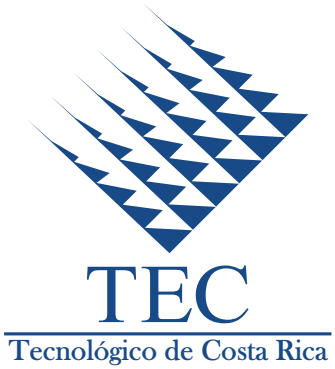
INSTITUTO TECNOLÓGICO DE COSTA RICA



GRADUATION PROJECT REPORT TO QUALIFY FOR ELECTRONICS ENGINEER DEGREE.

Low power embedded software optimization for the NuttX RTOS

Author:
Diego Sánchez López

January 17, 2013

Declaración de Autenticidad

Declaro que el presente Proyecto de Graduación ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía, he procedido a indicar las fuentes mediante las respectivas citas bibliográficas.

En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.

Cartago, January 17, 2013

Diego Sánchez
ID: 1-1345-0406

INSTITUTO TECNOLOGICO DE COSTA RICA
ESCUELA DE INGENIERIA ELECTRONICA
PROYECTO DE GRADUACIÓN
TRIBUNAL EVALUADOR
ACTA DE EVALUACIÓN

Proyecto de Graduación defendido ante el presente Tribunal Evaluador como requisito para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

Estudiante: **Diego Sánchez López**

Nombre del Proyecto:

Optimización de software para bajo consumo de potencia en NuttX RTOS

Miembros del Tribunal

Ing. Javier Pérez Rodríguez

Profesor Lector

Ing. William Marín Moreno

Profesor lector

Ing. Johan Carvajal Godinez

Profesor Asesor

Los miembros de este Tribunal dan fe de que el presente trabajo de graduación ha sido aprobado y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica

Cartago 23 de Enero del 2013

Abstract

This paper presents the study of the implementation for a new feature that allows the NuttX RTOS, handling the power consumption in order to optimize it. The project was focused on controlling different power states, following the global trend in portable devices.

In this document, it's explained all the necessary information that was necessary before to start developing all the modules, and also contains all the steps that were followed in order to get the final system. After this project, NuttX is able to handle 4 different power modes, this power modes were designed in order to reduce the power consumption gradually when the RTOS is idle, and also the system is able to return to its higher performance mode when it's required.

Part of the requirements in this project was to built a benchmark application in order to verify the functionality of this new feature, in the document is explained not only how this application works, but also all the hardware used in order to acquire the data that were used in the analysis.

Contents

List of Figures

List of Tables

Chapter 1

Project Background

1.1 Introduction

The global trend for all the applications and systems is to develop and increase their portability, this is the reason why, power management turns into a high priority issue in the world of the embedded software.

The need of power consumption optimization in embedded systems comes from the increase in the production of green energy and the usage of battery powered devices, like cellphones, tablets and laptops. This is even more imperative in embedded system given their applications in medical devices, aeronautical systems, alarms, among others, due to the need for extending the battery life.

Military research budget in power management is constantly increasing[11], given the advantages of portable devices in warfare, like the use of a fully integrated and safe fuel cell and battery. This could mean a significant aid for soldiers, because of safe and lightweight power sources for non-stop equipment operation in the field as an alternative to carrying extra batteries and recharging systems.

Zuquim explains in [18] that due to the evolution in the component miniaturization process and the development of high-speed wireless network technologies, there was an increase in use of mobile devices in the last few years. So, the development of reliable technologies enabled and increased demand for mobility and changed the focus of computing systems.

Nowadays, all the processors and microcontrollers have several power-saving modes, but that feature of the hardware needs to be controlled by software routines. NX-Engineering costumers are concerned about the power management of their projects, which being developed using NuttX RTOS[1], despite of the fact that NuttX has a very small footprint (tens of KBs), it does not have a

[1]See section 2.2 for more NuttX details

power management sub-system on any platform.

As explained in [3], power management (PM) for computer systems has traditionally focused on regulating the power consumption in static modes such as *sleep* and *suspend*, but this approach results in significant latencies and overheads for such states when user action is required..

1.1.1 Problem statement

The NuttX RTOS works in several platforms[2], such as: PIC32MX (MIPS), ARM, Atmel AVR, Intel, Renesas/Hitachi and Zilog. Nowadays, the system has been tested with many development boards that support peripherals like: USB, Network (Ethernet), SDIO, buttons, CAN, USART, touchscreens, etc.

At the moment of starting this project, NX-Engineering has a the commercial need for an improvement in the RTOS in order to run an application in a portable device, with the most important peripherals including USB, Color LCD, touchscreen, etc. However, the system is not able to work with a battery as power supply for a long time, due his high power consumption, so, this turns into a high priority issue for some of the company projects. Hence, NX-Engineering requires a substantial reduction in the power consumption of the NuttX RTOS in a specific platform which will be used for a portable application.

So, it's possible to summarize the problem statement as a lack of energy efficiency of the NuttX RTOS in battery-powered platforms.

1.2 Power management strategy

The solution presented consists in creating a power management sub-module, which needs to estimate the activity of the operating system, therefore, there are some crucial aspects to consider; such the fact that the PM sub-module must apply an algorithm in order to compute if the system should lower down the power consumption or increase it. To achieve this the PM will take into account the activity, that will basically, be an accumulator of the activity given by the applications and drivers. Also, it's important to select a good platform for the first power management in NuttX, which at least must have the following features.[3]

A) Must be a prototyping board.

[2] see table B.1

[3] All these features have been suggested by NX-Engineering in order to develop a suitable example for the customers.

B) Microprocessor or microcontroller which provides at least 3 power-down modes.

C) LED's.

D) Two or more user buttons.

E) Color LCD with resolution 240×230 or higher, with controllable backlight pin.

F) NuttX support for all the necessary peripherals.

G) RTC.

H) USART or USB, to run NSH.

According with the requirements listed above, the table 1.1 shows each evaluation board that has been considered for this project.

Table 1.1: Platforms considered for the project development

Feature \ Platform	STM3210e[13]	STM3220e[14]	STM3240g[15]	SAM3u-ek[2]
A	✔	✔	✔	✔
B	✔	✔	✔	✘
C	✔	✔	✔	✔
D	✔	✔	✔	✔
E	✔	✘	✘	✔
F	✔	✔	✔	✔
G	✔	✔	✔	✔
H	✔	✔	✔	✔

✔ Board meets the requirement.
✘ Board does not fulfil the requirement.

As shown in the table 1.1 the *STMicroelectronis* boards analyzed as potential platforms have basically the same hardware built in all of them, but with a different processor. The case of the *STM3220e & STM3240g* were ruled out because the pin which controls the backlight is connected to VDD and there is no control over the energy consumed for that device. As consequence, the

selected platform was the *STM3210e-eval*, from *STMicroelectronics* as shown in the figure 1.1, since it's the only one which satisfies all the requirements of the project. A description of the peripherals for this board is described in the layout, see figure 1.2.

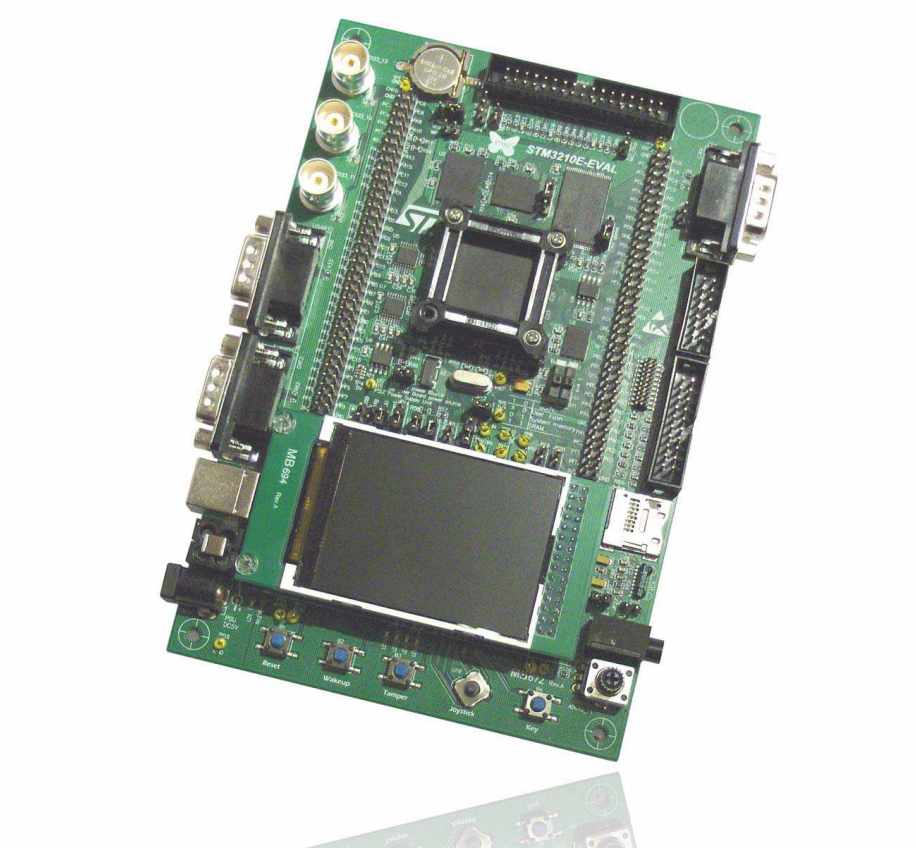


Figure 1.1: STM3210e-eval evaluation board.

As well as being a high performance microcontroller, the STM32 has several low power modes (see table 2.1). *The insiders's guide to the STM32 ARM based microcontroller* [17] comments that, when used judiciously, the SLEEP, STOP and STANDBY low power modes make powering applications from batteries a practical prospect. The STM32 is a low power microcontroller with a high performance processor. In the section 2.1.1 there's a summary of each of the power modes, with a comparison of their power consumption and wake up timing as well.

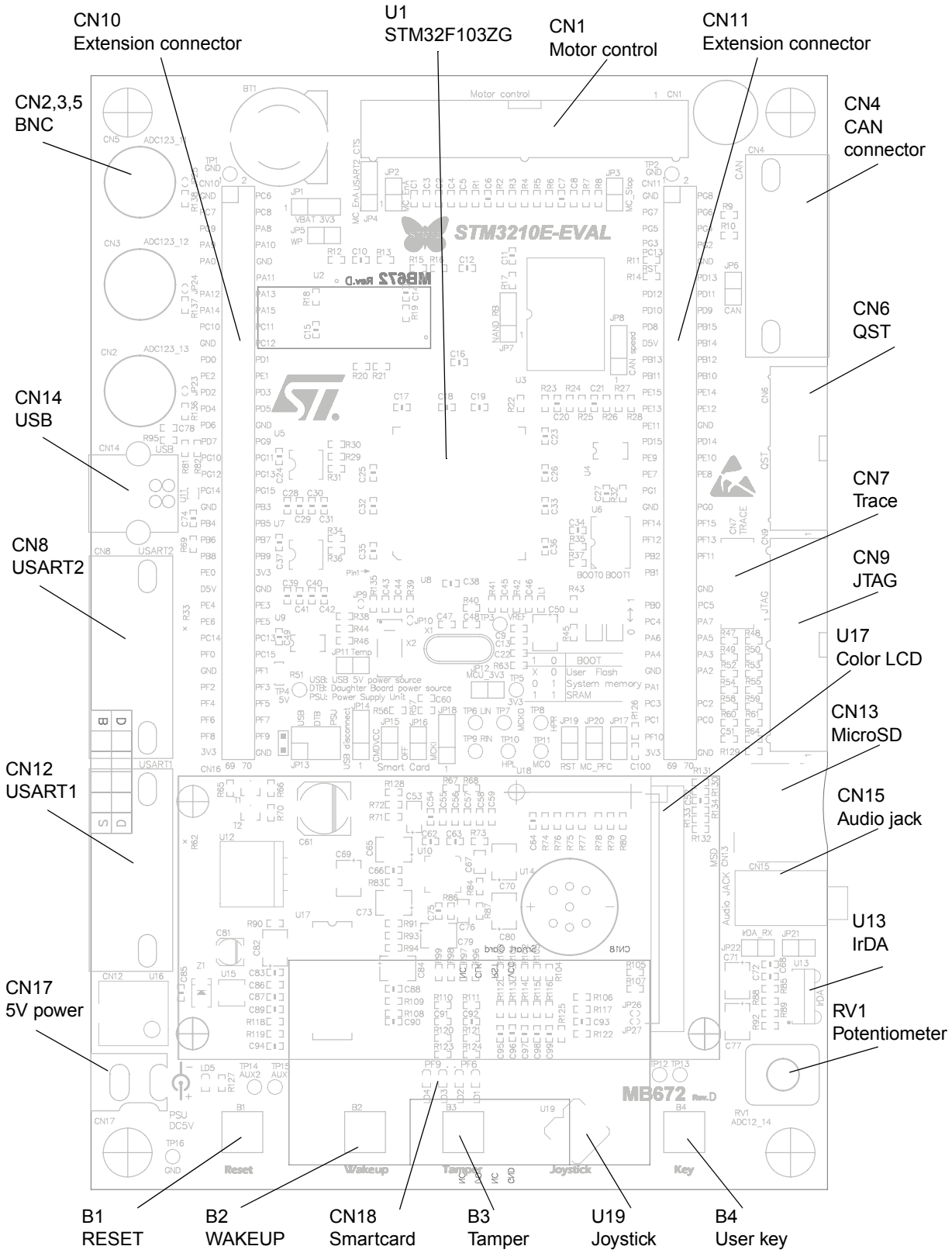


Figure 1.2: STM3210e-eval evaluation board layout.

1.3 Goal and objectives

1.3.1 Goal

Achieve at least a 50% reduction in the power consumption of NuttX RTOS.

1.3.2 Overall Objective

Design the necessary modules and driver modifications in the NuttX RTOS, in order to double the battery life in portable applications.

1.3.3 Specific Objectives

Research about potential platforms (development boards) for the project implementation, taking into account the power handling features and the NuttX support.[4]

Design the power management logic that provoke a change in the power mode and the action to be taken in each power state.

Implement all the necessary modifications in the drivers and applications, in order to handle specific actions when a power state change event occurs, and report this individual activity to the PM sub-module.

Develop a benchmark application in order to demonstrate the functionality of the power management in a portable application.

1.4 Document structure

This document contains the methodological process, and the results of the design and implementation of a power management subsystem for the NuttX RTOS. The structure of this document is the following: the chapter 3 contains a briefly explanation of the design and implementation process, the chapter 2 includes all the theory necessary for the implementation of the solution, the chapter 4 gives in-depth explanation of the whole development that has been done; the chapter 5 contains all the obtained results in this project and a comparison between the system with power management sub-system and without it, and the chapter 6 is reserved for all the transcendental conclusions and recommendations.

[4]The requirements for the platform is shown in the section 1.2

Chapter 2

Theoretical framework

This chapter includes all the necessary research, and previous information, that was required for the development of a low power embedded software optimization for the NuttX RTOS.

2.1 STM32 Platform

NuttX supports a list of 31 different boards which uses ARM MCUs (see table B.1). The present work has been developed in a STM32 board, the reason why this evaluation board has been selected is explained in the chapter 5. In addition, NuttX supports 3 different STM32Fxx boards[1], and the *STM3210e-eval*[2] board was chosen as explained above.

2.1.1 MCU Low-power modes

It's important to notice that the power modes designed for the NuttX (see figure 4.2) do not apply for a specific MCU, some of them have various names for the lower power modes and the most of the time are used in conflicting ways. For example, the lowest power consumption in the STM32F1xx microcontroller is called `Standby`, while in NuttX, the lowest power state defined in the section 4.1 is `PM_SLEEP`, which at the same time, might be contradictory with one of the energy states in the STM32F1 MCU.

The STM32F10xxx devices feature three low-power modes [12]:

- **Sleep mode:** CPU clock off, all peripherals including Cortex-M3 core peripherals like NVIC, SysTick, etc. are kept running.
- **Stop mode:** All clocks are stopped.

[1] STM3210e-eval, STM3220e-eval, STM3240g-eval
[2] see figure 1.1

- **Standby mode:** $1.8V$ domain powered-off.

The table 2.1 shown the summary of the low-power modes for the STM32 MCUs.

Table 2.1: STM32 low-power mode summary [12]

Mode name	**Entry**	**Wakeup**	**Effect on 1.8V domain clocks**	**Effect on V_{DD} domain clocks**	**Voltage regulator**
Sleep(Sleep now or Sleep-on-exit)	`WFI`	Any interrupt	CPU clock OFF no effect on other clocks or analog clock sources	None	ON
	`WFE`	WakeUp Event			
Stop	`PDDS` and `LPDS` bits `+ SLEEPDEEP` bit `+ WFI` or `WFE`	Any EXTI line (configured in the EXTI registers)	All 1.8V domain clocks OFF	HSI and HSE oscillators OFF	ON or in low-power mode (depends on Power control register)
Standby	`PDDS` bit `+ SLEEPDEEP` bit `+ WFI` or `WFE`	WKUP pin rising edge, RTC alarm, external reset in NRST pin, IWDG reset			OFF

STM32 RUN mode

When the system is in `PM_NORMAL` or `PM_IDLE` (see figure 4.2) the processor will be in RUN mode. In the STM32 processors this occurs when the MCU is executing program instructions at its highest level of power consumption, however, another actions should be taken for reducing the power consumption in the system.

During normal operation the Cortex processor and most of the STM32 in NuttX can run at 72MHz, when it is running at full speed, the STM32 consumes in excess 30mA. The power consumption of the STM32 can be reduced first by gating the clocks of any unused peripherals. This stops any unused areas of the chip from consuming power. The peripheral clocks can be switched on and off dynamically through the Reset clock control module.

STM32 SLEEP mode

This is the first level of low power operation, notice that it might lead to confusion with the `PM_SLEEP` (see figure 4.2). By default, when an `WFE` or `WFI` instruction is executed the Cortex processor will halt its internal clocks and stop executing the application code. The STM32 will leave SLEEP mode when a peripheral generates an interrupt.

STM32 STOP mode

The STM32 can be configured to enter the low power STOP mode by setting the `SLEEPDEEP` bit in the Cortex power control register [16]. Like in the SLEEP mode, this mode could be confused, in

this case with the `PM_STOP` mode (see figure 4.2) from the NuttX low power modes.

When the STOP mode is set, execution of `WFE` or `WFI` instruction will halt the Cortex processor and switch off the `HSI` `HSE` oscillators. In spite of this, the FLASH, SRAM and peripherals are still powered, so the state of the STM32 is preserved. Like SLEEP, the STOP mode can be awakened via an STM32 peripheral generate an interrupt. The use of the EXTI peripheral allows the STM32 to exit STOP mode when there is a state change on any GPIO pin.

STM32 STANDBY mode

Like the above modes, this power mode should not be confused with the `PM_STANDBY` mode (see figure 4.2). This architecture can be configured to enter its standby mode by setting the `SLEEPDEEP` bit in the Cortex power control register while setting the Power Down Deep bit. In this way, when the `WFE` or `WFI` instructions are executed, the MCU will drop into its lowest power mode. In this mode the core is completely switched off. In this mode the STM32 consumes a mere 2uA [17].

2.2 NuttX RTOS

NuttX is a real time embedded operating system (RTOS). NuttX, as with all RTOSs, is a collection of various features bundled as a library[7]. Its goals are:

- Small footprint usable in deeply embedded environments.
- Fully scalable from tiny (8-bit) to moderate (32-bit).
- Standards compliance.
- Real time.
- Totally open source.
- GNU toolchains.

2.2.1 Real Time

An operating system is composed by different parts: a file system, memory allocation, handling I/O, network, command shell and scheduler [1]; it is said that an operating system runs in real time because, the amount of time it takes to accept and complete an application's task is estimable, the variability of this time is the jitter, in an RTOS the jitter is lower in comparison with a standard operating system. This time is directly related with the scheduler of the OS.

The scheduler of an operating system decides when to run which process. There are different types of scheduler algorithms, NuttX implements a combination of these algorithms, if a new application needs to be prioritized, it's important to know how its scheduler works. In this section it is briefly explained the different scheduling techniques in operating systems.

The states of a process, at any given time, is comprised of the minimal set that is described in the figure 2.1, where the state *running* specifies when the CPU is currently executing the code belonging to the process, *ready* when the process could be running, but another process has the CPU, and the state *waiting* when some external events must occur once that the process finished its execution [1].

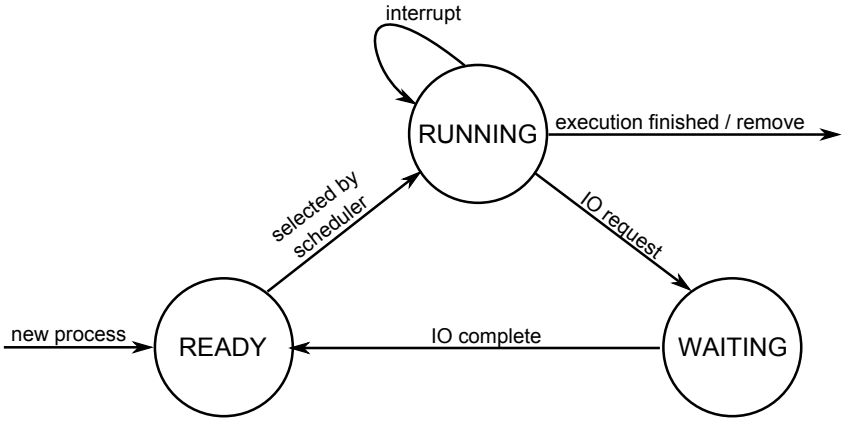


Figure 2.1: States of a process in an operating system.

Scheduler FCFS

The FCFS (First-come, First-Served) scheduler simply executes processes to completion in the order they are submitted. The FCFS is implemented using a queue data structure. Given a group of processes to run, insert them all into the queue and execute them in that order.

Scheduler SFJ

The SJF (Shortest-Job-First) scheduler is exactly like FCFS except that instead of choosing the job at the front of the queue, it will always choose the shortest job (i.e. the job that takes the least time) available.

Scheduler RR

RR (Round-Robin) is a preemptive scheduler, which is designed especially for time-sharing systems. In other words, it does not wait for a process to finish or give up control. In RR, each process

is given a time slot to run. If the process does not finish, it will "get back in line" and receive another time slot until it has completed.

Scheduler PRI

A PRI (Priority) scheduler is associated with each process. Actually the SJF algorithm is a special case of PRI. Processes with equal priorities may be scheduled in accordance with FCFS, but when are prioritized this is one of the best algorithms.

NuttX Scheduler

The NuttX RTOS implements a fully pre-emptible, fixed priority and round-robin scheduling. It ensures that the whole time it will be executing the highest priority process. Pre-emptible means that once a process starts executing, allow it to continue until it voluntarily yields the CPU.

In this project the scheduler is used to create a new task, and ensure that this will be executed in the lowest priority, this should be the idle tread, and is the only process which is able to have priority zero in the NuttX RTOS.

2.2.2 NuttX directory structure

It is very important to know the whole structure of the NuttX directories, in order to choose the right place to add the new functions and methods. The general directly layout for NuttX is very similar to the directory structure of the Linux kernel (at least at the most superficial layers) [5]. At the top level is the main makefile and a series of sub-directories as identified in the appendix C.

NuttX is used globally for industrial purposes, individual projects, even in worldwide universities, due it's non-restrictive BSD license, this is why its important to put the code in the right place, otherwise, might cause problems to many users.
To give a concrete example: when the STM32-specific logic is added, this code should goes in the next path:

Directory Structure

```
- nuttx
   '-- arch/
       '-- arm/
           |-- include/
           |   |--stm32/
           |   |  '-- (chip-specific header files)
           '-- src/
```

```
            '--stm32/
               '-- (chip-specific source files)
```

On the other hand, when the PM logic that corresponds to the board specific (such as: LED's or LCD) is added, all the code should goes in the next directory:

Directory Structure

```
- nuttx
  '-- configs/
       '-- stm3210e-eval/
           |-- include/
           |   '-- (board-specific header files)
           |-- src/
           |   '-- (board-specific source files)
           '-- pm/
               '-- (board configuration-specific for power management)
```

2.2.3 Configurations

A new configuration was created in this project, in order to create an application reference (see section 4.6) that can be tested in an specific board, that is the reason why this topic takes a high importance for the design and further implementation of this project.

The `configs/<board-name>/` sub-directory holds all of the files that are necessary to configure NuttX for a particular platform. A board may have various different configurations using the common source files. Each board configuration is described by three files: `Make.defs`, `defconfig`, and `setenv.sh`. Typically, each set of configuration files is retained in a separate configuration sub-directory (`<config1-dir>`, `<config2-dir>`, ..., `<confign-dir>`).

Make.defs

This makefile fragment provides architecture and tool-specific build options. It will be included by all other makefiles in the build (once it is installed). This make fragment defines:

- Tools: CC, LD, AR, NM, OBJCOPY, OBJDUMP.
- Tool options: CFLAGS, LDFLAGS.
- COMPILE, ASSEMBLY, ARCHIVE, CLEAN, and MKDEP macros.

When this makefile fragment runs, it will be passed TOPDIR which is the path to the root directory of the build. This makefile fragment may include `${TOPDIR}/.config` to perform configuration specific settings. For example, the CFLAGS will most likely be different if `CONFIG_DEBUG=y`.

defconfig

This is a configuration file similar to the Linux configuration file. In contains variable/value pairs like:

- `CONFIG_VARIABLE=value`

This configuration file will be used at build time, as a makefile fragment included in other makefiles, and to generate `include/nuttx/config.h` which is included by most C files in the system.

appconfig

Is a special configuration file used to configure which applications are to be included in the build. This file is copied into the application build directory when NuttX is configured. The `appconfig` file is copied into the `apps/` directory as `.config` when NuttX is configured. `.config` is included in the toplevel `apps/Makefile`.

As a minimum, this configuration file must define files to add to the `.CONFIGURED_APPS` list like:

- `CONFIGURED_APPS  += examples/nx_hello`

setenv.sh

This is a script that the user includes, it shall be installed at the top level of the directory structure and can be sourced to set any necessary environment variables. The user have to customize the default `setenv.sh` script in order for it to work correctly in an specific environment.

It includes the path of the path of the toolchain that the user prefers for compiling the Nuttx; for example, this project was built using cygwin under Windows, with the toolchain CodeSourcery provided by mentor graphics sourcery tools, under this conditions this file should includes a line as the following:

- `export TOOLCHAIN_BIN="/cygdrive/c/Program Files (x86)/CodeSourcery/Sourcery G++ Lite/bin"`

2.2.4 NuttShell

The `apps/nshlib` sub-directory contains the NuttShell (NSH) library [9]. This library can easily to linked to produce a NSH applications, in this project two built-in applications have been added in order to contribute to the application reference which will are explained in the section 4.6.

Using settings in the configuration file, NSH may be configured to use either the serial `stdin/out` or a `telnet` connection as the console or both. When NSH is started, it shows the following welcome on either console:

```
NuttShell (NSH)
nsh> |
```

NSH is a simple shell-like application. At present [9] , NSH supports the following commands forms:

```
Simple command:                    <cmd>
Command with re-directed output: <cmd> > <file>
                                   <cmd> >> <file>
Background command:                <cmd> &
Re-directed background command:  <cmd> > <file> &
                                   <cmd> >> <file> &
```

Where:

`<cmd>` is any one of the simple commands (see [9] for more details).

`<file>` is the full or relative path to any writeable object in the file-system name space (file or character driver).

NSH executes at the mid-priority (128) (see [9]). Backgrounded commands can be made to execute at higher or lower priorities using nice:

`[nice [-d <niceness>>]] <cmd> [> <file>|>> <file>] [&]`

Where `<niceness>` is any value between -20 and 19 where lower (more negative values) correspond to higher priorities. The default niceness is 10.

An if-then[-else]-fi construct is also supported in order to support conditional execution of commands. This works from the command line but is primarily intended for use within NSH scripts. The syntax is as follows:

```
if <cmd>
then
```

```
    [sequence of <cmd>]
  else
    [sequence of <cmd>]
  fi
```

2.3 Drivers and applications

This section includes all the necessary information for the drivers and applications which are intended to support energy management. It's of prime importance to understand how the drivers work before to try to modify them. In this case, all these drivers are being modified, so they can have extra functionalities for the power handling.

2.3.1 Buttons GPIOs

A driver that controls the buttons embedded in the board was implemented, the idea is to detect the activity of a possible user. When the initialization function is called, it configures all the buttons of the STM3210e-eval board as external interrupts, so any button is able to wake up the MCU from almost all the low power modes, since the `PM_SLEEP` mode can be awakened only if it goes through a reset event.

2.3.2 Color LCD

The key in the power management for this driver is how it responds to a state change in the system. This is more important than reports activity, because the activity of this driver does not reflect any user interaction, contrary to the keyboard (which works trough serial driver) or the buttons, just to mention some examples.

Once the power management is registered with this driver, it should be able to control the backlight using a PWM. The figure 2.2 shows the schematic of the TFT LCD connection in the STM3210e-eval board. It was extremely important to remove the resistor R107 and fit it in the R105, this is because this evaluation board has the LCD backlight control pin directly connected to VDD by default. This change was necessary for controlling the pin #22, as shown in the 2.2.

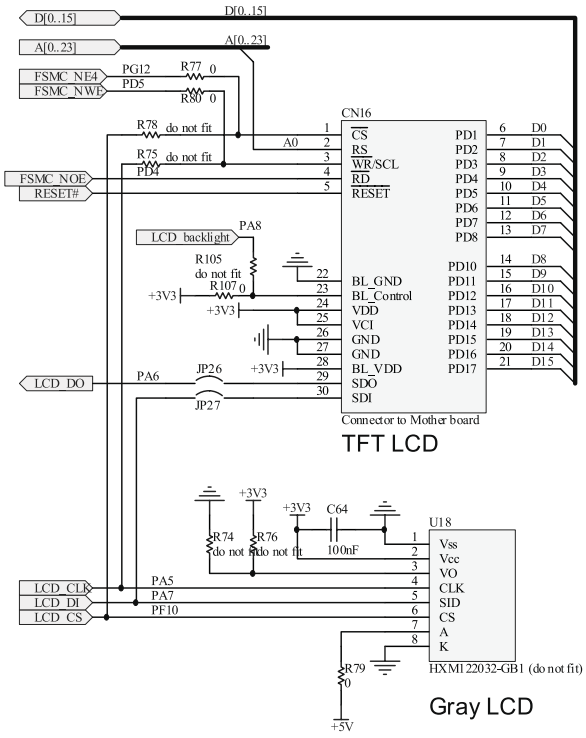


Figure 2.2: Schematic of the TFT LCD in the STM3210e-eval board [13].

2.3.3 Serial driver

This was one of the drivers which required some extra modifications to accept all the callback functions, and perform some logic for both, reduce the power consumption and taking the system back to the normal power mode. In order to achieve that, it was necessary to understand how the driver handles the serial communication and how to introduce new functions without affecting the behavior of the communication. It was important to analyze the schematic as well, in order to know if there will be a way to reduce the power consumption with this hardware.

This driver is one of the most important for this project, since this is the UI with the RTOS. Despite that NuttX can be executed through the color LCD or ethernet, this project use the serial driver, due to its simplicity and because it wasn't an objective of this project to use a complex interface, in addition, using the serial protocol and an application reference the power management can be tested out.

The figure 2.3 shows the schematic for the serial hardware which is embedded in the STM3210e-eval board. All the schematics were always necessary in order to verify the power consumption for each peripheral.

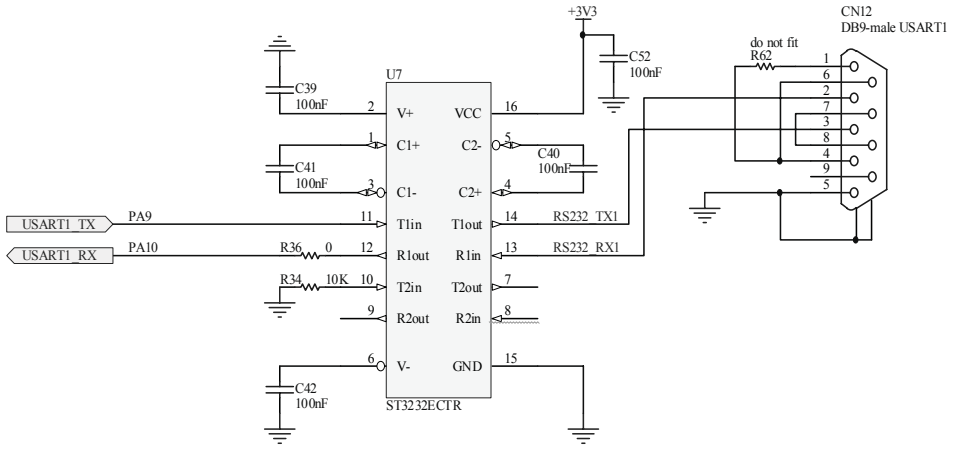


Figure 2.3: Schematic of the USART in the STM3210e-eval board [13].

2.3.4 LEDs Application

The STM3210e-eval board, has 4 different LEDs that are used in this architecture (see figure 2.4), in order to debug the code execution, using these four LEDs the NuttX represents 8 different events, for example when occurs: an interruption, an assertation, or a panic warning light. It is important to understand these states, whose are defined in the file `../configs/stm3210e-eval/include/board.h`, and encoded as follows:

- `LED_STARTED` is the value that describes the setting of the LEDs when the LED logic is first initialized. This LED value is set but never cleared.
- `LED_HEAPALLOCATE` indicates that the NuttX heap has been configured. This is an important state in the boot sequence because if the memory is configured wrong, it will probably crash leaving this LED setting. This LED value is set but never cleared.
- `LED_IRQSENABLED` indicates that interrupts have been enabled. Again, during bring-up (or if there are hardware problems), it is very likely that the system may crash just when interrupts are enabled, leaving this setting on the LEDs. This LED value is set but never cleared.
- `LED_STACKCREATED` is set each time a new stack is created. If set, it means that the system attempted to start at least one new thread. This LED value is set but never cleared.
- `LED_INIRQ` is set and cleared on entry and exit from each interrupt. If interrupts are working okay, this LED will have a dull glow.
- `LED_SIGNAL` is set and cleared on entry and exit from a signal handler. Signal handlers are tricky so this is especially useful during bring-up or a new architecture.
- `LED_ASSERTION` is set if an assertion occurs.

- `LED_PANIC` will blink at around 1Hz if the system panics and hangs.

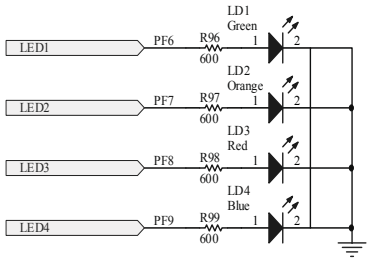


Figure 2.4: Schematic of the LED's in the STM3210e-eval board [13].

Since this is a debugging tool, we won't want to disable the LEDs for reducing the energy consumption, unless the system gets into the lower power consumption mode. Another important consideration, is that this driver initializes very early in power-up initialization, when the power management is still uninitialized, as result, the power management for this driver it's called later, when the LEDs are already working.

2.4 Power consumption

There are a number of approaches that can be taken when trying to determine the power consumption of a device. In this project, the current was acquired and logged into a file for further analysis, as long as the voltage is known the formula for power might be applied, as follows:

$$W = \frac{I}{V} \tag{2.1}$$

Where:

$W =$ Power $[Watts]$

$I =$ Current $[Amperes]$

V Voltage $[Volts]$

The final device for this project, is a portable hardware that optimizes the battery life as much as possible, so, the energy in joules is a good variable to analyze due it involves the usage time.

$$E = W \cdot t \tag{2.2}$$

Where:

$E =$ Energy [$Joules$]

$t =$ Time [$Seconds$]

With this considerations is important to notice that, the energy consumed for a specific device could be calculated if a graph of current-vs-time is available, with a software like MATLAB ®, the AUC (area under curve) can be computed, and multiplying the result by the voltage (assumed constant) gives as a result, the total energy in joules.

2.5 Current measurement

This section explains the application that was necessary to implement for the acquisition and final tests of this project, it include the hardware that was used and the code developed for the logging and analysis of the data. Further information is available in the section 4.7.

2.5.1 Module NI-9227

This is a module with isolated analog inputs for current acquisition. It was designed to measure 5 A_{rms} nominal and up to 14 A_{PEAK} on each channel. This device was used for the current measuring, and important considerations should be taken.

This module could be connected ground-referenced or floating current sources, as is shown in the figures 2.5 and 2.6.

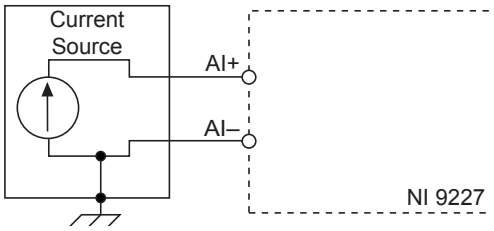


Figure 2.5: Connecting a Grounded Current Source to the NI 9227 [6]

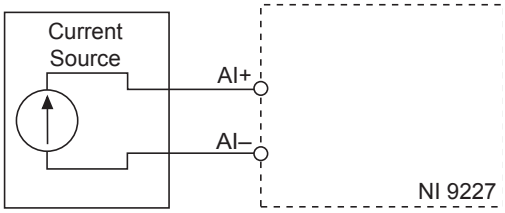


Figure 2.6: Connecting a Floating Current Source to the NI 9227 [6]

The frequency of a master timebase (f_M) controls the data rate (f_s) of the NI9227. This device includes an internal master timebase with a frequency of $12.8MHz$ but it also can accept an external master timebase or exports its own master timebase. The equation 2.5.1 provides the available data rates of this module:

$$f_s = \frac{f_M \div 256}{n} \tag{2.3}$$

Where n is any integer from 1 to 31.

Chapter 3

Methodology

This is a summary of the process that was followed in this project, this chapter contains a briefly but crucial information to understand how this project has been done and the steps followed for getting the final success.

1. Select a platform for the development of the project, taking into account the power modes of the microprocessor, the peripherals, and also thinking in a further application that can be tested for a final demonstration; the selection of this board depends of a list of constraints and/or conditions that were shown in the section 1.2.

2. Design and implement all the necessary software, (including: structures, functions, methods, headers, so on), to reduce the power consumption in the NuttX RTOS, no matter which platform is being used. This part of the project was designed as a library, which could be called for any platform that supports NuttX.

3. With the selected board, design all the drivers and libraries modifications, in order to reduce the power consumption that is related with platform aspects, such as: LCD, USB or LEDs. Some necessary drivers were not supported by NuttX et al, or were partially working, all those drivers were provided with fully support in this project.

4. A benchmark application was developed (section 4.6), in order to simulate the usage of the board for a possible user, running a portable application with NuttX, which use all the resources that has been added or modified for the power management.

5. Finally, all the work explained above, required to be demonstrated in a measurable way. That was the reason why a current acquire application was developed, using hardware from National Instruments including a special DAQ for current acquisition (see appendix D and E), and also an application in the software LabVIEW 2012 that logs the data into a file, so it could be analysed for the goals demonstration.

Chapter 4

Power Management Implementation

This chapter includes all the details about the implementation of this project, it contains aspects about the software modifications, new drivers that had been developed and everything regarding the benchmark application and how was it tested out.

4.1 General aspects

NuttX support a power management (PM) sub-system [5], which has been implemented in this project, and was released in the NuttX 6.23. This sub-system is able to:

- Monitors driver activity, and
- Provides hooks to place drivers (and the whole system) into reduce power modes of operation.

The figure 4.1 is the big picture of the solution approach, it show how the drivers needs to interact with the module, that means that, specific modifications were needed for each driver. The PM sub-system integrates the MCU idle loop with a collection of device drivers to support:

- Reports of relevant driver or other system activity.
- Registration and callback mechanism to interface with individual device drivers.
- IDLE time polling of overall driver activity.
- Coordinated, global, system-wide transitions to lower power usage states.
- It handle different energy modes, once it need to make a transition from one state to another, this could involve: lower display brightness, handle the frequency operation, gating the

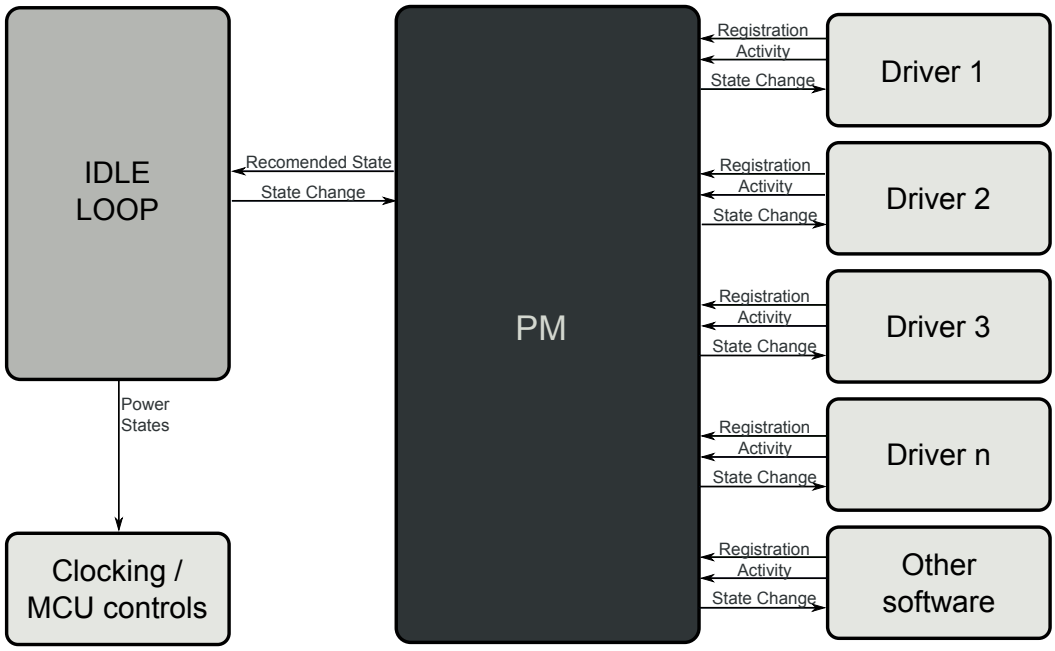


Figure 4.1: Power management sub-system.

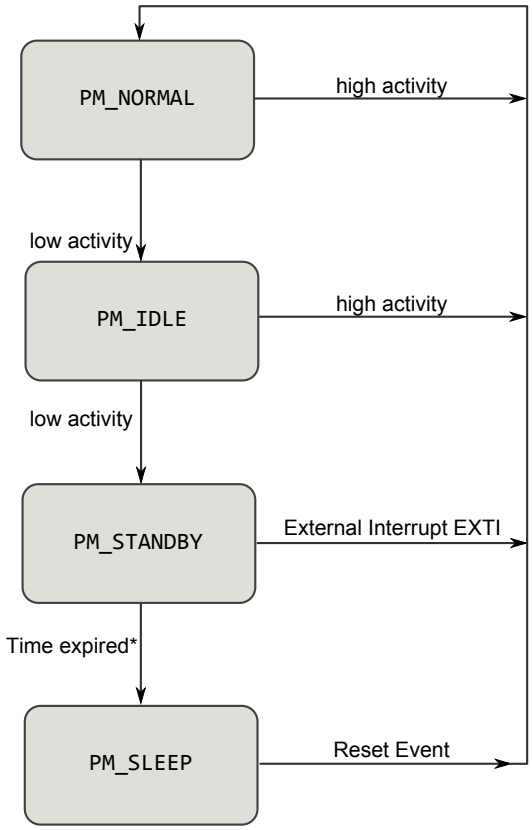
clocks to the APBx and AHBx peripherals when they are unused. Various "sleep" and low power consumption states have various names and are sometimes used in conflicting ways. In the NuttX PM logic, we will use the following terminology:

- `PM_NORMAL`: The normal, full power operating mode.
- `PM_IDLE`: This is still basically normal operational mode, the system is, however, `IDLE` and some steps to reduce power consumption are taken at this point, without affecting the normal operation. Simply dimming the a backlight might be an example that would be done when the system is idle.
- `PM_STANDBY`: Standby is a lower power consumption mode that may involve more extensive power management steps such has disabling clocking or setting the processor into reduced power consumption modes. In this state, the system should still be able to resume normal activity almost immediately.
- `PM_SLEEP`: The lowest power consumption mode. The most drastic power reduction measures possible should be taken in this state. It may require some time to get back to normal operation from SLEEP (some MCUs may even require going through reset).

4.2 Power management sequence

An important consideration is that all the processes that runs in NuttX, are executed in the state `PM_NORMAL`, the power management sub-system guarantees that it won't take any power reduction action until the processor is running in the idle loop, this idle loop is one of the tasks that

runs in NuttX and is also the only one that has a priority equals to zero, this ensures that the MCU is performing nothing relevant. This consideration is critical, because in this way, the processor always does the hard work with the highest performance.



* **Timeframe selected by the user**: is the time that PM submodule waits before to go to sleep.

Figure 4.2: Power management states sequence.

If the system is executing the code in the idle loop, that means that the processor doesn't have any other task being executed or waiting to be executed. The code in the idle loop is keeping track of the low activity time, after a specific period of time with low activity, the system switch to a lower power mode and continue keeping track of the amount of time with low activity, this times are configurable in the NuttX `.config` file, and those values will be depend of the application that the user wants to implement.

Since all the task with priorities greater than zero are executed in the `PM_NORMAL` mode, the system should switch from `PM_IDLE` or `PM_STANDBY` to the highest performance mode in a very fast step, for this project, the wake up times are really fast, it only needs to send a callback function to every registered driver or application in order to notify them the new state which is the `PM_NORMAL` in this case. In every driver those actions consists in a few lines of code, making the wake up timing less than one nanosecond for the STM32F4 processor.

The PM_SLEEP mode works slightly different, in this case the processor is within a very deep sleep mode, and the only way to wake it up is going through a reset event, either, dispatched for the user or triggered by an alarm. All these modes are also configurable, in the .config file, the user is able to select to run the PM system with only three states, so the NuttX never goes to the deepest sleep mode, and it still fulfils the objective of more than 50% of power reduction with this module enable. All the considerations that was explained above are illustrated in the figure 4.2.

4.2.1 PM_NORMAL

This is the higher energy consumption mode, but it is also the power state with the higher performance, since within this mode all the clocks are running at the highest rate. As long as the registered drivers are reporting activity via the callback functions, the PM will ensures that the RTOS will remain in this state. Some actions are taken after the systems returned from a lower power mode to the PM_NORMAL, such as: re-enable the clocks if these had been slowed down. Some of the actions taken at this points are listed as follows:

- Re-enable clocking if needed.
- Set a duty cycle of 100% in the PWM Color LCD backlight control.
- Re-synchronize clock with the RTC time.

4.2.2 PM_IDLE

All the actions taken for reducing the power consumption at this point, are non-processor related, that means that all the energy optimization was implemented in the drivers, some of these actions are the following:

- Gating the clocks to the APB and AHB peripherals if they are unused.
- Set a duty cycle of 60% in the PWM Color LCD backlight control.

4.2.3 PM_STANDBY

In this mode the PM module starts taking actions over the processor, at this point the system calls the STM32 stop mode that has been explained before in the section 2.1.1. Since this is a long period of time with no activity, the system assumes that the user is not needing the screen, like in a smart phone with a touchscreen display, and it proceeds turning the backlight off, but remaining all the display information in case that it needs to return to the normal state.

In addition, an important consideration is that, since the processor is within a sleep mode, it can't be awakened with standard activity report as in the `PM_IDLE`, it will be woken up only if an interruption occurs. Also, the system is not longer able to keep track of the amount of time with no activity, so, just before entering in this state, an EXTI alarm is set with a customizable time, which could be a few seconds or some years if the user wants to, when this time has been elapsed and no other interruption has occurred, the system goes to the next low power mode, otherwise it goes to the `PM_IDLE` and continue processing the tasks. Some of the most important actions taken in this state are listed below:

- Switch off the `HSI` and `HSE` oscillators.
- Get into the STM32 stop mode, explained in the section 2.1.1.
- Set a duty cycle of 0% in the PWM Color LCD backlight control, in order to turn it off.
- Set the alarm as an EXTI Line, for notifying the system when it should goes to next low power mode.
- Cancel the alarm if an interruption occurs.
- Set `SLEEPDEEP` bit of Cortex System Control Register

4.2.4 PM_SLEEP

At this point the processor is completely halted, this is an optional mode, is configurable from the NuttX `.config` file, and also, the fact of waking up from this state is also an option; for example, if the user wants to keep the system off until someone manually reinitialize it, like in a computer when automatically turn itself off because of a long period of no activity. Some other users might

want to perform some functions and calculations with a long time frame between every execution; for example, in a system that acquires temperature data, the user wants to take measures every 10 minutes, since the behaviour of the temperature changes pretty slow, so they can turn everything off, and programmatically wake up the system and performs the same calculations every 10 minutes. Some of the actions taken in this mode are the following:

- Set a standard alarm, to wake up the system through a reset event.

- Get into the STM32 standby mode, explainedin the section 2.1.1

4.3 Callback functions

Callback is a piece of executable code passed to functions. In the PM sub-sytem, consists in a structure which contains pointers callback functions in the driver. These callback functions can be used to provide power management information to any driver or application. It was implemented as a library and it doesn't depend of the processor, that means that it can be used in any platform supported by NuttX.

These functions are defined as a structure that supports a singly linked list, which means that the operating system is able to interface all the drivers with these functions. The structure declaration is shown in the appendix A.

4.3.1 Callback function `prepare`

Description:

Request the driver to prepare for a new power state. This is a warning that the system is about to enter into a new power state. The driver should begin whatever operations that may be required to enter power state. The driver may abort the state change mode by returning a non-zero value from the callback function.

Function Prototype

```
include/nuttx/power/pm.h
int (*prepare)(FAR struct pm_callback_s *cb, enum pm_state_e pmstate);
```

Input Parameters:

- `cb:` Returned to the driver. The driver version of the callback structure may include additional, driver-specific state data at the end of the structure.

- `pmstate:` Identifies the new PM state.

Returned Value:

Number 0 (`OK`) means the event was successfully processed and that the driver is prepared for the PM state change. Non-zero means that the driver is not prepared to perform the tasks needed achieve this power setting and will cause the state change to be aborted.

Notice that the prepare method will also be recalled when reverting from lower back to higher power consumption modes (say because another driver refused a lower power state change). Drivers are not permitted to return non-zero values when reverting back to higher power consumption modes.

4.3.2 Callback function `notify`

Description:

Notify the driver of new power state. This callback is called after all drivers have had the opportunity to prepare for the new power state.

Function Prototype

```
include/nuttx/power/pm.h
void (*notify)(FAR struct pm_callback_s *cb, enum pm_state_e pmstate);
```

Input Parameters:

- `cb:` Returned to the driver. The driver version of the callback structure may include additional, driver-specific state data at the end of the structure.
- `pmstate:` Identifies the new PM state.

Returned Value:

None. The driver already agreed to transition to the low power consumption state when when it returned OK to the `prepare()` call.

At that time it should have made all preparations necessary to enter the new state. Now the driver must make the state transition.

4.4 Power Management Interfaces

This section explains the interfaces that handles power management functions in the top level. To understand the role of these functions, the abstraction layers of this project is shown in the figure 4.3, which also illustrates what parts of the NuttX RTOS were actually developed from the beginning and which others were modified.

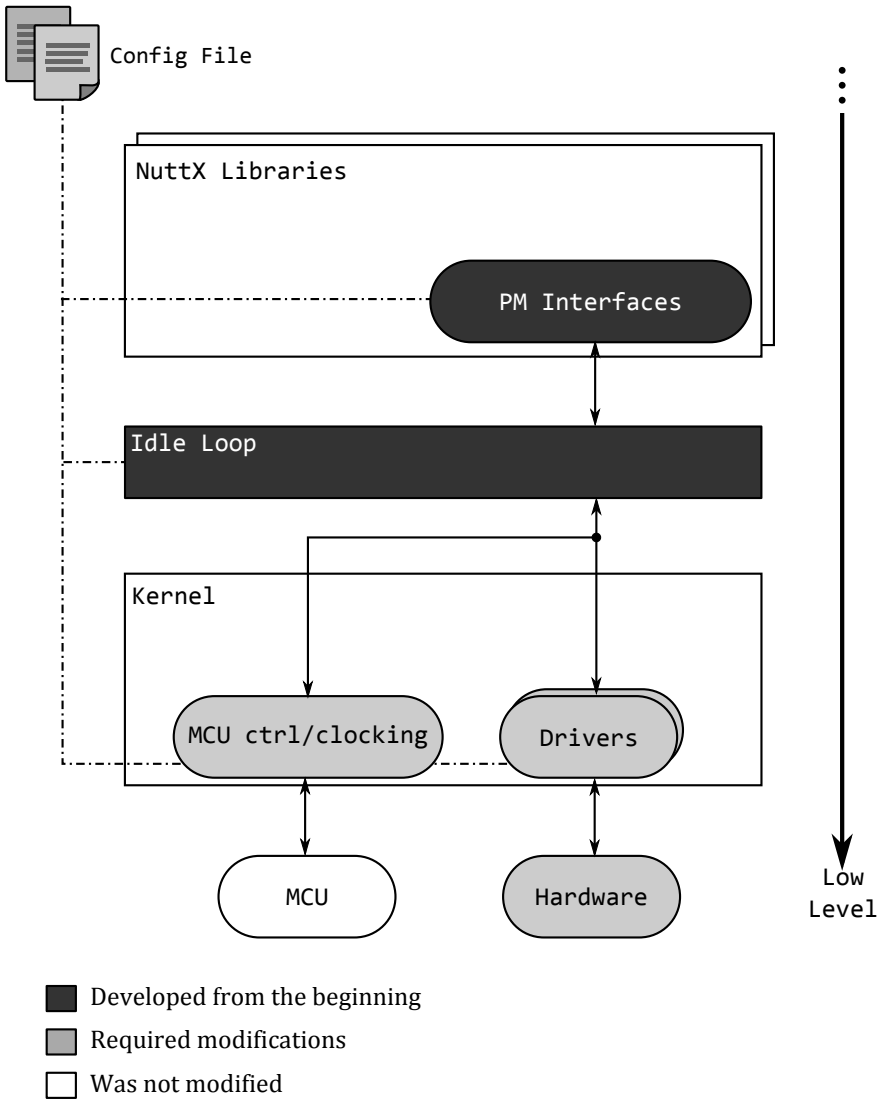


Figure 4.3: Abstraction layers of the PM project.

The PM interfaces is a event-based state machine, which means that it doesn't follow the same path trough the different states every time it runs, rather it goes from one state to another, depending of a set of conditions, such as: drivers activity, initialization of new drivers, and so on.

This functions are the responsible to estimate if a change in the power mode would be necessary,

one of the most important ones is the `pm_update()` which is performed in the worker thread. In this module, the PM keeps all the information in one structure called `pm_global_s`.

Public Types

```
#include "pm_internal.h"
struct pm_global_s
  uint8_t state        - The current state (as determined by an explicit
                         call to pm_changestate()
  uint8_t recommended  - The recommended state based on the PM algorithm
                         in function pm_update().
  uint8_t mndx         - The index to the next slot in the memory[] array
                         to use.
  uint8_t mcnt         - A tiny counter used only at start up.  The actual
                         algorithm cannot be applied until CONFIG_PM_MEMORY
                         samples have been collected.
  int16_t accum        - The accumulated counts in this time interval
  uint16_t thrcnt      - The number of below threshold counts seen.

  int16_t memory[CONFIG_PM_MEMORY-1]; - This is the averaging "memory."

  uint32_t stime      - The time (in ticks) at the start of the current
                         time slice.
  sem_t regsem        - This semaphore manages mutually exclusive access to
                         the power management registry.
  struct work_s work  - For work that has been deferred to the worker thread

  sq_queue_t registry - Registry is a singly-linked list of registered power
                         management callback structures
```

This module computes the activity every time slice. The power management module collects activity counts in time slices. At the end of the time slice, the count accumulated during that interval is applied to an averaging algorithm to determine the activity level.

The value `CONFIG_PM_SLICEMS` could be added to the configuration file, it provides the duration of the time slice. The default value is 100 milliseconds, as long as it's undefined in the `.config` file.

4.4.1 Interface: `pm_update()`

Function Prototype

```
drivers/power/pm_internal.h
EXTERN void pm_update(int16_t accum);
```

Input Parameters: `accum:` The value of the activity accumulator at the end of the time slice.

Returned Value: None.

This internal function is called at the end of a time slice in order to update driver activity metrics and recommended states. At the end of every time slice, the count accumulated during that interval is applied to an averaging algorithm (see equation 4.1) to determine the activity level.

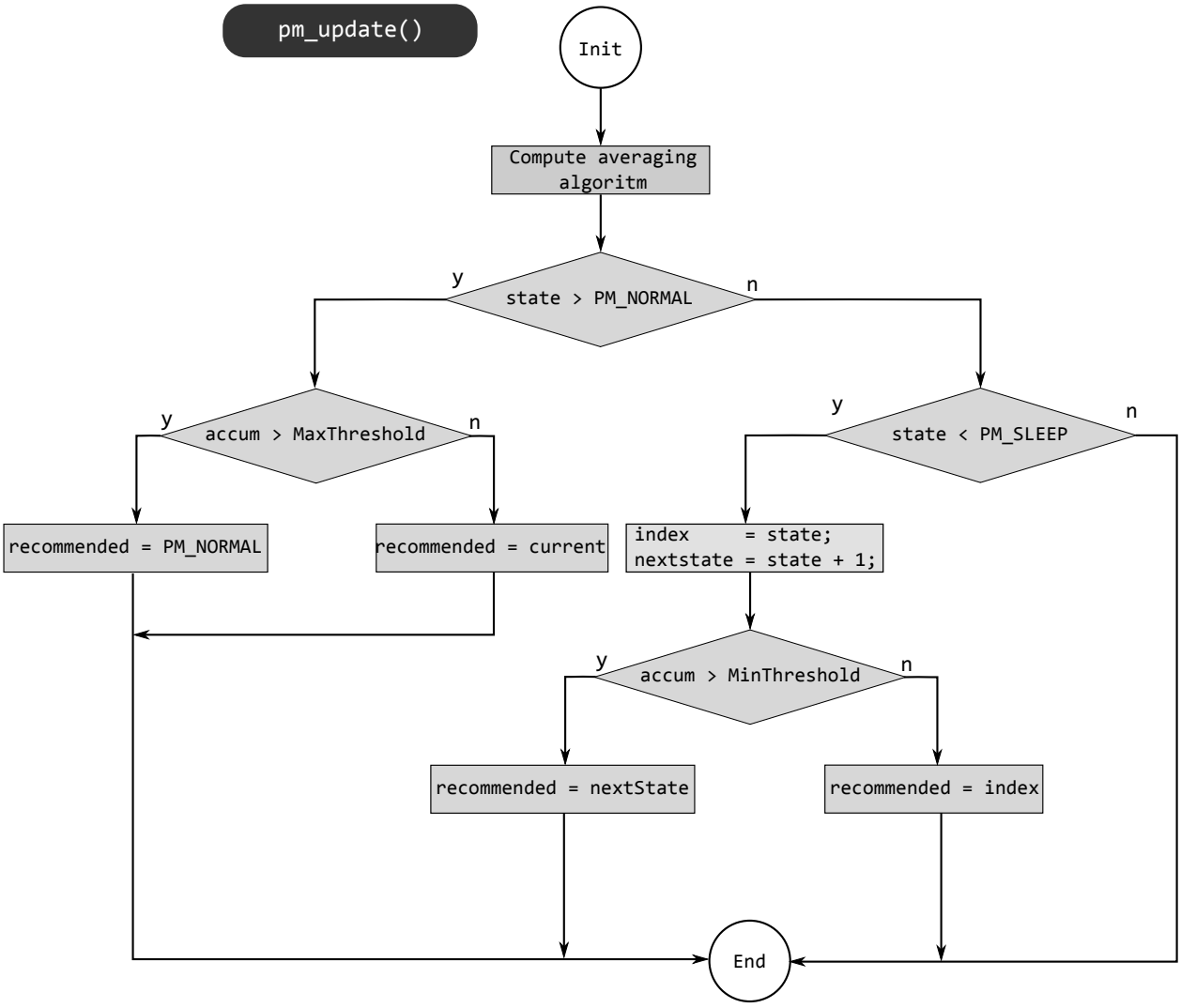


Figure 4.4: Data flow diagram of the function `pm_update()`

$$Y = \frac{A_n \cdot X + \sum_{i=1}^{n-1} A_i Y_i}{\sum_{j=1}^{n} A_j} \tag{4.1}$$

Where:

Y: Represents the activity value.

n: Is the length of the "memory" stored in `CONFIG_PM_MEMORY`.

X: Is the current activity.

The only input parameter in this function is the `accum` variable of the structure `pm_global_s`. This worker function is queue at the end of a time slice in order to update driver activity metrics and recommended states.

The figure 4.4 shows the data flow diagram for this function, when this functions is called, it computes the algorithm shown in the equation 4.1, with the value of "`accum`" that has been passed as an argument of this function, and with an amount of iterations equal to the value of `CONFIG_PM_MEMORY` in the `.config` file.

In summary, all the important definitions are the following:

- `CONFIG_PM_MEMORY` is the total number of time slices (including the current time slice). The history or previous values is then `CONFIG_PM_MEMORY-1`. (Default value = 3)
- `CONFIG_PM_SLICEMS` provides the duration of one time slice in milliseconds. (Default value = 100ms)
- `CLOCKS_PER_SEC` provides the number of timer ticks in one second. (Default value = 100)

With the above information, the time slice interval is converted into system clock with the equation 4.2.

$$TIME\_SLICE\_TICKS = \frac{CONFIG\_PM\_SLICEMS \cdot CLOCKS\_PER\_SEC}{1000} \tag{4.2}$$

The figure 4.5 illustrates the averaging event trough the time, assuming all the default values that are used in this project. Every time slice the activity accumulator is stored in memory, then when the amount of samples have been taken, the average activity is computed. With that value, the algorithm computes the recommended state.

In order to estimate the new recommended state, the function `pm_update()`, not only computes the activity, but also follows necessary logic for the estimating of this new state (see figure 4.4). After the activity was computed, this function verify if the system needs to go to the `PM_NORMAL` state, this happens if it detects an activity value grater than 10, or, if he needs to reduce the power

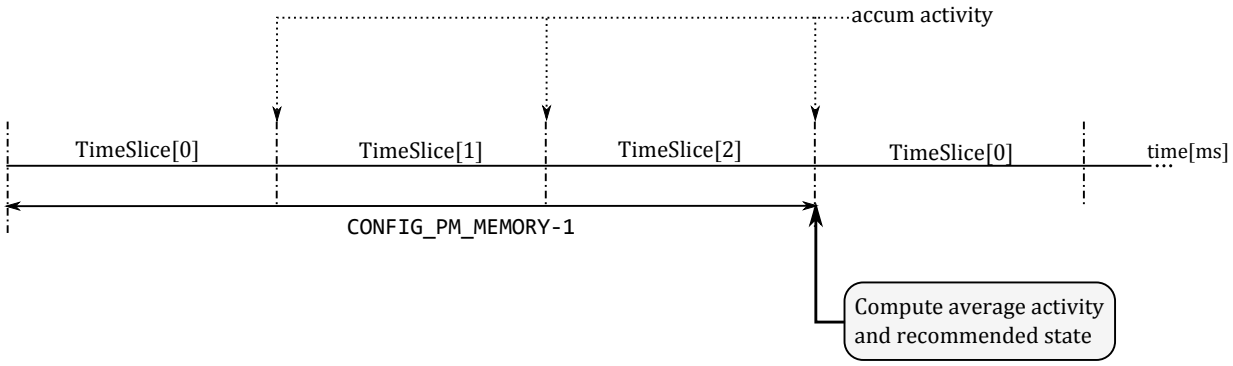


Figure 4.5: Time representation of the average activity and the new recommended state.

consumption mode, which happens when the activity value is zero, that means that the system was idle for all the last time slices.

Some other considerations was needed for this function, like if the system exceeds the a maximum threshold activity, in this case the system just recommend the `PM_NORMAL` state; otherwise, the system ask if the threshold to enter the next lower power consumption state has been exceeded, this allows the system to decide if it needs to recommend the same state than the last time, or if it needs to enter in the next low power mode.

4.4.2 Interface: `pm_checkstate()`

Function Prototype

```
include/nuttx/power/pm.h
EXTERN enum pm_state_e pm_checkstate(void);
```

Input Parameters: None.

Returned Value: The recommended power management state.

This function is called from the MCU-specific IDLE loop to monitor the power management conditions. This function returns the "recommended" power management state based on the PM configuration and activity reported in the last sampling periods. The power management state is not automatically changed, however. The IDLE loop must call `pm_changestate()` in order to make the state change.

These two steps are separated because the plaform-specific IDLE loop have additional situational information that is not available to the the PM sub-system. For example, in a future improvement of this project, the IDLE loop may know that the battery charge level is very low and may force lower power states even if there is activity.

It's important to notice that these two steps are separated in time and, hence, the IDLE loop could be suspended for a long period of time between calling `pm_checkstate()` and `pm_changestate()`. The IDLE loop need to make these calls atomic[1], by either disabling interrupts until the state change is completed.

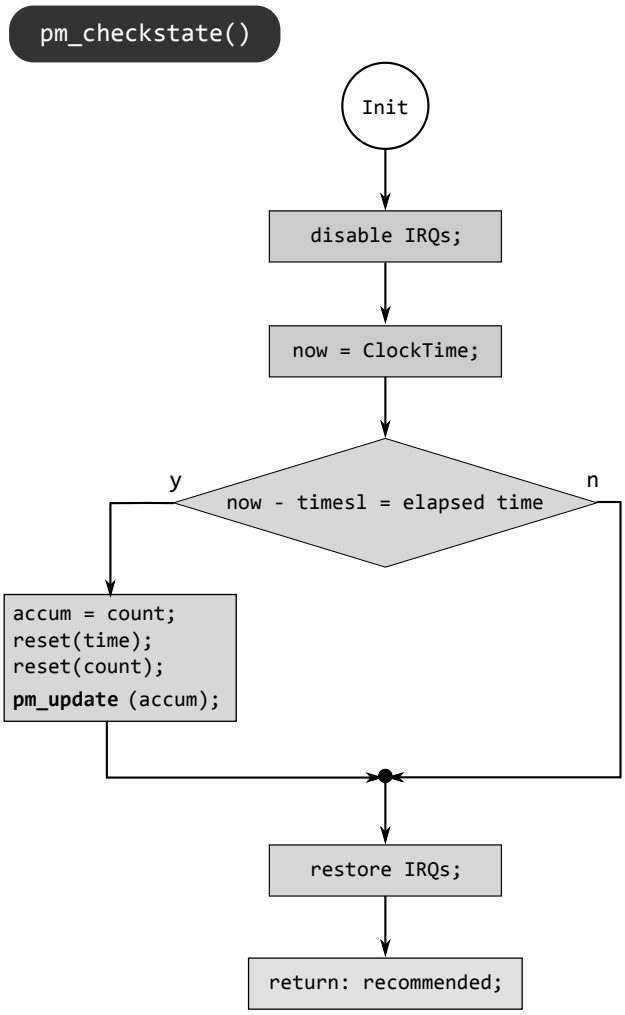


Figure 4.6: Data flow diagram of the function `pm_checkstate()`

The data flow diagram of this function is shown in the figure 4.6, notice how the interrupts must be disable during the execution of `pm_checkstate()`, after disabling the interrupts it check the elapsed time. In periods of low activity, time slicing is controlled by IDLE loop polling; in periods of higher activity, time slicing is controlled by driver activity. In either case, the duration of the time slice is only approximate; during times of heavy activity, time slices may be become longer and the activity level may be over-estimated.

[1]Atomic: Execute code with interrupts disable.

Finally, it return the recommended state, which was calculated in the `pm_update()` function and is stored in the global structure `pm_global_s.recommended`.

4.4.3 Interface: `pm_changestate()`

Function Prototype

```
include/nuttx/power/pm.h
EXTERN int pm_changestate(enum pm_state_e newstate);
```

Input Parameters: `newstate:` Identifies the new PM state.

Returned Value: 0 (OK) means that the callback function for all registered drivers returned OK (meaning that they accept the state change). Non-zero means that one of the drivers refused the state change. In this case, the system will revert to the preceding state.

This function is used to platform-specific power management logic. It will announce the power management state change to all drivers that have registered for the power management event callbacks.

As is shown in the data flow diagram of the figure 4.7, after disabling the interrupts, it prepares the drivers for the state change. In this phase, drivers may refuse the state state change, if that is the case, the system returns to its preceding power mode, otherwise, this function returns the value zero (`OK`).

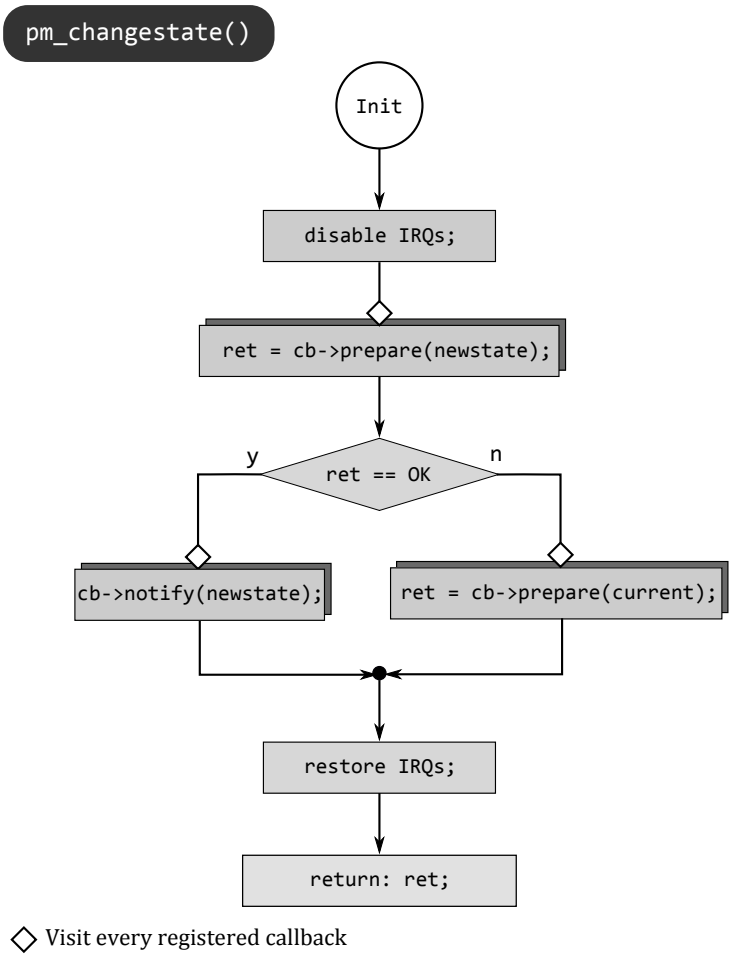


Figure 4.7: Data flow diagram of the function `pm_changestate()`

4.4.4 Interface: `pm_activity()`

Function Prototype

```
include/nuttx/power/pm.h
EXTERN void pm_activity(int priority);
```

Input Parameters: `priority:` Activity priority, range 0-9. Larger values correspond to higher priorities. Higher priority activity prevent the system from entering reduced power states for a longer period of time.

Returned Value: None.

This function is called by a device driver to indicate that it is performing meaningful activities (non-idle). This increment an activity count and will restart a idle timer and prevent entering reduced power states.

The `pm_activity()` may be called from an interrupt handler (this is the only PM function that may be called from an interrupt handler). As an example, a button press will be the higher priority activity because it means that the user is actively interacting with the device.

This function simply add the priority to the accumulated counts `accum`, and update it to the global PM structure, after that, if the time slice has been elapsed, it calls the function `pm_update(accum)`, with the new accumulated count value.

4.4.5 Interface: `pm_register()`

Function Prototype

```
include/nuttx/power/pm.h
EXTERN int pm_register(FAR struct pm_callback_s *callbacks);
```

Input Parameters: `callbacks:` An instance of `struct pm_callback_s` providing the driver callback functions

Returned Value: Zero (OK) on success; otherwise a negater errno value is returned.

This function is called by a device driver in order to register to receive power management event callbacks. This function simply add the new entry to the end of the list of registered callbacks.

4.4.6 Interface: `pm_initialize()`

Function Prototype

```
include/nuttx/power/pm.h
EXTERN void pm_initialize(void);
```

Input Parameters: None.

Returned Value: None.

This function is called by MCU-specific logic at power-on reset in order to provide one-time initialization the power management subsystem. This function is called very early in the initialization sequence, before any other device drivers are initialized (since they may attempt to register with the power management subsystem).

4.4.7 Interfaces and Callbacks Sequence

The figure 4.8 shows the sequence logic of the NuttX power management interfaces and callbacks, the figure is an example of a possible application, however the initialization that is shown in the step 1 is always the same, and happens only once, in the initialization of the NuttX, the rest of the interactions could be different depending on the application that is being executed.

If a certain application is running, say a few hours ago, and the system haven't initialized the display so far, when the display is being used the first time, it will be initialized, and in the initialization sequence of every driver occurs the registration of the callbacks for that specific driver, that means that the `pm_register(callbacks)` function could be called at any time that the application need it. This is actually a important feature of these interfaces, because, since the LEDs are used for debugging purposes in NuttX, the initialization of these LEDs happens before that the power management interfaces have been initialized, then, the callbacks for the LEDs controller is scheduled some time after the driver was already initialized.

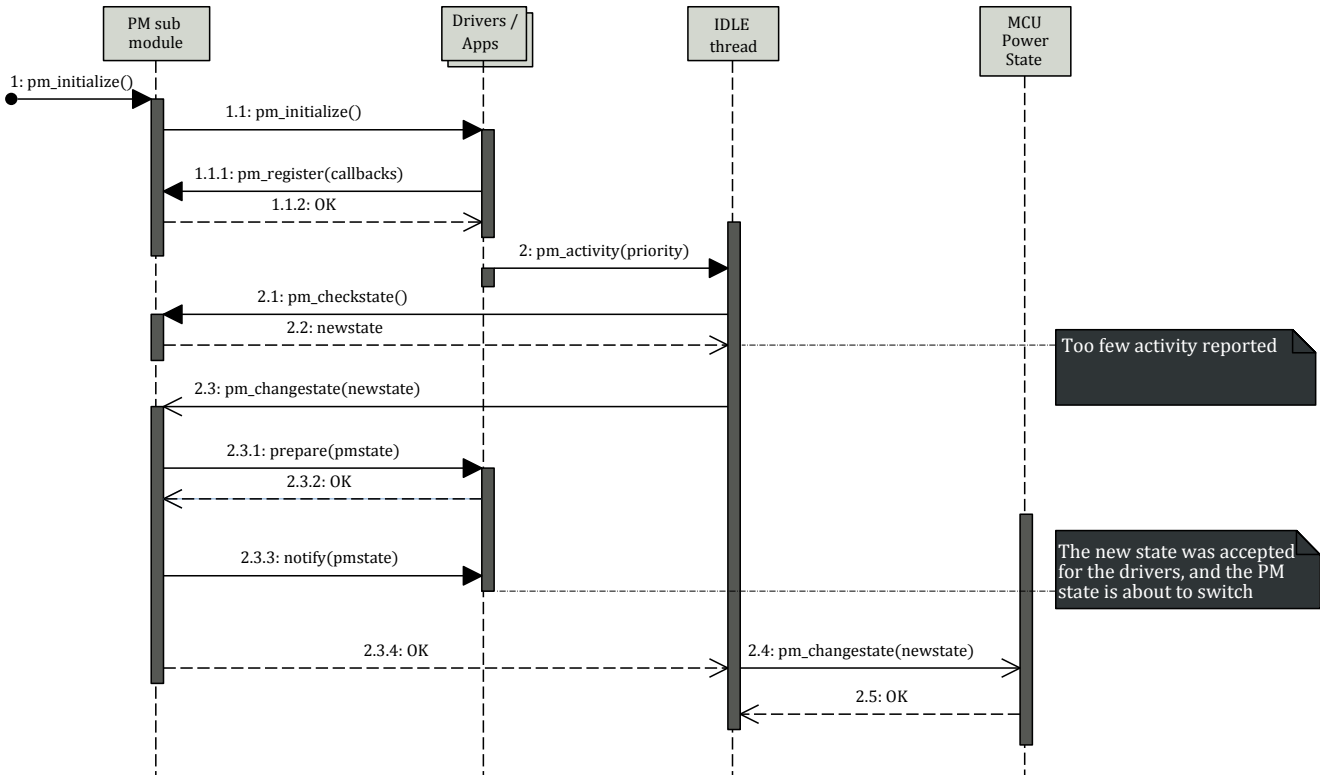


Figure 4.8: UML sequence diagram of the interfaces and callbacks interaction.

In the application that is being simulated in the figure 4.8, after the initialization of the power management interfaces, the drivers are registering only one driver with the callbacks, and it's returning a value `OK`, this driver at the point 2 is reporting some activity, and a time after the driver is not needed anymore, then when the system calls the `pm_checkstate` function it will return a

new lower state, based in the time that the system was idle. In the step 2.3 of the UML sequence diagram, the IDLE thread will recommend the new power mode, and here is when the callbacks are needed to notify every registered driver and application that a power state change is being attempted, and in this case all the drivers have returned `OK` in the step 2.3.2, that means that the system is able to continue with the process and the callback `notify` is sent to all the drivers, at this point every driver perform the necessary logic for the specific power state. When all the drivers and applications are done with the state change, the function `pm_changestate()` returns `OK` to the IDLE thread and the system continues with the changes in the MCU registers if required.

4.5 Power management for STM32 drivers and processor

These drivers had fully or partially support by NuttX, but all of them required power management enhancements. Those modifications usually consists of adding the logic for registering the callbacks, and choose the proper action when a power state change, and also for reporting the activity.

4.5.1 Color LCD

This driver required some extra work, at first because it required hardware modifications, as it was explained in the section 2.3.2, and also because, it was necessary to write a driver for the NuttX PWM, in order to handle the backlight of the screen. The modifications are listed as follows:

Callback prepare:

No preparation to change power modes is required by the LCD driver. It always accept the state change by returning OK.

Callback notify:

It depends of the recommended state, the main actions are listed below:

- `PM_NORMAL:` It restore the LCD to the normal operation, setting the duty cycle of the PWM backlight controller to 100%.
- `PM_IDLE:` Reduce LCD light, the information is still readable for the user, but the duty cycle is being set at 600% in this point.
- `PM_STANDBY:` Turn display backlight off, setting a duty cycle of 0, in the backlight controller.

- `PM_SLEEP:` It needs to go trough a display off sequence, because the information won't be needed, and because, when the MCU goes to sleep, the pin that controls the PWM backlight will be set in Hi-Z, the power off sequence avoids to display information in the LCD, even though the backlight pin has voltage or not.

Activity:

This driver doesn't report activity, since is used strictly for displaying information to the user, and not for receive information or feedback from the user. In other platforms, such as the STM3240g-eval, this feature was improved because the board has a touchscreen via SPI protocol, so, the system reports activity every time the user touch the screen and a interruption occurs in the SPI driver.

4.5.2 Serial

The modifications for the serial driver are listed as follows:

Callback prepare:

No preparation to change power modes is required by the serial driver. It always accept the state change by returning OK, however, it's recommended to add the logic to prepare this driver for a reduced power state, for example, should be a good improvement to reject the power change if this driver can't close the serial communication for a certain reason.

Callback notify:

It depends of the recommended state, the main actions are listed below:

- `PM_NORMAL:` Reconnect the serial communication if the last state was `PM_STANDBY`
- `PM_IDLE:` None.
- `PM_STANDBY:` Close the USART communication.
- `PM_SLEEP:` None.

Activity:

This driver reports activity in the USART interrupt handler, this is the logic that handles when the user is typing in the keyboard. It reports the highest activity in a driver with a value of 10.

4.5.3 LEDs application

As mentioned in the section 2.3.4, the LEDs are used in NuttX for debugging purposes, and because of that, the LEDs will remain working until the system is within the lowest power mode `PM_SLEEP`. Another important consideration is that, since the LEDs are initialized very early, in the initialization sequence of NuttX, the power management interfaces aren't initialized at this point, so a new function was added to this application in order to register it to receive PM callbacks, and this function is called later in the board specific power management initialization.

Callback prepare:

No preparation to change power modes is required by the LEDs driver.It always accept the state change by returning the value `OK`.

Callback notify:

It depends of the recommended state, the main actions are listed below:

- `PM_NORMAL:` No actions required.
- `PM_IDLE:` No actions required.
- `PM_STANDBY:` No actions required.
- `PM_SLEEP:` Turn off the LEDs via GPIO registers, as a preventing action in order to keep all LEDs off at this point.

Activity:

This driver doesn't report activity, since is used strictly to display debugging information to the user.

4.5.4 Other drivers

As it was explained in the section 4.3, the PM system is able to interface all the drivers with the callback functions, in the sections above, the most important drivers that required changes were explained, some other drivers are just reporting activity, such as the I2C, SPI and the CAN, and some others required minor changes, like the network or USB that simply close the communication

when the system is in `PM_STANDBY` mode.

In the case of the USB and the ethernet drivers, an important recommendation for this project is to add the logic to report activity in the critical sections of these drivers, when a high traffic of information is being transferred and the system should remain in the `PM_NORMAL` state.

4.5.5 Board-specific Actions

All the board-specific actions to reduce the power consumption are taken in the IDLE loop, since this is not a driver it doesn't require to register callbacks, it handles when the state change in all the drivers calling the function `pm_changestate`, and after that, this thread handles all the board-specific logic for every state as follows:

- `PM_NORMAL:` If it just awakened from `PM_STANDBY` mode, it reconfigure clocking.

- `PM_IDLE:` No actions required.

- `PM_STANDBY:` Set the alarm as an EXTI line, because only EXTI interruptions are able to wake it up from this mode, if the alarm expires with no other interruptions, means that the system should go to the `PM_SLEEP` mode.

  Enter in the STM32 stop mode explained in the section 2.1.1 of power management sequence, by calling the function `stm32_pmstop()`.

- `PM_SLEEP:` Configure the RTC alarm to auto reset the system (if it's set in the `.config` file).

  Enter in the STM32 standby mode explained in the section 2.1.1, by calling the function `stm32_pmstandby()`.

It's important to mention that all the routines used in the RTC controller wasn't supported for NuttX before to start this project, it had basic support for timing, so all the functions for setting the alarms were implemented as part of this project.

4.6 Benchmark Application

In order to verify the functionality of the power management project, a benchmark application has been developed. This is a configuration that is used to test STM32 power management, to test that the board can go into lower and lower states of power usage as a result of inactivity. This configuration is based on the `nsh2` configuration with modifications for testing power management. This configuration should provide some guideline for power management in the STM32 application of a NuttX user, and is available in the path `configs/stm3210e-eval/pm/`, see the README file `configs/stm3210e-eval/README` for further information about how to configure NuttX with this application.

4.6.1 Buttons (GPIO)

The benchmark application make use of all the buttons in the board as external interruptions, and as a way to either, keep the system in the `PM_NORMAL` state, or to wake it up from a low power mode. To achieve this, a new application was written in the file `up_pmbuttons.c`.

The main function in the `up_pmbuttons.c` file is `up_pmbuttons(void)`, and it configures all the buttons of the STM3210e-eval board as EXTI, so any button is able to wakeup the MCU from the `PM_STANDBY` mode. When an interruption occurs the MCU should have already awakened. The state change is handled in the IDLE loop when the system is re-awakened, the button interrupt handler is totally ignorant of the PM activities and reports button activity as if nothing special happened.

4.6.2 Color LCD

The color LCD is running an application that is part of the NX graphics subsystem examples, it was used to simulate information being displayed to the user, the figure 4.9 illustrates a screenshot of this example running on the simulated Linux X86 with X window, the image was taken from [10], in the NuttX site.

4.7 Current Measurement

This section explains all the technical details, that were necessary in order to demonstrate the functionality of the power management sub-system. The STM3210e-eval board uses a power supply of 5 volts, that means that in order to measure the power consumption, the more suitable variable to acquire will be the current. To achieve the current measurement in the hardware, an applications

was implemented using data acquisition hardware and the software LabVIEW from National Instruments.

4.7.1 Connection Diagram

The module NI-9227 is a cDAQ for current input, as explained in the section 2.5 using the grounded current source configuration (see figure 2.5) in order to keep the noise isolated.

The NI-9227 was designed for measuring power and energy consumption for applications such as appliance and electronic device test [6]. One of its features is that it allows up to $50KS/s$[2], it overcomes the necessary sample rate, because the objective is to analyse the energy usage trough long timeframes (tens of seconds).

The figure 4.10 is a simplified picture about the connection scheme, the NI-9227 module requires a chassis (cDAQ 9178), and also requires to implement an application that logs the data that has been acquired. A picture of the module with his chassis is shown in the figure E.1.

4.7.2 LabVIEW Application

The figure 4.11 illustrates the GUI of the program developed in the software LabVIEW for the acquisition of the current, this program that runs in windows allows the user to select: the input channel, the current range, the sample rate; it also shows the value of the current in real time and plots in a chart the current amplitude vs time. The code of this application is provided in the appendix D.

Although it plots the current value in real time, it also needs to log all the data in a file for further analysis, in this case the application stores all the data in a TDMS file, and this file is passed to the software Matlab in order to compute the energy and for plotting the data, so it can shows the current behaviour during the whole experiment.

[2]Thousands of samples per second

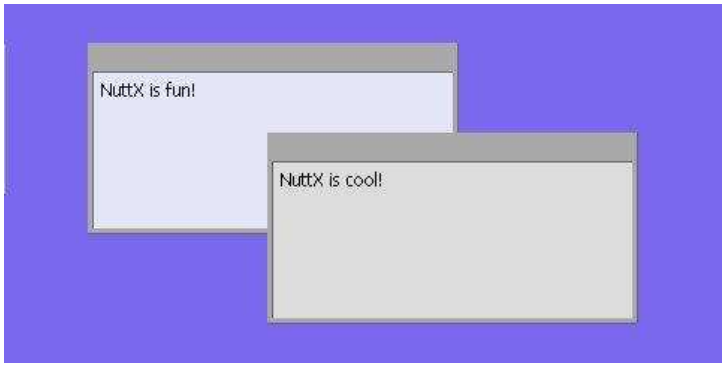


Figure 4.9: NuttX Screenshot example running on the simulated X window.

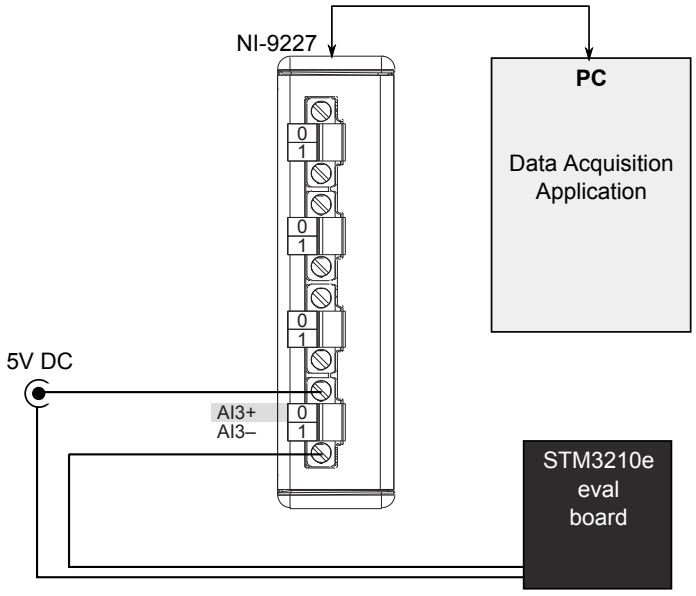


Figure 4.10: Schematic of the STM3210e-eval board and the acquisition system.

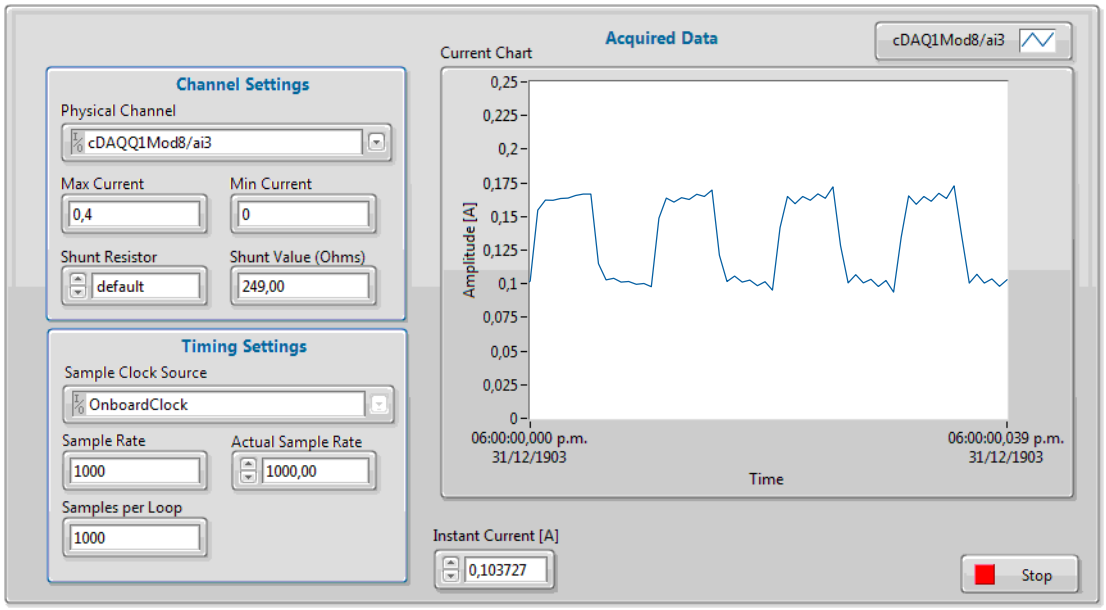


Figure 4.11: User interface for the data acquisition.

Chapter 5

Results and Analysis

This chapter includes all the findings obtained in the different experiments, that have been done for verifying all the objectives, a very detail explanation is provided from the results.

Is important to note that an acquisition hardware was necessary for the power consumption calculation, one of the best ways for measuring power consumption is the energy in joules that a system requires trough a specific timeframe, in this project the energy was determined using both equations 2.1 and 2.2.

5.1 Results analysis

The first part of this project involved the study of a proper platform for the power management development, this platforms required to satisfy with a list of features listed in the section 1.1.1, the first platform analysed was the multimedia for PIC32MX7 [4], and it was quickly discarded because this board had to few support by NuttX. Another platform studied was the SAM3U-EK [2], but its processor doesn't have enough low power modes, so it was discarded as well. The rest of available boards was the family of the STM32 processors, either boards, STM3220e [14] and STM3240g [15], doesn't have a backlight control pin for the Color LCD display, so both was also put away, and the last one STM3210e-eval accomplish with all the requirements [13], but in order to use the backlight control pin for the LCD, was necessary to remove a resistor and placed in a different place of the board, as it was explained in the section 2.3.2. Due all the above considerations, the STM3210e-eval board was the selected platform for the project implementation.

A total of 30 experiments were selected for this analysis, these experiments are tabulated in the table 5.2, the experiments consist of comparing two different configurations in similar conditions, specifically with the PM enable and with the PM disable. The experiment wasn't completely accurate, due only one board is available and its usage depends on the human factor when the replication of the experiment is attempted with a different configuration, with that constraint, is

not possible to run both configurations at the same time. However, it gives the idea of how the behaviour of the system is, working in both configurations.

The figure 5.1 illustrates one of the experiments, once it has been plotted using the software Matlab, it was selected because it clearly shows the different states occurring in a frame of around 20 seconds, all the experiments were taken using a sample rate of 1000 per second, and with 1000 samples per loop. In the figure is easy to see that it goes from `PM_NORMAL` to `PM_IDLE`, then the system detects activity, so it goes back to the higher performance mode, as is shown in the figure 4.2, and then it periodically reduce its power modes, until it gets in the very last low power mode `PM_SLEEP`. All of this state changes accomplish with the information given in the section 4.2.

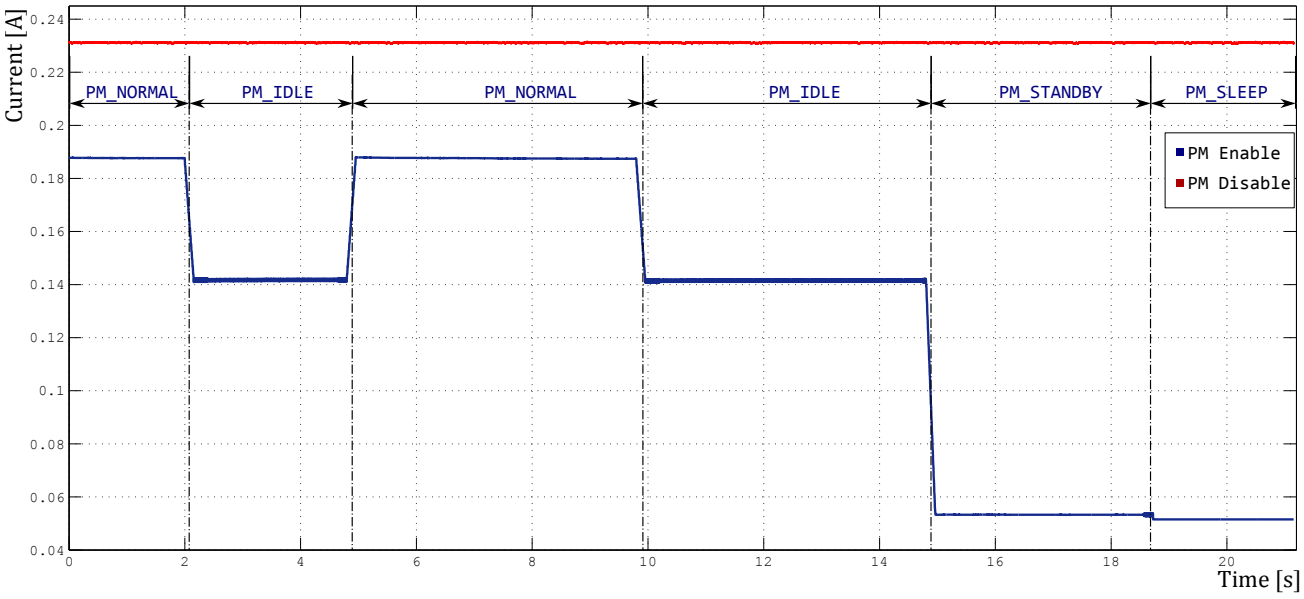


Figure 5.1: Graph of current vs time.

It is important to notice that in the `PM_NORMAL` state with the `PM` enable, the system is also reducing the power consumption, this is due mostly to the PWM that handles the Color LCD, in the benchmark application that was explained in the section 4.6, the display is working with an 80% of the backlight within `PM_NORMAL` mode, which is almost imperceptible for an user. In addition, all the drivers that aren't used at this point, remains uninitialized until the system need them.

From the figure 5.1, the energy for every individual state could be estimated, assuming that the conditions of the experiment of the figure 5.1 remain during one hour, the energy consumed in every state is shown in the table 5.1.

In the state `PM_IDLE` all the clocking and performance of the system is the same than in the `PM_NORMAL` mode, however, there are actions taken over the drivers, specially in the screen, the

Table 5.1: Energy consumption of the STM3210e-eval board during 1 hour

	Power $[mW]$		Energy$[J]$		Reduction
State	`PM` Enable	`PM` Disable	`PM` Enable	`PM` Disable	
`PM_NORMAL`	925	1130	55,50	67,80	18,14%
`PM_IDLE`	700	1130	42,00	67,80	38,05%
`PM_STANDBY`	275	1130	16,50	67,80	75,66%
`PM_SLEEP`	250	1130	15,00	67,80	77,88%

energy is being reduce in a 38%, the actions taken in this mode was explained in the section 4.5 and in the section 4.2.2.

The power reduction between the `PM_STANDBY` and `PM_SLEEP` is minimal if its compared between the other states, but the biggest difference happens in the processor. In the STM3210e-eval board, there are some parts of the hardware that are inaccessible, like the power LED, or the temperature sensor which is always powered. But in an application that runs with the minimal peripherals this configuration makes a big difference, as it was explained in the section 4.2, the STM32 MCU consumes around 24μA, in STM32 STOP mode, while the processor consumes 2μA in the STM32 STANDBY mode, this makes a big difference in an applications that needs to run standalone for long periods of time.

With the information of the table 5.2, the bar graph of the figure 5.2 was built in order to compare the results in a graphic way. This information illustrates that the power consumption was reduced in a given application more than 50% in 29 of 30 experiments performed, this means that the objective of double the life of a possible battery was fulfilled in a 96.67% of the samples. In fact, the power reduction was equal or greater than 65% in a 93.33% of the samples.

The benchmark application that was implemented for this project (see section 4.6), is located in the path `configs/stm3210/pm` and it was created in order to take advantage of all the PM interfaces explained in the section 4.4, the main drivers that it uses are the LCD, the buttons, the serial driver, and the LEDs, all of these drivers are registered with the callback functions (see section 4.3) and its functionality was tested in all the 30 samples with success in all of them, this was an important step in order to accomplish the objectives. This is because every driver is able to handle its own actions for every power mode as explained in the section 4.5, while the IDLE thread controls all the board-specific actions, how was illustrated before in the section 4.5.5.

A new power mode should be consulted, when the system remains idle for a long time, this time is configurable, one of the most important PM interfaces is the `pm_update()` that has been explained in the section 4.4.1, this is called every time slice in order to recommend a new power state, taking into account the averaging activity using the equation 4.1 and also depends of the current state. The PM sub-system ensures that the time slice has been elapsed by calling the `pm_update()`

Table 5.2: Energy consumption of the STM3210e-eval in 30 different experiments

Experiment	**Energy PM enable** $[J]$	**Energy PM disable** $[J]$	**Reduction**
1	152.54	242	36.97
2	40.31	121	66.68
3	54.43	181.50	70.01
4	78.99	435.60	81.87
5	64.69	266.20	75.70
6	93.12	314.60	70.37
7	84.91	411.40	79.35
8	77.15	326.71	76.38
9	100.51	350.90	71.36
10	38.35	205.12	81.36
11	39.12	205.00	80.98
12	22.18	254.12	91.26
13	114.83	363.02	68.37
14	38.40	217.7	82.37
15	56.82	302.21	81.21
16	45.19	157.30	71.26
17	43.42	145.17	70.09
18	51.39	181.50	71.68
19	99.91	375.09	73.37
20	41.37	169.43	75.57
21	46.88	169.38	72.32
22	79.32	399.30	80.13
23	65.36	217.79	69.98
24	45.15	193.60	76.68
25	71.50	254.10	71.86
26	161.85	411.41	60.65
27	117.85	411.37	71.36
28	41.16	121.00	65.98
29	45.41	459.80	90.12
30	50.13	181.50	72.38

from the `pm_checkstate()` function, which makes that call only if the time slice has exipred. This was deeply analysed in the section 4.4.

All the data of the table 5.2, was calculated using the equations 2.2 and 2.1, the board was powered in all the experiments with a power source of 5 volts, the current and the time was acquired using the acquisition application with the hardware of National Instruments, as it was explained in the section 4.7, this application was a very important step, since this is the tool that allows a measurable demonstration of this project.

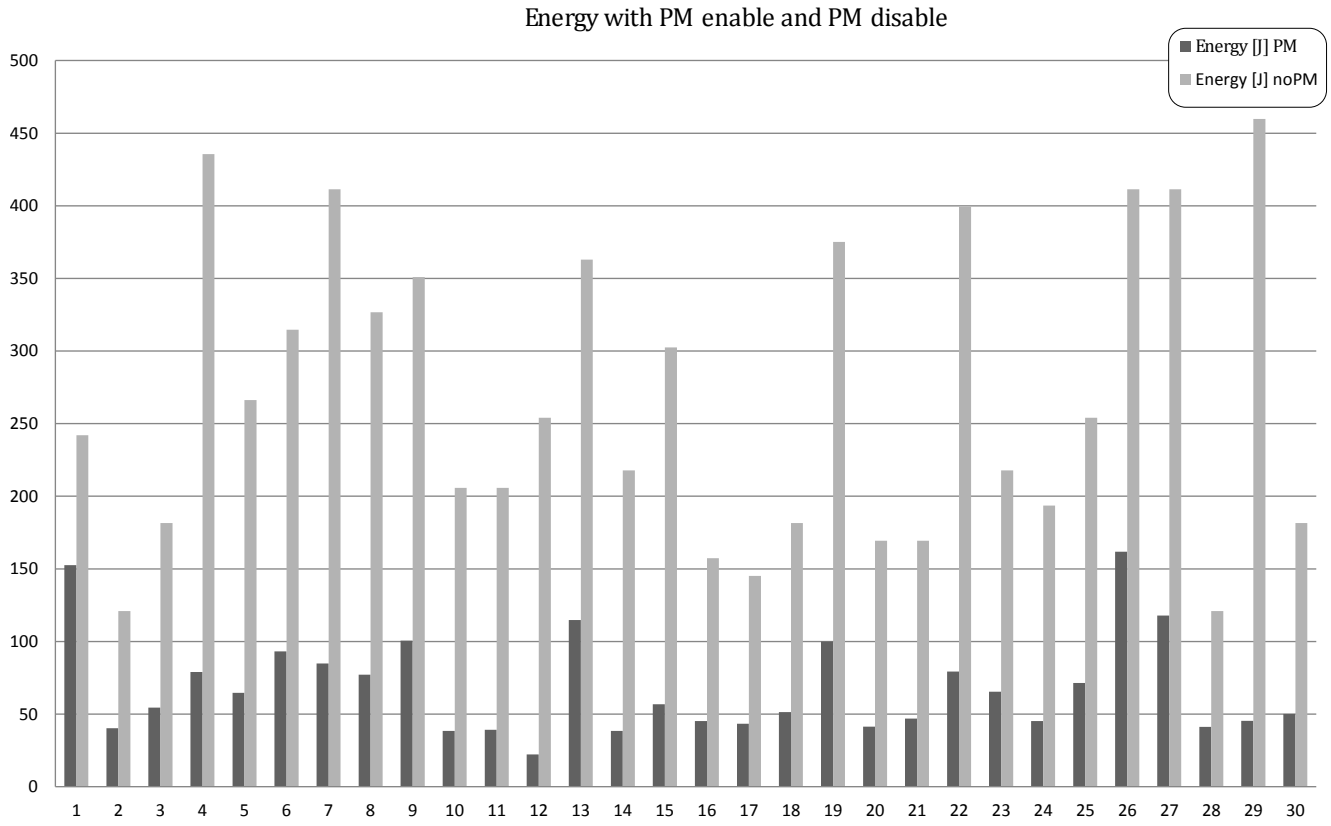


Figure 5.2: Bar Graph of 30 experiments with the PM Enable and the PM Disable

Chapter 6

Conclusions and recommendations

6.1 Conclusions

The power management subsystem reduced the energy consumption in more than a 50% in a 96.67% of an amount of 30 samples, running a certain application. This module will increase by double the battery life of a portable application running NuttX with a STM32 processor.

Using the NuttX power management interfaces, the system recommend a new state every given number of time slices, the amount of time slices needed to recommend a new state is customizable from the NuttX configuration file.

The STM3210e-eval board, is the best platform of a list of boards analysed, for the implementation of the power management sub-system in the NuttX RTOS.

With the power management enable, the drivers in NuttX are able to handle its own logic for every power mode, while the IDLE thread handles the board-specific logic.

A benchmark application was implemented and used, in order to verify the functionality of the project, and is available from the path `configs/stm3210/pm/`.

6.2 Recommendations

Provide PM support for the rest of the NuttX ports.

Add the logic to prepare the serial driver to enter in a lower power state, and the possibility to reject this state if required.

Add the logic in the USB and network drivers, to report activity in the critical sections, when a high traffic of information is being transferred and the system should remain in the `PM_NORMAL` state.

Bibliography

[1] Abielmona, R. (2000). **"Scheduling algorithmic research"** (Thesis No. 1029817). Canada: Ottawa-Carleton Institute.

[2] Atmel Corporation. (2009). **"AT91 ARM cortex-M3 based microcontrollers"** [datasheet]. Retrieved 07/02, 2012, from `http://www.atmel.com/Images/doc6430.pdf`

[3] IBM & Montavista. (2002). **"Dynamic power management"** Dynamic Power Management for Embedded Systems IBM and MontaVista Software, (1.1), 1-25.

[4] Microchip Technology Inc. (2007). **"PIC32MX7XX data sheet, section 10. power-saving modes"** (1st ed.)

[5] Nutt, G. (2011). **"NuttX RTOS porting guide"**, Retrieved 04/25, 2012, from `http://nuttx.sourceforge.net/NuttxPortingGuide.html`

[6] National Instruments. (2010). **"OPERATING INSTRUCTIONS AND SPECIFICATIONS NI 9227"**. Retrieved 07/22, 2012, from `http://www.ni.com/pdf/manuals/375101c.pdf`

[7] Nutt, G. (2012). In Nutt G. (Ed.), **"Nuttx overview."**, San José, Costa Rica: Corporative, NX-Engineering S.A.

[8] Nutt, G. (2012). **"NuttX RTOS home page"**. Retrieved June 27, 2012, from `http://nuttx.org/doku.php?id=nuttx`

[9] Nutt, G. (2012). **"NuttShell (NSH)."** Retrieved 09/23, 2012, from `http://nuttx.org/doku.php?id=documentation:nuttshell`

[10] Nutt, G. (2012). **"NX Graphics Subsystem."** Retrieved 09/30, 2012, from `http://nuttx.org/doku.php?id=documentation:nxgraphics`

[11] P. Podesser, (2009) Magazine:**"Military Embeded Systems"**, Portable power management for soldiers, May 16, 2009.

[12] STMicroelectronics. (2011). **"STM32F101xx and STM32F103xx advanced ARM-based 32-bit MCUs"** (1st ed.)

[13] STMicroelectronics. (2010). User manual: **"STM3210E-EVAL evaluation board."** Retrieved June 27, 2012, from `http://www.st.com/`

[14] STMicroelectronics. (2011). User manual: **"STM3220E-EVAL evaluation board."** Retrieved July 1, 2012, from `http://www.st.com/`

[15] STMicroelectronics. (2012). User manual: **"STM3240G-EVAL evaluation board."** Retrieved July 1, 2012, from `http://www.st.com/`

[16] STMicroelectronics. (2011). In www.st.com, **"Programming manual, cortex-M3 programming manual."** (Rev 4 ed.). United States of America: STMicro.

[17] Trevor M., Latchford S. (2009). In Michael Beach A. W. (Ed.), **"The insiders's guide to the STM32 ARM based microcontroller."** (1.8th ed.). Hitex (United Kingdom) Ltd.: hitex.

[18] Zuquim A.L, Viera. L.F. (2010). **"Efficient power management in real-time embedded systems"** [Abstract]. 1-2.

Appendix A

Important structures and definitions

A.1 Global data used by the PM module

This structure encapsulates all of the global data used by the PM module.

Public Types

```
#include "pm_internal.h"
struct pm_global_s
  uint8_t state        - The current state (as determined by an explicit
                         call to pm_changestate()
  uint8_t recommended  - The recommended state based on the PM algorithm
                         in function pm_update().
  uint8_t mndx         - The index to the next slot in the memory[] array
                         to use.
  uint8_t mcnt         - A tiny counter used only at start up.  The actual
                         algorithm cannot be applied until CONFIG_PM_MEMORY
                         samples have been collected.
  int16_t accum        - The accumulated counts in this time interval
  uint16_t thrcnt      - The number of below threshold counts seen.

  int16_t memory[CONFIG_PM_MEMORY-1]; - This is the averaging "memory."

  uint32_t stime      - The time (in ticks) at the start of the current
                        time slice.
  sem_t regsem        - This semaphore manages mutually exclusive access to
                        the power management registry.
  struct work_s work  - For work that has been deferred to the worker thread

  sq_queue_t registry - Registry is a singly-linked list of registered power
                        management callback structures
```

The next enumeration provides all power management states. Receipt of the state indication is the state transition event.

Public Types

```
#include <nuttx/power/pm.h>
enum pm_state_e
{
  PM_NORMAL = 0,   /* Normal full power operating mode.  If the driver is in
                    * a reduced power usage mode, it should immediately re-
                    * initialize for normal operatin.
                    *
                    * PM_NORMAL may be followed by PM_IDLE.
                    */
  PM_IDLE,         /* Drivers will receive this state change if it is
                    * appropriate to enter a simple IDLE power state.  This
                    * would include simple things such as reducing display
                    * backlighting.  The driver should be ready to resume
                    * normal activity instantly.
                    *
                    * PM_IDLE may be followed by PM_STANDBY or PM_NORMAL.
                    */
  PM_STANDBY,      /* The system is entering standby mode. Standby is a lower
                    * power consumption mode that may involve more extensive
                    * power management steps such has disabling clocking or
                    * setting the processor into reduced power consumption
                    * modes. In this state, the system should still be able
                    * to resume normal activity almost immediately.
                    *
                    * PM_STANDBY may be followed PM_SLEEP or by PM_NORMAL
                    */
  PM_SLEEP,        /* The system is entering deep sleep mode.  The most
                    * drastic power reduction measures possible should be
                    * taken in this state. It may require some time to get
                    * back to normal operation from SLEEP (some MCUs may
                    * even require going through reset).
                    *
                    * PM_SLEEP may be following by PM_NORMAL
                    */
};
```

The structure `pm_callback_s` contain pointers callback functions in the driver. These callback functions can be used to provide power management information to the driver.

Public Types

```
struct pm_callback_s
{
  struct sq_entry_s entry;   /* Supports a singly linked list */
  int (*prepare)(FAR struct pm_callback_s *cb, enum pm_state_e pmstate);
  void (*notify)(FAR struct pm_callback_s *cb, enum pm_state_e pmstate);
};
```

Appendix B

NuttX RTOS Supported Platforms

The next table shows the number of ports[1] for each platform.

Table B.1: Number of NuttX ports in each platform. [8]

Platfom	**Family**	**Number of ports**
Linux User mode simulation	-	1
ARM	ARM7TDMI	9
	ARM920T	1
	ARM926EJS	3
	ARM Cortex-M3	16
	ARM Cortex-M4	5
Atmel AVR	Atmel 8-bit AVR	3
	Atmel AVR32	1
Freescale	M68HCS12	2
Intel	Intel 8052 Microcontroller	1
	Intel 80x86	2
MicroChip	PIC32MX (MIPS)	4
Renesas/Hitachi	Renesas/Hitachi SuperH	1/2
	Renesas M16C/26	1/2
Zilog	Zilog Z16F	1
	Zilog eZ80 Acclaim!	1
	Zilog Z8Encore!	2
	Zilog Z80	2

[1] Boards with full NuttX RTOS support

Appendix C

NuttX directory structure

The general directly layout for NuttX is very similar to the directory structure of the Linux kernel (at least at the most superficial layers). At the top level is the main makefile and a series of sub-directories identified below [5].

Directory Structure

```
|- nuttx
|   |-- Makefile
|   |-- Documentation
|   |   '-- (documentation files)/
|   |-- arch/
|   |   |-- <arch-name>/
|   |   |   |-- include/
|   |   |   |   |--<chip-name>/
|   |   |   |   |  '-- (chip-specific header files)
|   |   |   |   |--<other-chips>/
|   |   |   |   '-- (architecture-specific header files)
|   |   |   '-- src/
|   |   |       |--<chip-name>/
|   |   |       |  '-- (chip-specific source files)
|   |   |       |--<other-chips>/
|   |   |       '-- (architecture-specific source files)
|   |   '-- <other-architecture directories>/
|   |-- binfmt/
|   |   |-- Makefile
|   |   |-- (binfmt-specific sub-directories)/
|   |   |   '-- (binfmt-specific source files)
|   |   '-- (common binfmt source files)
|   |-- configs/
|   |   |-- <board-name>/
|   |   |   |-- include/
|   |   |   |   '-- (other board-specific header files)
|   |   |   |-- src/
|   |   |   |   '-- (board-specific source files)
|   |   |   |---<config-name>/
|   |   |   |   '-- (board configuration-specific source files)
```

```
|   |   |   '---(other configuration sub-directories for this board)/
|   |   '-- <(other board directories)>/
|   |-- drivers/
|   |   |-- Makefile
|   |   |-- (driver-specific sub-directories)/
|   |   |   '-- (driver-specific source files)
|   |   '-- (common driver source files)
|   |-- fs/
|   |   |-- Makefile
|   |   |-- (file system-specific sub-directories)/
|   |   |   '-- (file system-specific source files)
|   |   '-- (common file system source files)
|   |-- graphics/
|   |   |-- Makefile
|   |   |-- (feature-specific sub-directories)/
|   |   |   '-- (feature-specific source files library source files)
|   |   '-- (common graphics-related source files)
|   |-- include/
|   |   |-- (standard header files)
|   |   |-- (standard include sub-directories)
|   |   |   '-- (more standard header files)
|   |   |-- (non-standard include sub-directories)
|   |       '-- (non-standard header files)
|   |-- lib/
|   |   |-- Makefile
|   |   '-- (lib source files)
|   |-- libxx/
|   |   |-- Makefile
|   |   '-- (libxx management source files)
|   |-- mm/
|   |   |-- Makefile
|   |   '-- (memory management source files)
|   |-- net/
|   |   |-- Makefile
|   |   |-- uip/
|   |   |   '-- (uip source files)
|   |   '-- (BSD socket source files)
|   |-- sched/
|   |   |-- Makefile
|   |   '-- (sched source files)
|   |-- syscall/
|   |   |-- Makefile
|   |   '-- (syscall source files)
|   '-- tools/
|       '-- (miscellaneous scripts and programs)
'- apps
    |-- netutils/
    |   |-- Makefile
    |   |-- (network feature sub-directories)/
```

```
|   |   '-- (network feature source files)
|   '-- (netutils common files)
|--  nshlib/
|   |-- Makefile
|   '-- NuttShell (NSH) files
|-- (Board-specific applications)/
|   |-- Makefile
|   |-- (Board-specific application sub-directories)/
|   |   '-- (Board-specific application source files)
|   '-- (Board-specific common files)
'-- examples/
    '-- (example)/
        |-- Makefile
        '-- (example source files)
```

Appendix D

LabVIEW code implemented for data acquisition

The image D.1 shows the source code used in the current data acquisition, this code is based in one of the examples provided for National Instruments in the software LabVIEW 2012, which is called `Current-Continuous Input.vi`.

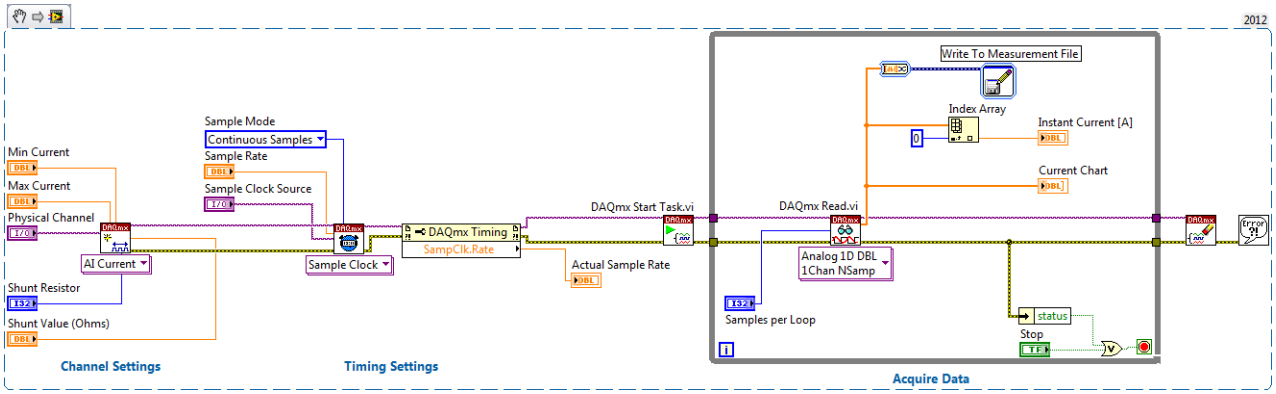


Figure D.1: LabVIEW block diagram (source code).

This application was implemented for data acquisition, it automatically detects any DAQ input connected in the computer, in this case the module NI-9227 was used in order to acquire the current consumed for the whole system, this device allows up to $5A_{RMS}$, see [6].

Appendix E

Connection Scheme

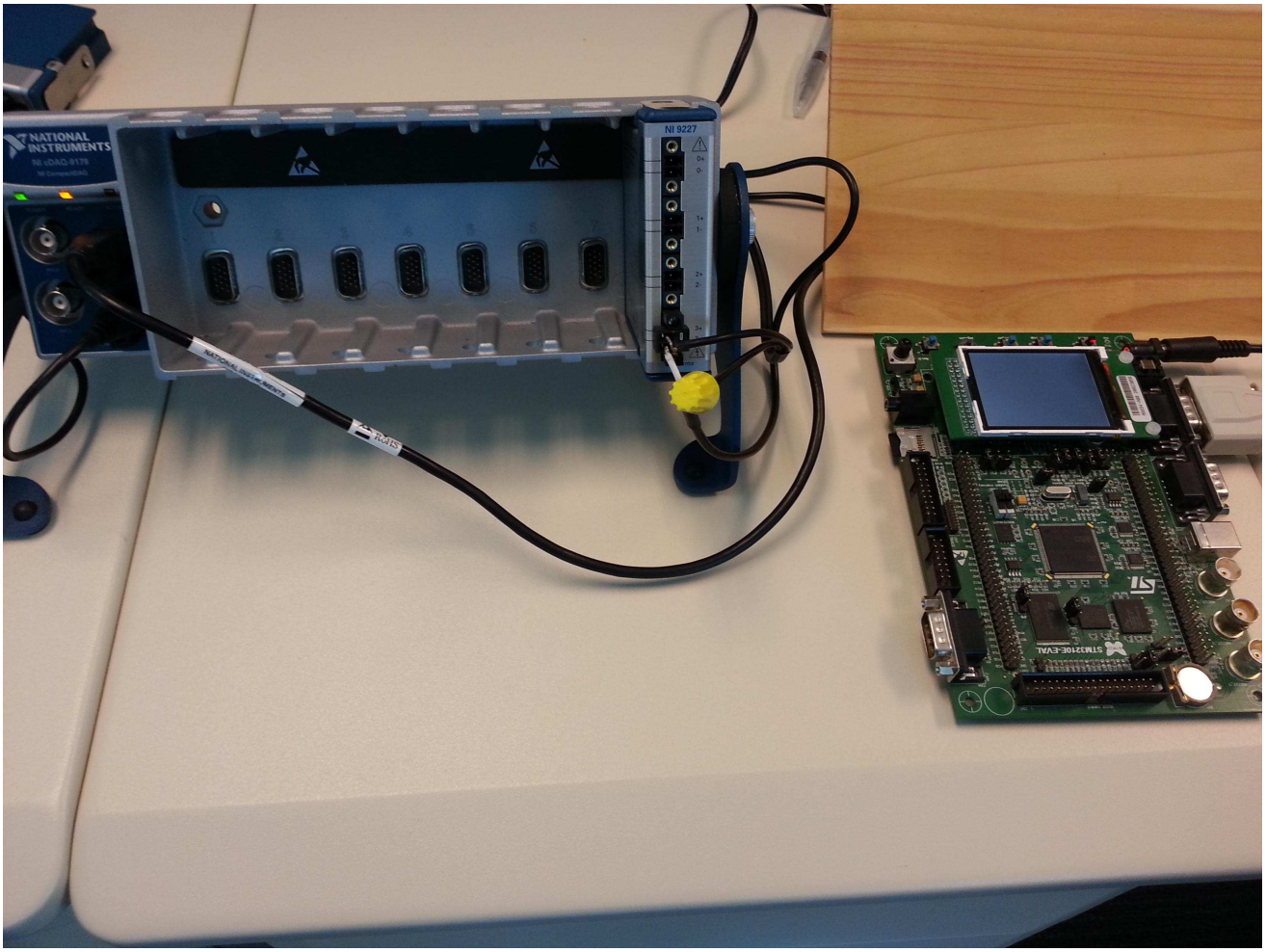


Figure E.1: Connection scheme of the hardware for the current acquisition.

Glossary

A

AR Name of environment variable, or Makefile variable., p. 12.

ARM ARM is a reduced instruction set computer (RISC), instruction set architecture (ISA) developed by ARM Holdings., p. 2.

Atmel Semiconductors manufacturer., p. 2.

AUC Area under curve., p. 19.

AVR AVR is a modified Harvard architecture., p. 2.

B

BSD Berkeley Software Distribution, are a family of permissive free software licenses., p. 11.

C

CAN Protocol, Controller Area Network., p. 2.

CC Name of environment variable, for compiling C code., p. 12.

CFLAGS Name of environment variable, or Makefile variable that can be set to specify additional switches to be passed to a compiler in the process of building computer software., p. 12.

CPU Central processing unit., p. 10.

D

DAQ Data acquisition system., p. 21.

E

Ethernet In the document it refers to Ethernet peripheral, p. 2.

F

FCFS First-come, First-Served scheduler., p. 10.

G

GNU Is a recursive acronym for "GNU's Not Unix!", chosen because GNU's design is Unix-like, but differs from Unix by being free software and containing no Unix code, p. 9.

GUI Graphical user interface., p. 45.

I

I2C Inter-Integrated Circuit protocol., p. 42.

L

LCD Liquid crystal display., p. vii.

LD Name of environment variable, or Makefile variable., p. 12.

LDFLAGS Name of environment variable, for the linking stage of compilation process., p. 12.

LED Light-Emitting Diode., p. vii.

M

MCU Microcontroller (also it is sometimes abbreviated as μC, uC), p. 7.

MIPS Acronym of "Microprocessor without Interlocked Pipeline Stages" or "Million Instructions Per Second"., p. 2.

MKDEP Construct Makefile dependency list, takes a set of flags for the C compiler and a list of C source files as arguments and constructs a set of include file dependencies which are written into a file., p. 12.

N

NM Name of environment variable, or Makefile variable., p. 12.

NSH NSH is a simple (bash-like) shell application for NuttX., p. 3.

NuttX Nuttx is a real-time embedded operating system (RTOS)., p. 1.

O

OBJCOPY The gnu objcopy utility copies the contents of an object file to another., p. 12.

OBJDUMP The gnu objsump, can be used as a disassembler to view executable in assembly form., p. 12.

OS Operating System., p. 9.

P

PM Power Management, usually refers to NuttX Power Management Sub-System., p. 2.

PRI Priority scheduler., p. 11.

PWM Pulse-width modulation., p. 15.

R

RR Round-Robin scheduler., p. 10.

RTC Real time clock., p. 3.

RTOS Real Time Operating System., p. 1.

S

SDIO Secure Digital Input Output, it's a type of Secure Digital card interface., p. 2.

SJF Shortest-Job-First scheduler., p. 10.

SPI Serial Peripheral Interface Bus., p. 42.

T

TDMS Technical Data Management System, a file that can be read for the program Excel., p. 45.

TFT Thin-film transistor., p. vii.

U

UI User interface., p. 16.

USART Universal Asynchronous Receiver-Transmitter, p. vii.

USB Universal Serial Bus., p. 2.

V

VDD Voltage Drain Drain, IC power supply pin, p. 3.