibilities</td>
<td>Being close to the hardware, we have many alternatives to solve a specific problem</td>
</tr>
<tr>
<td>Universality</td>
<td>A lot of tools and others languages are based on the C language (Octave, Matlab, Wire, …). Migrating existing developments to a C language jargon shouldn’t be a tough task in most cases (e.g. a modulation algorithm provided in Octave)</td>
</tr>
</tbody>
</table>
In C language it’s very easy to cause a *Segmentation Fault* which crashes the application. And it is even worse when developing drivers because the errors there can hang the whole operating system, since running on privileged mode.

Due to the direct access to the memory, indirect errors can provoke strange side-effects that sometimes give few clues about their root cause.

Table 6. Pros and cons of developing a framework in C language over other existing frameworks

For this project I have taken advantage of many features provided by the Linux Operating System, like:

<table>
<thead>
<tr>
<th>Area</th>
<th>Links for detailed information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device drivers</td>
<td><a href="http://elinux.org/Device_drivers">http://elinux.org/Device_drivers</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://lwn.net/Kernel/LDD3/">http://lwn.net/Kernel/LDD3/</a></td>
</tr>
<tr>
<td></td>
<td><a href="https://www.kernel.org/doc/Documentation/kbuild/modules.txt">https://www.kernel.org/doc/Documentation/kbuild/modules.txt</a></td>
</tr>
<tr>
<td>Device Tree Overlays</td>
<td><a href="https://learn.adafruit.com/introduction-to-the-beaglebone-black-device-tree/compiling-an-overlay">https://learn.adafruit.com/introduction-to-the-beaglebone-black-device-tree/compiling-an-overlay</a></td>
</tr>
<tr>
<td>Timers</td>
<td><a href="http://man7.org/linux/man-pages/man2/timer_create.2.html">http://man7.org/linux/man-pages/man2/timer_create.2.html</a></td>
</tr>
<tr>
<td>Dynamic loadable libraries (for plugins)</td>
<td><a href="http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html">http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html">http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://linux.die.net/man/3/dlopen">http://linux.die.net/man/3/dlopen</a></td>
</tr>
<tr>
<td>UI with <em>ncurses</em></td>
<td><a href="https://www.gnu.org/software/ncurses/">https://www.gnu.org/software/ncurses/</a></td>
</tr>
</tbody>
</table>

Table 7. Linux concepts applied in this project

Other sources of documentation are mentioned in the Bibliography.
2.4 PlcCape board

2.4.1 Overview

The development of the software will be based on a system composed of a BBB + a PlcCape board:

![PlcCape board attached to a BeagleBone Black](image)

The core of the PlcCape is the AFE031 chip located in the center of the board (AFE stands for Analog Front End). The main block of this integrated circuit is a **high speed DAC allowing the generation of any wave** within the covered frequency range. It also provides some other interesting features oriented to PLC:

- an integrated Power Amplifier (PA) to deliver power enough with minimum additional components
- several programmable gain amplifiers in the TX and RX chains
- integrated filtering for the band of interest (CENELEC A or CENELEC B/C/D)
- power supply phase detection for synchronization purposes

The PlcCape also integrates the line-coupling block to inject/extract the signal to/from an AC line:

![Blocks diagram of the assembly BBB (blue) + PlcCape (green)](image)

There are two board versions of the PlcCape designed for two different purposes:

- **PlcCape-v1**: for early development and testing. The AC-coupling is done through an auxiliary board
- **PlcCape-v2**: it’s a final candidate version with all the required components integrated
2.4.2 PlcCape-v1

The PlcCape version 1 was designed for the initial development stage to have a first contact with the AFE031. It provides a series of easily accessible probing points and some sockets to comfortably test with different resistors and capacitors. It has this look:

![PlcCape-v1 mounted board](image1)

These are the available probing points which will be described and referred to in different parts of this documentation:

![PlcCape-v1 probing points](image2)

There are some probing points (in a green background color) which are easily accessible (a wire can be easily plugged) while others (in a yellow background color) are SMD-based requiring the oscilloscope probe to be manually handled during the test, with care not to touch the surrounding area and cause a shortcut.
2.4.3 PlcCape-v2

The PlcCape version 2 was designed as a final BeagleBone cape with all the required components integrated. It has this look:

*Figure 7. PlcCape-v2 mounted board*

These are the available probing points:

*Figure 8. PlcCape-v2 probing points*

In this board the probing points are mainly SMD-based except for PA_F_OUT which is an auxiliary output that can be used for other purposes than the original Power Line Communications, as an audio player for example. It is the output of the Power Amplifier of the AFE031. Thus, it is like a high-speed DAC able to deliver 1.5A (if proper dissipation) and to generate a voltage swing of about 24Vpp (26V is the absolute max), all limited to 15W max power dissipation.
2.4.4 Software management

The PlcCape has the following connection points with the BBB that can be controlled by the software:

- SPI bus to send both, either commands or DAC samples
- Several GPIOs to flash three programmable LEDs in the PlcCape
- Several GPIOs to detect fault conditions from the PlcCape board
- Input data to be captured by the ADC at maximum speed (200 ksps by specifications)

Through SPI commands we can manage:

- the AFE031 blocks to enable (DAC, TX chain, RX chain, filters, PA, etc.)
- the gains of the involved programmable amplifiers
- the frequency band to be used (CENELEC A vs B/C/D)
- the calibration modes, a function of the AFE031 that allows some internal routing among blocks
- the source of a problem that caused a faulty condition

The DAC of the AFE031 doesn’t produce predefined waves (as other chips do). This is one of the main missions of the software developed in this project.
2.5 AFE031

The core of the PlcCape is the AFE031 integrated circuit which is composed of the following blocks:

A brief description of the main blocks and involved signals follows in the next pages.

For more detailed information consult the AFE031 datasheet.

The following two tables describe the blocks that compose the AFE031 and the related pins and signals.
### Block Description

**SPI**
- Receives and routes two type of data:
  - 8-bit based commands: to enable/disable the different blocks and configure them
  - 10-bits based samples: real-time data to be sent to the DAC

**DAC**
- Generates the analog voltages (from 0 to 3.3V) corresponding to the digital samples coming from the SPI block

**TxPGA**
- Transmission (Tx) Programmable Gain Amplifier (PGA) for better output control. It allows four programmable gains: 0.250, 0.500, 0.707, 1.000 (in V/V units)

**Tx Filter**
- Transmission Low Pass Filter (LPF). It filters the input signal at Tx_F_IN1 (usually the output from the TxPGA) with a configurable cut-off frequency: 95kHz for CENELEC A, 145kHz for CENELEC B/C/D.
  - The output goes from 0 to VDD. The maximum continuous DC current for both sourcing and sinking is about 25mA.

**Power Amplifier**
- Amplifies the voltage of the input signal (limited to the voltage supply VDD <= 5.5V) to the higher levels of another external power supply (PA_VS from +7V to +24V).
  - The PA offers a nominal gain of 6.5 V/V.
  - The PA is able to deliver 1.5A.
  - This project is designed for a +15V power supply

**RxPGA_1**
- First Programmable Gain Amplifier in the reception chain, before the integrated low-pass filter.
  - If provides four programmable gains: 0.25, 0.5, 1, 2 (in V/V units)

**Rx Filter**
- It’s a filter in the reception chain intended to remove the unwanted signals out of the CENELEC band. The cut-off frequency is affected by the parametrization of the CENELEC mode (A vs B/C/D)

**RxPGA_2**
- Second Programmable Gain Amplifier in the reception chain, after the low-pass filter.
  - It provides four programmable gains: 1, 4, 16, 64 (in V/V units)

**Two-Wire Rx/Tx**
- This block can be used for simple implementations of communications that use amplitude shift keying (ASK) with on-off keying (OOK) modulation.
  - It is not used in this project because this functionality is already covered by the DAC-based implementation.

**Zero Crossing**
- Zero Crossing module to easily detect the AC phase edges and thus allow a better use of the channel:
  - some protocols can relay on it for synchronization purposes
  - the data can be transmitted near the zero-pass-region where the consumption of connected machines or appliances may be lower

---

*Table 8. AFE031 main blocks*
<table>
<thead>
<tr>
<th>Block</th>
<th>Pin name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI</td>
<td>SCLK</td>
<td>Serial clock indicating when DI and DO are valid</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>Serial data sent from the BBB to the AFE031 (commands or DAC samples)</td>
</tr>
<tr>
<td></td>
<td>DO</td>
<td>Serial data sent back from the AFE031 to the BBB in response to BBB commands</td>
</tr>
<tr>
<td></td>
<td>CSN</td>
<td>Chip Select (Negated) used by the AFE031 as a strobe signal indicating the end of a 10-bits DAC sample</td>
</tr>
<tr>
<td>TX</td>
<td>TX_PGA_OUT</td>
<td>Analog samples after the SPI to DAC conversion + TxPGA attenuation. This signal has a “step-shape” because the constant level between DAC samples</td>
</tr>
<tr>
<td></td>
<td>Tx_F_OUT</td>
<td>In this project the TX_PGA_OUT is directly routed to Tx_F_IN1. This leads Tx_F_OUT to be the result of applying the low-pass TxFilter to the TX_PGA_OUT signal. The effect of this is a smoother signal removing part of the DAC-quantification shape</td>
</tr>
<tr>
<td>PA</td>
<td>PA_IN</td>
<td>It is the input to the Power Amplifier. The PA requires the signal to don’t have a DC component. To achieve this the TX_F_OUT traverses a decoupling capacitor</td>
</tr>
<tr>
<td></td>
<td>PA_OUT</td>
<td>This is the output of the Power Amplifier which magnifies the input signal using the PA_VS power supply</td>
</tr>
<tr>
<td>RX</td>
<td>Rx_PGA1_IN</td>
<td>Input signal coming from the AC decoupling stage + the external pass-band filter. A decoupling capacitor is also required to allow the AFE031 biasing the level of the received AC signal</td>
</tr>
<tr>
<td></td>
<td>Rx_PGA1_OUT = Rx_F_IN</td>
<td>Output signal of the RxPGA_1 block. It’s the RX_PGA1_IN after some programmable amplification factor. This signal is directly connected to the input of the RxFilter (RX_F_IN)</td>
</tr>
<tr>
<td></td>
<td>Rx_F_OUT</td>
<td>Rx_F_IN filtered signal in either CENELEC A or B/C/D band</td>
</tr>
<tr>
<td></td>
<td>Rx_PGA2_IN</td>
<td>Input to the RxPGA_2 block. It’s the RX_F_OUT after a decoupling capacitor to allow the AFE031 for proper internal biasing of the AC signal</td>
</tr>
<tr>
<td></td>
<td>Rx_PGA2_OUT</td>
<td>Output of the RxPGA_2 block after the last programmable amplification</td>
</tr>
<tr>
<td></td>
<td>ADCIN (not in the AFE but in the BBB)</td>
<td>Signal sent to the ADC of the BBB. It comes from Rx_PGA2_OUT after a voltage resistor divider to adjust the levels (from 0 to 3.3V) to the range of the ADC in the BBB (absolute max value 1.8V)</td>
</tr>
<tr>
<td>ZC</td>
<td>ZC_IN1</td>
<td>Low-level “clone” of the 240 VAC line for Zero Cross detection. The 240 VAC can be converted to a low-level clone through high-impedance resistors or through an optocoupler</td>
</tr>
<tr>
<td></td>
<td>ZC_OUT1</td>
<td>Digital signal indicating the zero-crossing points</td>
</tr>
</tbody>
</table>

*Table 9. AFE031 pins and signals*
3 Implementation

3.1 Overview

The main target of this project is to develop a versatile system to send data over Power Lines based on an existing assembly composed of a BBB + a PlcCape board, and evaluate the feasibility of the solution in different scenarios.

For that purpose, one of the main missions of the software is to convert data to a modulated signal within the allowed CENELEC band.

The main requirements for this project are quite common:

- high speed SPI communication to send real time data to a DAC
- high-speed ADC capturing

This leads to consider developing reusable code that may be adapted to other similar projects.

Note that this project is not focused on providing good modulation/demodulation algorithms. There already are existing packages doing so in efficient ways (as already discussed in 2.1.4 Free open software solutions). So, very basic modulators are going to be developed here, just to cover the feasibility analysis.

As the software is the key-point of this project, next chapters will focus on a detailed explanation about.

3.2 Software development strategy

To manage the BBB + PlcCape assembly, a dedicated software has been developed consisting on:

- Modifications of the existing drivers:
  - in the SPI driver:
    - to adapt it to the custom behavior of the AFE031 DAC which uses the Chip Select line as a clock synchronization mechanism
    - to use a DMA ping-pong buffer for real-time continuous data transmission to the DAC of the AFE031, leaving the CPU free for other tasks in the background
  - in the ADC driver, for faster data capturing

- Development of static libraries (*.a) for a convenient encapsulation of the reusable functionality

- Development of dynamic libraries (*.so) as a convenient way of developing extensions (plugins) with a short-learning curve

- Creation of dedicated applications to explore all the possibilities of the AFE031 chip

The whole ensemble has been called the “plc-cape framework”.

The plc-cape framework is customized for the PlcCape board but the idea (the tree architecture, the makefile strategy, etc) can be reused as a template or skeleton to build other similar projects.
For portability and maintenance reasons, these principles have been followed:

- The framework has been developed in ANSI-C (and not C++) for maximum portability: even though C++ compilers are frequent, C compilers are present in almost any platform (even for low-cost microcontrollers or DSPs). Moreover, note that being C++ a superset of C (C++ compilers can compile C code), the chances of having a C compiler for a specific platform is always higher.

- The dependencies with third party packages have been minimized for portability reasons too. When a third party library is used (e.g. the `ncurses` package), it has been encapsulated (within a plugin for example) to simplify the task of migrating the framework to other platforms. Thus, if a dependency doesn’t exist on a particular system (a common scenario when third-party components are only offered in binary format), only the affected plugin need to be adapted; the remaining parts of the framework can still work.

   Logically, the use of existing libraries for complex tasks (e.g. complex modulators plugins) is encouraged to avoid spending time reinventing the wheel.

For this project only one third-party library (not included in the standard Debian Wheezy distribution coming with the BBB) has been used, the `ncurses` package, to provide a rich-enough user interface for comfortable testing of the PlCCape board. Information about how to install this package can be found in Annex III. Getting started with the development environment.

The framework is based on the principles of **modularity** and **encapsulation**, implicitly imposed by the usage of libraries and plugins. This approach has the following benefits:

<table>
<thead>
<tr>
<th>Pros</th>
<th>Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customization</td>
<td>The source code is more readable because isolated per functional/logical area</td>
</tr>
<tr>
<td></td>
<td>The origin of the bugs can be quickly located</td>
</tr>
<tr>
<td></td>
<td>The decomposition of the framework in isolated building units allows for faster compilation times because only the affected component needs to be rebuilt (interesting when compiling on the BBB which has a medium speed processor)</td>
</tr>
</tbody>
</table>

**Table 10. Main benefits of a modular approach**

The source code has been documented following **Doxygen**\(^\text{\textsuperscript{48}}\) patterns. This provides:

- a good way to quickly produce an **always updated and synchronized documentation** of the source code

- a **rich and practical documentation** with very helpful and comfortable cross-referenced links between the source-code and other organized documentation

More information about Doxygen is available in the section **8.12 Doxygen**.
Despite the considerable programming work done, the software is still in an early development stage: on one hand the current foundations of the framework are quite consolidated but on the other hand the specific implementation of the components needs to be optimized (for better performance) and refactored (for better modularity and maintenance).

A brief explanation of the developed components and the key points of the software are exposed in the next sections with the aim of providing enough information to allow anyone that can be interested in, to be introduced to the framework with the minimum effort and with a short learning-curve.

Anyone who wants to play with the source-code should read first these annexes:
- Annex III. Getting started with the development environment: to quickly setup the development environment and be able to compile the plc-cape framework in a BBB platform
- Annex IV. Software design principles: to know some details about the source code implementation
- Annex V. Source code files: to get the sources

### 3.3 Components developed

Next sections will focus on the components developed within each category:

<table>
<thead>
<tr>
<th>Category</th>
<th>Components developed</th>
</tr>
</thead>
<tbody>
<tr>
<td>drivers</td>
<td>spi, adc</td>
</tr>
<tr>
<td>static libraries</td>
<td>libplc-tools, libplc-gpio, libplc-adc, libplc-cape</td>
</tr>
</tbody>
</table>
| plug-ins       | encoders: encoder-wave, encoder-pwm, encoderOOK
                | decoders: decoder-pwm, decoder-ook
                | user interfaces: uin-curses, ui-console               |
| applications   | plc-cape-demo-lib-usage, plc-ape-autotest, plc-cape-lab |

*Table 11. Components developed per category*

For each category some brief notes will be exposed first. Then a brief description of each developed component will be given (in tabular format) as a quick way to easily find the purpose and features covered by each element.

Links to the Doxygen’s documentation are included, though unfortunately they point to a hard-coded location, Y:\doc\doxygen\html. If you can map Doxygen’s documentation to that path, you’ll have additional information when clicking on those links. If not, this information can also be accessed just by browsing the Doxygen’s Wiki pages: double click on the Doxygen’s main page, doc/doxygen/html/index.html, and use the tree panel or the search box.

More information about Doxygen is included in the section 8.12 Doxygen.

Shortcut: plc-cape-lab is the main application used in the chapter 4 Analysis & results. It’s a kind of laboratory tool with multiple options. A detailed description can be found in the chapter 3.7.3.3 plc-cape-lab.
3.4 Drivers

3.4.1 Main target
The standard SPI and ADC drivers in the off-the-shelf Debian Linux distribution which comes with the BBB are not able to process the data with the particularities required by this project, which can be summarized in:

- ChipSelect auto-toggling per DAC sample
- continuous transmission of DAC samples in real-time (without jitter and with low CPU impact)
- high-speed ADC capturing

The underlying hardware provides functionality enough to cover these requirements. SPI and ADC drivers need to be adapted to maximize performance taking advantage of the BBB processor.

3.4.2 Basic notions
Drivers are made up of low-level code designed to manage devices. They have the following characteristics:

- run in kernel space in privileged mode. In contrast to user-space applications drivers have full access to the hardware. Thus, they must be developed with care as they can easily hang the whole system
- provide a high-level interface for communication with devices, aimed to applications running in user-space
- drivers cannot use the functionality provided by well-known standard C libraries as glibc, pthread, etc. They have their own syntax (e.g. printk is the dual to printf)

The separation between user-space and kernel-space is very convenient as a way to protect the system. For instance, it allows the system to nicely kill a user-space process that is hanging the system:

- a "while(1);" loop from a user space application can be killed
- a "while(1);" loop from a kernel space driver can hang the system

Most development is done in user-space.

The implementation of the different running modes (kernel-space vs user-space) requires the CPU to support privileged modes. Old processors, as the 8086 didn’t have that support. Subsequent evolutions of the x86 architecture (286, 386, 486, 586 = Pentium, etc) added support to privileged modes and memory mapping that increased robustness.

The main drawback of protected modes is only the addition of some development complexity, as we’ll see when patching the drivers.

The BeagleBone Back is based on the ARM Cortex-A8 (based on the ARM7 architecture) which also integrates privileged modes and memory mapping.

3.4.3 Loadable vs built-in
In the Linux Kernel we roughly have two types of drivers:

- built-in: they are integrated within the kernel image and loaded at operating system initialization
Data communication through power lines using a system based on BeagleBone

- loadable: they come in separate binary files with the extension .ko (standing for Kernel Object). They can be loaded and unloaded on-demand. They are called **kernel modules**

These are some interesting commands related with the management of modules:

<table>
<thead>
<tr>
<th>Command</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>insmod</td>
<td>Installs/loads a loadable kernel module</td>
</tr>
<tr>
<td>rmmod</td>
<td>Removes a loadable kernel module</td>
</tr>
<tr>
<td>modprobe</td>
<td>Checks if a kernel module is currently loaded. If it is not then <code>modprobe</code> tries to install it (as <code>insmod</code>)</td>
</tr>
<tr>
<td>modinfo</td>
<td>Gets information from a kernel module</td>
</tr>
</tbody>
</table>

*Table 12. Basic commands for modules management*

In the Linux Kernel that comes with the off-the-shelf BBB, the SPI and ADC drivers are *built-in*, a fact that make things a bit more complex. We have (at least) three methods to modify these drivers:

- patch the source code of the original drivers and then rebuild the whole Linux Kernel. This is an unaffordable procedure in initial development phases because even simple tests would take too long to be practical. This is only a possible solution when the source code of the drivers is stable enough

- rebuild the kernel configuring the SPI and ADC drivers as *loadable modules* instead of *built-in*. This is the typical way but still requires the whole kernel to be rebuilt once, which is an intimidating task (at least for newbies as I am)

- modify the source-code of the SPI and ADC drivers but load them as auxiliary modules, complementing the functionality of the standard ones. The complexity of this solution resides on redirecting standard SPI and ADC calls to our customized drivers. It’s a kind of hooking or hacking

In this project, the option chosen has been the last one: “driver redirection”. To do that, the standard Device Tree Overlays of SPI and ADC have been modified.

### 3.4.4 Device Tree Overlays

#### 3.4.4.1 Basic notions

Device Tree Overlays (DTO) is a Linux concept intended to simplify the configuration of devices among different platforms. They describe the resources required by a specific device and provide a comfortable way of configuring it “on-the-fly”. In old versions of Linux that information was set when building the kernel. So, even for basic modifications of the hardware, a kernel rebuilding process was required, which is a task that usually takes several hours.

In this project some parts of the standard DTOs for the SPI and ADC drivers have been customized with two goals in mind:

- define a hook to redirect the standard driver calls to the customized ones
- provide some specific configuration (e.g. the direction of the DIN and DOUT pins in the SPI bus)
Data communication through power lines using a system based on BeagleBone

DTOs are described in a human readable form (in a *.dts file) which needs to be compiled with the dtc tool to the binary format the Linux Kernel understands\(^\text{[27]}\) (a *.dtbo file, standing for Device Tree Blob). For that purpose a script has been provided together with the DTS. For instance, this is the one for the ADC driver:

```
drivers/adc/dtb/compile-overlay.sh
```

```bash
dtc -O dtb -o PLC_CAPE-ADC-00A0.dtbo -b 0 -@ PLC_CAPE-ADC-00A0.dts
mv PLC_CAPE-ADC-00A0.dtbo /lib/firmware
```

The script compiles the PLC_CAPE-ADC-00A0.dts to PLC_CAPE-ADC-00A0.dtbo and moves it to the /lib/firmware folder where all the DTBOs reside.

The DTS files used are enclosed in 10.2 Device Tree Overlays.

We can do the reverse process of getting the human readable format from a DTBO with the fdtdump tool:

```bash
# fdtdump /lib/firmware/BB-SPIDEV0-00A0.dtbo
```

3.4.4.2 SPI data pins direction

The standard SPI driver allows configuring the direction of the DIN and DOUT pins in the SPI bus through its DTO.

PlcCape-v1 was designed with DIN and DOUT following the opposite configuration of the default convention. PlcCape-v2 was designed to be conforming to.

So, the PlcCape-v1 requires the default DTO to be adapted. For that, PLC_CAPE_V1-SPI0-00A0.dts needs the DIN and DOUT pins to be configured in two places:

- a first modification in the am33xx_pinmux target to specify the hardware configuration (OUTPUT ↔ INPUT)
- a second modification in the spi0 target to indicate the logical behavior of the pins. For PlcCape-v1 we are requesting the SPI driver to work in the pindir-d0-out-d1-in configuration, the opposite of the standard mode

```
PLC_CAPE_V2-SPI0-00A0.dts
```

```bash
... fragment@0 {
  target = <&am33xx_pinmux>;
  __overlay__ {
    spi0_pins_s0: spi0_pins_s0 {
      pinctrl-single,pins = <
      0x150 0x30 /* spi0_sclk, INPUT_PULLUP | MODE0 */
      0x154 0x30 /* spi0_d0, INPUT_PULLUP | MODE0 */
      0x158 0x10 /* spi0_cs0, OUTPUT_PULLUP | MODE0 */
      >;
    }
  }
}
```

```
PLC_CAPE_V1-SPI0-00A0.dts
```

```bash
... fragment@0 {
  target = <&am33xx_pinmux>;
  __overlay__ {
    spi0_pins_s0: spi0_pins_s0 {
      pinctrl-single,pins = <
      0x150 0x30 /* spi0_sclk, INPUT_PULLUP | MODE0 */
      >;
    }
  }
}
/* D0 & D1 toggled with regards to std configuration */
0x154 0x10 /* spi0_d0, OUTPUT_PULLUP | MODE0 */
0x158 0x30 /* spi0_d1, INPUT_PULLUP | MODE0 */
0x15c 0x10 /* spi0_cs0, OUTPUT_PULLUP | MODE0 */;
};;
};
};
fragment12 {
  target = <&spi0>;
  __overlay__ {
    ...
    /* D0 & D1 toggled with regards to std configuration*/
    ti,pindir-d0-out-d1-in = <1>;
    ...
  }
};

3.4.4.3 Driver entry-point hooking

The redirection of the driver is carried out by changing the “compatible” fields in the DTS files:

**PLC CAPE V1-SPI0-00A0.dts**

```plaintext
... /* identification */
part-number = "PLC CAPE V1-SPI0";
...
compatible = "ti,omap4-mcspi_plc"
...
compatible = "spidev_plc"
...
```

PLC CAPE V2-SPI0-00A0.dts has the same type of redirection.

**PLC CAPE-ADC-00A0.dts**

```plaintext
... /* identification */
part-number = "PLC CAPE-ADC"
...

**PLC CAPE-ADC-00A0.dts**

```plaintext
... /* identification */
part-number = "PLC CAPE-ADC"
...

On the other hand, the customized drivers have to declare the same `compatible` field to do the proper matching between the DTO and the driver. For example, this is the definition of the `omap_mcspi_of_match` in the customized SPI driver:

**spi-omap2-mcsplc_plc.c**

```c
static const struct of_device_id omap_mcspi_of_match[] = {
  { .compatible = "ti,omap2-mcspi",
    .data = &omap2_pdata,
  },
  { .compatible = "ti,omap4-mcspi_plc",
    .data = &omap4_pdata,
  },
};
MODULE_DEVICE_TABLE(of, omap_mcspi_of_match);
```

3.4.4.4 DTO loading

Once we have the DTBs in the `/lib/firmware`, we can load our current configuration by sending the name of the DTB to the slots section of the `bone_capemgr` device. This can be done with a simple `echo`:

```bash
# echo PLC_CAPE_V2-SPI0 > /sys/devices/bone_capemgr./slots
```
We can check that the DTB is properly loaded by having a look at the last kernel messages:

```
# dmesg | tail
[ 5634.157030] bone-capemgr bone_capemgr.9: part_number 'PLC_CAPE_V2-SPI0', version 'N/A'
[ 5634.157239] bone-capemgr bone_capemgr.9: slot #7: generic override
[ 5634.157286] bone-capemgr bone_capemgr.9: bone: Using override eeprom data at slot 7
[ 5634.157334] bone-capemgr bone_capemgr.9: slot #7: 'Override Board Name,00A0,Override Manuf,PLC_CAPE_V2-SPI0'
[ 5634.160401] bone-capemgr bone_capemgr.9: slot #7: Requesting part number/version based 'PLC_CAPE_V2-SP10-00A0.dtbo'
[ 5634.160462] bone-capemgr bone_capemgr.9: slot #7: Requesting firmware 'PLC_CAPE_V2-SP10-00A0.dtbo' for board-name 'Override Board Name', version '00A0'
```

3.4.5 Drivers loading notes

Once the proper DTO is loaded, it is time to load the customized drivers. For that purpose, we can use `insmod`. If the driver emits debugging messages, we can see them with `dmesg`. For instance, in our case `edma_plc` prints a debugging message (`edma_plc:edma_init`) when loaded:

```
# insmod edma_plc.ko
# insmod spi-omap2-mcspi_plc.ko
# insmod spidev_plc.ko
# dmesg | tail -n3
```

For convenience, a script has been created (`env/bbb/setup/load_drivers.sh`) to simplify the task of loading the required drivers, an operation that must be done after each boot. There is more information about in 3.7.2 Applications execution.

3.4.6 Drivers developed

As already commented two kernel drivers have been customized for this project:

- **SPI**: the official driver that comes with the off-the-shelf Debian distribution of the BBB needs to be modified to accommodate to the specifics of the AFE031 chip, which uses SPI for both, classical SPI commands exchanges but also for the transmission of DAC samples at high-speeds

- **ADC**: the off-the-shelf ADC driver needs to be modified to optimize the capturing process and this way be able to reach the maximum allowed rate (200000 samples per second)
### 3.4.6.1 spi-driver

#### 3.4.6.1.1 Summary

<table>
<thead>
<tr>
<th>Targets</th>
<th>edma_plc.ko</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>spidev_plc.ko</td>
</tr>
<tr>
<td></td>
<td>spi-omap2-mcspi_plc.ko</td>
</tr>
<tr>
<td>Purpose</td>
<td>Modify the original SPI driver for optimized high-speed transmission (based on DMA)</td>
</tr>
<tr>
<td>Details</td>
<td></td>
</tr>
<tr>
<td>Source code</td>
<td>./drivers/spi</td>
</tr>
</tbody>
</table>

#### 3.4.6.1.2 Requirements

Due to the particular way the AFE031 uses the SPI bus to process DAC samples, two important modifications of the standard off-the-shelf driver are required:

- **Auto-CS toggling**: to fulfill with the timing diagrams expected by the AFE031
- **Data transmission with a ping-pong buffer through EDMA**: to achieve the performance required by the real-time exchanges

#### 3.4.6.1.3 Testing means

To test the behavior of the SPI driver we can use the options provided by the *plc-cape-lab* application (described in 3.7.3.3 *plc-cape-lab*) to send continuous DAC data.

Then, with the oscilloscope we have to probe the SPI lines involved:

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSN</td>
<td>Chip Select, Negated</td>
<td>Serial Data Input “Strobe” of each DAC sample</td>
</tr>
<tr>
<td>DIN</td>
<td>Serial Data Input (from AFE031’s perspective)</td>
<td>DAC samples as a serial frame of 10-bits</td>
</tr>
<tr>
<td>SCLK</td>
<td>Serial Clock</td>
<td>Clock indicating when DOUT is valid</td>
</tr>
</tbody>
</table>

To know where to probe these signals, take a look at the corresponding figure according to the PlcCape version: Figure 6. PlcCape-v1 probing points or Figure 8. PlcCape-v2 probing points.
3.4.6.1.4 Auto Chip Select toggling

**Problem:** the Chip Select line is usually intended to enable a specific SPI slave (in a multi-slave scenario). In our case, the AFE031 uses it to indicate the end of a serial DAC sample coming from the DIN line, as can be seen in the DAC chronogram enclosed in the AFE031 datasheet[^43]:

![Figure 11. Writing to the DAC register](image1)

**Target:** try to configure the Chip Select line to automatically toggle (with minimum CPU impact) after each DAC sample.

A continuous DAC transmission test with the off-the-shelf SPI driver results in these signals (green = CSN; yellow = DIN):

![Figure 12. Absence of CSN pulse with the standard SPI driver](image2)

We can see that the CSN keeps constant (0V) after the transmission of samples. In this situation the AFE031 doesn’t produce DAC output.
For CSN-auto-toggling, the SPI driver has been modified this way:

```c
//spi-omap2-mcspi.c [Original code]
static void omap2_mcspi_set_master_mode(struct spi_master *master)
{
    struct omap2_mcspi *mcspi = spi_master_get_devdata(master);
    struct omap2_mcspi_regs *ctx = &mcspi->ctx;
    u32 l;

    /*
     * Setup when switching from (reset default) slave mode
     * to single-channel master mode
     */
    l = mcspi_read_reg(master, OMAP2_MCSPI_MODULCTRL);
    l &= ~(OMAP2_MCSPI_MODULCTRL_STEST | OMAP2_MCSPI_MODULCTRL_MS);
    l |= OMAP2_MCSPI_MODULCTRL_SINGLE;
    mcspi_write_reg(master, OMAP2_MCSPI_MODULCTRL, l);
    ctx->modulctrl = l;
}
```

```c
//spi-omap2-mcspi_plc.c [Patched code]
static void omap2_mcspi_set_master_mode(struct spi_master *master)
{
    struct omap2_mcspi *mcspi = spi_master_get_devdata(master);
    struct omap2_mcspi_regs *ctx = &mcspi->ctx;
    u32 l;

    /*
     * Setup when switching from (reset default) slave mode
     * to single-channel master mode
     */
    l = mcspi_read_reg(master, OMAP2_MCSPI_MODULCTRL);
    l &= ~(OMAP2_MCSPI_MODULCTRL_STEST | OMAP2_MCSPI_MODULCTRL_MS);
    l &= ~(OMAP2_MCSPI_MODULCTRL_SINGLE);
    mcspi_write_reg(master, OMAP2_MCSPI_MODULCTRL, l);
    ctx->modulctrl = l;
}
```

That is, we have just removed one line in the off-the-shelf driver that configured the SPI in Single Mode (OMAP2_MCSPI_MODULCTRL_SINGLE). This leaded the CSN line to never be toggled because this is an unnecessary operation if only one SPI device is attached to the line. This is explained in the page 4834 of the AM335x Technical Guide[^10].

[^10]: AM335x Technical Guide

### 24.3.2.6 Single-Channel Master Mode

When the SPI is configured as a master device with a single enabled channel, the assertion of the SPI_CSX signal can be controlled in two different ways:

- In 3 pin mode: MCSPI_MODULCTRL[1] PIN34 and MCSPI_MODULCTRL[0] SINGLE bit are set to 1, the controller transmit SPI word as soon as transmit register or FIFO is not empty.
- In 4 pin mode: MCSPI_MODULCTRL[1] PIN34 bit is cleared to 0 and MCSPI_MODULCTRL[0] SINGLE bit is set to 1. SPIEN assertion/deassertion controlled by Software. (See Section 24.3.2.6.1) using the MCSPI_Ch(i)CONF[20] FORCE bit.

![Figure 13. SPI Single-Channel Mode behavior](image)

In our case we have only one device attached to the SPI line (the AFE031) but we need the CSN to be toggled as if several slaves would be attached to. So, we are lucky and can reuse this behavior for our customized DAC strobe signaling.
A continuous DAC transmission test with the patched SPI driver confirms the wanted behavior (green = CSN; yellow = DIN):

![Figure 14. Automatic CSN pulse with the patched SPI driver](image1)

The check the correct functionality of the patched driver, a sawtooth wave of 400Hz has been generated:

![Figure 15. Real-time test of DAC samples transmission using a sawtooth wave](image2)

The linear result, without any distortion, demonstrates that the SPI data is timely and properly processed.

The proper management of higher SPI rates are tested with a sinusoid at about 50kHz. Note that, in order to determine the required sampling rate, these factors must be taken into account:

- the Nyquist rate\(^2\): the sampling rate must be at least twice the bandwidth
- DAC resolution: 10-bits per sample
- CSN pulse gap: around 1-bit extra time

This leads to a sampling rate greater than \(50k \times 2 \times 11 = 1100 \text{ kbps}\).

By default, the SPI driver is configured to allow bit rates in multiples of 2, with 750 kbps a typical value. For our case we need at least the next step which is 1500 kbps (1.5 Mbps). Find following a capture of a 50kHz sinusoid at 1.5Mbps:

![Figure 16. Test of sinusoid generation at 50kHz](image1)

It seems good but, unfortunately, after several tests we realize that sometimes a wrong generation happens:

![Figure 17. Typical effect in the DAC generation of a background operation](image2)

The graph above seems to indicate that there is a small gap during which the DAC generation is paused and kept constant (here for about 30 microseconds).

To correct this effect we can:

- boost the priority of our dedicated thread and our main process: this will reduce the chance of another process to acquire the CPU for too much time, but the possibility still exists. Moreover, this
system will fail if other threads are also boosting their priority for other real-time tasks (as the high-speed ADC capturing process)

- implement a DAC feeding system through **Direct Memory Access** (DMA): the benefits of this approach are huge:
  - transmission will be stable and regular because the bus will be acquired quickly for each DAC sample
  - it won’t impact the CPU because the processor doesn’t participate in the automatic exchanges.

Note, however, that it will still have some little impact on the overall performance because it relies on using the common shared bus: if a CPU instruction requires accessing the memory (which is the most frequent operation) it may be delayed for a short time

### 3.4.6.1.5 Ping Pong buffer with EDMA

The main target here is to implement the transmission of SPI data through a ping-pong DMA buffer providing a notification of when toggling occurs in order the client application to update the free buffer with the subsequent data.

The implementation of this functionality has been quite complex (maybe the most complex part of the whole project). Due to my lack of knowledge on kernel drivers development I’ve used techniques which are not elegant and which result in a quite awkward patched code. But in any case the solution provided is functional and valid enough for the target of this project. These are some examples of things to improve in future releases:

- I’ve used a real time SIGNAL (**#define PLC_SIGNUM_APP_WARNING 44**) to notify the client applications when a buffer underrun (aka underflow) occurs. The overhead entailed by signal exchanges between kernel-space-drivers and user-space-applications discourages its usage here. In fact, I changed the initial implementation that was sending a **signal** per buffer underrun because it frequently hung the whole system. I put a contention mechanism to send the **signal** once per second at the most, just to notify the client application that a problem occurred.

Notice also the hard-coded **#define** to 44 that I’ve randomly chosen within the available range for real-time signals (from 33 to 64). This is a not recommended practice but is acceptable enough for the evaluation purposes of this project

- For simplicity, I’ve used the debugging file system (**debugfs**) to carry out some normal exchanges between the client application and the kernel driver. Obviously another better system must be put in place because, as mentioned in the official documentation[^24]:

  > [...] Debugfs is a simple to use RAM based file system especially designed for debugging purposes. Developers are encouraged to use debugfs instead of procsfs in order to obtain some debugging information from their kernel code. Debugfs is quite flexible: it provides the possibility to set or get a single value with the help of just one line of code but the developer is also allowed to write its own read/write functions, and he can use the seq_file interface described in the procsfs section.

  > [...] An intense refactoring of the source code for the SPI driver is therefore required.

You can take a look at the enclosed source code for the whole patch.

[^24]: To use a real-time SIGNAL for custom purposes it should be based on SIGRTMIN. More information can be obtained in the manual page for signals (**man 7 signal**)
Just to have an idea, find following some key-points of the current implementation, mainly done in the spi-omap2-mcspi.c file (identified as OMAP2 McSPI controller driver):

**DMA PING-PONG CONFIGURATION**

### spi-omap2-mcspi.c [Original code]

```c
static void omap2_mcspi_tx_dma(struct spi_device *spi,
    struct spi_transfer *xfer,
    struct dma_slave_config cfg)
{
    struct omap2_mcspi *mcspi;
    struct omap2_mcspi_dma *mcspi_dma;
    unsigned int count;

    mcspi = spi_master_get_devdata(spi->master);
    mcspi_dma = &mcspi->dma_channels[spi->chip_select];
    count = xfer->len;

    if (mcspi_dma->dma_tx) {
        struct dma_async_tx_descriptor *tx;
        struct scatterlist sg;

        dma_async glfw_slave_config(mcspi_dma->dma_rx, &cfg);

        sg_init_table(&sg, 1);
        sg_dma_address(&sg) = xfer->tx_dma;
        sg_dma_len(&sg) = xfer->len;

        tx = dmaengine_prep_slave_sg(mcspi_dma->dma_rx, &sg, 1,
            DMA_MEM_TO_DEV, DMA_PREP_INTERRUPT | DMA_CTRL_ACK);
        if (tx) {
            tx->callback = omap2_mcspi_tx_callback;
            tx->callback_param = spi;
            dmaengine_submit(tx);
        } else {
            /* FIXME: fall back to PIO? */
        }
    } else {
        dma_async_issue_pending(mcspi_dma->dma_rx);
        omap2_mcspi_set_dma_req(spi, 0, 1);
    }
}
```

### spi-omap2-mcspi_plc.c [Patched code]

```c
static void omap2_mcspi_tx_configure_dma_ping_pong(struct spi_device *spi,
    struct dma_slave_config cfg)
{
    struct omap2_mcspi *mcspi;
    struct omap2_mcspi_dma *mcspi_dma;

    mcspi = spi_master_get_devdata(spi->master);
    mcspi_dma = &mcspi->dma_channels[spi->chip_select];

    dev_dbg(mcspi->dev, "omap2_mcspi_tx_configure_dma_ping_pong:init");
    if (!mcspi_dma->ping_pong_enabled || !mcspi_dma->dma_tx)
        dev_err(mcspi->dev, "Ping pong not enabled!
        ");

    // [PLC] List of three items for an easy dual buffer implementation
    // An interm is required for the active ParAM, the other two maintain the ping-pong link
    
    struct dma_async glfw_slave_config(mcspi_dma->dma_rx, &cfg);

    sg_init_table(sg, 3);
    sg_dma_address(&sg[0]) = mcspi_dma->ping pong.buf1_dma;
    sg_dma_len(&sg[0]) = mcspi_dma->ping pong.buf1_count*2;
    sg_dma_address(&sg[1]) = mcspi_dma->ping pong.buf2_dma;
    sg_dma_len(&sg[1]) = mcspi_dma->ping pong.buf2_count*2;
    sg_dma_address(&sg[2]) = mcspi_dma->ping pong.buf1_count*2;

    tx = dmaengine_prep_slave_sg(mcspi_dma->dma_rx, sg, 3,
```
In the above code we can see how I have chained DMA addresses using the PaRAM for a ping-pong buffer: DMA uses first `ping_pong.buf1_dma`, then automatically switches to `ping_pong.buf2_dma` and finally it switches to `ping_pong.buf1_dma` again.

The `edma.c` file (identified as *TI EDMA DMA engine driver*) has also been required to be modified. For example you can take a look at the `edma_execute` function (among other changes):

**edma.c [Original code]**

```c
static void edma_execute(struct edma_chan *echan)
{
    struct virt_dma_desc *vdesc = vchan_next_desc(&echan->vchan);
    struct edma_desc *edesc;

    /* Write descriptor PaRAM set(s) */
    for (i = 0; i < edesc->pset_nr; i++) {
        /* Link to the previous slot if not the last set */
        if (i != (edesc->pset_nr - 1))
            edma_link(echan->slot[i], echan->slot[i+1]);
        /* Final pset links to the dummy pset */
        else
            edma_link(echan->slot[i], echan->ecc->dummy_slot);
    }
    edma_start(echan->ch_num);
}
```

**edma_plc.c [Patched code]**

```c
static void edma_execute(struct edma_chan *echan)
{
    struct virt_dma_desc *vdesc = vchan_next_desc(&echan->vchan);
    struct edma_desc *edesc;

    /* Write descriptor PaRAM set(s) */
    for (i = 0; i < edesc->pset_nr; i++) {
        /* [PLC] DMA linking on channel CHANNEL_SPITX */
        if (echan->ch_num == CHANNEL_SPITX) {
            // Single buffer mode
            if (edesc->pset_nr == 1) {
                edma_link(echan->slot[i], echan->ecc->dummy_slot);
            } else if (edesc->pset_nr == 2) {
                if (i == 0)
                    edma_link(echan->slot[0], echan->slot[1]);
                else
                    edma_link(echan->slot[1], echan->slot[1]);
            } else {
                edma_link(echan->slot[0], echan->slot[1]);
                edma_link(echan->slot[1], echan->slot[0]);
            }
        } else {
            // Continuous mode
            if (i == 0)
                edma_link(echan->slot[0], echan->slot[1]);
            else
                edma_link(echan->slot[1], echan->slot[0]);
        }
    }
    edma_start(echan->ch_num);
}```
Data communication through power lines using a system based on BeagleBone

```c
edma_link(echan->slot[i], echan->slot[i+1]);
} else {
    dev_err(echan->vchan.chan.device->dev, "Unexpected DMA data");
}
} else {
    /* Link to the previous slot if not the last set */
    if (i != (edesc->pset_nr - 1))
        edma_link(echan->slot[i], echan->slot[i+1]);
    /* Final pset links to the dummy pset */
    else
        edma_link(echan->slot[i], echan->ecc->dummy_slot);
}
edma_start(echan->ch_num);
```

For more information about EDMA and PaRAM management consult the pages 1538 and 1542 of the AM335x Technical Guide[10], within the chapter 11.3.3.2 EDMA3 Channel PaRAM Set Entry Fields.

**BUFFER UNDERFLOW NOTIFICATION WITH FREQUENCY CONTENTION**

```c
spi-omap2-mcspi_plc.c
static void omap2_mcspi_plc_send_warning_signal(struct device *dev, struct task_struct *app_task)
{
    struct siginfo info;
    int ret;
    memset(&info, 0, sizeof(struct siginfo));
    info.si_signo = PLC_SIGNUM_APP_WARNING;
    info.si_code = SI_QUEUE;
    info.si_int = 1234;
    ret = send_sig_info(PLC_SIGNUM_APP_WARNING, &info, app_task);
    if (ret < 0) {
        dev_err(dev, "error sending signal\n");
        return;
    }
}
```

```c
static void omap2_mcspi_tx_callback(void *data)
{
    struct spi_device *spi = data;
    struct omap2_mcspi *mcspi = spi_master_get_devdata(spi->master);
    struct omap2_mcspi_dma *mcspi_dma = &mcspi->dma_channels[spi->chip_select];
    // [PLC] Send signals
    if (mcspi_dma->ping_pong_enabled) {
        // TODO: Improve access to 'dma_buffer_waiting'. Locking required?
        if (mcspi_dma->ping_pong.check_missed_dma_buffer
            & (mcspi_dma->ping_pong.dma_buffer_waiting == 0)) {
            u64 stamp = jiffies;
            if (stamp > next_warning_stamp) {
                next_warning_stamp = stamp+usecs_to_jiffies(WARNING_CONTENTION_US);
                omap2_mcspi_plc_send_warning_signal(&spi->dev, mcspi_dma->app_task);
            }
        }
        mcspi_dma->ping_pong.cur_buffer_in_dma_index =
            (mcspi_dma->ping_pong.cur_buffer_in_dma_index+1) & 0x1;
        complete(&mcspi_dma->dma_tx_completion);
    }
    else if (mcspi_dma->stop_dma_after_tx && !mcspi_dma->dma_completion_notified)
    {
        mcspi_dma->dma_tx_in_progress = 0;
        complete(&mcspi_dma->dma_tx_completion);
        /* We must disable the DMA TX request */
        omap2_mcspi_set_dma_req(spi, 0, 0);
    }
}
```

In the code of the omap2_mcspi_tx_callback we can see how the signal PLC_SIGNUM_APP_WARNING is triggered when the `mcspi_dma->ping_pong.dma_buffer_waiting` is equal to 0 indicating that the client application is not waiting for the DMA buffer yet, probably because it is still busy processing the previous
buffer. If so, this would mean a buffer underrun condition (i.e. DMA dispatches data faster than the client application is able to fill) and hence, the SIGNAL notification.

**EXCHANGES THROUGH ‘DEBUGFS’ TO NOTIFY PING-PONG TOGGLING**

```c
#define DEBUGFS_IOC_MAGIC 'd'
#define DEBUGFS_IOC_RD_WAIT_DMA_BUFFER _IOR(DEBUGFS_IOC_MAGIC, 1, __u8)

static long
debugfs_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int err = 0;
    int retval = 0;

    struct omap2_mcspi *mcspi = spi_master_get_devdata(spi_master_global);
    struct omap2_mcspi_dma *mcspi_dma = &mcspi->dma_channels[chip_select_global];

    /* Check type and command number */
    if (_IOC_TYPE(cmd) != DEBUGFS_IOC_MAGIC)
        return -ENOTTY;

    /* Check access direction once here; don't repeat below. */
    /* IOC_DIR is from the user perspective, while access_ok is */
    /* from the kernel perspective; so they look reversed. */
    if (_IOC_DIR(cmd) & _IOC_READ)
        err = !access_ok(VERIFY_WRITE,
                         (void __user *)arg, _IOC_SIZE(cmd));
    if (err == 0 && _IOC_DIR(cmd) & _IOC_WRITE)
        err = !access_ok(VERIFY_READ,
                         (void __user *)arg, _IOC_SIZE(cmd));

    if (err)
        return -EFAULT;

    switch (cmd) {

    /* read requests */
    case DEBUGFS_IOC_RD_WAIT_DMA_BUFFER:
        // TODO: Improve to notify an idle or aborted DMA transfer instead of
        // just sending a hard-coded zero or a partially transmitted buffer
        if (!mcspi_dma->dma_tx_in_progress) {
            retval = __put_user(0, (__u8 __user *)arg);
            break;
        }
        mcspi_dma->ping_pongdma_buffer_waiting = 1;
        init_completion(&mcspi_dma->dma_tx_completion);
        wait_for_completion(&mcspi_dma->dma_tx_completion);
        mcspi_dma->ping_pong_dma_buffer_waiting = 0;
        retval = __put_user((mcspi_dma->ping_pong.cur_buffer_in_dma_index,
                             (__u8 __user *))arg);
        break;
    }
    return retval;
}
```

In the `debugfs_ioctl` there is an example of how I’ve used the `debugfs` to implement the synchronization with the Ping Pong buffer.

**3.4.6.1.6 Analysis**

The results of the data transmission using the Ping-Pong DMA buffer demonstrates the importance of this implementation. When using it, the output waves are stable even if the CPU is switching among other tasks in the background (we’ll see some graphs about in the chapter 3.4.6.2 adc-driver). In fact, the wave generation continues to run even if the BBB freezes (due to the standalone working mode of the DMA).

But the source code must still be optimized: the signal `PLC_SIGNUM_APP_WARNING`, notifying about buffer underrun conditions, is sometimes fired, specially when using high SPI bit rates. An easy way to improve it may be through the use of larger DMA buffers for example (now limited by the page size = 4096 bytes, for implementation simplicity).
### 3.4.6.1.7 Effective DAC frequency

The tests done to validate the SPI functionality show that there is an extra gap between samples of about half a clock cycle. In fact, this gap is configurable through the TCS field of the CF0CONF register, being 0.5 clock cycles the minimum value, as described in this capture from the AM335x Technical Manual[^10] at page 4838:

![Chip-Select timing control](image)

For this project we need to **accurately** know the DAC sampling rate because the application requires it to produce an output signal of a given frequency.

For medium bit rates we have confirmed with the oscilloscope that the DAC sampling frequency is equal to “SPI_Bit_Rate/10.5”. But for higher frequencies we observe an unexpected slippage which increases with the bitrate.

By analyzing the involved signals, we can observe that the cause of this deviation is the fixed width of the Chip Select toggling pulse, negligible at low bit rates, but significant at high ones.

The following captures show the signals SCLK (in blue) and CSN (in red) at different capturing rates:

![SCLK (blue) and CSN (red) at SPI bitrate = 375kbps](image)

At low SPI rates the width of the CSN pulse is negligible.
At 1.5Mbps:

- SCLK cycle = 1/1.5E6 = 667ns
- Expected gap between words (from falling to rising SCLK) = ½ cycle due to usual clock toggling rate + ½ extra cycle due to TCS = 1 full SCLK cycle = 667ns
- Measured gap from falling to rising SCLK edges = 834ns
- Difference = 834ns-667ns = 167ns

So, there is a noticeable extra gap due to the CSN pulse.

At 3Mbps:

- SCLK cycle = 1/3.0E6 = 333 ns
- Expected falling-to-rising-edges = 1 SCLK cycle = 333ns
- Measured gap = 480ns
- Difference = 480-333 = 147ns

At 3Mbps the difference has a similar value in absolute terms which means a higher deviation in relative terms.
At 6Mbps:

- SCLK cycle = 1/6.0E6 = 167 ns
- Expected falling-to-rising-edges = 1 SCLK cycle = 167 ns
- Measured gap = 335 ns
- Difference = 335-167 = 168 ns

At 6Mbps the measured gap doubles the expected one.

The captures of the CSN seem to indicate a pulse width close to 160 ns:

In the above capture, the CSN pulse exposes an oscillating smooth shape rather than the expected squared form. This is due to the bandwidth limitation of the USB oscilloscope and the interpolation algorithm applied.

For maximum accuracy the following process has been used to find the exact formula that converts bit rates (bits per second) to sampling rates (samples per second):

1. With the plc-cape-lab application a sinusoid of 2 kHz has been generated and captured with the oscilloscope
2. Initially, low bit rates have been tested confirming the 0.5-clock-cycle-extra-gap and the corresponding basic formula, \( f_{sps} = \text{bps}/10.5 \)
3. Then, higher bit rates have been tested. As expected, the resulting frequency, measured with the oscilloscope, is then smaller than the 2kHz because the effect of the CSN width. Octave has also been used to measure the effective frequency by locating the peak of [Abs(FFT)]

4. The 160ns-CSN-pulse factor has been included into the formula and checked again with much better results but with still some little deviation

5. The CSN-pulse-width factor has then been tuned until found the one giving the best results at high bit rates. The best matching occurred for a CSN width of 165ns

So the formula to convert from bit-rate to sample-rate has finally resulted in:

\[
\text{freq\_DAC [samples per second]} = \frac{1}{\frac{10.5}{\text{SPI rate in bits per second}}} + 165 \times 10^{-9}
\]

Or in source-code terms:

`freq_dac_sps = 1.0/(10.5/(float)settings->tx.freq_bps+165e-9);`
3.4.6.2 adc-driver

3.4.6.2.1 Summary

| Targets   | adc_hs.ko  
ti_am335x_tscadc_pll.ko |
| Purpose   | Modify the original ADC driver for high-speed captures |
| Details   | |
| Source code | /drivers/adc |

3.4.6.2.2 Requirements

**Target:** modify ADC driver to maximize the capturing rate.

At the page 3 of the AM3358 Manual\(^1\) it is indicated that we should be able to reach 200 ksps.

On the other hand the current Linux Kernel already offers the IIO subsystem\(^4\) for high-speed data acquisition.

3.4.6.2.3 Patch

The `ti_am335x_tscadc_pll.c` file (*TI Touch Screen / ADC MFD driver*) has been modified with minor changes, just to allow the hooking of the ADC custom driver without interfering the standard one. For example the `am335x_tsc_se_update` function has been renamed to `am335x_tsc_se_update_pll` to export the function with a different public name and this way to allow the coexistence of both drivers in memory:

```c
// [PLC] EXPORTS renames to avoid conflicts with modules already loaded
void am335x_tsc_se_update_pll(struct ti_tscadc_dev *tsadc)
{
    tscadc_writel(tsadc, REG_SE, tsadc->reg_se_cache);
}
EXPORT_SYMBOL_GPL(am335x_tsc_se_update_pll);
```

The `ti_am335x_adc.c` file (*TI ADC MFD driver*) has been modified for high speed ADC capturing:

- the official driver is focused on Touch Screen controllers where the reliability of the measures is the main goal and hence, strong averaging of the ADC data is applied
- for high ADC data captures this file has been cloned to `adc_hs.c` and modified removing the averaging and tuning the FIFO for better acquisition times

The initial goal was to implement DMA capturing, as done for the SPI, but due to time constraints and the complexity of the implementation, this has not been possible. Anyway the modifications done without DMA have been demonstrated to be good enough for the evaluation purposes of this project.

---

For example find following some changes done:

**drivers/iio/adc/ti_am335x_adc.c [Original code]**

```c
static void tiadc_step_config(struct iio_dev *indio_dev)
{
    struct tiadc_device *adc_dev = iio_priv(indio_dev);
    unsigned int stepconfig;
    int i, steps, chan;

    /*
     * There are 16 configurable steps and 8 analog input
     * lines available which are shared between Touchscreen and ADC.
     * Steps backwards i.e. from 16 towards 0 are used by ADC
     * depending on number of input lines needed.
     * Channel would represent which analog input
     * needs to be given to ADC to digitalize data.
     */
    steps = TOTAL_STEPS - adc_dev->channels;
    if (iio_buffer_enabled(indio_dev))
        stepconfig = STEPCONFIG_AVG_16 | STEPCONFIG_FIFO1 |
                     STEPCONFIG_MODE_SWCNT;
    else
        stepconfig = STEPCONFIG_AVG_16 | STEPCONFIG_FIFO1;
    for (i = 0; i < adc_dev->channels; i++) {
        chan = adc_dev->channel_line[i];
        tiadc_writel(adc_dev, REG_STEPCONFIG(steps),
                     stepconfig | STEPCONFIG_INP(chan));
        tiadc_writel(adc_dev, REG_STEPDELAY(steps),
                     STEPCONFIG_OPENDLY);
        adc_dev->channel_step[i] = steps;
        steps++;
    }
}
```

**adc_hs.c [Patched code]**

```c
static void tiadc_step_config(struct iio_dev *indio_dev)
{
    struct tiadc_device *adc_dev = iio_priv(indio_dev);
    unsigned int stepconfig;
    int i, steps, chan;

    steps = TOTAL_STEPS - adc_dev->channels;
    // [PLC] The standard version comes with a default value of 'STEPCONFIG_AVG_16' for better
    // averaging. Changing to 'STEPCONFIG_AVG(0)' to maximize capture speed
    if (iio_buffer_enabled(indio_dev))
        stepconfig = STEPCONFIG_AVG(0) | STEPCONFIG_FIFO1 |
                     STEPCONFIG_MODE_SWCNT;
    else
        stepconfig = STEPCONFIG_AVG(0) | STEPCONFIG_FIFO1;
    for (i = 0; i < adc_dev->channels; i++) {
        chan = adc_dev->channel_line[i];
        tiadc_writel(adc_dev, REG_STEPCONFIG(steps),
                     stepconfig | STEPCONFIG_INP(chan));
        // [PLC] The standard version came with a default value of 'STEPDELAY_OPEN(0)' =
        // 'STEP_DELAY_OPEN(0x098)'. Set to 'STEP_DELAY_OPEN(0x0)' to maximize capturing speed
        tiadc_writel(adc_dev, REG_STEPDELAY(steps), STEPDELAY_OPEN(0));
        adc_dev->channel_step[i] = steps;
        steps++;
    }
}
```

In the above code we can see that for our driver we have changed the averaging from
STEPCONFIG_AVG_16 to STEPCONFIG_AVG(0) and also the STEPDELAY_OPENLY to
STEP_DELAY_OPEN(0).
Other changes following the same line:

```c
adc_hs.c [Patched code]
static u32 get_adc_step_mask(struct tiadc_device *adc_dev) {
    u32 step_en;
    step_en = ((1 << adc_dev->channels) - 1);  
    step_en <<= TOTAL_STEPS - adc_dev->channels + 1;
    // [PLC] Activate only 1 step (AIN0) to maximize speed (0x200)
    step_en = 0x200;
    return step_en;
}
```

And also, some examples of the `debugfs` use (to be refactored when possible) to start and stop the continuous ADC capturing mode (note that despite the `dma` prefix used on some variables, DMA is not currently used):

```c
// TODO: Refactor. Based on 'debugfs_ioctl' from 'spi-omap2-mcspi_plc.c'
#define ADCHS_IOC_MAGIC 'a'
#define ADCHS_IOC_WR_START_CAPTURE _IOW(ADCHS_IOC_MAGIC, 1, __u8)
#define ADCHS_IOC_WR_STOP_CAPTURE _IOW(ADCHS_IOC_MAGIC, 2, __u8)
static long adchs_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {
    ...  
    switch (cmd) {
    /* read requests */
    case ADCHS_IOC_WR_START_CAPTURE:
        dev_dbg(&adchsdev->indio_dev->dev, "IOCTL: Start capture mode\n");
        if (!continuous_transfer)
            return -EFAULT;
        abort_continuous_transfer = 0;
        adc_dev->data_avail = 0;
        continuous_transfer_started = 1;
        adc_dev->dma_samples_cur_write = adc_dev->dma_samples;
        adc_dev->dma_samples_cur_read = adc_dev->dma_samples;
        {
            unsigned int config;
            config = tiadc_readl(adc_dev, REG_CTRL);
            tiadc_writel(adc_dev, REG_CTRL, config & ~CNTRLREG_TSCSSENB);
            tiadc_writel(adc_dev, REG_CTRL, config | CNTRLREG_TSCSSENB);
            tiadc_writel(adc_dev, REG_IRQSTATUS, IRQENB_FIFO1THRES | IRQENB_FIFO1OVRRUN | IRQENB_FIFO1UNDRFLW);
            tiadc_writel(adc_dev, REG_IRQENABLE, IRQENB_FIFO1THRES | IRQENB_FIFO1OVRRUN);
        }
        break;
    case ADCHS_IOC_WR_STOP_CAPTURE:
        dev_dbg(&adchsdev->indio_dev->dev, "IOCTL: Stop capture mode\n");
        abort_continuous_transfer = 1;
        wake_up_interruptible(&adc_dev->wq_data_avail);
        if (!iio_std_exchange)
        {
            unsigned int config;
            tiadc_writel(adc_dev, REG_IRQCLR, (IRQENB_FIFO1THRES | IRQENB_FIFO1OVRRUN | IRQENB_FIFO1UNDRFLW));
            config = tiadc_readl(adc_dev, REG_CTRL);
            tiadc_writel(adc_dev, REG_CTRL, config & ~CNTRLREG_TSCSSENB);
        }
        continuous_transfer_started = 0;
        break;
    }
    return retval;
    }
```
And finally, some tuned code in the IRQ for better real-time response:

```c
static irqreturn_t tiadc_irq(int irq, void *private)
{
    struct iio_dev *idev = private;
    ...
} else if (status & IRQENB_FIFO1THRES) {
    int fifo1count;
    /* Wake adc_work that pushes FIFO data to iio buffer */
    // tiadc_writel(adc_dev, REG_IRQCLR, IRQENB_FIFO1THRES);
    fifo1count = tiadc_readl(adc_dev, REG_FIFO1CNT);
    for (; fifo1count > 0; fifo1count--) {
        adc_dev->dma_samples_cur_write++ = tiadc_readl(adc_dev, REG_FIFO1)
        & FIFOREAD_DATA_MASK;
        // Circular buffer overflow
        if (adc_dev->dma_samples_cur_write == adc_dev->dma_samples_end)
            adc_dev->dma_samples_cur_write = adc_dev->dma_samples;
        if (adc_dev->dma_samples_cur_write == adc_dev->dma_samples_cur_read)
            printk(KERN_INFO "adc_hs_plc: Read Underflow!\n");
    }
    tiadc_writel(adc_dev, REG_IRQSTATUS, IRQENB_FIFO1THRES);
    // tiadc_writel(adc_dev, REG_IRQENABLE,IRQENB_FIFO1THRES);
    adc_dev->data_avail = 1;
    wake_up_interruptible(&adc_dev->wq_data_avail);
    return IRQ_HANDLED;
} else
    return IRQ_NONE;
}
```

For more details take a look at the source code provided.

### 3.4.6.2.4 Analysis

To check that the patched ADC driver is working as expected, the *plc-cape-lab* (3.7.3.3 plc-cape-lab) application has been configured like this:

- **AFE Mode** = *calib_adc_txpga*: this makes an internal bridge between the TX and RX blocks. We can then send custom data to the DAC and check that the ADC is properly capturing it
- Samples-to-file = 200 samples
- SPI TX = 6Mbps
- Offset = 800, Range = 400 (these values have been found by experimentation in the calibration mode. Other values can saturate some intermediate stage and produce wrong results)
- RX DC Offset = 1100
- Freq = 10kHz
- ADC capturing rate = 200ksps (max value supported by the hardware)

The captured samples have been checked with Octave with:

```bash
>> run "X:/tools/octave/plot_adc_fft.m"; execute_test("adc", "Y:/applications/plc-cape-lab")
```

There are more details about Octave usage in 3.9 Development with Octave.
This is the result for a generated sinusoid:

![AD Capture](image1)

![FFT](image2)

*Figure 25. Sinusoid of 10kHz captured by the ADC and plotted with Octave*

In the top plot we have the samples captured by the ADC. We confirm that the sinusoid is properly captured.

In the bottom plot we have the absolute value of the Fast Fourier Transform which give us the composition of the received signal in the frequency domain. Notice that, while the ADC capture at top has a DC component (1150 in average), the corresponding FFT at bottom doesn’t reflect any value at 0. That’s because, for better readability, the DC component has been explicitly removed.

The X-axis represents the frequency in the digital domain. That frequency is usually represented as the ratio over the ADC capturing rate, which would result in an X-axis between 0 and 1. In the graph with Octave I’ve kept the index-based units because give also an idea of the accuracy of the graph: 200 means we have calculated the FFT over 200 points.

The middle value of the X-axis corresponds to half the ADC capturing rate. So, the 100 corresponds to 100 kHz.

The shape is a bit abrupt because few samples per cycle has been used to conform the wave:

- ADC capturing rate = 200 kbps. Wave freq = 10kHz (= 10k cycles per second) → 200ksps/10kcps = 20 samples per cycle.
This can be appreciated with a steam graph in Octave:

![Steam graph in Octave](image)

*Figure 26. Zoom of the ADC captured data to the sample level*

Other types of waves have been generated to confirm the ADC is properly capturing them. For example a 10kHz ramp because, being linear, it’s easy to detect any distortion at human-eye:

![ADC capture of a sawtooth wave](image)

*Figure 27. ADC capture of a sawtooth wave*

The linearity of the results confirms that the ADC is working OK. We see some irregularities but they are normal because the low-level of the signal we are capturing. The ADC of the BBB has 12-bits of resolution which gives a capturing margin of 4096 values ($2^{12}$) for a range between 0 and 1.65V (3.3V/2). That means
0.4mV per ADC unit (1.65/4096). The captured graph covers 600 ADC units (between 800 and 1400) which are equivalent to $600 \times 0.4\text{mV} = 240\text{mVpp}$.

Note that the ramp-wave expands over a lot of frequencies (many FFT peaks) because the asymmetry of the signal. If we use a symmetric triangular-wave instead, that’s what we get:

![ADC capture](image1)

![FFT](image2)

*Figure 28. ADC capture of a triangular wave*

With all these tests we have confirmed that the patched ADC driver is properly working at 200ksps.

Next series of tests will explore the behavior of the ADC at the frequencies we need to use in this project, which are over the maximum supported.
Test with a sinusoid of 80kHz:

![ADC capture of a sinusoid of 80kHz](image1)

Note that in the captured image in the top it’s difficult to identify the sinusoid shape. That’s because we are using only 2.5 samples per cycle \( (200k/80k) \). The FFT, however, clearly exposes the sinusoidal behavior at the 80kHz (peak at index 80). Repeating the test over 90kHz results in this:

![ADC capture of a sinusoid of 90kHz](image2)

We have now 2.22 samples per cycle \( (200k/90k) \).
Finally, find following the result of increasing the frequency to 100kHz, 110kHz and 120kHz:

**Frequency = 100 kHz**

![ADC capture of a sinusoid of 100kHz](image1)

![FFT](image2)

*Figure 31. ADC capture of a sinusoid of 100kHz*

**Frequency = 110 kHz**

![ADC capture of a sinusoid of 110kHz](image3)

![FFT](image4)

*Figure 32. ADC capture of a sinusoid of 110kHz*
The point to mention here is that when increasing the generated frequency over the capturing rate (Nyquist frequency\(^5\) = half of the sampling rate = 100kHz), the ADC captures decreasing frequencies. For a generated wave of 110k we have the same results as for 90kHz. And in 120kHz the same as in the 80kHz case. That’s due to the undersampling theory\(^6\).

For the feasibility tests done in this project it doesn’t pose a big problem (as we’ll seen in 4 Analysis & results) because the AFE031 filters out the frequency content out of the band of interest (CENELEC A or B/C/D) and because our tests are based on just one carrier. But we should have this limitation in mind.

Note that the CENELEC B/C/D band (95kHz to 125kHz as mentioned in Table 4. CENELEC bands) falls in the middle of the capturing frequency (=100kHz). When using modulation schemes that extend over all the band, the captured signal would be a mix that couldn’t be decoded. A possible solution to that could be selecting another capturing ADC rate allowing to have the correct replica over all the band.

This ends the chapter dedicated to drivers and kernel-space. From now on we’ll lie in user-space.

---

5 Some notes about in https://en.wikipedia.org/wiki/Nyquist_frequency
6 Basic information on Undersampling theory in https://en.wikipedia.org/wiki/Undersampling
3.5 Static libraries

3.5.1 Overview

Static libraries are a convenient and comfortable way of reusing source code. The result of a static library is a *.a file with the executable code. Developers that want to use a library must know the publicly exported symbols (functions or variables).

The encapsulation of reusable code in libraries (versus cloning source code everywhere) has the following benefits:

- Having the shared code located in one single place simplifies the maintenance: bug fixing or improvements need to be done just in one single place
- When the source code of a static library is updated, the dependent projects need to be just re-linked which is a faster process than recompiling

To simplify maintenance, all the public symbols offered by a library are declared in headers (*.h) in a sub-folder called api (from Application Programming Interface).

The api sub-directory strategy has these goals:

- for the consumers of libraries, it simplifies the task of discovering the functionality offered by a library because all the public declarations are located in one well-known easily accessible point
- for the developers of libraries, it points out that a function has been declared as public and, therefore, shouldn’t be modified in a backwards incompatibility way

3.5.2 Libraries developed

For this project these libraries have been defined:

<table>
<thead>
<tr>
<th>Name</th>
<th>Source code</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libplc-tools</td>
<td>./libraries/libplc-tools</td>
<td>Generic functionality to be reused by any plc-cape-based component.</td>
</tr>
<tr>
<td>libplc-gpio</td>
<td>./libraries/libplc-gpio</td>
<td>GPIO helpful extra functionality.</td>
</tr>
<tr>
<td>libplc-adc</td>
<td>./libraries/libplc-adc</td>
<td>ADC functionality provided by the plc-cape modified ADC driver.</td>
</tr>
<tr>
<td>libplc-cape</td>
<td>./libraries/libplc-cape</td>
<td>Encapsulation of the PlcCape board functionality.</td>
</tr>
</tbody>
</table>

Table 13. Static libraries developed

Next sections will provide a brief description of each library and an extract of the public API offered (from the headers in the api folder). This is a good means of having an idea about the main available functionality. I’ve removed the Doxygen’s comments to present this information in a more compact format.
3.5.2.1 libplc-tools

<table>
<thead>
<tr>
<th>Target</th>
<th>libplc-tools.a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>This library offers generic functionality to be used by any project belonging to the plc-cape framework</td>
</tr>
<tr>
<td>Details</td>
<td>This library relies only on functionality provided by the Linux Operating system. Therefore, it could be used in other Linux-based platforms besides the BeagleBone Black platform</td>
</tr>
<tr>
<td></td>
<td>This version of the library covers these areas:</td>
</tr>
<tr>
<td></td>
<td>• <strong>application</strong>: Helper functions related to the management of applications.</td>
</tr>
<tr>
<td></td>
<td>• <strong>file</strong>: Generic file functionality as writing data in Octave-compatible format.</td>
</tr>
<tr>
<td></td>
<td>• <strong>plugin</strong>: Helper functions related to the plugin management.</td>
</tr>
<tr>
<td></td>
<td>• <strong>terminal_io</strong>: Extended input/output functionality in terminal consoles, as checking for key pressing in non-blocking mode.</td>
</tr>
<tr>
<td></td>
<td>• <strong>time</strong>: Time measurement facilities.</td>
</tr>
<tr>
<td></td>
<td>• <strong>trace</strong>: Tracing helper functions.</td>
</tr>
<tr>
<td>Dependencies</td>
<td>librt: for some time-measurement functions</td>
</tr>
<tr>
<td>API help</td>
<td>.libraries/libplc-tools/api</td>
</tr>
<tr>
<td>Source code</td>
<td>.libraries/libplc-tools</td>
</tr>
</tbody>
</table>

Extract of the provided API:

**application.h**

```c
char *plc_application_get_abs_path(void);
char *plc_application_get_abs_dir(void);
char *plc_application_get_output_abs_dir(void);
```

**file.h**

```c
void plc_file_write_string(const char *base_path, const char *rel_path, const char *value);
int plc_file_write_csv(const char *filename, enum csv_type_enum csv_type_enum, void *buffer, ...
```

**plugin.h**

```c
char *plc_plugin_get_abs_path(enum plc_plugin_category category, const char *plugin_name);
struct plc_plugin_list *plc_plugin_list_create(enum plc_plugin_category category);
void plc_plugin_list_release(struct plc_plugin_list *plc_plugin_list);
int plc_plugin_list_find_name(struct plc_plugin_list *plc_plugin_list, const char *plugin_name);
```

**terminal_io.h**

```c
struct plc_terminal_io *plc_terminal_io_create(void);
void plc_terminal_io_release(struct plc_terminal_io *plc_terminal_io);
int plc_terminal_io_kbhit(struct plc_terminal_io *plc_terminal_io);
int plc_terminal_io_getchar(struct plc_terminal_io *plc_terminal_io);
```

**time.h**

```c
uint32_t plc_time_get_tick_ms(void);
struct timespec plc_time_get_hires_stamp(void);
uint32_t plc_time_hires_interval_to_usec(struct timespec t1, struct timespec t2);
```
### 3.5.2.2 libplc-gpio

<table>
<thead>
<tr>
<th>Target</th>
<th>libplc-gpio.a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>It encapsulates some useful GPIO extended functionality like callbacks-on-toggling.</td>
</tr>
</tbody>
</table>
| Details | This library doesn't rely on the PlcCape board but only on the BBB. It implements two sets of functions to access the GPIO:  
  - using the standard file system '/sys/class/gpio/...'
    - **pros**: encapsulates the access to 'gpio' through a driver. This allows a better use of shared resources. For example if a GPIO pin is already used, a cape could automatically (or manually through jumper) use a different one.
  - using direct access to memory
    - **pros**: optimum performance because not intermediate wrappers
    - **cons**: requires 'root' access. |
| Dependencies | **pthread**: some gpio functions spawn new threads  
**libplc-tools**: used to get some information from device-based files |
| API help | .libraries/libplc-gpio/api |
| Source code | .libraries/libplc-gpio |

**Extract of the provided API:**

```c
#include <gpio.h>

void plc_gpio_set_soft_emulation(int soft_emulation);

struct plc_gpio;
struct plc_gpio *plc_gpio_create(void);
void plc_gpio_release(struct plc_gpio *plc_gpio);

struct plc_gpio_pin_out;
struct plc_gpio_pin_out *plc_gpio_pin_out_create(struct plc_gpio *plc_gpio, uint32_t pin_bank, uint32_t pin_mask);
void plc_gpio_pin_out_release(struct plc_gpio_pin_out *plc_gpio_pin_out);
void plc_gpio_pin_out_set(struct plc_gpio_pin_out *plc_gpio_pin_out, int on);
void plc_gpio_pin_out_toggle(struct plc_gpio_pin_out *plc_gpio_pin_out);

struct plc_gpio_pin_in;
struct plc_gpio_pin_in *plc_gpio_pin_in_create(struct plc_gpio *plc_gpio, uint32_t pin_bank, uint32_t pin_mask);
void plc_gpio_pin_in_release(struct plc_gpio_pin_in *plc_gpio_pin_in);
int plc_gpio_pin_in_get(struct plc_gpio_pin_in *plc_gpio_pin_in);

struct plc_gpio_sysfs_pin;
struct plc_gpio_sysfs_pin *plc_gpio_sysfs_pin_create(uint32_t gpio_number, int output);
void plc_gpio_sysfs_pin_release(struct plc_gpio_sysfs_pin *plc_gpio_sysfs_pin);
int plc_gpio_sysfs_pin_get(struct plc_gpio_sysfs_pin *plc_gpio_sysfs_pin);
void plc_gpio_sysfs_pin_set(struct plc_gpio_sysfs_pin *plc_gpio_sysfs_pin, int on);
void plc_gpio_sysfs_pin_callback(struct plc_gpio_sysfs_pin *plc_gpio_sysfs_pin, void (*callback)(void *data, int flag_status), void *data);
```
3.5.2.3 libplc-adc

<table>
<thead>
<tr>
<th>Target</th>
<th>libplc-adc.a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Provides comfortable access to the high speed capturing system based on the modified ADC driver</td>
</tr>
<tr>
<td>Details</td>
<td>The functionality offered doesn't depend on the PlcCape but only on the BBB. Thus, it can be used directly on an isolated BBB or with others capes</td>
</tr>
<tr>
<td>Dependencies</td>
<td>libplc-tools: time functions used</td>
</tr>
<tr>
<td>API help</td>
<td>.libraries/libplc-adc/api</td>
</tr>
<tr>
<td>Source code</td>
<td>.libraries/libplc-adc</td>
</tr>
</tbody>
</table>

Extract of the provided API:

```
adc.h

typedef void (*rx_buffer_completed_callback_t)(
    void *data, sample_rx_t *samples_buffer, uint32_t samples_buffer_count);

float plc_adc_get_sampling_frequency(void);
struct plc_adc *plc_adc_create(int plc_driver);
void plc_adc_release(struct plc_adc *plc_adc);
void plc_adc_set_rx_buffer_completed_callback(struct plc_adc *plc_adc,
    rx_buffer_completed_callback_t rx_buffer_completed_callback,
    void *rx_buffer_completed_callback_data);
sample_rx_t plc_adc_read_sample(struct plc_adc *plc_adc);
void plc_adc_start_capture(struct plc_adc *plc_adc, uint32_t buffer_samples, int kernel_buffering);
void plc_adc_stop_capture(struct plc_adc *plc_adc);

analysis.h

struct rx_statistics
{
    uint32_t buffers_handled;
    uint32_t buffer_preparation_us;
    uint32_t buffer_preparation_min_us;
    uint32_t buffer_preparation_max_us;
    uint32_t buffer_cycle_us;
    uint32_t buffer_cycle_min_us;
    uint32_t buffer_cycle_max_us;
    sample_rx_t buffer_min;
    sample_rx_t buffer_max;
    float buffer_dc_mean;
    float buffer_ac_mean;
};

struct plc_rx_analysis *plc_rx_analysis_create(void);
void plc_rx_analysis_release(struct plc_rx_analysis *plc_rx_analysis);
void plc_rx_analysis_configure(struct plc_rx_analysis *plc_rx_analysis, float freq_adc_sps,
    uint32_t data_bit_us, sample_rx_t data_offset, sample_rx_t data_hi_threshold_detection);
void plc_rx_analysis_set_statistics_mode(struct plc_rx_analysis *plc_rx_analysis,
    enum plc_rx_statistics_mode mode);
ATTR_EXTERN int plc_rx_analysis_analyze_buffer(struct plc_rx_analysis *plc_rx_analysis,
    sample_rx_t *buffer, uint32_t buffer_samples);
84 const struct rx_statistics *plc_rx_analysis_get_statistics(struct plc_rx_analysis *plc_rx_analysis);
```
3.5.2.4 libplc-cape

<table>
<thead>
<tr>
<th>Target</th>
<th>libplc-cape.a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>It encapsulates the low-level access to the PlcCape board providing a high-level API for interacting with</td>
</tr>
<tr>
<td>Details</td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td></td>
</tr>
<tr>
<td>API help</td>
<td>.libraries/libplc-cape/api</td>
</tr>
<tr>
<td>Source code</td>
<td>.libraries/libplc-cape</td>
</tr>
</tbody>
</table>

Extract of the provided API:

```c
// cape.h
struct plc_cape *plc_cape_create(int plc_driver);
void plc_cape_release(struct plc_cape *plc_cape);
struct plc_afe *plc_cape_get_afe(struct plc_cape *plc_cape);
struct plc_leds *plc_cape_get_leds(struct plc_cape *plc_cape);

// afe.h
void plc_afe_configure(struct plc_afe *plc_afe, const struct afe_settings *settings);
char *plc_afe_get_info(struct plc_afe *plc_afe);
void plc_afe_set_flags_callback(struct plc_afe *plc_afe, void (*callback)(void *data, int tx_flag, int rx_flag, int ok_flag), void *data);
void plc_afe_set_standby(struct plc_afe *plc_afe, int standby);
void plc_afe_activate_blocks(struct plc_afe *plc_afe, enum afe_block_enum blocks);
void plc_afe_set_calibration_mode(struct plc_afe *plc_afe, enum afe_calibration_enum calibration_mode);
void plc_afe_set_gain_tx(struct plc_afe *plc_afe, enum afe_gain_tx_pga_enum gain);
void plc_afe_set_gains_rx(struct plc_afe *plc_afe, enum afe_gain_rx_pga1_enum gain1, enum afe_gain_rx_pga2_enum gain2);
void plc_afe_disable_all(struct plc_afe *plc_afe);
void plc_afe_clear_overloads(struct plc_afe *plc_afe);
uint8_t plc_afe_get_overloads(struct plc_afe *plc_afe);
void plc_afe_configure_spi(struct plc_afe *plc_afe, uint32_t spi_dac_freq, uint16_t spi_dac_delay);
void plc_afe_set_dac_mode(struct plc_afe *plc_afe, int enable);
void plc_afe_transfer_dac_sample(struct plc_afe *plc_afe, uint16_t sample);

// leds.h
void plc_leds_toggle_app_activity(struct plc_leds *plc_leds);
void plc_leds_set_app_activity(struct plc_leds *plc_leds, int on);
void plc_leds_set_tx_activity(struct plc_leds *plc_leds, int on);
void plc_leds_set_rx_activity(struct plc_leds *plc_leds, int on);

// tx.h
struct plc_tx *plc_tx_create(tx_fill_cycle_callback_t tx_fill_cycle_callback,
  tx_fill_cycle_callback_h tx_fill_cycle_callback_handle,
  tx_on_buffer_sent_callback_t tx_on_buffer_sent_callback,
  tx_on_buffer_sent_callback_h tx_on_buffer_sent_callback_handle,
  enum spi_tx_mode_enum rx_mode, uint32_t rx_buffers_len, struct plc_afe *plc_afe);
const struct tx_statistics *tx_get_tx_statistics(struct plc_tx *plc_tx);
void plc_tx_release(struct plc_tx *plc_tx);
void plc_tx_fill_buffer_iteration(struct plc_tx *plc_tx, uint32_t t, uint32_t t buffer_samples);
void plc_tx_start_transmission(struct plc_tx *plc_tx);
void plc_tx_stop_transmission(struct plc_tx *plc_tx);
```
3.6 Plugins

3.6.1 Overview

Plugins are a kind of dynamic libraries that can be loaded on-demand.

Like static libraries, plugins are a convenient way of reusing code. The file with the executable code has the extension *.so. The main differences with static libraries are:

- instead of being embedded at the linking stage (case of static libraries) the plugins are loaded at run-time
- the address of the plugin-exported functions must be found via the standard libdl library. Functions are retrieved by their name. The suitability of the parameters is not checked. Thus, special care must be taken to avoid compatibility problems. Moreover, incompatible evolutions of public functions may result in old-applications to crash (until adapted to the new interfaces). Note that, by contrast, compatibility problems are easily detected at compile-time when using static libraries
- a positive side effect of dynamic libraries (in contrast to static libraries) is that if a bug is fixed in the *.so, all the applications relying on it will automatically use the fix, with no rebuilding involved

For maintenance purposes and to guarantee compatibility these rules need to be followed:

- plugins are grouped into categories. Each category defines the exact API that plugins belonging to must satisfy
- when an API is officially released, exported methods become immutable: they cannot be removed nor modified (either in name, arguments or behavior). The only safe operation in future releases is to add new methods

3.6.2 Categories

Plugins are divided in categories depending on the target function:

<table>
<thead>
<tr>
<th>Category</th>
<th>Public API</th>
<th>Source code</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoders</td>
<td>encoder_api</td>
<td>./plugins/encoder</td>
<td>Generate RAW samples to feed a DAC from input data or given settings.</td>
</tr>
<tr>
<td>Decoders</td>
<td>decoder_api</td>
<td>./plugins/decoder</td>
<td>Generate output data from RAW samples coming from an ADC.</td>
</tr>
<tr>
<td>User Interfaces</td>
<td>ui_api</td>
<td>./plugins/ui</td>
<td>Implement a specific user interface mechanism.</td>
</tr>
</tbody>
</table>

Table 14. Plugin categories
3.6.2.1 Encoders

The mission of encoders is to generate RAW samples to feed a DAC from input data or from given settings.

Any Encoder-based plugin must implement and provide this interface:

```
encoder_api_h *create(void)

Constructor of a plugin-specific encoder object.

Returns
It must return a handle to the created object to be used on the other methods of the api interface

void release(encoder_api_h *handle)

Destructor of a previously created plugin-specific encoder object.

Parameters
handle Handle to the encoder-plugin

int begin_settings(encoder_api_h handle)

Switch the plugin to Configuration mode

Parameters
handle Handle to the encoder-plugin

Returns
0 if the Configuration mode can be started; -1 if error

int set_setting(encoder_api_h handle, enum encoder_setting_enum setting, const void *data)

Set the value of a specific setting. It requires the plugin to be in Configuration mode

Parameters
handle Handle to the encoder-plugin
setting Identifier of the setting to be configured
data New data to be set. The type of it depends on each specific setting

Returns
0 if the provided setting and value are valid; -1 if error

int end_settings(encoder_api_h handle)

End the Configuration mode process. The configuration of the plugin is then effectively updated with the settings provided.

Parameters
handle Handle to the encoder-plugin

Returns
0 if the whole configuration is coherent and can be applied; -1 if error
```
### void reset(encoder_api_h *handle)

Reset the encoder-plugin to start a new cycle from a fresh condition.

**Parameters**
- handle Handle to the encoder-plugin

### void prepare_next_samples(
encoder_api_h *handle, sample_tx_t *buffer, uint32_t buffer_count)

Request for filling next consecutive samples. Function called by the plc-cape framework each time a buffer is required to be filled with successive content.

**Parameters**
- handle Handle to the encoder-plugin
- buffer A pointer to the buffer that must be filled by the plugin
- buffer_count The length of the buffer in items

---

**Note:** The description of this interface has automatically been extracted by Doxygen from the embedded comments in the file `plugins/encoder/api/encoder.h`. 
3.6.2.2 Decoders

Decoders find data from the RAW samples coming from an ADC.

```c
decoder_api_h create(void)
```
Constructor of a plugin-specific decoder object.

**Returns**
It must return a handle to the created object to be used on the other methods of the *api* interface.

```c
void release(decoder_api_h handle)
```
Destructor of a previously created plugin-specific decoder object.

**Parameters**
- `handle` Handle to the decoder-plugin

```c
void initialize_demodulator(decoder_api_h handle, uint32_t chunk_samples,
sample_rx_t demod_data_hi_threshold, sample_rx_t data_offset, float samples_per_bit, uint32_t samples_to_file)
```
Intialize a decoding session.

**Parameters**
- `handle` Handle to the decoder-plugin

```c
void terminate_demodulator(decoder_api_h handle)
```
Terminate a decoding session.

**Parameters**
- `handle` Handle to the decoder-plugin

```c
uint32_t parse_next_samples(decoder_api_h handle, const sample_rx_t *buffer_in,
data_tx_rx_t *buffer_data_out, uint32_t buffer_data_out_count)
```
Decode a chunk of raw received samples to the correponding data.

**Parameters**
- `[in]` `handle` Handle to the decoder-plugin
- `[in]` `buffer_in` Buffer with the incoming raw samples
- `[out]` `buffer_data_out` Buffer with allocated space for the data output
- `[in]` `buffer_data_out_count` Space available in the output buffer in bytes

**Returns**
The number of data effectively decoded and stored in `buffer_data_out`
3.6.2.3 User Interface

User interfaces are not usually defined as plugins. For the plc-cape-lab application UIs has been designed like this in order to:

- provide a way to encapsulate third-party packages (as ncurses) to simplify the migration to other platforms that could miss these packages
- allow developers customizing the UI without requiring the main application to be affected
- optimize loading time and memory consumption depending on UI needs. For example, a graphical interface would need a considerable amount of resources that are not necessary if the application is going to be operated through a remote terminal session or through the command line

```c
ui_api_h create(const char *title, const struct ui_menu_item *main_menu_items, uint32_t main_menu_items_count, ui_callbacks_h callbacks_handle)
```

Constructor of a plugin-specific ui object.

**Parameters**
- title: Title of the main menu
- main_menu_items: List of main menu items and associated operations
- main_menu_items_count: Number of main menu items
- callbacks_handle: Handle sent to the menu operations for custom usage

**Returns**
- A handle to the created object

```c
void release(ui_api_h handle)
```

Destructor of a previously created plugin-specific ui object.

**Parameters**
- handle: Handle to the ui-plugin

```c
void do_menu_loop(ui_api_h handle)
```

Starts the menu loop in charge of capturing the key pressed and executing the corresponding menu operation. It's a blocking function until 'quit' is called from a different thread.

**Parameters**
- handle: Handle to the ui-plugin

```c
void refresh(ui_api_h handle)
```

Forces a refreshment of the whole user interface.

**Parameters**
- handle: Handle to the ui-plugin
### void log_text(ui_api_h handle, const char *text)

Specifies a text to be logged in the dedicated window.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Handle to the ui-plugin</td>
</tr>
<tr>
<td>text</td>
<td>The text to be displayed</td>
</tr>
</tbody>
</table>

### void set_event(ui_api_h handle, uint32_t id, uint32_t data)

Notifies the user interface that one of the predefined events (asynchronous notifications) has occurred in order for the user interface to be updated accordingly.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Handle to the ui-plugin</td>
</tr>
<tr>
<td>id</td>
<td>Identifier of the event</td>
</tr>
<tr>
<td>data</td>
<td>Extra data associated to the event</td>
</tr>
</tbody>
</table>

### void active_panel_close(ui_api_h handle)

Closes the active panel (floating menu, dialog, etc.) and then displays the parent one. Used for example when typing ESC on a menu to go back.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Handle to the ui-plugin</td>
</tr>
</tbody>
</table>

### void quit(ui_api_h handle)

Closes the menu.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Handle to the ui-plugin</td>
</tr>
</tbody>
</table>

### void open_menu(ui_api_h handle, const struct ui_menu_item *menu_items, uint32_t menu_items_count, const char *title, void(*on_cancel)(ui_callbacks_h))

Opens a sub-menu.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Handle to the ui-plugin</td>
</tr>
<tr>
<td>menu_items</td>
<td>List of menu items and associated operations</td>
</tr>
<tr>
<td>menu_items_count</td>
<td>Number of menu items</td>
</tr>
<tr>
<td>title</td>
<td>Title of the menu window</td>
</tr>
<tr>
<td>on_cancel</td>
<td>Function to be called by the plugin if the user cancels the menu (by pressing ESC or similar)</td>
</tr>
</tbody>
</table>

**Returns**

A handle to the created object
void open_dialog(ui_api_h handle, const struct ui_dialog_item *dialog_items, uint32_t dialog_items_count, const char *title, void(*on_cancel)(ui_callbacks_h), void(*on_ok)(ui_callbacks_h))

Opens a dialog box with a list of user-editable fields.

**Parameters**

- **handle**
  - Handle to the ui-plugin
- **dialog_items**
  - List of dialog fields
- **dialog_items_count**
  - Number of dialog fields
- **title**
  - Title of the dialog window
- **on_cancel**
  - Function to be called by the plugin if the user cancels the dialog (by pressing ESC or similar)
- **on_ok**
  - Function to be called by the plugin if the user validates the dialog (by pressing RETURN or similar)

void set_info(ui_api_h handle, enum ui_info_enum info_type, const void *info_data)

Sets a predefined-type information in order the user interface decides if rendering it and how.

**Parameters**

- **handle**
  - Handle to the ui-plugin
- **info_type**
  - The category of the information to show
- **info_data**
  - Data associated to the 'info_type'
3.6.3 Plugins developed

For this project these plugins have been developed:

## ENCODERS

<table>
<thead>
<tr>
<th>Name</th>
<th>Source code</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>encoder-wave</td>
<td>./plugins/encoder/encoder-wave</td>
<td>Typical waves generator.</td>
</tr>
<tr>
<td>encoder-pwm</td>
<td>./plugins/encoder/encoder-pwm</td>
<td>Basic PWM modulator.</td>
</tr>
<tr>
<td>encoder-ook</td>
<td>./plugins/encoder/encoder-ook</td>
<td>Basic modulator of data in OOK codifications.</td>
</tr>
</tbody>
</table>

Table 15. Plugins developed within the 'encoder' category

## DECODERS

<table>
<thead>
<tr>
<th>Name</th>
<th>Source code</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decoder-pwm</td>
<td>./plugins/decoder/decoder-pwm</td>
<td>Basic demodulator of data in PWM codifications.</td>
</tr>
<tr>
<td>decoder-ook</td>
<td>./plugins/decoder/decoder-ook</td>
<td>Basic demodulator of data in OOK codifications.</td>
</tr>
</tbody>
</table>

Table 16. Plugins developed within the 'decoder' category

## USER INTERFACES

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Source code</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui-ncurses</td>
<td>./plugins/ui/ui-ncurses</td>
<td>Basic UI plugin based on the ncurses package.</td>
</tr>
<tr>
<td>ui-console</td>
<td>./plugins/ui/ui-console</td>
<td>Basic UI plugin with the classical console look and feel.</td>
</tr>
</tbody>
</table>

Table 17. Plugins developed within the 'user interface' category

All these plugins are in an early development stage. They aren’t optimized but are good-enough to test the feasibility of the whole system. In particular encoders and decoders should be optimized for computational efficiency and use better modulation/demodulation algorithms for reinforced robustness and protection against interferences, noise, attenuation, etc.

In the following sections a brief description of all the plugins implemented for this project will be given. Fragments of the source code in the core functions will also be included because it can give a quick idea of what’s implemented and the complexity around.

For encoders the core function is: `prepare_next_samples`

For decoder the core function is: `parse_next_samples`

In User Interfaces there is not a single core function, so, just basic concepts will be introduced.

For additional info refer to the source code.
3.6.3.1 encoder-wave

<table>
<thead>
<tr>
<th>Target</th>
<th>encoder-wave.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Generator of typical waves: sinusoids, ramps, constant values, ...</td>
</tr>
<tr>
<td>Details</td>
<td>Encodings supported:</td>
</tr>
<tr>
<td></td>
<td>• Sinuosoid</td>
</tr>
<tr>
<td></td>
<td>• Ramps</td>
</tr>
<tr>
<td></td>
<td>• Triangular waves</td>
</tr>
<tr>
<td></td>
<td>• ...</td>
</tr>
<tr>
<td>Source code</td>
<td>./plugins/encoder/encoder-wave</td>
</tr>
</tbody>
</table>

This plugin generates the raw samples of the typical waves depending on the settings provided.

<table>
<thead>
<tr>
<th>Stream type</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>Fills the output buffer with just a constant value</td>
</tr>
<tr>
<td>freq_max</td>
<td>Fills the output buffer with alternating values between min and max. It produces a signal with the maximum frequency the DAC can generate, which is $freq_{dac_sps}/2$ (Nyquist frequency)</td>
</tr>
<tr>
<td>freq_max_div2</td>
<td>Fills the output buffer with the repetitive pattern $[0, \text{max_value}, 0, \text{min_value}, 0, \text{max_value}…]$. This is a computational efficient way of generating an output frequency equivalent to $freq_{dac_sps}/4$.</td>
</tr>
<tr>
<td>sinus</td>
<td>Fills the output buffer with a configurable sinusoid</td>
</tr>
<tr>
<td>ramp</td>
<td>Fills the output buffer with a ramp</td>
</tr>
<tr>
<td>triangular</td>
<td>Fills the output buffer with a triangular wave</td>
</tr>
<tr>
<td>stream_file</td>
<td>Fills the output buffer with the samples read from a file. The file is restarted to repetitively fill the output buffer</td>
</tr>
<tr>
<td>bit_padding_per_cycle</td>
<td>Fills the output buffer by introducing 1-bit after each DAC sample. In other words, this repetitive pattern is followed: $[0x1, 0x3, 0x7, 0x15… 0x3FF, 0x1…]$. It can be helpful for testing purposes</td>
</tr>
<tr>
<td>freq_sweep</td>
<td>Generates a sinusoid with a frequency which varies in time</td>
</tr>
<tr>
<td>am_modulation</td>
<td>Generates a sinusoid with an amplitude which varies in time</td>
</tr>
</tbody>
</table>

*Table 18. Type of waves generated by the 'encoder-wave' plugin*
This is a fragment of the main function in the plugin in charge of generating the raw data:

```c
void encoder_prepare_next_samples(struct encoder *encoder, sample_tx_t *buffer,
                                  uint32_t buffer_count)
{
    switch (encoder->settings.stream_type)
    {
    case stream_ramp:
        for (i = 0; i < buffer_count; i++, encoder->stream_ramp.value_last += encoder->stream_ramp.value_delta)
        {
            if (encoder->stream_ramp.value_last >= encoder->stream_ramp.value_max)
                encoder->stream_ramp.value_last -= encoder->settings.range;
            buffer[i] = (sample_tx_t) round(encoder->stream_ramp.value_last);
        }
        break;
    ...
    case stream_constant:
        for (i = 0; i < buffer_count; i++)
            buffer[i] = encoder->settings.offset;
        break;
    case stream_freq_max:
        {
            int nMin = encoder->settings.offset - encoder->settings.range / 2 + 1;
            int nMax = encoder->settings.offset + encoder->settings.range / 2 - 1;
            for (i = 0; i < buffer_count; i++)
                buffer[i] = (i % 2) ? nMin : nMax;
        }
        break;
    case stream_freq_sinus:
        {
            float amp = (encoder->settings.range - 1) / 2;
            float freq = 2.0 * M_PI * (float) encoder->settings.freq / encoder->freq_dac_sps;
            for (i = 0; i < buffer_count; i++, encoder->counter++)
                buffer[i] = (sample_tx_t) round((float) encoder->settings.offset + amp * sin(freq * encoder->counter));
        }
        break;
    ...
    }
}
```
3.6.3.2 encoder-pwm

<table>
<thead>
<tr>
<th>Target</th>
<th><code>encoder-pwm.so</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Encodes data in a simple Pulse Width Modulation codification</td>
</tr>
<tr>
<td>Details</td>
<td>Configurable settings</td>
</tr>
<tr>
<td></td>
<td>• <code>us_per_bit</code></td>
</tr>
<tr>
<td></td>
<td>• <code>guard_time</code></td>
</tr>
<tr>
<td>Source code</td>
<td><code>.plugins/encoder/encoder-pwm</code></td>
</tr>
</tbody>
</table>

The `encoder-pwm` plugin generates a sinusoidal carrier of configurable frequency with duration proportional to the value of the incoming data.

To check it, we can use the `plc-cape-lab` tool with the following parameters:

- AFE mode = `calib_dac_txpga`
- SPI bitrate = 1.5Mbps
- Carrier freq = 1kHz, DC Offset=800, AC Range=400
- Store to file = 20000 samples
- Msg to encode = `~Hi!`

This message incorporates 4 characters distributed among the ASCII code in a way we can appreciate the PWM encoding implementation.

![ADC capture](image)

*Figure 36. Example of PWM-encoded message*

We can see in the top, in dark green, the ASCII codes of the characters sent, and in the bottom, in dark red, the corresponding characters.
This is the main routine in charge of composing the encoded data:

```c
void encoder_prepare_next_samples(struct encoder *encoder, sample_tx_t *buffer,
                                  uint32_t buffer_count)
{
    int i;
    for (i = 0; i < buffer_count; i++)
    {  
        if (encoder->guard_samples == 0)
        {
            buffer[i] = (sample_tx_t) round(
                encoder->settings.offset + encoder->amp * sin(encoder->freq * (encoder->pwm_counter++)));
            if (encoder->pwm_counter == encoder->pwm_next)
                encoder->guard_samples = GUARD_SAMPLES;
        }
        else
        {
            buffer[i] = (sample_tx_t) round(encoder->settings.offset);
            if (!encoder->message_end_reached)
            {
                encoder->guard_samples--;  
                if (encoder->guard_samples == 0)
                {
                    encoder->pwm_counter = 0;
                    if (encoder->message_index == encoder->message_length)
                    {
                        encoder->message_index = 0;
                        if (!encoder->settings.loop_message)
                        {
                            encoder->message_end_reached = 1;
                            encoder->guard_samples = (uint16_t) -1;
                        }
                    }
                    encoder->pwm_next = round(
                        encoder->samples_per_bit * encoder->settings.message[encoder->message_index++]);
                }
            }
        }
    }
}
```
3.6.3.3 decoder-pwm

<table>
<thead>
<tr>
<th>Target</th>
<th>decoder-pwm.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Decodes data from a simple Pulse Width Modulation codification</td>
</tr>
<tr>
<td>Details</td>
<td>Configurable settings</td>
</tr>
<tr>
<td></td>
<td>• us_per_bit</td>
</tr>
<tr>
<td></td>
<td>• guard_time</td>
</tr>
<tr>
<td>Source code</td>
<td>./plugins/decoder/decoder-pwm</td>
</tr>
</tbody>
</table>

This plugin decodes PWM-encoded data. It’s the dual of the encoder-pwm plugin.

This is the main function in charge of the decoding:

```c
uint32_t decoder_parse_next_samples(struct decoder *decoder, const sample_rx_t *buffer_in, uint8_t *buffer_data_out, uint32_t buffer_data_out_count) {
    uint8_t *buffer_data_out_ini = buffer_data_out;
    const sample_rx_t *buffer_in_end = buffer_in + decoder->chunk_samples;
    for (; buffer_in < buffer_in_end; buffer_in++) {
        int hi_level_detected = (abs((int16_t) *buffer_in - (int16_t) decoder->offset) >= decoder->carrier_threshold);
        if (decoder->samples_with_carrier == 0) {
            if (hi_level_detected)
                decoder->samples_with_carrier++;
        } else {
            decoder->samples_with_carrier++;
            if (hi_level_detected) {
                // Whenever carrier detected reset the 'samples_without_carrier' counter
                decoder->samples_without_carrier = 0;
            } else {
                if (++decoder->samples_without_carrier >= decoder->stop_samples) {
                    // Reject the spurius
                    if (decoder->samples_with_carrier < round(decoder->samples_per_bit / 2.0)) {
                        decoder->samples_with_carrier = 0;
                        decoder->samples_without_carrier = 0;
                    } else {
                        // Adjust removing the stop-samples extra-counting
                        decoder->samples_with_carrier -= decoder->stop_samples;
                        *buffer_data_out++ = (uint8_t) round((
                            decoder->samples_with_carrier / decoder->samples_per_bit);
                        decoder->samples_with_carrier = 0;
                        decoder->samples_without_carrier = 0;
                    }
                } else {
                    // Adjust removing the stop-samples extra-counting
                    decoder->samples_with_carrier -= decoder->stop_samples;
                    *buffer_data_out++ = (uint8_t) round((
                        decoder->samples_with_carrier / decoder->samples_per_bit);
                    decoder->samples_with_carrier = 0;
                    decoder->samples_without_carrier = 0;
                }
            }
        }
    }
    return buffer_data_out - buffer_data_out_ini;
}
```
3.6.3.4 encoder-ook

<table>
<thead>
<tr>
<th>Target</th>
<th>encoder-ook.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Encodes data in a simple On-Off-Keying codification</td>
</tr>
<tr>
<td>Details</td>
<td>Encodes data in a simple On-Off-Keying codification</td>
</tr>
<tr>
<td>Source code</td>
<td>../plugins/encoder/encoder-ook</td>
</tr>
</tbody>
</table>

For the analysis in this project I’ve basically used the PWM encoding (encoder-pwm and decoder-pwm plugins) because it’s very simple to identify in the oscilloscope captures. However, a basic OOK modulation has also been provided to prove the adaptability of the framework to other codifications.

The core implementation is quite simple: a carrier is generated or not depending on each bit to be sent. A start bit is also required to delimit each character (like typical UARTs\(^7\)).

```c
void encoder_prepare_next_samples(struct encoder *encoder, sample_tx_t *buffer, uint32_t buffer_count)
{
  int i;
  for (i = 0; i < buffer_count; i++)
  {
    if (((encoder->guard_bits == 0) && ((encoder->bit_index == 0xFF) || ((encoder->settings.message[encoder->message_index] << encoder->bit_index) & 0x80))
    {
      buffer[i] = (sample_tx_t) round(
        encoder->settings.offset + encoder->amp * sin(
          encoder->freq * (encoder->counter - encoder->counter_bit)));
    }
    else
    {
      buffer[i] = (sample_tx_t) round(encoder->settings.offset);
    }
    encoder->counter++;
    if ((encoder->counter % encoder->samples_per_bit) == 0)
    {
      if (encoder->guard_bits > 0)
      {
        encoder->guard_bits--;
      }
      else
      {
        if (++encoder->bit_index == 8)
        {
          encoder->guard_bits = GUARD_BITS;
          encoder->bit_index = 0xFF;
          // -1 because the '\0' ending
          if (++encoder->message_index == encoder->message_length)
            encoder->message_index = 0;
        }
        // 'encoder->counter' to 0 to reset frequency phase per symbol.
        encoder->counter_bit = encoder->counter;
      }
    }
  }
}
```

### 3.6.3.5 decoder-ook

<table>
<thead>
<tr>
<th>Target</th>
<th>decoder-ook.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Decodes data from a simple On-Off-Keying codification</td>
</tr>
<tr>
<td>Details</td>
<td>This plugin includes some code that applies a low pass filter to the input (either through FIR or IIR). It’s not optimized but it gives an overview of how digital filters can be implemented in C.</td>
</tr>
<tr>
<td>Source code</td>
<td>./plugins/encoder/decoder-ook/decoder-ook.c</td>
</tr>
</tbody>
</table>

This plugin includes some code that applies a low pass filter to the input (either through FIR or IIR). It’s not optimized but it gives an overview of how digital filters can be implemented in C.

```c
uint32_t decoder_parse_next_samples_buffer(struct decoder *decoder, const sample_rx_t *buffer_in, uint8_t *buffer_data_out, uint32_t buffer_data_out_count) {
    // CHUNK_PUSH: Filtering
    // int n;
    float *in_f = decoder->buffer_in_f + decoder->iir_b_count;
    float *out_f = decoder->buffer_out_f + decoder->iir_a_count;
    // Chain previous in/out values
    for (n = 0; n < decoder->iir_b_count; n++)
        decoder->buffer_in_f[n] = decoder->buffer_in_f[decoder->chunk_samples + n];
    for (n = 0; n < decoder->iir_a_count; n++)
        decoder->buffer_out_f[n] = decoder->buffer_out_f[decoder->chunk_samples + n];
    // Push new values
    // Demodulation
    for (n = 0; n < decoder->chunk_samples; n++)
        { 
            in_f[n] = abs(buffer_in[n] - decoder->data_offset);
            out_f[n] = 0.0;
        }
    // Apply filter
    // Filtering algorithm (1-based):
    // y[n]*iir_a[k] = Sum(k=0..M) [iir_b[k+1]*x[n-k]] - Sum(k=1..N) [iir_a[k+1]*y[n-k]]
    // N=len(iir_a)-1; M=len(iir_b)-1
    // More info about filtering can be found in 'Octave' in 'doc filter'
    for (n = 0; n < decoder->chunk_samples; n++)
        { 
            int k;
            for (k = 0; k < decoder->iir_b_count; k++)
                out_f[n] += decoder->iir_b[k] * in_f[n - k];
            for (k = 1; k < decoder->iir_a_count; k++)
                out_f[n] -= decoder->iir_a[k] * out_f[n - k];
            out_f[n] /= decoder->iir_a[0];
            // Quantification
            decoder->buffer_out_filter[n] = (sample_rx_t) round(out_f[n]);
        }
    // CHUNK_SIGNAL_TO_DATA: Interpretation
    // if (buffer_data_out_count == 0)
    //    return 0;
    uint8_t *buffer_data_out_end = buffer_data_out + buffer_data_out_count;
    uint8_t *buffer_data_out_cur = buffer_data_out;
    while (1) ... 
```

---

FIR and IIR are computationally efficient ways of implementing filters in source code:
The proper behavior of the pair encoder-ook and decoder-ook has been checked with the plc-cape-lab application with the following configuration:

- calibration mode (i.e. internal bridge between TX and RX)
- low frequency carrier of 10kHz to simplify the analysis with Octave
- SPI clock = 750k bps
- Message to be send and received continuously: Hi
- ADC recording = 50k samples

This is a screenshot of the plc-cape-lab with the results of the test:

![Screenshot of plc-cape-lab](image)

We can see that the message has been properly decoded: h.HiHiHiHiHiHiH. We see an error at beginning of the stream (h.). That’s due to the required ADC setup time.

With Octave we can take a look at the first 50k ADC stored samples and identify there the symbols transmitted with the OOK modulation. In the following figure we have three graphs:

- the 50k samples received
- the 50k samples after having applied the IIR low-pass filter
- the decoded data
Data communication through power lines using a system based on BeagleBone

When zooming, we can see the composition of the H and i symbols in detail (1 start-bit + 8 data bits):

If we zoom to one bit of data, we can see the carrier and check that has the expected frequency (10kHz):

Figure 38. OOK captured data + IIR filtering result + Data decoded

Figure 39. OOK encoding for the 'H' character = 72 = 0b01001000

Figure 40. OOK encoding for the 'i' character = 105 = 0b01101001

Figure 41. OOK 10kHz carrier detail
### 3.6.3.6 ui-ncurses

<table>
<thead>
<tr>
<th>Target</th>
<th>ui-ncurses.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Implement a menu-oriented user interface based on the ncurses package</td>
</tr>
<tr>
<td>Details</td>
<td>Features implemented</td>
</tr>
<tr>
<td></td>
<td>• <strong>Popup menus</strong>: the main application provides the hierarchy of the menus and operations related</td>
</tr>
<tr>
<td></td>
<td>• <strong>Popup dialogs</strong>: allows edition of multi-fields in a generic and comfortable way</td>
</tr>
<tr>
<td></td>
<td>• <strong>Popup list boxes</strong>: asks the user for a choice among a list of predefined options</td>
</tr>
<tr>
<td>Source code</td>
<td>./plugins/ui/ui-ncurses</td>
</tr>
</tbody>
</table>

ncurses is a GNU package that simplifies the task of developing menu-based user interfaces. To install the package on the BBB you can follow the instructions given in 8.7 Required packages: ncurses.

Just to have a basic idea, find following a fragment of the function initializing the user interface:

```c
struct tui *tui_create(const char *title, const struct ui_menu_item *main_menu_items, uint32_t main_menu_items_count, ui_callbacks_h callbacks_handle)
{
    struct tui *tui = (struct tui*) calloc(1, sizeof(struct tui));
    // Start ncurses mode
    initscr();
    // hide_cursor
curs_set(0);
    // Use 'cbreak' to hook CTRL+C
    // cbreak();
    noecho();
    // We get F1, F2 etc..
    keypad(stdscr, TRUE);
    // ncurses uses by default "ESCaPse sequences" to manage the combination ALT+<key>
    // A simpler alternative is to use 'set_escdelay(0)' is the ALT+<key> is not used
    if (has_colors())
        {
            start_color();
            init_pair(COLOR_ID_RED, COLOR_RED, COLOR_BLACK);
            init_pair(COLOR_ID_YELLOW, COLOR_YELLOW, COLOR_BLACK);
            init_pair(COLOR_ID_RED_INVERTED, COLOR_BLACK, COLOR_RED);
        }
    // Menu Window
    tui->callbacks_handle = callbacks_handle;
    struct tui_menu *tui_menu = tui_menu_create(
        NULL, main_menu_items, main_menu_items_count, title, NULL, callbacks_handle);
    int menu_y, menu_x, menu_rows, menu_cols;
    tui_set_active_panel(tui, tui_menu_get_as_panel(tui_menu));
    tui->active_panel->tui_panel_get_dimensions(tui->active_panel,
        &menu_y, &menu_x, &menu_rows, &menu_cols);
    struct tui_rect layout[tui_windows_COUNT];
    tui->main_menu_width = menu_x + menu_cols - 1;
    ...
```

If an application based on ncurses is unexpectedly aborted, it is possible that the console output goes to an undefined state exposing some strange behavior. For example you can face invisible typing or wrong indentations. If so, execute the reset command on the console to reinitialize it.
3.6.3.7 ui-console

<table>
<thead>
<tr>
<th>Target</th>
<th>ui-console.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Implement a menu-oriented user interface based on a sequential classical terminal</td>
</tr>
<tr>
<td>Details</td>
<td></td>
</tr>
<tr>
<td>Source code</td>
<td>./plugins/ui/ui-console</td>
</tr>
</tbody>
</table>

This plugin (just partially implemented) offers a basic console-based UI.

The output is based on the classical sequential `printf`. The user-friendliness of this implementation is much worse than then one offered by `ui-ncurses` but could be an alternative on platforms not having the possibility to install the `ncurses` package.

![Figure 42. Main screen of the 'plc-cape-lab' when running with ui-console](image)

Just to have a basic idea of the implementation, find following a fragment of the typical source code:

```c
plugins/ui/ui-console/tui.c

struct tui *tui_create(const char *title, const struct ui_menu_item *main_menu_items,
    uint32_t main_menu_items_count, ui_callbacks_h callbacks_handle)
{
    struct tui *tui = (struct tui*) calloc(1, sizeof(struct tui));
    tui->plc_terminal_io = plc_terminal_io_create();
    tui->callbacks_handle = callbacks_handle;
    tui_open_menu(tui, main_menu_items, main_menu_items_count, "Main Menu", NULL);
    return tui;
}

void tui_log_text(struct tui *tui, const char *text)
{
    // static uint32_t log_counter = 0;
    // fprintf(stdout, "  [%03d] %s\n", log_counter++, text);
    // "." just as an indicator the subsequent text is logging text
    putchar('.');
    fputs(text, stdout);
}
```
3.7 Applications

3.7.1 Overview

Different applications have been developed to interact with the PlcCape. Applications use the functionality provided by static libraries and plugins.

The applications have been designed to be executed directly from a remote console.

3.7.2 Applications execution

The applications developed for this project usually depend on the custom drivers that must be loaded before launching them. To do that a dedicated script has been written. In future versions this should be automated within the C code.

Find following the script used because it’s important to understand how the current framework is initialized:

```bash
#!/bin/bash
if [ $# != 2 ]
then
    echo "Error: incorrect params"
    echo "Usage: $(basename "$0") PlcCapeVersion BBBVersion"
    exit 1
fi

# Symbol definition for overlays
export SLOTS=/sys/devices/bone_capemgr.*/slots

# Load modules
if ! (lsmod | grep -q spidev_plc)
then
    # # Load SPI modules
    insmod $DEV_BIN_DIR/drivers/spi/edma_plc.ko
    insmod $DEV_BIN_DIR/drivers/spi/spi-omap2-mcspi_plc.ko
    insmod $DEV_BIN_DIR/drivers/spi/spidev_plc.ko

    if [ $1 = 1 ]; then
        # Overlay for PLC_CAPE Version 1 (custom SPI+DMA)
        echo PLC_CAPE_V1-SPI0 > $SLOTS
    elif [ $1 = 2 ]; then
        # Overlay for PLC_CAPE Version 2 (SPI with D0 & DI std + DMA)
        echo PLC_CAPE_V2-SPI0 > $SLOTS
    else
        echo "Unknown PlcCapeVersion param"
        exit 1
    fi
fi

if ! (lsmod | grep -q adc_hs)
then
    # # ADC
    #
    # Load ADC modules
    insmod $DEV_BIN_DIR/drivers/adc/ti_am335x_tscadc_plc.ko
    insmod $DEV_BIN_DIR/drivers/adc/adc_adc_hsi.ko

    # Overlay to enable custom ADC
```

Figure 43. Applications tree
Data communication through power lines using a system based on BeagleBone

```bash
echo PLCCAPE-ADC > $SLOTS

# Create device node
# NOTE:
# * In BBB1 assigned node to ADC = 241
# * In BBB2 assigned node to ADC = 239
if [ $2 = 1 ]; then
  mknod /dev/adchs_plc c 241 0
elif [ $1 = 2 ]; then
  mknod /dev/adchs_plc c 239 0
else
  echo "Unknown BBBVersion param"
  exit 1
fi
```

If an application is based on the custom SPI and ADC drivers (the usual case) the above script must be executed before to load them (it has to be done once after each reboot). There are two facts that need to be considered in the script:

- As already seen, the PlcCape-v1 has the DIN and DOUT pins of the SPI bus toggled compared to PlcCape-v2. The configuration of these pins come in the corresponding Device Tree Overlay. The best way to deal with this would be by implementing a system that automatically detects the version of the cape and loads the corresponding DTO, but for the moment it must be done manually, by specifying the version of the cape as the first parameter of the script (1 for PlcCape-v1, 2 for PlcCape-v2)

- Concerning the ADC driver we need to create a node in the file system (with `mknod`) specifying the major version assigned to the device. In future versions this should be done automatically but for the moment we can rely on hard-coded numbers. Experimentally, it has been found that the older BBB always assigns 241 to it; the newer allocates 239. The version of the BBB (1 for old, 2 for new) must be provided as a second parameter.

A `run.sh` script is also provided in the root folder of each application as a launcher shortcut:

```
root@beaglebone:/mnt/plc-cape-src/applications/plc-cape-lab# ./run.sh 2 2
```

### 3.7.3 Applications developed

For this project these applications have been developed:

<table>
<thead>
<tr>
<th>Category</th>
<th>Source code</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plc-cape-demo-lib-usage</td>
<td>./applications/plc-cape-demo-lib-usage</td>
<td>Basic demo application to show how to link and use the libplc-xxx libraries.</td>
</tr>
<tr>
<td>plc-cape-autotest</td>
<td>./applications/plc-cape-autotest</td>
<td>PlcCape board autotest.</td>
</tr>
<tr>
<td>plc-cape-lab</td>
<td>./applications/plc-cape-lab</td>
<td>Laboratory tool to work with the PlcCape board.</td>
</tr>
</tbody>
</table>

*Table 19. Applications developed*

In the following sections a more detailed description of the applications developed will be provided.
3.7.3.1 plc-cape-demo-lib-usage

<table>
<thead>
<tr>
<th>Target</th>
<th>plc-cape-demo-lib-usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Basic example to show how to create a standalone application that uses the plc-cape libraries, one of the key points of this project.</td>
</tr>
<tr>
<td>Details</td>
<td>The application just measures the execution time of a function using the functionality encapsulated by the libplc-tools library</td>
</tr>
<tr>
<td>Source code</td>
<td>./applications/plc-cape-demo-lib-usage</td>
</tr>
</tbody>
</table>

Static libraries can be easily called from any building unit in the project (applications, plugins, other static libraries).

The following example is a very basic application that just displays the time consumed by a function (usleep in the code below). It is based on a function provided the libplc-tools, plc_time_get_tick_ms, which gives an accurate stamp of current time in milliseconds. It is helpful to illustrate how to use an existing library function within the plc-cape project:

```c
#include <unistd.h>  // usleep
#include "+common/api/+base.h"
#include "libraries/libplc-tools/api/time.h"

int main(void)
{
    uint32_t tick_ini = plc_time_get_tick_ms();
    usleep(10000);
    uint32_t tick_end = plc_time_get_tick_ms();
    printf("'usleep' executed in %d ms\n", tick_end - tick_ini);
    return 0;
}
```

```makefile
ADDITIONAL_LIBS = -lrt
ADDITIONAL_PLC_LIBS = plc-tools
TARGET = $(notdir $(CURDIR))
include $(DEV_SRC_DIR)/+common/make_object.mk
```

We can build the application by running make:

```
# cd $DEV_SRC_DIR/applications/plc-cape-demo-lib-usage
# make
  -> /mnt/plc-cape-bin/applications/plc-cape-demo-lib-usage/main.o
```

And then we can execute the new generated application:

```
# cd $DEV_BIN_DIR/applications/plc-cape-demo-lib-usage
# ./plc-cape-demo-lib-usage
'usleep' executed in 10 ms
```
3.7.3.2 plc-cape-autotest

<table>
<thead>
<tr>
<th>Target</th>
<th>plc-cape-autotest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Application to test a PlcCape board</td>
</tr>
<tr>
<td>Details</td>
<td>It checks that a PlcCape board is properly working through a series of guided tests</td>
</tr>
<tr>
<td>Source code</td>
<td>./applications/plc-cape-autotest</td>
</tr>
</tbody>
</table>

This tool shows an initial menu where the user can check the different blocks interactively.

![Figure 44. Main screen of the 'plc-cape-autotest' application](image)

Tests currently implemented:

<table>
<thead>
<tr>
<th>Test area</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFE LEDs</td>
<td>The three LEDs on the PlcCape-v2 board are flashed one after the other with one-second interval</td>
</tr>
<tr>
<td>AFE Info</td>
<td>Checks that the AFE031 answers with the correct expected identifier. It would confirm that the SPI block and the AFE are both working properly</td>
</tr>
<tr>
<td>BBB ADC</td>
<td>Continuously display the value captured by the ADC of the BBB. Useful to test the ADC without the PlcCape</td>
</tr>
<tr>
<td>AFE DAC steps + BBB ADC [AFE in TX RX LOOP]</td>
<td>The AFE is put on internal routing mode: the TX samples (from DAC) are routed to the RX pins which are connected to the ADC of the BBB. A constant value is sent each second and the ADC value captured displayed</td>
</tr>
<tr>
<td>AFE DAC freq=0.5 [AFE in TX]</td>
<td>Sends an alternating sequence of values to the DAC (hence freq=0.5) that can be checked with an external oscilloscope</td>
</tr>
<tr>
<td>AFE DAC freq=0.25 [AFE in TX]</td>
<td>Sends repeatedly the pattern [Max/2, Max, Max/2, 0] to the DAC (which is a periodic wave of freq=0.25*freq_dac_sps) that can be checked with an external oscilloscope</td>
</tr>
<tr>
<td>AFE DAC freq=0.25 + BBB ADC + File-capturing</td>
<td>Like in AFE DAC freq=0.25 but in loop-mode (TX &amp; RX internally connected). The captured data by the ADC is stored in a file to be checked with Octave</td>
</tr>
</tbody>
</table>

Table 20. Tests available in the 'plc-cape-autotest' application
3.7.3.3 plc-cape-lab

<table>
<thead>
<tr>
<th>Target</th>
<th>plc-cape-lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Laboratory tool to explore the main features offered by the PlcCape board</td>
</tr>
<tr>
<td>Details</td>
<td>It is the main application for &quot;laboratory work&quot;. It offers access to the different configurable options of the AFE031. Features:</td>
</tr>
<tr>
<td></td>
<td>• Multi wave generation</td>
</tr>
<tr>
<td></td>
<td>• Real-time and deferred DAC transmission</td>
</tr>
<tr>
<td></td>
<td>• Real-time and deferred ADC reception</td>
</tr>
<tr>
<td></td>
<td>• Capturing to file in Octave-compatible format for post-analysis</td>
</tr>
<tr>
<td></td>
<td>• Time measurements</td>
</tr>
<tr>
<td></td>
<td>• Gains...</td>
</tr>
</tbody>
</table>

| Source code  | ./applications/plc-cape-lab |

3.7.3.3.1 Main concepts
The plc-cape-lab is the core application of this project. It implements different tests and tools to analyze the behavior of the BBB + PlcCape assemblies.

It offers a text-based user interface (TUI) relying on the ncurses package for a user-friendly menu-based interface. The main screen looks like this:

![Figure 45. Main screen of the 'plc-cape-lab' application (running with 'ui-ncurses')](image)

A graphical-based plugin (GUI) could be developed in a future version to provide some extra functionality, like showing the ADC captured data without requiring external tools.
Note that a TUI has also some advantages over a GUI. For example, it can be executed remotely from a
desktop PC through a simple SSH session.

`plc-cape-lab` depends only on standard built-in libraries. It logically depends on the `ncurses` package when
using the `ui-ncurses` plugin (the usual operating mode). When completed, the `ui-console` plugin could be
used as an alternative on systems that doesn’t provide `ncurses`.

### 3.7.3.3.2 Features

The `plc-cape-lab` application implements the following features:

- **AFE031 functionality**
  - TX and RX programmable gains
  - CENELEC band configuration (A or B/C/D)
  - Calibration modes
  - Asynchronous error notification: thermal & current protection flag
  - Per block enabling/disabling control to optimize consumption
- **Signal generation (TX) in real-time or deferred mode**
- **Signal capture (RX) in real-time or deferred mode**
- **Periodic transmission (to send short packets of data at intervals)**
- **Modulation/demodulation (encoding/decoding)**
- **Plugins management**
  - Dynamic loading and unloading of encoders and decoders
  - Automatically generated lists of available plugins for user selection
- **LED control**
- **Monitoring**
  - Errors: buffer underflow conditions
  - Processing Time: reporting of the time required to prepare a buffer of samples when using the
current encoder/decoder
  - Input signal statistics: reporting of some real-time measurements, as the AC mean and DC mean
  of each buffer received. It is specially useful to simplify the PlcCape tuning. It can also be
disabled to reduce the involved CPU load
- **Profiles concept, to predefined sets of settings**

### 3.7.3.3.3 Basic operation

The recommended way to start the application is by using the `run.sh` script (specifying the version of the
BBB and PlcCape as indicated in 3.7.2 Applications execution):

```
# cd $DEV_SRC_DIR/applications/plc-cape-lab
# ./run.sh 2 2
```

Then the main screen shown above is displayed.
**Main screen**

In the main screen of the application we have four areas:

| Main menu (in the left side) | With Up and Down arrows we can move among the different menu items
|                            | With RETURN we open pop-up sub-menus or execute operations
|                            | The characters in square brackets (or numbers, when indicated) are shortcuts for the menu items
| Settings (in the right)     | This panel displays all the current configured settings grouped by scope area: settings affecting the transmission (TX), the reception (RX), the AFE031 generic behavior, etc.
| Log window (in the center)  | This is a generic-purpose output console (a “stdout”). All the messages reported by the application are shown here. The real-time received data is also reported here if the applications is configured to do so.
| Status bar (at bottom)      | The bottom line of the screen is used to report some brief real-time data, like statistics about the received data, DMA buffers processed on transmission, some AFE031 flags, etc.

| Table 21. Main screen panels |

**Main menu**

The Main menu at the left offers these options:

| ON    | Starts the communication (transmission and/or reception of data) |
| OFF   | Stops the communication |
| Profiles | Configures the application by selecting one of the predefined profiles |
| Settings Global | Configuration of some generic-purpose settings |
| Settings TX | Configuration of transmission-related settings |
| Settings RX | Configuration of reception-related settings |
| Monitoring profiles | Offers the possibility to select the type of information to show in the status bar (processing time, data received statistics, etc.) |
| Tools | Several one-shot operations |
| Quit | Terminates the application |

| Table 22. Main menu options |

Most options in the main menu open pop-up sub-menus. Following, there are some captures enclosed with a brief explanation about.
This sub-menu shows first a categorization of the different available profiles, as:

**TX+RX CAL**: Profiles based on the AFE031 calibration modes that internally route the transmitter to the receiver. These profiles allow us to easily check the whole communication path: data modulation + TX + RX + data demodulation

**TX**: Profiles that only enable the low-power transmitter blocks (DAC + TxPGA + TxFilter). The generated signal can be measured with an oscilloscope on the TX_F_OUT pin (see 2.5 AFE031)

**TX+PA**: Like in the TX profile but also enabling the Power Amplifier stage. So, this will generate amplified output in the PA_OUT pin

**TX+PA+RX**: Like in TX+PA but also enabling the reception blocks. This configuration can be useful to do a TX-RX loop (like in the TX+RX CAL profile) but using external circuitry.

**RX**: Profiles that only enable the reception-related blocks

**TX+RX CAL PROFILES**

This is an example of the list of profiles we get in the TX+RX CAL category.

For example we have TX+RX CAL 2kHz, a typical profile for quick testing and check that all the system works OK.

It is pre-configured with these settings:

- The AFE is configured to operate in the calibration mode that enables the DAC and the TxPGA blocks (shown in the UI as *calib_dac_txpga*)

- At TX it generates a sinusoid of 2 kHz with a DC Offset value of 800 (in DAC units) and an AC Range of 400 (in DAC units too). The SPI rate is configured at 1.5Mbps, which allows for a maximum wave frequency of 69 kHz (because the 10-bits per sample and the Nyquist frequency).

- At RX it is configured to store in a file (*adc.csv*) 10000 samples in Octave-compatible format, for post-analysis
The settings panel is automatically updated with these values when selecting the profile, as can be observed in the below figure:

![Figure 46. Settings corresponding to the profile 'TX+RX CAL 2kHz'](image)

A profile configures a group of settings in one go (for convenience).

But anyway, these settings can also be configured individually through the corresponding sub-menus (Global, TX, RX):
This sub-menu selects the information to be displayed in the status-bar:

- **None**: No information is provided. This saves CPU for other tasks.
- **Buffers processed**: Just information about the amount of buffers transmitted and received is displayed. The impact on the CPU is very low.

**TX values**: Statistics about the missed versus successful buffers transmitted

**TX time**: Real-time statistics (max, min, average) of the time consumed to generate a buffer for transmission. It will give an idea about the computational effort required for a given modulation/encoding algorithm.

**RX values**: Real-time statistics over each received buffer. The values monitored here are specially important to tune the system at reception. The most important data provided is:

- **DC Mean** (or DC Offset): is the average of all the samples captured by the ADC
- **AC Mean**: is the average of the absolute values once removed the DC Mean. It is a method computationally efficient that gives a good measure of the amplitude of the signal received. We can increase the gains of the RxPGA1 and/or RxPGA2 to increase this value
- **Range**: gives the difference between the maximum value and the minimum one. It’s like a peak-to-peak voltage. However, it is not very reliable because affected by spurious or glitches

**TOOLS**

Different one-shot tools are provided here:
Log overloads: Just reports the status of the AFE031 register in charge of notifying the source cause of an error condition, either Thermal overload or Current overload.

Clear overloads: This explicitly clears the Overloads register.

View current settings: Just prints the list of currently configured settings in the Log panel.

Measure processing time on modulation: Does a modulation test with the current settings and displays the time required to complete it. It gives an idea of the maximum transmission rate we can achieve to have on-the-fly encoding (i.e. continuous modulation and transmission of incoming data).

3.7.3.3.4 Command line parameters

For simpler usage the plc-cape-lab application gives the possibility to pre-configure some of its parameters through the command line:

```
# ./plc-cape-lab --help
Usage: plc-cape-lab [OPTION]...
"Laboratory" to experiment with the PlcCape board

-A:MODE       Select ADC receiving mode
-d            Forces the application to use the standard drivers
-D:INTERVAL   Max duration for the test [ms]
-F:BPS        SPI bit rate [bps]
-H:FREQ       Signal frequency [Hz]
-I:INTERVAL   Repetitive test interval [ms]
-L:DELAY      Samples delay [us]
-N:SIZE       Buffer size [samples]
-O:OFFSET     Signal offset [DAC value]
-P:PROFILE    Select a predefined profile
-R:RANGE      Signal range [DAC value]
-S:MODE       SPI transmitting mode
-T:MODE       Operating mode
-U:NAME       UI plugin name (without extension)
-W:SAMPLES    Received samples to be stored in a file
-x            Auto start
--help     display this help and exit

For the arguments requiring an index from a list of options you can get more information specifying the parameter followed by just a colon
```

For more information refer to the source-code.

3.8 Pending tasks

This is the first version of the framework (0.1), an early draft. Many things must be improved, bugs fixed and new features implemented.

For example, these are possible improvements or new features to be covered in future versions of the framework:

- Zero-crossing detection of the AC voltage to be used as a synchronization mechanism
- Better management of the AFE031 asynchronous error notifications
- Implement simple protocols (proprietary or based on some standard)
- FEC (Forward Error Correction), 16bit CRC (Cyclic Redundancy Check), encryption algorithms
• Define a whole Device Tree Overlay for the PlcCape to indicate which GPIOs are used by the board and thus allow other Capes to be stacked without conflicts.

There are also some optimizations that can be explored for better computational efficiency:

• take advantage of the Floating Point Unit (FPU) of the BBB CPU\(^9\) (VFPLite)
• replace floating point calculations by 32-bit integer whenever possible.

In fact, the BBB is specially powerful when managing integers but it’s behind other platforms on floating-point support. Here attached, an extract\(^{10}\) of an interesting conclusion obtained from a benchmark with the \textit{nbench} tool over some popular embedded platforms:

\begin{quote}
Looking at the results, the Beaglebone Black has the strongest memory and integer performance. However, the Beaglebone Black’s floating point performance is slightly behind the Raspberry Pi. This can be explained because the ARM Cortex-A8 processor on the Beaglebone Black has a ‘VFPLite’ floating point unit which isn’t as fast as other ARM FPUs. If you only care about performance and don’t have a floating point heavy workload, the Beaglebone Black is a good board to consider.
\end{quote}

\begin{quote}
• use preloaded tables + interpolation for the \textit{sin}(x) values
• take advantage of one of the specific features of the BBB microcontroller: the PRUs\(^{13}\), which are parallel processors able to do some job in real-time
\end{quote}

Optimization note: in the present version of the software we have used single precision floating-points (\textit{float} type) to have a small gain in performance. Usually \textit{floats} give accuracy enough but it’s likely that \textit{double} precision will be required when dealing with more sophisticated modulations.

Concerning bugs:

• the software still freezes in some scenarios that have not been protected. For example the application hangs when the ADC is not able to reach the capturing speed due to other threads running in the background and performing CPU-intensive tasks
• the framework is multithreaded (MT), though only some preliminary resource-sharing control has been implemented. Full MT-safe code must be implemented as soon as possible

\begin{flushleft}
\footnotesize
\textsuperscript{9} Some notes about how to use full hardware floating point in \url{https://wiki.debian.org/ArmHardFloatPort}
\textsuperscript{10} Full article available at \url{https://learn.adafruit.com/embedded-linux-board-comparison/performance}
\end{flushleft}
3.9 Development with Octave

Octave is the Open Source alternative to Matlab. It's a very powerful tool for mathematical analysis. In this project it has been used to analyze the captured data by the ADC of the BBB. It has also been used to get the coefficients of the low-pass IIR filter used in the decoder-ook plugin.

A single Octave file (*.m) has been created to store all the tests done during the development of this project. It is not the best way to do things but it’s quick and functional enough. Each test has been identified with a name which is specified on the calling function (execute_test). A big switch selects the test to run.

The whole source-code of the Octave file used is enclosed in Annex V. Source code files, in the section 10.3 Octave.

3.9.1 Post-analysis

The typical steps to analyze ADC captured data are:

1. Run the plc-cape-lab application configured to store the ADC captures to a file with name adc.csv
2. Start Octave
3. Launch the execute_test function specifying one of the available test types. For example, to see the captures of the ADC execute this sentence:
   ```
   >> run "X:/tools/octave/plot_adc_fft.m"; execute_test("adc", "Y:/applications/plc-cape-lab")
   ```
   where X: is the mapped unit where the source code resides, and Y: is the mapped unit where the output data is stored.
4. For the “adc” test type, Octave will plot the samples received and the corresponding FFT. There are some examples about in 3.4.6.2.4 Analysis within the adc-driver chapter.

3.9.2 IIR coefficients

Octave can be used to easily get the necessary coefficients to implement a basic FIR/IIR filter.

For example we can use the butter function within the signal package to get the coefficients of a 1st order filter with a cut-off frequency = 0.1:

```
>> pkg load signal
>> [iir_b,iir_a] = butter(1, 0.1)
iir_b =
     0.13673   0.13673
iir_a =
     1.00000  -0.72654
```

Then we can update the source code with these coefficients to apply the corresponding IIR filter to a signal:

```
static const float iir_a_default[] = { 1.00000, -0.72654};
static const float iir_b_default[] = { 0.13673, 0.13673};
```

An example of this can be found in the decoder-ook plugin.

Note that the more coefficients the more the accuracy in the cutoff frequency but the more the computational effort required, which is the limiting factor in real-time communications.
4 Analysis & results

4.1 Overview

In the previous chapters, I’ve described the software that has been developed for this project. Now it’s time to use it to evaluate the feasibility of the assembly composed of a BeagleBone Black and a PlcCape board as an alternative for data transmission over Power Lines.

For the tests, the PWM encoding has been used because:

- it’s easy to manage and to identify characters in the resulting captures
- it’s protected against attenuation: PWM characters will be properly decoded while the carrier is over a predefined configurable threshold

4.2 Test equipment

The preliminary tests are focused on checking that the electronics and the software are working as expected. For that purpose, we just need one BeagleBone Black equipped with a PlcCape and a USB connection with a desktop PC.

For the tests involving data communication between boards, we’ll need another BBB and extra power supplies. In particular these are the electronic devices that have been used for the analysis:

![Test kit](image-url)
At the top of the previous figure we have the power supplies with the following characteristics (from left to right):

<table>
<thead>
<tr>
<th>Model</th>
<th>Input</th>
<th>Output</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACO POWER TXM015-115</td>
<td>100-240VAC 50-60Hz 0.7A max</td>
<td>15VDC 1.0A</td>
<td>AC adaptor for the Power Amplifier stage of the AFE031. It is only required for transmission</td>
</tr>
<tr>
<td>DVE C39280-Z4-C501/SNG 30-a</td>
<td>230VAC 50Hz 50mA</td>
<td>9.5VAC 400mA</td>
<td>Basic AC transformer to test the data transmission over a low-voltage AC line (for first tests, it’s a safer way than testing with the 240VAC of the mains)</td>
</tr>
<tr>
<td>Power Pax SW4310B</td>
<td>100-240V 50-60Hz 0.18A</td>
<td>5VDC 1.0A</td>
<td>AC adaptor to power the BeagleBone Black when required</td>
</tr>
</tbody>
</table>

*Table 23. Power supplies used on the tests*

At the bottom we have (from left to right):

<table>
<thead>
<tr>
<th>Device</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeagleBone Black + PlcCape-v2</td>
<td>This device is used to receive data</td>
</tr>
<tr>
<td></td>
<td>You can notice a bridge done through wire-wrapping. That’s because the Power Amplifier stage of the AFE031 was damaged in one of tests done.</td>
</tr>
<tr>
<td></td>
<td>Other blocks still operate correctly. In particular, this board can be used for data reception (where the Power Amplifier doesn’t participate).</td>
</tr>
<tr>
<td></td>
<td>This assembly is connected to a desktop PC through the USB port which is also used to power it</td>
</tr>
<tr>
<td></td>
<td>This BBB (purchased on 2016) has a Linux Kernel 3.8.13-bone70</td>
</tr>
<tr>
<td>BeagleBone Black + PlcCape-v1</td>
<td>This device is used to send data.</td>
</tr>
<tr>
<td></td>
<td>It is connected to a desktop PC through Ethernet</td>
</tr>
<tr>
<td></td>
<td>The logic circuitry is powered by the PowerPax 5VDC</td>
</tr>
<tr>
<td></td>
<td>The TracoPower 15VDC supplies the AFE031 Power Amplifier stage</td>
</tr>
<tr>
<td></td>
<td>This BBB (purchased on 2015) has a Linux Kernel 3.8.13-bone47</td>
</tr>
<tr>
<td>Small breadboard</td>
<td>The small protoboard is used to place some electronics simulating loads on the Power Supply rails.</td>
</tr>
<tr>
<td></td>
<td>You can see a small burned area there: this was because a wrong connection when testing with the 240VAC! Needless to say, be very careful if you are going to experiment with the mains!</td>
</tr>
</tbody>
</table>

*Table 24. Devices and tools used on the tests*

Concerning the PlcCape boards, practical information can be found in previous sections:

- main blocks and signals of the core chip: 2.5 AFE031
• probing points: Figure 6. PlcCape-v1 probing points or Figure 8. PlcCape-v2 probing points

Some remarks about the oscilloscope captures (go to the Figure 67 for a typical example):

◦ On the left margin there is a mark indicating the 0V reference for each channel. Channel 1 is always in dark-blue, Channel 2 is always in dark-red

◦ In some captures, cursor information is provided in dark-purple giving information about the magnitude and the time interval between two points

◦ The USB oscilloscope doesn’t have AC coupling mode. This is not a problem for DC measurements or AC with a low DC component, but it doesn’t produce the best results for AC with considerable offset

### 4.3 Data transmission in loop-mode

#### 4.3.1 Purpose

The idea of this first test is to check that the basic PWM-based modulation and demodulation algorithms implemented by software are working properly. For that purpose, a very helpful functionality offered by the AFE031 has been used: the calibration modes.

In Calibration modes, some blocks of the AFE031 are internally interconnected providing a quick way for simple testing and system characterization. They can be used to tune the communication parameters by isolating the AFE031, hence the calibration term.

Calibration modes can also be helpful to check if there is some AFE031 module that doesn’t work. For example, if the Power Amplifier (PA) stage is damaged, the communication in the calibration modes will still work, since the internal routing doesn’t go across the PA.

There are three different calibration modes available. For this test we have used the one involving the TX Programmable Gain Amplifier (TxPGA in Figure 9. AFE031 blocks and pins) and the TX Low Pass Filter (TxFilter):

![Figure 48. Calibration mode diagram](image)

Note: C2000 MCU is the microcontroller proposed by Texas Instruments in the datasheet of the AFE031. In this project we use the BBB instead.
4.3.2 Testing environment

For the analysis in this section, this testing environment has been arranged:

We can see the USB oscilloscope configured to measure the signal output at one of the PlcCape-v2 provided connectors. That connector is intended for generic purposes (other than PLC communication). It gives easy access to the output of Power Amplifier stage (PA_F_OUT in Figure 8. PlcCape-v2 probing points).

However, as the Power Amplifier of PlcCape-v2 was damaged, a bridge with a wire was made (see at right) to gain easy access to the signal TX_F_OUT instead, which corresponds to the output after the TxPGA + TxFilter (see Figure 9. AFE031 blocks and pins).
In the screen of the desktop PC we have the required applications running:

![Testing tools in the desktop PC](image)

*Figure 51. Testing tools in the desktop PC*

In detail:

<table>
<thead>
<tr>
<th>Application</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>plc-cape-lab</strong></td>
<td>It’s the main applications developed for this project. It performs all the required signal processing and configures the PlcCape board for the different operating modes (see 3.7.3.3 plc-cape-lab for detailed information) It runs on the BBB but it’s remotely controlled by a Putty client</td>
</tr>
<tr>
<td>(at top left)</td>
<td></td>
</tr>
<tr>
<td><strong>Oscilloscope tool</strong></td>
<td>It’s the tool included with the Hantek’s USB oscilloscope. It’s used here to check the TX_F_OUT pin of the AFE031 which corresponds to the low-voltage signal (0 to 3.3V) after the DAC + TxFilter</td>
</tr>
<tr>
<td>(at bottom left)</td>
<td></td>
</tr>
<tr>
<td><strong>Octave</strong></td>
<td>It’s a mathematical tool used here to plot the received data at the ADC of the BBB. The plc-cape-lab application captures it in real-time and stores a given number of samples in Octave-compatible format for post-analysis</td>
</tr>
<tr>
<td>(both windows at right)</td>
<td></td>
</tr>
</tbody>
</table>

*Table 25. Testing tools summary*
4.3.3 Results

4.3.3.1 Test with PlcCape-v2

Find following a screen capture of the plc-cape-lab application configured to continuously send and receive the message This is PlcCape. Hello! \n (a 24-bytes message):

![Screen capture of plc-cape-lab application](image)

Figure 52. Communication test with plc-cape-lab in calibration mode

In the Log panel, plc-cape-lab only displays the incoming data received and demodulated in real-time. Thus, the above figure corresponds to a successful communication test: we are properly getting the message that we have transmitted.

The modulation and demodulation process are carried out by the active encoder (encoder-pwm.so) and decoder (decoder-pwm.so) plugins.

In the Settings panel (at the right) we can see the configuration used, like:

- **AFE configuration mode** = **calib_dac_txpga_tx_filer**

This corresponds to the calibration mode described in Figure 48. Calibration mode diagram

- In the TX section we can see:
  - the settings of the carrier: frequency = 110 kHz, DC Offset = 800 (in DAC units), AC Range = 400 (in DAC units). This results in a sinusoid between 600 (=800-400/2) and 1000 (=800+400/2) in the DAC-side. This range has been experimentally found. Lower values produce some strange effects in the output wave (only faced when operating in calibration mode)
  - the SPI bit rate = 6 Mpbs

- In the RX section we have:
  - capturing mode = **real_time**
This mode shows the message received and demodulated on-the-fly. It’s the opposite of deferred, which would buffer the data received for post-demodulation once the communication is finished.

- ToFile: 10000 samples. The first 10k samples captured by the ADC will be stored on a file for Octave plotting.

The Calibration mode of this test (calib_dac_txpga_tx_filer) also enables the output at TX_F_OUT pin which can then be probed with the oscilloscope.

For example, in the next capture we can recognize the chunks corresponding to the 24-bytes-message periodically sent:

The graph shows that the interval between messages is 385ms. This means that, in average, the communication rate of this test is about:

\[
\text{Average bitrate} = \frac{24 \text{ bytes}}{\text{message}} \times \frac{8 \text{ bits}}{\text{byte}} \times \frac{\text{message}}{0.385 \text{ seconds}} = 499 \text{ bits/second}
\]
By zooming in, we can get a detailed view of the carrier:

![Figure 54. Detail of the 110kHz carrier](image)

Notice that despite the few samples per cycle used in the DAC generation we can see a quite good sinusoid of about 350mVpp and 110kHz (the 10.93kHz measured by the cursor is because I’ve considered 10 cycles for better accuracy).

With Octave, we can also take a look at the 10000 first samples captured by the ADC:

![Figure 55. Data captured by the ADC of the BBB](image)
From the Octave’s graph, we can deduce some interesting conclusions:

- In the top plot, which shows the value of the samples captured by the ADC, we can see that, in our test, a small subset (from 1550 to 1800) of the whole ADC available range (from 0 to 4095) is used. Even in that low-level reception conditions, the system is properly demodulating the PWM signal. Note that 10000 samples at 200 ksps correspond to a capturing interval of 10k/200k = 50ms. In a previous oscilloscope capture we have seen that our full message is repeated each 385ms. So, in this Octave’s capture, we are not seeing whole messages but just some characters of the composing message, encoded in PWM.

- In the bottom plot, we have the absolute value of the Fast Fourier Transform which give us the composition of the received signal in the frequency domain. More details about the FFT graphs where explained in the section 3.4.6.2.4 Analysis when describing the adc-driver.

To get more details about the ADC capture, I have zoomed in the graph to 2000 and 100 samples:

We can see that at the current frequency of the carrier (110kHz) there is only a couple of samples per cycle. We almost get an alternating signal between the minimum (1550) and the maximum values (1800).
4.3.3.2 Test with PlcCape-v1

To test the real communication between two devices we will need the PlcCape-v1 to be used as the transmitter (the PlcCape-v2 has the Power Amplifier stage damaged). So, I’ve repeated the previous test, but now with the PlcCape-v1 and with this probing configuration (measuring TX_F_OUT too):

![Testing environment with PlcCape-v1](image)

First, the proper operation of the TX-RX loop is checked by repeating the steps described in 4.3.3.1:

![Communication test with PlcCape-v1 in calibration mode](image)
I’ve taken some captures with the oscilloscope to compare the output with the one in PlcCape-v2.

The output obtained at 100ms/div is more or less the same one as in the PlcCape-v2 case. If we zoom to 20ms/div, we can see the PWM modulation of each character composing the message sent. I’ve superimposed in red the characters of the message with the corresponding ASCII code in green:
By zooming in, we get a detailed view of the carrier:

![Detail of the 110kHz carrier generated by the PlcCape-v1](image)

The shape is similar to the one in PlcCape-v2 but the output voltage is slightly higher, 447mVpp (+27%). This could be due to the different components and layout of both PlcCapes. In any case, the qualitative behavior of both PlcCape boards is the expected one.

### 4.3.4 Conclusions

From the experiments done in this section, we can draw these conclusions:

- The transmission and reception chains (including the PWM encoder and PWM decoder plugins) are working properly
- The configuration and parameters of this test are suitable for further tests (not only for Calibration Modes)
4.4 Data output at the Power Amplifier

4.4.1 Purpose

In this section we’ll check that the transmission stages and, in particular, the Power Amplifier of the PlcCape-v1, work as expected.

4.4.2 Testing environment

The testing environment is composed of BBB + PlcCape-v1 + 15VDC power supply + OSC probing:

![Figure 63. Testing environment to check the Power Amplifier of the PlcCape-v1](image)

Note that for this test we need the AC-coupling auxiliary board that attaches to the PlcCape-v1 and which provides easy connection points through practical screw-based connectors.

From top to bottom, this is the purpose of the pins in the column of connectors:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PA_GND (in)</td>
</tr>
<tr>
<td>2</td>
<td>PA_VS (in)</td>
</tr>
<tr>
<td>3</td>
<td>VCC_5V (out)</td>
</tr>
<tr>
<td>4</td>
<td>VCC_GND (out)</td>
</tr>
<tr>
<td>5</td>
<td>AC-coupling (in/out)</td>
</tr>
</tbody>
</table>

Table 26. Pinout of the AC-coupling auxiliary board

In this test we’ll measure:

- the TX_F_OUT signal (green-wire probe), which corresponds to the output of the TxPGA + TxFilter
- the TXRX signal (orange-wire probe) which corresponds to the output of the Power Amplifier stage (PA_OUT) after a decoupling capacitor of 10uF (which removes any DC component)
4.4.3 Results

To check the Power Amplifier of the PlcCape, the `plc-cape-lab` application has been configured this way:

![Configuration settings](image)

**Figure 65. Configuration settings to generate output through the Power Amplifier**

This is the oscilloscope capture obtained (CH1 in blue = TX_F_OUT, CH2 in red = TXRX):

![Oscilloscope capture](image)

**Figure 66. Signals at TX_F_OUT (CH1) and TXRX (CH2), 100ms/div**

We can see observe here the gain of the in TXRX (Power Amplifier output) over the TX_F_OUT.
A zoom to 10us/div allows us to better measure the amplitude of the TXRX signal:

The AFE031 data sheet mentions that the typical gain of the Power Amplifier in relation to the input is about 6.5V/V which corresponds to the results got.

The next step is to check the output after the AC coupling circuit. The same testing configuration has been used; only the probing point associated to CH2 has been changed from TXRX to the line-coupling connector (white and green wires):
These are the corresponding oscilloscope captures:

![Figure 69](image1)

**Figure 69. TX_F_OUT (CH1) and coupled-line (CH2), 100ms/div**

And a zoom in to 10us/div:

![Figure 70](image2)

**Figure 70. TX_F_OUT (CH1) and coupled-line (CH2), 10us/div**

We can notice a small attenuation in the coupled-line output (426mVpp) compared to the previously measured TXRX (655mVpp).

### 4.4.4 Conclusions

- The Power Amplifier stage of the PlcCape-v1 and the line-coupling circuitry provided by the auxiliary board are working as expected.
- The line-coupling circuitry slightly attenuates the signal coming from the Power Amplifier.
4.5 Data transmission over an unpowered line

4.5.1 Purpose
The target of this section is to check that we can communicate data between two device assemblies (BBB + PlcCape) through an unpowered line. This will confirm the proper behavior of the whole communication path (TX + RX + line-coupling circuitry) for the particular case of 0 VAC.

4.5.2 Testing environment
Here, the testing material has been arranged like this:

![Testing environment for communication over an unpowered line](image)

We can see:

- On the top (at left), the 15VDC power supply that feeds the Power Amplifier stage of the PlcCape-v1
- Next, we have the transmitter, the assembly composed of a BBB plus a PlcCape-v1
- Finally, we have the receiver, consisting on a BBB and a PlcCape-v2
- The PlcCape-v1 and the PlcCape-v2 are interconnected through a couple of direct wires (green and white) through the line-coupling connectors of both boards

The oscilloscope probe is arranged to measure the TXRX line (the output of the Power Amplifier after a decoupling capacitor).
At software level, two Putty instances have been launched in the desktop PC:

We can see:

- at the top left, the Putty terminal session (via Ethernet) that manages the PlcCape-v1, the transmitter
- at the top right, the Putty terminal session (via USB) that manages the PlcCape-v2, the receiver
- at the bottom, the oscilloscope tool

Note here one of the benefits of using resource sharing to connect the BBB with the desktop PC (7.7 Sharing files): both BBB execute the exact same plc-cape-lab binary because it is retrieved from the same shared folder. Thus, both BBB are always implicitly synchronized, a good point that avoids one of the typical sources of errors.
4.5.3 Results

A look at the Putty consoles after a communication test gives us helpful information:

![Figure 73. Screen capture of the plc-cape-lab at transmission](image)

![Figure 74. Screen capture of the plc-cape-lab at reception](image)

We can observe that the message is received, though with errors!
In the previous captures there are some interesting points to comment:

- The errors at reception usually result in a neighbor character: Hello → Gello, PlcCape → OtcCape, etc.
- The default configuration of the receiver didn’t report any data. An extra amplification gain on the RX_PGA2 has been required. You can see it in the Settings panel of the corresponding plc-cape-lab capture: the rx2 enumeration, which is 0 by default (standing for 1V/V), has been configured to 1, which means a gain of 4V/V
- The status bar of the receiver is configured to report the real-time statistics on the incoming data (for more details go to 3.7.3.3 plc-cape-lab). The values reported are:
  - **DC Mean** = 1301.6 → In the current plc-cape-lab application it is necessary to manually specify the DC Mean value at reception for proper demodulation (in future versions the receiver should offer the option to do an automatic calibration).
  - **AC Mean** ≈ 40 (the 0.8 value in the above figure is because I stopped the transmitter before getting the capture) → This value indicates that the incoming ADC captured data spans over a low range. Or in other words, we are getting a signal with a very low voltage. The low value of the signal received would likely be the root cause of the errors. Note: when rx2 was 0 (= 1V/V), the AC Mean value was 10 in average, which is below the minimum default threshold in plc-cape-lab to consider the presence of a carrier. Hence, the lack of decoded data at reception for that test.

A look at the oscilloscope capture (TXRX pin) shows the 110 kHz carrier we have already seen in previous tests:

![Figure 75. Detail of the carrier signal in the TXRX pin](image-url)
To better analyze what is happening, it is necessary to also measure the signal just before the ADC. For that, we need to probe some point in the ADCIN path (see Figure 8. PlcCape-v2 probing points). One comfortable point is the via displayed in the following picture:

![Probing point for the ADCIN signal](image)

The following oscilloscope graph displays, in CH1 (blue), the TXRX signal at the PlcCape-v1 (the transmitter) and, in CH2 (red), the ADCIN signal at the PlcCape-v2 (the receiver):

![TXRX at PlcCape-v1 (CH1) and ADCIN at PlcCape-v2 (CH2)](image)

We can see that the received signal in ADCIN has a DC offset of about 580mV and an amplitude of about 80mVpp.
If we zoom out (to 500mV/div and 100ms/div), we can appreciate the PWM data and emphasize the attenuation experienced:

![Graph](image)

**Figure 78. TXRX at PlcCape-v1 (CH1) and ADCIN at PlcCape-v2 (CH2), 100ms/div**

We have different possibilities to increase the level of the signal at ADCIN:

- increase the amplitude of the wave generated by the DAC at transmission. In the present test, we are using a Range of 100 (in DAC units). The maximum allowed value, due to the 10-bits resolution of the DAC, would be 1024, although we’ll probably reach some saturation before, in some AFE block
- configure the TX_PGA for less attenuation (the default configuration applies a 0.25V/V)
- increase the gain of the RX_PGA1
- increase the gain of the RX_PGA2

Increasing the gain at reception has been the option chosen to minimize the current that should be delivered by the Power Amplifier stage of the AFE031 and so, the heat produced (the current PlcCape boards doesn’t have good heat dissipation).

RX_PGA2 has been used because it allows for higher gains. It has been configured to the next step \( (rx2 = 2) \), which corresponds to a gain of 16V/V. The improvement is evident, as demonstrates the following capture:
Data communication through power lines using a system based on BeagleBone

Figure 79. Correct reception of data when configuring RX_PGA2 to 16V/V (rx2:2)

Results:

- Now there aren’t errors in the communication
- AC Mean = 161. That’s about four times the value we had with rx2=1 (4V/V), which is according to the expected results

On the other hand, that’s the oscilloscope capture measuring TXRX and ADCIN:

![Oscilloscope Capture](image)

Now, the ADCIN has a magnitude of 281mVpp, a much more manageable level for a proper demodulation.
4.5.4 Conclusions

- The transmission between boards through the line coupling circuitry works OK.
- The signal that arrives to the receiver (from the coupled-line) is strongly attenuated by the integrated filters before reaching the ADC.
- In the conditions of this test, these are the recommended values for a proper demodulation:
  - RX_PGA2 gain of 16V/V
  - DC Offset = 1301
4.6 Data transmission over a low-voltage AC line

4.6.1 Purpose
To minimize the risks of damage, the first series of AC tests have been done over a 12V AC line.

4.6.2 Testing environment
This is the equipment we have used for this test:

![Testing environment in a 12VAC line](image)

Figure 81. Testing environment in a 12VAC line

We can identify:

- PlcCape-v1 + PlcCape-v2
- 15VDC
- 12VAC
- Load simulation with a pair of opposed LEDs with a current limiting series resistor of 10kΩ
- Probing TXRX + AC-Line: note that we can use a different ground for each probe because the measurement is done over a floating transformer
A zoom:

![Image](image.jpg)

*Figure 82. Detail of the wiring in the testing environment*

We can identify:

- Orange and White = 12VAC Power Supply rail
- Pair of yellow wires: connection between the power supply rail and the coupling circuit at the transmitter
- Pair of green wires: connection between the power supply rail and the coupling circuit at the receiver
- Blue: probe over the power supply rail
- Red and black: 15VDC power supply for the Power Amplifier stage of the transmitter

To simulate a load, two LEDs have been disposed in opposite sense. They are alternatively lighted on each half-cycle. That’s also a basic way of checking that our AC power supply is working as expected:

![LEDs](image2.jpg)

Although the power supply is marked as 9.5VAC (root-mean-square amplitude) it seems to provide 12VAC because the peak-to-peak voltage measured with the oscilloscope (33.5Vpp) corresponds to it:

\[
V_{pp} = 2\sqrt{(2)} \times V_{RMS} \approx 2.8 \times V_{RMS} \rightarrow V_{pp}(12\text{ VAC}) = 2.8 \times 12 = 33.6\text{ Vpp}
\]

The deviation between RMS values (expected and real) could be due to the fact we are using the power supply with a high-impedance load (to minimize the power delivered by the AFE) rather than using the nominal conditions of the power supply. In any case this doesn’t mean any trouble for our tests.

---

4.6.3 Results

With the transmitter OFF:

![Graph 1](image1)

Figure 83. Capture of the 12VAC power rail (CH2) with no transmission (CH1)

With the transmitter ON:

![Graph 2](image2)

Figure 84. Capture of the 12VAC power rail (CH2) with PWM transmission (CH1)

We can appreciate now a thicker line for the 12VAC signal. That’s due to the overlapping of the PWM signal over the 12VAC supply.
Let’s see that overlapping in more detail with the help of three more captures:

In the above capture we can see the difference between both scenarios: the first raising edge corresponds to the 12VAC 50Hz without transmission; the falling edge is thicker because the data is superposed.

I’ve also taken a capture at the end of the transmission of a character (at 100us/div):

---

**Figure 85. Detail of the 12VAC with superimposed data and without**

**Figure 86. Detail of the end of a character transmission**
Data communication through power lines using a system based on BeagleBone

A zoom in to the 50us/div scale allows us to clearly identify the PWM carrier of the transmitted data:

![Figure 87. Detail of the PWM carrier in the 12VAC](image)

In the Putty terminal of the receiver we can observe that the message is properly received over the 12VAC powered line:

![Figure 88. Correct reception of data](image)

There is an error by the middle of the logging. It is due to a glitch when switching OFF the 12VAC power. Since then, we are in the unpowered line scenario analyzed in previous sections.
A capture of the TXRX & ADCIN signals (at 10ms/div and 10us/div) shows the expected behavior:

**Figure 89.** TXRX at PlcCape-v1 (CH1) and ADCIN at PlcCape-v2 (CH2), 10ms/div

**Figure 90.** TXRX at PlcCape-v1 (CH1) and ADCIN at PlcCape-v2 (CH2), 10us/div
Data communication through power lines using a system based on BeagleBone

The Octave captures are also the expected ones, without any trace of the 12VAC signal:

![ADC capture](image1)

![FFT](image2)

*Figure 91. Data captured by the ADC plotted with Octave*

And just a final zoom in of the captured data over 100 samples to take a look at the carrier:

![ADC capture](image3)

*Figure 92. Detail of the data captured by the ADC over 100 samples*

### 4.6.4 Conclusions

**Important conclusion:** We have been able to reliably communicate data between two BBB + PlcCape assemblies over a 12VAC Power Line.
4.7 Data transmission over a high-voltage AC line

4.7.1 Purpose
The final test to be done is to check that the system works also over the mains.

Obviously, we have to be very careful here: 240VAC (~680Vpp) is enough voltage to kill people.

4.7.2 Testing environment
The main idea is to repeat the previous test, but replacing the 12VAC by the 240VAC mains:

Comparing to the 12VAC case:

- we have connected directly the mains to the shared coupled-line (power and data)
- the 10 kΩ serial resistor used to limit the current in the low-power LEDs has been replaced here by two low-power resistors (250mW) of 3.3 MΩ each. With this configuration, the LEDs get very little current (< 1mA) but enough to be slightly lighted. In fact, that’s just to have a visual notification that the 240VAC are OK
4.7.3 Results

Unfortunately, I’ve not been able to do a successful test with the current PlcCape board.

In this environment, the AFE031 heats too much and too quickly. In theory, the internal thermal limiter of the AFE should automatically stop any transmission to avoid any further damage, but in this case it seems it has not been fast enough. As with the PlcCape-v2, the Power Amplifier stage of the PlcCape-v1 has been destroyed after this test.

Note that the use of the mains (compared to the 12VAC scenario) has two significant differences:

- obviously, the PlcCape circuitry must deal with much higher voltages (x20 times)
- a lot of domestic appliances are connected to the mains (the TV, the desktop PC, the refrigerator…). All these consume a considerable amount of energy. Or in other words, they represent low impedance sources. That means that our AFE031 in the PlcCape must be able to deliver significant current to attain the required voltage. And, maybe this has been the cause of the excessive heat produced

4.7.4 Conclusions

This test has revealed one important point we must be considered when looking at the PLC technology: the management of the power consumption and the design of proper heat dissipation mechanisms.

In our specific case, a probable cause of the excessive heat has been the lack of a good thermal design in the PlcCape board. For example, the AFE031 chip has a conductive layer for heat dissipation in the bottom side that MUST be soldered to the board. This was not the case for the boards I’ve used for the tests.

After the result of this analysis, we should consider other alternatives to evaluate in the future, like:

- provide an option into the plc-cape-lab software to just enable the AFE031 for a very short lapse, to send only one message (or one char). Current software is focused more on repetitive tests
- provide an external Power Amplifier stage through a high-power transistor. The benefit is that we should be able to deliver much more power and dissipate heat more easily, but at the expense of higher costs and more board space

In any case, this electronic problem is out of the scope of this project
5 Conclusions

5.1 Feasibility

In this project we have demonstrated that the assembly composed of a BeagleBone Black and a PlcCape is a cost-effective solution that can be used to communicate data over a Power Line, respecting the bandwidth constraints imposed by the CENELEC regulations.

The core of the PlcCape (the AFE031) is based on the modulation-by-software strategy, which allows implementing any protocol or customization by just updating the software. The hardware is generic enough to deal with most scenarios.

That concept has brought important challenges to the software to be developed:

**HIGH SPEED SPI**

We have seen that the off-the-shelf SPI driver that came with the BBB wasn’t able to reliably cover the real-time constraints of this project, which required the DAC to be fed with continuous data at 6Mbps with no latency or jitter.

To overcome that, the SPI driver has been adapted to implement a Ping-Pong buffer managed by DMA, taking advantage of the EDMA unit available in the AM335x processors.

This solution has proven to be very effective, reliable and optimized (minimum CPU-consumption and minimum impact to other applications in background).

**HIGH SPEED ADC**

To be able to decode the modulated signals, a high speed ADC is also required. By specifications, the ADC of the BBB is limited to 200ksp/s but the off-the-shelf driver didn’t allow reaching these speeds because it was more focused on Touch Screen controllers, where averaging of the captured samples is prioritized (for reliability) over speed.

To deal with that, the ADC driver has been modified to be able to reach the maximum capturing rate:

- on one hand, the averaging has been disabled
- on the other hand, the FIFO strategy implemented by the driver has been slightly modified for better throughput

The solution has demonstrated to be effective, although not optimized in performance. It is quite CPU-intensive. In fact, the initial target was to implement a DMA-based strategy (instead of the FIFO-based one) but it has not been possible due to timing constraints.

**BBB PERFORMANCE**

Floating-point computations have supposed another important challenge to deal with.

The CPU of the BBB offers high computational efficiency when managing integers but is relatively slow in floating-point calculations. In this project I have started to face real-time problems when generating a simple carrier (with a \( \sin(x) \) function) at 3Mbps. The BBB was not able to fill the Ping-Pong buffer in time, resulting in underflow errors and wrong output waves.
I’ve tried to use single precision floats instead of double ones but it has not been enough.

The final solution I have applied is a “deferred modulation”. That is, prior to the transmission, I have generated the wave to be sent to the DAC in a large enough buffer. After that, only fast memcpy operations are required to fill the small DMA Ping-Pong buffer in real-time.

This solution has demonstrated to be very effective and has finally been used on most tests.

The “deferred modulation” strategy is something that can be accepted in many applications where only small packets of data are involved. But it can be a limiting factor on others requiring real-time communication.

The ARM processor of the BBB provides a “VFP Lite” co-processor that could drastically improve the floating-point computations. I’ve not been able to determine if the C compiler is already using it or not. It’s something to evaluate in the future.

**PlcCape electronics**

At hardware level the PlcCape has been presented as a cost-effective board that could do the job.

Unfortunately, the breakdown of the AFE031 in the last tests with the 240VAC has revealed that there is still some work to be done on the electronics of the board for more robustness and reliability. But this is out of the scope of this particular software project.

In any case, the tests I’ve managed to do with a 12VAC power supply demonstrate the feasibility of the system. For the 240VAC it’s just a matter of increasing the supported voltage and current.

On the software side, the framework developed has demonstrated to be highly customizable and versatile. A couple of basic modulation schemes has been successfully tested, and more complex ones should be easily integrable thanks to the “plugin” concept.

### 5.2 Reusability

The AFE031, including a high speed DAC and a Power Amplifier stage, is a good complement to the BBB, which has an embedded ADC but misses a DAC.

The ensemble BBB + PlcCape + plc-cape framework constitutes a complete inexpensive analog system that can be easily adapted to other uses than PLC, like (in brackets the subsystem involved):

- a highly customizable function generator [DAC]
- a frequency-response analyzer [DAC + ADC]
- a basic but programmable oscilloscope [ADC] with some extra features
  - the small transformer in the PlcCape board gives AC coupling and a practical isolation
  - the RX chain of the AFE031 provides interesting filtering and some programmable gain functionality
  - the plc-cape software should allow for customizable capturing intelligence (for example to record samples only after a pulse of some predefined width or amplitude)
- audio tools: player [DAC], recorder [ADC], software equalizer or real-time-distorter [ADC + DAC], etc.
At academic scope this tool may offer the opportunity to play with real hardware (instead of through just mathematical tools) when studying the typical problems on communication systems: modulations, noise, interferences...

5.3 Future development

The software framework delivered along with this documentation is still in an early phase of the development (hence released as version 0.1). It has been designed with the focus on evaluating the feasibility of the system proposed (BBB + PlcCape) in the PLC field, putting source code quality and best practices in a second level (though I’ve always had those in mind). Therefore, there is still a considerable amount of work to be done to have some usable package.

On the other hand, there are a lot of interesting subjects that can be explored in future developments, like:

- Improve current modulation/demodulation schemes (PWM, OOK) to gain in speed and robustness
- Implement other more efficient modulations (OFDM,...)
- Develop smarter standalone applications with typical features as: auto-discovering, auto-calibration, etc.
- Optimize the system to minimize CPU-impact: ADC captures by DMA, floating-point optimizations, use of the PRUs (mini parallel real-time CPUs integrated in the BBB processor), etc.
- ...

5.4 Final assessment

This project has been a challenge for me.

When I started it, I was a newbie on Linux development (I had some experience on other operating systems). Moreover, I had never programmed for the BeagleBone before. I didn’t either know some of the fascinating free software tools that I’ve used for this project as Octave, LibreOffice or Doxygen (to mention a few).

The current implementation is still far from being a usable framework. But in any case I think I’ve covered some interesting problems and proposed “creative” solutions that may help other developers (newbies too) to be introduced to the embedded areas addressed by this project (Linux drivers, SPI, EDMA, ADC, etc), and maybe, with a shorter learning curve than it has been for me.

:-)
6 Annex I. Development tools

6.1 Hardware
The following equipment has been used:

- A couple of BeagleBone Black boards equipped with a PlcCape each to test the transfer between two devices from different points
- A USB-based Oscilloscope for comfortable measurements and captures. Any mid-range model is enough for this project because the medium frequency of the signals involved. In particular a Hantek 6022 BE has been used here

6.2 Software development tools
This project is intended to be developed in a Desktop PC equipped with a Linux-based system. Unfortunately the manufacturer of the USB-based oscilloscope (Hantek) does not currently provide any driver to work with on a Linux-based system. This obliged me to use my old Microsoft Windows XP-based system.

The main software development components used on this project have been:

- A BeagleBlone Black with a Debian Wheezy 7.8 distribution to compile and run the code
- An editor (or IDE) to write the source code. Depending on the running platform different software has been used:
  - in Windows XP: `notepad.exe`, Visual Studio suite
  - in Ubuntu: `gedit`, Eclipse 4.5 Mars
  - in the BeagleBone Black: `nano`
- A remote console tool to connect to the BeagleBone boards. With it we can easily compile the source code in the device and check the results in the desktop PC
  - in Windows XP/Ubuntu: Putty 0.64
  - in Ubuntu: ssh, `screen`

6.3 Other tools
Other third-party tools that have been used in this project are:

- **Doxygen 1.8.11 + Graphviz 2.38**: to generate automatic documentation from the source code
- **Octave 4.0.1**: for comfortable mathematical analysis
- **Libre Office 5.1.3.2**: to write this document
7 Annex II. Connection between a Desktop PC and the BBB

The aim of this section is to describe the basic recommended steps to have a comfortable environment and be able to compile the software developed for this project.

For maximum versatility the steps are given for Linux-based OSes (Ubuntu) and for Microsoft Windows-based ones (Windows XP).

At a first glance it may seem cumbersome but once used to it’s a comfortable and efficient development environment.

7.1 Configuration key points

For this project the BeagleBone Black platform is configured as follows:

- Linux distribution based on Debian Wheezy 7.8
- Linux Kernel version 3.8.13
- Specific customization for the BeagleBone Black identified as 3.8.13-bone70
- Third-party packages not included by default and required by the app of this project:
  - ncurses: for rich console-based user interfaces

Next chapters describe the steps to configure an off-the-shelf BeagleBone Black (bought to Element14 on February 2016) to allow building and running the software developed for this project.

7.2 Connection from a desktop PC to the BBB

The first thing we need is a comfortable way to interact with the BeagleBone Black platform. For example:

- through a remote console
- directly through a USB keyboard and an HDMI display

In this project I’ve used the console-based because is simpler and cheaper (keyboard & display are shared for both the desktop PC and the BBB).

We can open a remote session by USB or by Ethernet.

In Ubuntu we have a couple of built-in tools to quickly establish client connections:

- **ssh**: to connect to a remote Linux device through a network connection

  For example, assuming the BBB has the IP address 192.168.7.2 we establish a connection with it this way:

  $ ssh root@192.168.7.2

- **screen**: to connect to a remote Linux device through a serial connection (the USB connection supports both, a TCP/IP connection but also a virtual serial port)
First we have to find out the path of the serial connection (with $ls$), then we can open the connection:

```
$ ls /dev/serial/by-id/
   usb-Circuitco_BeagleBoneBlack-if02
$ screen /dev/serial/by-id/usb-Circuitco_BeagleBoneBlack-if02
```

There is also a dedicated application available for remote telnet connections: **Putty**. It is an Open Source SSH and telnet client (http://www.putty.org/). It’s available for both Linux and Windows. It allows to comfortably communicate with the BBB through a USB cable or an Ethernet port.

For the initial setup the USB method is quite simple:

1. Connect a USB cable from the desktop PC to the BBB and wait for some seconds for the OS initialization.

If you were using Windows new hardware will be detected (e.g. “Linux USB Ethernet/RNDIS Gadget” + “Gadget Serial” + “Mass Storage” + ...). Install it as usually. You can then check that the USB-network interface is properly working with the **ipconfig** command:

```
C:\>ipconfig
```

This screen capture shows a typical configuration:

- a Wi-Fi connection in IP 192.168.1.3 (with Internet access through the 192.168.1.1 gateway)
- the USB connection with the BBB identified with the IP 192.168.7.1
- an Ethernet interface is available but nothing is connected through

The BBB assigns by default the IP address 192.168.7.2 to the USB interface. To check that the connection from the PC to the BBB is ok the **ping** command can be used:

```
C:\>ping 192.168.7.2
Haciendo ping a 192.168.7.2 con 32 bytes de datos:
Respuesta desde 192.168.7.2: bytes=32 tiempo=1ms TTL=64
Respuesta desde 192.168.7.2: bytes=32 tiempo=1ms TTL=64
```
2. Start Putty and specify the BBB IP Address (192.168.7.2) in the Session panel:

![Putty Configuration](image)

*Figure 95. Main screen of the Putty application*

The default BBB SSH connection requires using `root` as the user name. You can type it in each connection or indicate it in the `Connection > Data > Auto-login username` field of Putty for quicker access.

**Tip:** for comfort purposes custom sessions can be defined (in previous capture JMO-BBB-Usb, JMO-BBB-Eth...) with the specific required settings (IP address, login name, specific UI colors, etc.). Then double clicking on the saved profile will open the connection.
3. If the connection is properly established a console window will be opened:

![Figure 96. BBB login message (via a remote console on Putty)](image)

### 7.3 Initial interaction with the BBB

When working with a remote client (ssh, screen or Putty) all the commands typed on the console will be executed on the BeagleBone. For example to get the version of the OS we can execute `uname -a`:

```
root@beaglebone:~# uname -a
Linux beaglebone 3.8.13-bone70 #1 SMP Fri Jan 23 02:15:42 UTC 2015 armv7l GNU/Linux
```

![Figure 97. Version of the Linux Kernel of the BBB](image)

Note: for a BeagleBone Black bought on 2015 the original version was a previous one, 3-8-13-bone47:

```
root@beaglebone:~# uname -a
Linux beaglebone 3.8.13-bone47 #1 SMP Fri Apr 11 01:36:09 UTC 2014 armv7l GNU/Linux
```

We can get information about the distribution with the command `lsb_release -a`:

```
# lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:    Debian GNU/Linux 7 (wheezy)
Release:        7.8
Codename:       wheezy
```

Note: exactly the same distribution was present in the BBB purchased on 2015.
Following a list of other possible interesting informative commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>df -h</code></td>
<td>File system information</td>
</tr>
<tr>
<td><code>dmesg -head</code></td>
<td>Show initial boot messages (with interesting information)</td>
</tr>
<tr>
<td><code>dpkg -l</code></td>
<td>List of all the installed packages in a Human Readable format</td>
</tr>
<tr>
<td><code>dpkg-query -W</code></td>
<td>List of all the installed packages in a compact way</td>
</tr>
<tr>
<td>`dpkg-query -W</td>
<td>wc -l`</td>
</tr>
</tbody>
</table>

And the corresponding execution on the off-the-shelf BBB:

```
# df -h
Filesystem Size Used Avail Use% Mounted on
rootfs 3.5G 1.8G 1.6G 54% /
udev 10M 0 10M 0% /dev
tmpfs 100M 500K 99M 1% /run
/dev/disk/by-uuid/746f634c-67a5-4e5a-b14d-1ec025ae9095 3.5G 1.8G 1.6G 54% /
tmpfs 240M 0 240M 0% /dev/shm
tmpfs 240M 0 240M 0% /sys/fs/cgroup
tmpfs 100M 0 100M 0% /run/user
tmpfs 5.0M 0 5.0M 0% /run/lock
/dev/mmcblk0p1 96M 64M 33M 67% /media/BEAGLEBONE
```

```
# dmesg | head
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 3.8.13-bone70 (root@a5-imx6q-wandboard-2gb) (gcc version 4.6.3
(Debian 4.6.3-14) ) #1 SMP Fri Jan 23 02:15:42 UTC 2015
[    0.000000] CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=50c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] Machine: Generic AM33XX (Flattened Device Tree), model: TI AM335x BeagleBone
[    0.000000] Memory policy: ECC disabled, Data cache writeback
[    0.000000] On node 0 totalpages: 130816
[    0.000000] free_area_init_node: node 0, pgdat c08c09c0, node_mem_map c093e000
[    0.000000] Normal zone: 1024 pages used for memmap
```

```
# dpkg-query -W -f='${binary:Package;-20} ${Version;18} ${binary:Summary}
'
accl 2.2.51-8 Access control list utilities
acpid 1:2.0.16-1+deb7u1 Advanced Configuration and Power Interface event daemon
adduser 3.113+mmu3 add and remove users and groups
alsa-base 1.0.25-4 Utilities for configuring and using ALSA
alsa-utils 1.20141125-1-bpo70 AM335x PRU PACKAGE
apache2 2.2.22-13+deb7u4 Apache HTTP Server metapackage
```

```
# dpkg -l | wc -l
957
```

```
# ifconfig
eth0 Link encap:Ethernet HWaddr ec:12:34:56:78:aa
  UP BROADCAST MULTICAST MTU:1500 Metric:1
  RX packets:0 errors:0 dropped:0 overruns:0 frame:0
  TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
  collisions:0 txqueuelen:1000
  RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
  Interrupt:40

lo Link encap:Local Loopback
  inet addr:127.0.0.1 Mask:255.0.0.0 inet6 addr: ::1/128 Scope:Host
  UP LOOPBACK RUNNING MTU:65536 Metric:1
  RX packets:0 errors:0 dropped:0 overruns:0 frame:0
  TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
  collisions:0 txqueuelen:0
```
7.4 Internet access

For a comfortable development environment it’s very convenient to get access to the Internet. It will allow updating packages on the BBB or download specific ones, to get access to a NTP server to synchronize date/time, etc.

A quick way to test if we have an Internet connection is with `ping`:

```
# ping www.google.com
ping: unknown host www.google.com
```

We can also try with one of the IP address associated to `www.google.com`, 172.217.16.68:

```
# ping 172.217.16.68
connect: Network is unreachable
```

We can check the connection with the PC:

```
# ping 192.168.7.1
PING 192.168.7.1 (192.168.7.1) 56(84) bytes of data.
64 bytes from 192.168.7.1: icmp_req=1 ttl=128 time=0.475 ms
64 bytes from 192.168.7.1: icmp_req=2 ttl=128 time=0.477 ms
64 bytes from 192.168.7.1: icmp_req=3 ttl=128 time=0.610 ms
...
```

A quick simple way to gain Internet access could be connecting a gateway to the Ethernet port.

Another way is sharing the Internet connection of the desktop PC: the BBB connects with the PC through the USB and the desktop PC routes the packets to the Internet gateway through the proper adapter (the Wi-Fi interface in my case).

7.4.1 Steps in the Desktop PC

If the Desktop PC is equipped with Windows XP we can share the Internet connection following these steps:

1. Open the `Control Panel` and double-click the `Network Connections` icon
2. Then select the adapter that is connected to Internet, right click on it and select `Properties > Advanced options` and enable the option `Allow other network users to connect through this computer’s Internet connection`
If the Desktop PC is equipped with Ubuntu we can share the Internet connection this way:

1. Gain root access
   
   $ sudo su

2. With `ifconfig` find the name of the interface that is connected to Internet. For this example assume it is `wlan4` corresponding to a Wi-Fi connection

3. With `ifconfig` find the name of the interface that is connected to the BBB. For this example assume it is `eth5`, corresponding to the Ethernet Port

4. Set the bridge between the BBB interface and the Internet one:
   
   $ iptables --table nat --append POSTROUTING --out-interface wlan4 -j MASQUERADE
   $ iptables --append FORWARD --in-interface eth5 -j ACCEPT

5. Enable the forwarding:
   
   $ echo 1 > /proc/sys/net/ipv4/ip_forward

### 7.4.2 Steps in the BBB

We need to specify the IP address of the desktop PC sharing the Internet connection:

```
# route add default gw 192.168.7.1
```

Now we can check the ping with an Internet server IP address or with a domain name:

```
# ping 172.217.16.68
PING 172.217.16.68 (172.217.16.68) 56(84) bytes of data.
64 bytes from 172.217.16.68: icmp_req=1 ttl=53 time=41.9 ms
64 bytes from 172.217.16.68: icmp_req=2 ttl=53 time=51.0 ms
64 bytes from 172.217.16.68: icmp_req=3 ttl=53 time=58.3 ms
...
# ping www.google.com
PING www.google.com (216.58.208.228) 56(84) bytes of data.
64 bytes from par10s22-in-f228.1e100.net (216.58.208.228): icmp_req=1 ttl=52 time=48.5 ms
64 bytes from par10s22-in-f4.1e100.net (216.58.208.228): icmp_req=2 ttl=53 time=47.0 ms
64 bytes from par10s22-in-f4.1e100.net (216.58.208.228): icmp_req=3 ttl=53 time=47.6 ms
...
```

Note that the `route` command applies only to current session. The configuration is lost after reboot. To make permanent the access to the gateway we should update the file `/etc/network/interfaces` adding the default gateway. However, if we take a look at that file we can see that it's already defined there:

```
# tail /etc/network/interfaces
# wpa-psk "password"

# Ethernet/RNDIS gadget (g_ether)
# ... or on host side, usbnet and random hwaddr
# Note on some boards, usb0 is automatically setup with an init script
iface usb0 inet static
   address 192.168.7.2
   netmask 255.255.255.0
   network 192.168.7.0
   gateway 192.168.7.1
```
It doesn’t work because in the BBB there is a custom initial script that enables the USB with a basic configuration without reading the interfaces file. Thus, for our basic configuration we can even replace the previous route command by ifup with similar results:

```bash
# ping www.google.com
ping: unknown host www.google.com
# ifup usb0
# ping www.google.com
PING www.google.es (172.217.16.67) 56(84) bytes of data.
64 bytes from par03s13-in-f3.1e100.net (172.217.16.67): icmp_req=1 ttl=53 time=46.3 ms
64 bytes from par03s13-in-f3.1e100.net (172.217.16.67): icmp_req=2 ttl=52 time=44.3 ms
... 
```

A difference between these two commands is that ifup automatically synchronizes the clock with an NTP server after some seconds (see next sections for more information about NTP).

### 7.4.3 Date time management

The BBB doesn’t have a battery to keep a RTC (Real Time Clock). Thus, it always starts with a predefined date:

```bash
# date
Sun Mar  1 20:46:26 UTC 2015
```

Having the proper date and time is convenient when using make because it uses the file time stamps to check if there is a change in the source code beyond the date of the compiled binary and then recompile it.

We can manually set the clock time with the date command but if an Internet connection is available a more comfortable way to synchronize it is via a NTP server. We can use the ntpdate command and a public NTP server (http://www.pool.ntp.org/en/):

```bash
# date
Sun Mar  1 20:46:26 UTC 2015
# ntpdate europe.pool.ntp.org
10 Jun 17:28:13 ntpdate[1220]: step time server 193.145.15.15 offset 40336754.693659 sec
# date
Fri Jun 10 17:28:36 UTC 2016
```

Note: if the Internet access is available at boot-time the ntpdate is executed automatically and the clock synchronized at start-up.

By default, ntpdate assumes the UTC+0 time zone. To change it to the local time zone we can use the interactive command tzselect which asks for the local country and then give the steps to configure it permanently. For example this is the response for Spain:

```bash
# tzselect
Please identify a location so that time zone rules can be set correctly.
...
You can make this change permanent for yourself by appending the line
TZ='Europe/Madrid'; export TZ
to the file '.profile' in your home directory; then log out and log in again.
...
```

So next step is to edit the .profile as indicated. One simple way to do so is as follows:

```bash
# echo "TZ='Europe/Madrid'; export TZ" >> ~/.profile
```
Be careful: the appending operator ‘>>>’ MUST be used to don’t overwrite the existing .profile contents.

Then close the current session with `logout`, reopen it again and execute the `date` command:

```
# date
```

Now the clock is properly synchronized using the local time zone (CEST instead of UTC).

If an Internet connection is not available we can manually set the date and time we the `date` command. Note that the default input format is [MMDDhhmm[[CC]YY][.ss]] (see `date --help`):

```
# date 0713120016
Wed Jul 13 12:00:00 CEST 2016
```

### 7.5 Source code edition

For source code edition we have many strategies:

- Edit and compile the code directly in the BBB
- Edit the code in the desktop PC, send it to the BBB and then compile it there
- Edit the code in the desktop PC using a shared folder (located in the PC, in the BBB or elsewhere) and compile it in the BBB
- Edit the code and compile it in the desktop PC (cross-compilation), then send the binary to the BBB

In this project the option of editing the code in the desktop PC through a shared folder and compile it in the BBB was chosen because its simplicity in management and maintenance: it allows to have the source code in the desktop PC where it is easier to manage (edit, backup…) than in the BBB.

### 7.6 Sending files to the BBB

The direct transfer from the PC to the BBB has also been used in some occasion because its simplicity.

For instance a simple way is using the `pscp` tool that comes with Putty. Assume you have a `hello.c` in your `C:\dev\hello` folder and the Putty installed on `D:\Archivos de programa\Putty`. To send the file to the BBB you can use:

```
C:\dev\hello>"D:\Archivos de programa\Putty\pscp" main.c root@192.168.7.2:main.c
main.c                   | 0 kB |   0.1 kB/s | ETA: 00:00:00 | 100%
```

Then with Putty you can see the transferred file in the BBB:

```
root@beaglebone:~# ls -l
total 4
-rw-r--r-- 1 root root 99 Jun 10 18:12 main.c
```

The `pscp` tool allows transferring a complete folder too. In the BBB create first a folder to keep the project files (e.g. `mkdir projects`). Then send all the files from the desktop PC `C:\dev\hello` folder:

```
C:\dev>"D:\Archivos de programa\Putty\pscp" -r hello root@192.168.7.2:projects
main.c                   | 0 kB |   0.1 kB/s | ETA: 00:00:00 | 100%
```

You can see now all the project files in the BBB proper folder:
The advantage of this method is that it doesn’t require any specific configuration task. The drawback is that it’s inefficient and slow.

7.7 Sharing files

A better strategy for the plc-cape project is to use shared resources:

- it avoids redundant and prone-to-error copy/paste processes
- we have always the up-to-date version of the code in both the Desktop PC and the BBB
- using logical shared resource names instead of hard-paths to physical media makes the maintenance easier. For example the project can be easily relocated

7.7.1 Sharing files through SAMBA

SAMBA is a convenient way to share files because supported by both Windows XP and Ubuntu.

To share a folder in Windows XP just right click on it, select the option Sharing and Security... and then the option Share this folder on the network, specifying a public name for it\textsuperscript{12} (e.g. plc-cape-src):

\textbf{Figure 98. Resource sharing dialog in Windows XP}

\textsuperscript{12} More information on \textit{How to use the Simple File Sharing feature to share files in Windows XP} is available in https://support.microsoft.com/en-us/kb/304040
We can also use a batch file to automate the procedure. For example the following batch \textit{(init\_env.cmd)} would create a shared resource \textit{plc-cape-src} pointing to the source code in a USB Pen drive in F: (“slow” flash media) while the binary generation is done in the HDD C: (fast hard disk drive) and shared as \textit{plc-cape-bin}:

\begin{verbatim}
@echo off
echo Sharing resources...
net share plc-cape-bin=C:\dev\bin\beaglebone\plc-cape-bin
net share plc-cape-src=F:\dev\src\beaglebone\plc-cape-src
echo Resources shared
pause
\end{verbatim}

To share a folder in Ubuntu we have a similar procedure within the Nautilus file explorer.

In the BBB we will use the \texttt{mount} command to connect to the shared folder:

\begin{verbatim}
# mkdir /mnt/dev
# mount -t cifs //192.168.7.1/dev /mnt/dev -o user=,password=
# find /mnt/dev
# /mnt/dev/hello
# /mnt/dev/hello/main.c
\end{verbatim}

### 7.7.2 Sharing files through NFS

NFS is another protocol to share files commonly used on Linux-based systems.

To share a resource in Ubuntu we have to add the shared resources to \texttt{/etc/exports}. For example, this will be the line to share a folder \texttt{/home/my-user-name/dev}:

\begin{verbatim}
/home/my-user-name/dev 192.168.7.0/255.255.255.0(rw,sync,no_subtree_check)
\end{verbatim}

Then we need to request a refreshing of the NFS server exports:

\begin{verbatim}
$ sudo exportfs -a
\end{verbatim}

In the BBB we will use a \texttt{mount} command like this:

\begin{verbatim}
# mkdir /mnt/dev
# mount 192.168.7.1:/home/my-user-name/dev /mnt/dev
# find /mnt/dev
# /mnt/dev/hello
# /mnt/dev/hello/main.c
\end{verbatim}
8 Annex III. Getting started with the development environment

Assuming that a Desktop PC and the BBB are configured as described in the Annex II. Connection between a Desktop PC and the BBB, this chapter describes how to set up the development environment to be able to develop and compile the plc-cape framework.

8.1 Tree structure

8.1.1 Source code

The source code is distributed among these folders:

- **+common**: common resources (*.h headers with public declarations, *.mk shared makefiles, etc.) to be used by any application, library or plugin belonging to the plc-cape framework
- **drivers**: source code of the modified drivers. The aim of the code there is to optimize performance of low-level operations and minimize the overall impact to other running processes, taking advantage of the BBB and PlcCape board specific features
- **libraries**: source code for the reusable static libraries (*.a)
- **plugins**: source code of the reusable dynamic libraries (*.so). Plugins are grouped among predefined categories
- **applications**: source code of the standalone executables developed for specific purposes
- **tools**: sources corresponding to other external tools used in the project (e.g. Octave modules)
- **env**: bash scripts, batch files and other resources to simplify the setup of the development environment
8.1.2 Output warehouse

For convenience a separated folder has been used for the data that is generated from the source code like:

- **binaries**: executables, *.a for static libraries, *.so for plugins, *.ko for drivers
- **intermediate compiler results**: *.o, *.symvers (for drivers)
- **Doxygen documentation**: lots of HTML pages, RTFs, MAN pages, etc.

This way of separating source-code vs building-results is known as *out-of-tree building*.

The path for the building-results is configurable. This allows to comfortable switch among different environments. For example we can have a folder for the results corresponding to the building targeted to the BeagleBone Black and its ARM processor (*plc-cape-bbb* in the Figure 100) or the results corresponding to the building of a PlcCape emulator for a x86 architecture\(^\textsuperscript{13}\) (*plc-cape-x86* in the figure).

8.2 Basic equipment

The first step to be able to operate with the source code is to prepare the development environment.

To cut down the effort required to get used to the *plc-cape framework* this *project relies on direct compilation from the BBB*.

For a comfortable development environment these points are suggested:

- Edit the source-code from a desktop PC and share the directory where it resides to get direct access to it from the BBB. This avoids the need and drawbacks of keeping the source code synchronized back and forth between the desktop PC and the BBB.
- Operate the BBB through a remote console from the desktop PC.
- Compile the code and test the binaries in the BBB through the remote console.

At hardware level the development environment is also very simple and versatile:

- A PC equipped with a USB 2.0 or Ethernet port to connect to the BBB and with the tools for development:
  - a source-code editor: it can be a full featured IDE (like *Eclipse, Microsoft Visual Studio, ...*) or just a text editor (like *gedit, pico, nano, vi, vim, notepad.exe...*).
  - a tool to open a remote console on the BBB (like *ssh, screen, Putty...*).
- A BBB equipped with the PlcCape board.

For example, you can develop for the *plc-cape* framework with just this:

---

\(^{13}\) Although not fully operative yet this project will accept the compilation against a x86 processor (a desktop PC) by "emulating" part of the PlcCape components.
For mobility purposes the source code framework is based on two environment variables when referring to paths:

- **DEV_SRC_DIR**: pointing to the root folder of the source-code framework
- **DEV_BIN_DIR**: pointing to the folder where to place the resulting binaries and intermediate compiler files

This replaces absolute paths by pseudo-relative-paths (based on easily configurable environment variables).

### 8.3 Checking the building environment

To check that you are ready to build the *plc-cape* source code try first to build a simple application.

Firstly, you need to open a session in the BBB. To power the BBB the usual comfortable way is through USB because it avoids the need of an external 5V power supply:

1. Connect the BBB to your desktop PC by the USB port. The default IP addresses for the USB-based connection is 192.168.7.1 for the PC and 192.168.7.2 for the BBB

2. Open a remote client session with the BBB (with ssh, screen or Putty)

3. With your preferred editor (*nano*, *vi*, etc.) create a file with the typical *Hello world* application in `~/projects/hello/main.c`:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello world\n");
    return 0;
}
```
To compile it just go to the project folder and type **make main**:

```
root@beaglebone:~# cd ~/projects/hello
root@beaglebone:~/projects/hello# make main
cc     main.c   -o main
root@beaglebone:~/projects/hello# ls -l
total 6
-rwxr-xr-x 0 root root 5055 Jun 10 23:01 main
-rwxr-xr-x 0 root root   99 Jun 10 22:41 main.c
root@beaglebone:~/projects/hello# ./main
Hello world
```

### 8.4 Basic setup: compiling the ‘plc-cape’ framework

To have an initial quick contact with the *plc-cape* source code with minimum complexity this procedure can be followed to set up the entire framework:

1. Download the whole *plc-cape* project to a directory in your desktop PC
2. Share the directory using *plc-cape-src* as the identifier for the public resource. Use SAMBA as the sharing protocol (commonly present in Windows and Linux)
3. Connect the BBB to the PC by the USB port
4. Open a remote client session with the BBB
5. In the BBB declare the required environment variables (**DEV_SRC_DIR** and **DEV_BIN_DIR**) and connect to the shared folder in the PC with the corresponding `mount` command. In the simplest approach we can configure the environment to just compile the binaries in the same folder as the sources:

   ```
   # export DEV_SRC_DIR=/mnt/plc-cape-src
   # export DEV_BIN_DIR=/mnt/plc-cape-src
   # mkdir $DEV_SRC_DIR
   # mount -t cifs //192.168.7.1/plc-cape-src $DEV_SRC_DIR -o user=,password=
   ```

6. Then go the root folder of the *plc-cape* project and execute **make**. This will launch a recursive build operation for the whole framework:

   ```
   # cd $DEV_SRC_DIR
   # make
   ...
   ```

7. It all the required dependencies are present the building process should succeed after some seconds. The compiled applications can then be tested:

   ```
   # cd $DEV_BIN_DIR/applications/plc-cape-demo-lib-usage
   # ./plc-cape-demo-lib-usage
   ```

### 8.5 Advanced setup: out-of-tree building

The **makefile** files of the project are all based on the global environment variables **DEV_SRC_DIR** and **DEV_BIN_DIR**.
For simplicity we can create a batch file defining the global variables and mounting the shared folders:

```bash
#!/bin/bash
PROJECT_SRC=plc-cape-src
PROJECT_BIN=plc-cape-bin

# Remember to execute this batch with './init_env.sh' or 'source init_env.sh' in order the exported
# variables to become global
export DEV_SRC_DIR=/mnt/${PROJECT_SRC}
export DEV_BIN_DIR=/mnt/${PROJECT_BIN}

mount -t cifs //192.168.7.1/${PROJECT_SRC} ${DEV_SRC_DIR} -o ro,user=,password=
mount -t cifs //192.168.7.1/${PROJECT_BIN} ${DEV_BIN_DIR} -o user=,password=
```

Once the BBB is powered on, the telnet session established and the shared folder declared we can execute the batch with the `source` command:

```bash
root@beaglebone:~# source init_env_samba_usb.sh
root@beaglebone:~# cd $DEV_BIN_DIR/applications/plc-cape-lab
root@beaglebone:/mnt/plc-cape-bin/applications/plc-cape-lab# ./plc-cape-lab
```

Due to the out-of-tree strategy we have the source-code and the binaries on different folders. The `make` tool must be executed from the source-code folder. To avoid changing of directory at each build we can use the `-C` option (or `--directory`) to indicate where to run the `make`:

```bash
root@beaglebone:~# make -C $DEV_SRC_DIR/plugins/encoder/encoder-ook
```

Any `makefile` in the framework include and use the functionality provided by a common `makefile` named `make_object.mk` (or `make_driver.mk` for the special case of kernel drivers compilation). This file supports two types of building: the out-of-tree recommended strategy but also the classical compiling strategy with the resulting binaries in the same folder as the source code:

- If `DEV_SRC_DIR` and `DEV_BIN_DIR` are defined and different → out-of-tree building
- If `DEV_BIN_DIR` is not defined or equal to `DEV_SRC_DIR` → in-place building

**Bash shell tips**

In previous sections we have seen that the out-of-tree strategy leads to use longer commands which are more difficult to type than the classical `make` call but this is not a big problem thanks to the shortcuts offered by the `bash` shell:

- Arrow keys: up and down keys allow a comfortable navigation over the previously typed commands
- ‘!’ shortcut: the exclamation symbol leads the shell to repeat the most recent command starting with the chars typed after it. Note also that the history of executed commands is preserved even after reboot (see the `history` command). This, for instance, provides an easy way to set up the environment on each new session, which requires the execution of a custom batch file (the `rc.local` booting script could also be used for this task):

14 The `source` command executes the batch file within the current process. If omitted, the batch is executed in a new isolated sub-shell which loses the modifications done in the environment once finished.
Data communication through power lines using a system based on BeagleBone

```bash
#! source init_env_samba_usb.sh
```

- CTRL+R functionality: this shortcut allows retrieving the most recent command matching the pattern typed next. For example, CTRL+R+ook will display the most recent command containing the text ook, which would be our long `make` sentence.
- Finally, you can also open two remote terminals with the BBB: one for compiling, another one for executing the built apps.

### 8.6 Benefits of the out-of-tree building

Some benefits and drawbacks of the out-of-tree building strategy are summarized in the following table:

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simpler backups</td>
<td>More complex handling</td>
</tr>
<tr>
<td>Backups of source code are cleaner and optimized in size because don’t include the data that can be easily regenerated from the sources (binaries, automatic documentation, temporary created files from applications, etc.).</td>
<td><code>make</code> calls are more complex because the path of the source-code must be included each time</td>
</tr>
<tr>
<td>Simpler installers</td>
<td></td>
</tr>
<tr>
<td>As all the binaries are automatically stored over a dedicated base folder, a package with all the binaries of the framework and with the proper tree can be easily created (in a TAR file for instance)</td>
<td></td>
</tr>
<tr>
<td>Extra protection</td>
<td></td>
</tr>
<tr>
<td>Allows an extra protection level to avoid accidental deletion of source code. For example in this project the BBB had read-only access to the source-code to avoid deleting or corrupting some file by mistake. Edition was only allowed in the Desktop-PC</td>
<td></td>
</tr>
<tr>
<td>Flexibility on project organization</td>
<td></td>
</tr>
<tr>
<td>Source-code and compiled-data have different characteristics:</td>
<td></td>
</tr>
<tr>
<td>• modifications on source-code involves just a few kilobytes per saving</td>
<td></td>
</tr>
<tr>
<td>• compilation of the sources usually involves several hundredths of kilobytes</td>
<td></td>
</tr>
<tr>
<td>For this project a Micro SD Card (flash-based media) has been used for the source-code and a hard disk drive for the compiled-data:</td>
<td></td>
</tr>
<tr>
<td>• the Micro SD Card allows the whole project to be easily accessible across different devices (computers, laptops or tablets) without requiring an Internet connection^{15}</td>
<td></td>
</tr>
<tr>
<td>• the compilation of binaries to a HDD avoids the premature wearing-out of the flash media (limited on writable cycles) because the intensive writings associated to the continuous rebuilding process</td>
<td></td>
</tr>
<tr>
<td>• the building process is slightly faster because the higher writing speeds of HDDs (or SSDs) compared to SD Card media</td>
<td></td>
</tr>
</tbody>
</table>

^{15} If an Internet connection is always available a better alternative could be to upload the source code to the cloud via one of the popular source-control systems (GIT, Subversion…) and source-code-hosting services (GitHub, GitLab, SourceForge…)
8.7 Required packages: ncurses

The ui-ncurses plugin contains the source code for a rich User Interface, in text mode but with floating panels and other widgets. It relies on the ncurses third-party library:

```bash
# cd $DEV_SRC_DIR/plugins/ui/ui-ncurses
# make
```

In file included from common.c:8:0:

```
commom.h:15:21: fatal error: ncurses.h: No such file or directory
compilation terminated.
make: *** [/mnt/plc-cape-bin/plugins/ui/ui-ncurses/common.o] Error 1
```

The build fails because the off-the-shelf distribution that comes with the BBB doesn’t include the ncurses package for development:

```bash
# dpkg-query -W -f='${binary:Package;-20} ${Version}	${binary:Summary}
' | grep ncurses
```

<table>
<thead>
<tr>
<th>Package</th>
<th>Version</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>libncurses5:armhf</td>
<td>5.9-10</td>
<td>shared libraries for terminal handling</td>
</tr>
<tr>
<td>libncursesw5:armhf</td>
<td>5.9-10</td>
<td>shared libraries for terminal handling (wide character support)</td>
</tr>
<tr>
<td>ncurses-base</td>
<td>5.9-10</td>
<td>basic terminal type definitions</td>
</tr>
<tr>
<td>ncurses-bin</td>
<td>5.9-10</td>
<td>terminal-related programs and man pages</td>
</tr>
</tbody>
</table>

libncurses5-dev and libncursesw5-dev need to be installed.

Assuming a working Internet connection is available, we can use `apt-get`.

First we need to resynchronize the local index database with fresh content:

```bash
# apt-get update
```

Then we can download and install the required packages:

```bash
# apt-get install libncurses5-dev libncursesw5-dev
```

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  libtinfo-dev
Suggested packages:
  ncurses-doc
The following NEW packages will be installed:
  libncurses5-dev libncursesw5-dev libtinfo-dev
0 upgraded, 3 newly installed, 0 to remove and 173 not upgraded.
Need to get 500 kB of archives.
After this operation, 1915 kB of additional disk space will be used.
Do you want to continue [Y/n]?
Get:1 http://ftp.us.debian.org/debian/ wheezy/main libtinfo-dev armhf 5.9-10 [83.9 kB]
Get:2 http://ftp.us.debian.org/debian/ wheezy/main libncurses5-dev armhf 5.9-10 [194 kB]
Get:3 http://ftp.us.debian.org/debian/ wheezy/main libncursesw5-dev armhf 5.9-10 [222 kB]
Fetched 500 kB in 1s (250 kB/s)
Selecting previously unselected package libtinfo-dev:armhf.
(Reading database ... 59626 files and directories currently installed.)
Unpacking libtinfo-dev:armhf (from .../libtinfo-dev_5.9-10_armhf.deb) ...
Selecting previously unselected package libncurses5-dev.
Unpacking libncurses5-dev (from .../libncurses5-dev_5.9-10_armhf.deb) ...
Selecting previously unselected package libncursesw5-dev.
Unpacking libncursesw5-dev (from .../libncursesw5-dev_5.9-10_armhf.deb) ...
Setting up libtinfo-dev:armhf (5.9-10) ...
Setting up libncurses5-dev (5.9-10) ...
Setting up libncursesw5-dev (5.9-10) ...

After the installation we can check again with `dpkg-query`:

```
# dpkg-query -W -f='${binary:Package;-20} ${Version}	${binary:Summary}
' | grep ncurses
libncurses5:armhf    5.9-10     shared libraries for terminal handling
libncurses5-dev     5.9-10     developer's libraries for ncurses
libncursesw5:armhf   5.9-10     shared libraries for terminal handling (wide character support)
libncursesw5-dev     5.9-10     developer's libraries for ncursesw
ncurses-base         5.9-10     basic terminal type definitions
ncurses-bin          5.9-10     terminal-related programs and man pages
```

Now we can try to compile the ui-ncurses plugin again:

```
# cd $DEV_SRC_DIR/plugins/ui/ui-ncurses
# make
gcc -g -Wall -fPIC -c common.c -o /mnt/plc-cape-bin/plugins/ui/ui-ncurses/common.o
gcc -g -Wall -fPIC -c tui.c -o /mnt/plc-cape-bin/plugins/ui/ui-ncurses/tui.o
gcc -g -Wall -fPIC -c tui_dialog.c -o /mnt/plc-cape-bin/plugins/ui/ui-ncurses/tui_dialog.o
gcc -g -Wall -fPIC -c tui_menu.c -o /mnt/plc-cape-bin/plugins/ui/ui-ncurses/tui_menu.o
-o /mnt/plc-cape-bin/plugins/ui/ui-ncurses/ui-ncurses.so
```

Now we can try to compile the ui-ncurses plugin again:

```
# ls -l $DEV_BIN_DIR/plugins/ui/ui-ncurses
```

8.8 Building hierarchy

There are different `makefile` levels to simplify hierarchical building:

<table>
<thead>
<tr>
<th>Level</th>
<th>Examples</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td># make -C $DEV_SRC_DIR</td>
<td>Compiles the whole framework</td>
</tr>
<tr>
<td>Category</td>
<td># make -C $DEV_SRC_DIR/drivers</td>
<td>Compiles all the projects within a specific category (applications, plugins, libraries or drivers)</td>
</tr>
<tr>
<td></td>
<td># make -C $DEV_SRC_DIR/libraries</td>
<td></td>
</tr>
<tr>
<td></td>
<td># make -C $DEV_SRC_DIR/plugins</td>
<td></td>
</tr>
<tr>
<td></td>
<td># make -C $DEV_SRC_DIR/applications</td>
<td></td>
</tr>
</tbody>
</table>

| Project level| #make -C $DEV_SRC_DIR/applications/plc-cape-lab                        | Compiles just a specific project                 |

Table 29. Building hierarchy

The table indicates the “`make -C`” way. The “`cd <path>`” + “`make`” way can be also used if preferred. For example to build all the applications you can do:

```
# cd $DEV_SRC_DIR/applications
# make
PROJECT building started
LIBRARIES building started
  /mnt/plc-cape-bin/libraries/libplc-tools/application.o
  /mnt/plc-cape-bin/libraries/libplc-tools/file.o
  /mnt/plc-cape-bin/libraries/libplc-tools/plugin.o
...  
  /mnt/plc-cape-bin/applications/plc-cape-lab/ui.o
-> /mnt/plc-cape-bin/applications/plc-cape-lab/plc-cape-lab
APPLICATIONS building finished
```
8.9 Building options

Other remarks about makefile:

- The typical cleaning target (clean) has been defined to delete all the make-generated-output, including the final executable:

  root@beaglebone:~# make clean

  If you face strange problems it is always worth cleaning the project and rebuilding to be sure all the project is up-to-date.

- A special target (clean-build-objects) has been defined in the shared make_object.mk to remove all the intermediate objects of the building process leaving only the final binaries

  root@beaglebone:~# make clean-build-objects

  This simplifies the task of a version generation: it gives the whole tree architecture with only the required binaries. This way the resulting application can be easily packed and installed into another BBB. An installable package (*.deb) will be a better alternative but it’s out of the scope of this project.

- A PROFILE variable has been defined to quickly compile the whole project in Debug or Release modes. By default, make compiles in Debug. If we want to get a Release version (smaller and optimized in speed) we can trigger it when invoking the makefile:

  # make PROFILE=release.bbb

8.10 makefile strategy on user-space components

Pure makefiles are something complex to maintain. We need to update the dependencies depending on source code evolutions. For example if a new #include to a project header is added, we would need to add it to the makefile. Keep this synchronized it’s a tedious task. Development IDEs as Eclipse do it automatically: source codes files are preprocessed, the #includes analyzed and makefiles automatically rebuilt with the proper dependency tree.

In this project I wanted to use the minimum tools for simplicity on the getting started phase and also for portability.

This decision has been taken: makefiles will be split in two parts, a shared makefile with the common declarations (located in $DEV_SRC_DIR/+common/make_object.mk), and an individual makefile for each project with only the particularities.

In this generalization I have accepted a penalty: makefiles can specify more dependencies than necessary if this simplifies the maintenance. The drawback of this approach is that unnecessary recompilations are sometimes triggered, but the simpler maintenance is worth the inconvenience.

In any case, as the project is spread among different building units (libraries, plugins and applications) the compilation time is still contained.
The project-specific makefiles indicate their particularities through variables that the `make_object.mk` file will process.

For the up-to-dated description of the shared variables supported and its meaning take a look at the header of the shared makefile:

```
make_object.mk

# PURPOSE
# Common makefile to be included by plc-cape based makefiles to build executables
# or libraries (dynamic or static)
#
# EXTERNAL VARIABLES
# TARGET: [Required] Compiler to be used (e.g. gcc, g++)
# PROFILE: [Option, default: debug.bbb] Predefined profiles to configure some settings
# (e.g. release.bbb)
# CC: [Optional, default defined by make] Compiler to be used (e.g. gcc, g++)
# ADDITIONAL_CFLAGS: [Optional] Flags to be added to effective CFLAGS
# (e.g. `pkg-config --cflags gtk+-3.0`)  
# ADDITIONAL_LIBS: [Optional] Libraries to be included at linking (e.g. -lrt -lm -ldl -lpthread)
# ADDITIONAL_PLC_LIBS: [Optional] Additional plc-cape libraries (e.g. plc-tools plc-adc)
# ADDITIONAL_PLC_PLUGIN_CATEGORIES: [Optional] Dependencies to plug-in categories (e.g. decoder)
# ADDITIONAL_HEADERS: [Optional] Other dependencies on compiling (e.g. api/*)
# SOURCE_PATH: [Optional] If the source code and the 'makefile' are in different folders
# this variable indicates the relativa path to source (e.g. for makefile in subdir -> ../../)
#
# ENVIRONMENT VARIABLES
# DEV_BIN_DIR: [Optional] absolute path for resulting objects and executables
# DEV_SRC_DIR: [Optional, but required if DEV_BIN_DIR declared] absolute path of the
# root source of the 'plc-cape' project
# Note: if DEV_BIN_DIR is not defined in-place compilation is applied -> compilation
# is done in the folder where the code resides
#
# Make typical targets not to be related to any file
.PHONY: default all display-vars clean clean-build-objects

ifndef TARGET
$(error 'TARGET' undefined)
endif
...
```

On the other hand the private makefiles have this format:

```
makefile

# Specific dependencies for a given component
ADDITIONAL_LIBS = -lrt
ADDITIONAL_PLC_LIBS = plc-tools

# Trick to automatically assign the TARGET name to a component to simplify maintenance
# It is based on the name of the containing folder
TARGET = $(notdir $(CURDIR))

# Inclusion of the main makefile customized for the plc-cape framework
include $(DEV_SRC_DIR)/+common/make_object.mk
```

### 8.11 `makefile` strategy on kernel-modules

The compilation of drivers (kernel modules) has some specific challenges.

Firstly, the `makefile` must follow specific rules because needs to be compliant with the *kbuild* system.

For the SPI driver in this project that condition would lead to a `makefile` like this:

```
drivers/spi/makefile

obj-m += spidev_plc.o spi-omap2-mcspi_plc.o edma_plc.o
```
Data communication through power lines using a system based on BeagleBone

```
all:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Note that if we try to build a driver with the off-the-shelf BBB distribution the process will fail:

```
# cd $DEV_SRC_DIR/drivers/spi

# ls -l
```
```
total 106
-rwxr-xr-x 0 root root 192 Jun  9 14:36 Makefile
-rwxr-xr-x 0 root root 2281 Dec  8 2015 dmaengine.h
-rwxr-xr-x 0 root root 23132 Feb 23 17:36 edma_plc.c
-rwxr-xr-x 0 root root 56336 May 16 17:42 spi-omap2-mcsipi_plc.c
-rwxr-xr-x 0 root root 18449 May  6 10:19 spidev_plc.c
-rwxr-xr-x 0 root root 3907 Dec  7 2015 virt-dma.h
```

```
# make
make -C /lib/modules/3.8.13-bone70/build M=/mnt/plc-cape-src/drivers/spi modules
make: *** /lib/modules/3.8.13-bone70/build: No such file or directory. Stop.
make: *** [all] Error 2
```

To compile a kernel module we need to download and install the headers of the Linux Kernel version we have in our BBB:

```
# apt-get install linux-headers-$uname
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:          
  linux-headers-3.8.13-bone70
0 upgraded, 1 newly installed, 0 to remove and 170 not upgraded.
Need to get 8430 kB of archives.
After this operation, 45.3 MB of additional disk space will be used.
Fetched 8430 kB in 20s (419 kB/s)
Selecting previously unselected package linux-headers-3.8.13-bone70.
(Reading database ... 60058 files and directories currently installed.)
Unpacking linux-headers-3.8.13-bone70 (from .../linux-headers-3.8.13-bone70_1+wheezy_armhf.deb) ...
Setting up linux-headers-3.8.13-bone70 (1+wheezy) ...
```

Then we can retry `make`:

```
# make
make -C /lib/modules/3.8.13-bone70/build M=/mnt/plc-cape-src/drivers/spi modules
make[1]: Entering directory `/usr/src/linux-headers-3.8.13-bone70'
  CC [M]  /mnt/plc-cape-src/drivers/spi/spidev_plc.o
  CC [M]  /mnt/plc-cape-src/drivers/spi/spi-omap2-mcsipi_plc.o
  CC [M]  /mnt/plc-cape-src/drivers/spi/dmaengine.o
Building modules, stage 2.
  MODPOST 3 modules
  CC /mnt/plc-cape-src/drivers/spi/dmaengine.mod.o
  LD [M] /mnt/plc-cape-src/drivers/spi/dmaengine.ko
  CC /mnt/plc-cape-src/drivers/spi/spi-omap2-mcsipi_plc.mod.o
  LD [M] /mnt/plc-cape-src/drivers/spi/spi-omap2-mcsipi_plc.ko
  CC /mnt/plc-cape-src/drivers/spi/spidev.ko
  LD [M] /mnt/plc-cape-src/drivers/spi/spidev.mod.o
make[1]: Leaving directory `/usr/src/linux-headers-3.8.13-bone70'

# ls -l
```
The final binaries of the drivers are the *.ko files: edma_plc.ko, spi-omap2-mcspi_plc.ko and spidev_plc.ko.

The second important consideration, is that, due to the rules imposed by the kbuild system, I’ve not found a proper way to apply the out-of-tree building strategy on drivers. Notice that in the above example we have compiled the driver in the same place as the source-code.

After a deep research, I’ve just found an effective but not-nice alternative based on these steps:

1. The source code of the drivers is temporarily copied to the building folder
2. A basic makefile is then created on-the-fly in that location
3. Then the makefile is executed and the drivers compiled on the output folder
4. The temporary source code is then deleted for cleanliness and avoid edition mistakes

This has been carried out with a common makefile (located in $DEV_SRC_DIR/+common/make_driver.mk) which is included by the makefile of the specific drivers:

```
+common/make_driver.mk

# PURPOSE
# Common make file to build drivers in 'out-of-tree' mode
#
# EXTERNAL VARIABLES
# TARGET: [Required] Name of the folder where the customized driver resides
#
.PHONY: default all on-the-fly-makefile clean clean-build-objects

ifndef TARGET
$(error 'TARGET' undefined)
endif

KDIR := /lib/modules/`uname -r`/build
DRIVER_BIN_DIR = $(DEV_BIN_DIR)/drivers/$(TARGET)
SOURCES = $(wildcard *.c *.h)
OBJECTS = $(patsubst %.c,%.o,$(wildcard *.c))
TMP_SOURCES_AT_BIN_DIR = $(addprefix $(DRIVER_BIN_DIR)/,$(SOURCES))

default: all

$DRIVER_BIN_DIR: 
  @echo " Creating target dir '$@'"
  @mkdir -p $@

# Note: the kernel build system requires a 'makefile' on the source code directory with the 'obj-m' declaration
on-the-fly-makefile: 
  @echo Building temporary '$(DRIVER_BIN_DIR)/makefile'
  ifndef "$$(wildcard $(DRIVER_BIN_DIR)/makefile)"
    @echo "WARNING! There is a makefile in the binary folder '$(DRIVER_BIN_DIR)' that is going..."
    @mv --interactive $(DRIVER_BIN_DIR)/makefile $(DRIVER_BIN_DIR)/makefile.bck
  endif
  @echo obj-m += $(OBJECTS) > $(DRIVER_BIN_DIR)/makefile

all: $DRIVER_BIN_DIR on-the-fly-makefile

# Note 1: The typical 'M=$(PWD)' points to the Makefile's location instead of '$(DRIVER_BIN_DIR)'
```

```
Data communication through power lines using a system based on BeagleBone

# This is required to clean the temporary duplicated files even in case of compilation errors
-@make -C $(KDIR) M=$(DRIVER_BIN_DIR) modules
@echo Cleaning duplicated source code files in '$(DRIVER_BIN_DIR)'
@rm $(TMP_SOURCES_AT_BIN_DIR)
@rm $(DRIVER_BIN_DIR)/makefile

clean: on-the-fly-makefile
@echo Full cleaning of $(DRIVER_BIN_DIR)
@make -C $(KDIR) M=$(DRIVER_BIN_DIR) clean
@rm $(DRIVER_BIN_DIR)/makefile

clean-build-objects: on-the-fly-makefile
@echo Cleaning intermediate objects in $(DRIVER_BIN_DIR)
@rename 's/\.ko/\.koo/' $(DRIVER_BIN_DIR)/*.ko
@make -C $(KDIR) M=$(DRIVER_BIN_DIR) clean
@rename 's/\.koo/\.ko/' $(DRIVER_BIN_DIR)/*.koo
@rm $(DRIVER_BIN_DIR)/makefile

drivers/spi/makefile

TARGET = $(notdir $(CURDIR))
include $(DEV_SRC_DIR)/+common/make_driver.mk

Some notes about the implementation of make_driver.mk:

- To keep the stamps of the files (for the proper operation of the dependency-analyzer on makefiles) the flag --preserve has been used
- Besides, the --interactive flag has been used just to notify if there exists some unexpected code in the output folder in order to avoid overwriting some possible valid code wrongly edited there by mistake

Note that due to the extra processing time and operations required to check if a driver needs to be rebuilt, I’ve removed them from the list of objects to check and compile in the main makefile at the root folder. The comments there are self-explanatory:

makefile

GROUP_DESCRIPTION = PROJECT

# NOTE: Compilation of drivers takes a while. As they are rarely modified
# they are not included by default in the compilation list
# Add a 'drivers' lines to 'MODULES' to include them in the loop or go
# directly to the 'drivers' directory and execute 'make' from there

MODULES = \libraries \plugins \applications

include $(DEV_SRC_DIR)/+common/make_group.mk
8.12 Doxygen

Main target: create automatically fresh documentation from source code avoiding redundant job

Benefits: documentation generated in different formats, with fast cross linking and navigation possibilities

The comments have been written in Doxygen format. This way automatic useful documentation can be created with the up-to-date source code. Moreover, Doxygen allows for different output formats as HTML, PDF, LaTeX, man pages, etc.

Doxygen is available for different platforms. In Windows XP there is a GUI front-end called Doxywizard that simplifies the configuration of the project-based settings (stored by default in a file called doxygen.conf):

![Figure 102. Main screen of the Doxywizard application](image)

Doxygen GUI frontend (X:/doxygen.conf)

File Settings Help

Step 1: Specify the working directory from which Doxygen will run

X:

Step 2: Configure Doxygen using the Wizard and/or Expert tab, then switch to the Run tab to generate the documentation

Wizard Export Run

Provide some information about the project you are documenting

- Project name: pic-capa
- Project summary: Software framework for the PicCape project
- Project version or ID: 0.1

Project logo: [Select...]

Specify the directory to scan for source code

- Source code directory: .
- Scan recursively

Specify the directory where Doxygen should put the generated documentation

- Destination directory: [GEY_BIN_DIR]/doc/doxygen

[Previous] [Next]
And this, the main page (index.html) generated by the tool:

**Figure 103. Home page of the Doxygen documentation**
To be compatible with Doxygen the comments in the source code should follow a specific format, like this:

```c
/**
 * @file
 * @brief Helper functions related to the management of applications
 *
 * @cond COPYRIGHT_NOTES
 * Copyright (C) 2016 Jose Maria Ortega
 * Distributed under the GNU GPLv3. For full terms see the file LICENSE
 * @endcond
 */

#ifndef LIBPLC_TOOLS_SHELL_H
#define LIBPLC_TOOLS_SHELL_H

/**
 * @brief Gets the absolute full path of the running application
 * @return A pointer to the path. Release it with _free_ when no longer required
 */
char *plc_application_get_abs_path(void);

/**
 * @brief Gets the absolute base directory of the running application
 * @return A pointer to the base directory. Release it with _free_ when no longer required
 */
char *plc_application_get_abs_dir(void);

/**
 * @brief Gets the absolute directory where the output results (if any) will be stored
 * @return A pointer to the output directory. Release it with _free_ when no longer required
 */
char *plc_application_get_output_abs_dir(void);
#endif /* LIBPLC_TOOLS_SHELL_H */
```

Notice the double asterisk indicating Doxygen-compatible-format. This the corresponding output:
9 Annex IV. Software design principles

9.1 Object Oriented Programming

Despite the C language (structure-based) a kind of Object-Oriented Programming strategy\(^{16}\) (OOP) has been followed for simpler maintenance.

To accomplish this some patterns have been followed in the code:

- the different *.c files are usually “objects” with the public declarations defined in the corresponding header *.h. The filename correspond to the name of the object
- all the functions are prefixed with the name of the object they belong to (and an underscore)
- the data belonging to the object is always enclosed in a struct (the class concept in C++)
- all the functions of the object (methods in C++ terms) receive a handle as the first parameter. It is indeed a pointer to the data of the object (similar to the implicit this pointer in C++). This data can be directly accessed by the functions belonging to the object but not by other external caller functions (similar to the private clause in C++)
- there is always a ‘create’ function that dynamically allocates memory for the object and initializes it. It is the dual of the C++ constructor
- there is always a ‘release’ function that releases the resources and memory of the object. The first and single parameter of this function is the pointer to the object (previously allocated with a ‘create’ call). It is the dual of the C++ destructor

This could be an example of a basic object, gpio, which would encapsulate the access to the Generic Purpose Input Outputs of the BeagleBone:

```c
#include <gpio.h>

struct gpio
{
    int pin_index;
    void gpio_set(int activate);
    int gpio_get();
};
```

The struct gpio is defined in the *.c file. In the *.h header it is only declared because this way the internal definition of the structure doesn’t affect to other modules depending on (encapsulation principle): the only way other source code files can use the functionality exposed by the object is through the functions publicly declared in the header. This is a key-point that hugely simplifies the maintenance of the source code.

Note that although the definition of the struct gpio is not provided in the header it must still be declared before its first use (forward declaration) due to the C language syntax.

Find following the typical source code implementing the object:

16 Only basic concepts of OOP has been used in this project. The inheritance concept, for example, has not been exploited
Data communication through power lines using a system based on BeagleBone


gpio.c
#include <stdio.h>
#include "+common/api/+base.h"
#include "gpio.h"

struct gpio {
  int pin_index;
  // Other object private data
};

struct gpio *gpio_create(int pin_index)
{
  struct gpio *gpio = (struct gpio*)calloc(1, sizeof(struct gpio));
  gpio->pin_index = pin_index;
  return gpio;
}

void gpio_release(struct gpio *gpio)
{
  free(gpio);
}

void gpio_set(struct gpio *gpio, int activate)
{
  bbb_set_gpio(gpio->pin_index, activate);
}

int gpio_get(struct gpio *gpio)
{
  return bbb_get_gpio(gpio->pin_index);
}

Notice in this file the #include "+common/api/+base.h". It’s a shared file with common declarations (includes, macros, constants, etc.) included in all the plc-cape-based projects. This simplifies maintenance (it’s a good place to comfortably add declarations frequently used).

9.2 Other source code guidelines

These are some other rules followed in the source code design in different areas:

COMMENTS

For comments targeted to the auto-documentation-tool (Doxygen) special syntax is used consisting on a slash plus two asterisks and other control commands (read the chapter 8.12 Doxygen in the annexes for more info):

/** Some Doxygen compatible comment */

For normal comments the double slash format is typically used:

// Some basic comment

HEADERS

For headers which are private to a project the classical pattern is followed:

#ifndef CONTROLLER_H
#define CONTROLLER_H
...
// Project declarations
...
#endif

For public headers shared among different building units (the ones within the api folders) extra care must be taken to avoid conflicts with headers located in other folders that can have the same file name. To
Data communication through power lines using a system based on BeagleBone

avoid this the parent folders are also included in the #define clause. For example this is the header for the error.h file in the libplc-tools project:

```c
#define LIBPLC_TOOLS_ERROR_H
... // Publicly shared declarations
... #endif
```

**INCLUDES**

Includes are grouped in two blocks:

- **system headers** (as `<math.h>`, `<stdio.h>`, …) are delimited by angle brackets (`<>`)
- **project headers** (as “+common/api/error.h”, “libraries/libplc-tools/api/application.h”, …) are delimited by double-quotes

Each block is alphabetically ordered for maintenance simplicity.

There are some base headers (common to all the projects) that need to be included first in the list because other headers rely on. For this, a special syntax has been used: these headers have been prefixed with a plus sign (“+”) which carries out two functions:

- obey with the alphabetic ordering rule (mandatory if using tools that give the possibility to automatically sort the includes like Eclipse)
- inform the user about the special consideration of this type of headers and the importance of any modification made on: any change must be backwards compatible or all the dependent projects will need to be updated. Note also that a modification on a common shared file will force a complete rebuild of the whole project (because it’s a dependency of most components).

An example of a global definition extensively used within the plc-cape framework is the macro `ATTR_EXTERN` used to explicitly declare a public symbol (opposite to static) which is used in libraries or plugins to declare the functions that can be called from outside. There are also types used on this project (e.g. `uint32_t`) which requires the system header `<stddef.h>` to be always present. These are examples of why these special header files need to appear first.

The main shared file (`base.h`) has in fact a double plus sign (`#include ”+common/api/+base.h”`) to appear the first in any case: before any other include in another folder but also before any file within the `+common/api` directory.

**COLUMNS**

A typical limit on Linux applications is 80 columns from the old screen era. Now with typical 16:9 displays with higher resolutions a greater number of columns can be used. For this project the limit has been relaxed to **100 columns** as it renders better readability (mainly when using long names for variables and functions) and it’s still conservative enough to be printed in DIN-A4 pages without the awkward auto word wrapping.

17 The symbol ‘+’ has been chosen among others because seems to be accepted enough by the tools used on this project. Other symbols as ‘#’ are more complex to be used because have special meaning (e.g. make consider it as the beginning of a comment)
INDENT STYLE

All the source code sticks to the Allman-based style (aka BSD-style):

```c
while (x == y)
{
    something();
    somethingelse();
}
```

ENCODING

Source code is in English. The sources have been saved with the encoding Windows-1252 (Western European), a superset of ISO-8859-1. This is only important if special characters (code >= 128) have been used in the source code or in the comments (e.g. accents or the Euro symbol ‘€’).

For the carriage returns two conventions may be present because the use of two Operating Systems in the development:

- Unix based: one LF per line
- DOS/Windows based: one CR + one LF per line

In Linux, the gcc compiler in the BBB is able to automatically manage both conventions. By contrast, bash fails executing scripts with the CR LF format. In such cases, we can normalize a file by using the sed tool:

```
# sed 's/\r$//' cr_lf_content.sh > lf_content.sh
```

In Windows XP we can have some problems reading a LF-based file with some tools. For example the Notepad.exe will display all the content of a LF-based file in a single line; the Visual Studio IDE however manage both conventions properly

PATTERNS AND SYNTAX

For homogeneity I’ve tried to always use the same patterns. Examples:

- variable `ret` for functions returning an `int` indicating success (0) or error (-1)
- suffix `_count` or `_len` to indicate “amount of items”
- extensive use of `calloc` (instead of `malloc`) to initialize objects and structures with zero content

ERROR CONTROL

Due to time constraints, good error control has been postponed for future evolutions. In this first version just basic error checking is carried out through `assert` statements. This is good enough in the development phase because:

- easily detects and indicates unexpected results
- easily points where it will be required to implement a good error control routine

---

19 More related information about character encoding available at:
- [http://www.w3schools.com/charsets/ref_html_ansi.asp](http://www.w3schools.com/charsets/ref_html_ansi.asp)
10 Annex V. Source code files

10.1 Main C source code

For convenience, a copy of the documentation generated by Doxygen, with all the source code included, has been provided together with this memory in an external PlcCapeSoftwareFrameworkDoxygen.zip file.

To see the code, decompress the ZIP, open index.html, and click on the Files button in the top bar.

10.2 Device Tree Overlays

10.2.1 SPI

For PlcCape-v1

```c
/dts-v1/
/plugin/
/{
  compatible = "ti,beaglebone", "ti,beaglebone-black";

  /* identification */
  part-number = "PLC_CAPE_V1-SPI0";

  /* version */
  version = "00A0";

  fragment0 {
    target = &am33xx_pinmux;
    __overlay__ {
      spi0_pins_s0: spi0_pins_s0 {
        pinctrl-single.pins = <
          0x150 0x30 /* spi0_sclk, INPUT_PULLUP | MODE0 */
          0x154 0x10 /* spi0_d0, OUTPUT_PULLUP | MODE0 */
          0x158 0x30 /* spi0_d1, INPUT_PULLUP | MODE0 */
          0x15c 0x10 /* spi0_cs0, OUTPUT_PULLUP | MODE0 */
        >;
      }
    }
  };

  fragment12 {
    target = &spi0;
    __overlay__ {
      /* Std definition hook for redirection to patched drivers */
      compatible = "ti,omap4-mcspi_plc";
      #address-cells = <1>;
      #size-cells = <0>;
      status = "okay";
      pinctrl-names = "default";
      pinctrl-0 = <&spi0_pins_s0>;
      /* D0 & D1 toggled with regards to std configuration*/
      ti,pindir-d0-out-d1-in = <1>;
      spidev0 {
        spi-max-frequency = <24000000>;
        reg = <0>;
        compatible = "spidev_plc";
      }
    }
  };
};
```
Data communication through power lines using a system based on BeagleBone

For PlcCape-v2

```
drivers/spi/dtb/PLC_CAPE_V2-SPI0-00A0.dts

/dts-v1/;
/plugin/;
/
{ 
  compatible = "ti,beaglebone", "ti,beaglebone-black";
  /* identification */
  part-number = "PLC_CAPE_V2-SPI0";
  /* version */
  version = "00A0";

  fragment@0 {
    target = <&am33xx_pinmux>;
    __overlay__ {
      spi0_pins_s0: spi0_pins_s0 {
        pinctrl-single,pins = <
          0x150 0x30 /* spi0_sclk, INPUT_PULLUP | MODE0 */
          /* D0 & D1 with std configuration */
          0x154 0x30 /* spi0_d0, INPUT_PULLUP | MODE0 */
          0x158 0x10 /* spi0_d1, OUTPUT_PULLUP | MODE0 */
          0x15c 0x10 /* spi0_cs0, OUTPUT_PULLUP | MODE0 */
        >;
      };
    };
  }

  fragment12 {
    target = <&spi0>;
    __overlay__ {
      /* Std definition hook for redirection to patched drivers */
      compatible = "ti,omap4-mcspi_plc";
      #address-cells = <1>;
      #size-cells = <0>;
      status = "okay";
      pinctrl-names = "default";
      pinctrl-0 = <&spi0_pins_s0>;

      spidev@0 {
        spi-max-frequency = <24000000>;
        reg = <0>;
        compatible = "spidev_plc";
      };
    };
  }
};
```

10.2.2 ADC

For both versions of PlcCape:

```
drivers/adc/dtb/PLC_CAPE-ADC-00A0.dts

/dts-v1/;
/plugin/;
/
{ 
  compatible = "ti,beaglebone", "ti,beaglebone-black";
  /* identification */
  part-number = "PLC_CAPE-ADC";
  /* state the resources this cape uses */
  exclusive-use =
    /* the pin header uses */
    "P9.39", /* AIN0 */
    "P9.40", /* AIN1 */
    "P9.37", /* AIN2 */
};
```
Data communication through power lines using a system based on BeagleBone

/* the hardware IP uses */
"tscadc";

fragment@ {
    target = <&ocp>;
    __overlay__ {
        /* avoid stupid warning */
        #address-cells = <1>;
        #size-cells = <1>;

        tscadc_hs {
            compatible = "ti,tisccadc_plc";
            reg = <0x44e0d000 0x1000>;
            interrupt-parent = <&intc>;
            interrupts = <16>;
            ti,hwmods = "adc_tsc";
            status = "okay";

            adc {
                ti,adc-channels = <0 1 2 3 4 5 6 7>;
            }
        }
    }
};

10.3 Octave

This is the Octave developed module with the different routines used along this project:

% PURPOSE
% Display results of different tests within the 'plc-cape' project
%
% REMARKS
% Interesting info on plotting control here:
% https://www.gnu.org/software/octave/doc/v4.0.1/Plot-Annotations.html
% https://www.gnu.org/software/octave/doc/v4.0.0/Axes-Properties.html
%
% Main function: execute_test(test_id, base_dir)
% test_id:
%   "adc": adc.csv + FFT
%   "adc+demod": adc.csv + Octave filtering
%   "predef_seq+demod": hard-coded data + Octave filtering
%   "adcN": adc<N>.csv
%   "adc_filter": adc_filter.csv
%   "adc+adc_filter+adc_data": adc.csv + adc_filter.csv + adc_data.csv
% base_dir:
% Linux example: "/media/y/applications/plc-cape-lab"
% Windows example: "Y:/applications/plc-cape-lab"

global adc_sampling_freq = 200000;

function seq = generate_predef_sequence(period)
    data=[0, 1, 0, 1, 1, 1, 0, 1];
    seq_len = 1024;
    assert(mod(seq_len, length(data)) == 0);
    seq_symbol_len = seq_len/length(data);
    seq = sinewave(seq_len,period);
    data_index = 0;
    for i = 1:seq_len
        if (mod(i, seq_symbol_len) == 1)
            data_index++;
        endif
        seq(i) = seq(i)*data(data_index);
    endfor
endfunction
function plot_seq(seq, title_arg, xlabel_arg, ylabel_arg)
    plot(seq);
    axis([1, length(seq)]);
    title(title_arg); xlabel(xlabel_arg); ylabel(ylabel_arg);
endfunction

function f_max = plot_seq_and_display_max(seq)
    figure(1);
    subplot(1,1,1)
    plot_seq(seq)
    [seq_max_value seq_max_index] = max(seq);
    printf("Sequence max = \%f at index \%u (1-based)\n", seq_max_value, seq_max_index);
endfunction

function f_max = plot_seq_and_fft(seq, figure_num)
    global const adc_sampling_freq;
    seq_len = length(seq);
    figure(figure_num);
    subplot(2,1,1)
    plot_seq(seq, 'ADC capture', 'Sample', 'Value')
    subplot(2,1,2)
    dc_mean = mean(seq);
    ac_mean = mean(abs(seq_mean));
    ac_range = max(seq_mean) - min(seq_mean);
    seq_mean_fft = abs(fft(seq_mean));
    plot_seq(seq_mean_fft, 'FFT', 'Digital Frequency', 'Abs(FFT)');
    axis([1, length(seq_mean_fft)]);
    [seq_mean_fft_max seq_mean_fft_max_index] = max(seq_mean_fft);
    f_max = (seq_mean_fft_max_index-1)/seq_len;
    printf("DC mean = %.2f, AC mean = %.2f, AC Range = %.2f, FFT max at %.2f Hz\n", dc_mean, ac_mean, ac_range, f_max*adc_sampling_freq);
endfunction

% 'f_demod' is the frequency to demodulate. Can be 0 for autocalculation
function plot_seq_and_demod(seq, f_demod)
    f_max = plot_seq_and_fft(seq, 2);
    printf("Freq=%d\n", f_max);
    % Note: 'f_demod' should be equal to 'seq_mean_fft_max_index/seq_len'
    % If 'f_demod' not accurately set -> autodetect (for accuracy enough only use on low freqs)
    % Modulation
    % mod-dem = seq_mean.*sin((0:length(seq_mean)-1)*f_demod*2*pi);
    % mod-dem = seq_mean.^2;
    seq_mean = seq - mean(seq);
    demod = abs(seq_mean);
    figure(3)
    subplot(2,1,1)
    plot(demod)
    subplot(2,1,2)
    plot(abs(fft(demod)))
    pkg load signal
    [b, a] = butter(10, 0.1); % Cutoff = 0.1*pi radians
    demod_filt = filter(b,a,demod);
    figure(4)
    subplot(2,1,1)
    plot(demod_filt)
    subplot(2,1,2)
    plot(abs(fft(demod_filt)))
endfunction

function execute_test(test_id, base_dir)
    switch (test_id)
        case "adc"
            in = transpose(load([base_dir "/adc.csv"]));
            plot_seq_and_fft(in, 2);
        case "adc+demod"
            in = transpose(load([base_dir "/adc.csv"]));
            % Experimentalmente encontrados carriers para si Arduino:
            % tx = 23kHz -> f_demod=0.11781 -> Auto=0.11816 (precisión insuficiente)
            % tx = 47kHz -> f_demod=0.2973 -> Auto=0.29785
            plot_seq_and_demod(in, 0);
        case "predef_seq+demod"
            period = 20;
            seq = generate_predef_sequence(period);
            plot_seq_and_demod(seq, 1/period);
case "adcN"
    adc_captures = 4;
    figure(2);
    for i = 0:adc_captures-1
        adc_filename = sprintf("%s/adc%d.csv", base_dir, i);
        in = transpose(load(adc_filename));
        in_len = length(in);
        subplot(adc_captures,1,i+1)
        plot(in)
        axis([1, length(in)]);
    endfor
    return;
endfunction

case "adc_filter"
    in = transpose(load([base_dir "/adc_filter.csv"]));
    figure(4);
    subplot(1,1,1)
    plot(in)
    axis([1, length(in)]);
endfunction

case "adc+adc_filter+adc_data"
    in = transpose(load([base_dir "/adc.csv"]));
    subplot(3,1,1)
    plot(in)
    in = transpose(load([base_dir "/adc_filter.csv"]));
    subplot(3,1,2)
    plot(in)
    axis([1, length(in)]);
    in = transpose(load([base_dir "/adc_data.csv"]));
    % bar, barh, stairs are other interesting graphs
    subplot(3,1,3)
    stem(in)
    axis([1, length(in)]);
    return;
otherwise
    printf("Unexpected 'test_id' type %d
", test_id);
endswitch
endfunction

% execute_test("adc")
11 Bibliography

All the bibliographic references (except if noted otherwise) are provided as online URL links which has been accessed and validated on September 2016.

11.1 BeagleBoneBlack

[1] BeagleBoard.org - community supported open hardware computers for making
http://www.beagleboard.org/

http://beagleboard.org/support/bone101

http://elinux.org/Beagleboard:BeagleBoneBlack

http://elinux.org/BeagleBone_and_the_3_8_Kernel

http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

http://elinux.org/Beagleboard:BeagleBone_Capes

http://elinux.org/Beagleboard:Cape_Expansion_Headers

https://cdn-shop.adafruit.com/datasheets/BBB_SRM.pdf

[9] AM3358 | AM335x | ARM Cortex-A8 | Description & parametrics
http://www.ti.com/product/AM3358

http://www.ti.com/lit/ug/spruh73n/spruh73n.pdf


http://exploringbeaglebone.com/

http://exploringbeaglebone.com/chapter13/

[14] The Beaglebone for Embedded Linux Development | derekmolloy.ie
http://derekmolloy.ie/beaglebone/

11.2 Linux

[15] Linux - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/Linux

[16] GitHub - torvalds/linux: Linux kernel source tree
https://github.com/torvalds/linux/

[17] Linux/ - Linux Cross Reference - Free Electrons
http://lxr.free-electrons.com/
11.3 GNU packages

[29] Software - Proyecto GNU - Free Software Foundation
   http://www.gnu.org/software/

[30] ncurses
   https://www.gnu.org/software/ncurses/

[31] GNU - Free Software Directory
   https://directory.fsf.org/wiki/GNU

[32] Welcome to IT++!
   http://itpp.sourceforge.net/4.3.1/

11.4 C Language

   https://www.cs.rit.edu/~ats/books/ooc.pdf

11.5 PLC

[34] Power-line communication - Wikipedia, the free encyclopedia
   https://en.wikipedia.org/wiki/Power-line_communication

[35] Machine to machine - Wikipedia, the free encyclopedia
   https://en.wikipedia.org/wiki/Machine_to_machine

[36] System on Module for Industrial Power Line Communication (CENELEC Frequency Band) - TIDM-SOMPLC-INDUSTRIAL-CENELEC - TI Tool Folder
   http://www.ti.com/tool/tidm-somplc-industrial-cenelec
Data communication through power lines using a system based on BeagleBone

[37] IEEE International Symposium on Power Line Communications and its Applications
http://www.ieee-isplc.org/

[38] European Committee for Electrotechnical Standardization - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/European_Committee_for_Electrotechnical_Standardization

[39] Welcome to CENELEC – European Committee for Electrotechnical Standardization
https://www.cenelec.eu/

11.6 SDR

[40] Software-defined radio
https://en.wikipedia.org/wiki/Software-defined_radio

[41] GNU Radio
http://gnuradio.org/
http://gnuradio.org/doc/doxygen/index.html

11.7 AFE031

[42] AFE031 | Precision Op Amps | Operational Amplifiers (Op Amps) | Description & parametrics
http://www.ti.com/product/afe031


11.8 Tools

[45] GNU Octave
https://www.gnu.org/software/octave/index.html

[46] GNU Octave Wiki – Octave
http://wiki.octave.org/GNU_Octave_Wiki

[47] Digital Modulation - MATLAB & Simulink

[48] Doxygen: Main Page
http://www.doxygen.org/
http://www.stack.nl/~dimitri/doxygen/