Università degli studi di Padova
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Linux and MQX RTOS in Asymmetric Multiprocessing environments: application in a drone navigation system.

Laureando
Laura Nao

Relatore
Michele Moro

Correlatore
Matteo Petracca
Scuola Superiore Sant'Anna, Pisa

Anno accademico 2015-2016

Abstract

This thesis aims at studying the Asymmetric Multi-Processing architecture on the NXP's i.MX 6SoloX SABRE board, featuring a i.MX 6SoloX processor that couples an ARM Cortex-A9 core and an ARM Cortex-M4 core. As a tangible application of the exploitability and of the potential employment of such an heterogeneous architecture, the prospect of using the board as a quadcopter's on-board flight control system is illustrated. A demo application is implemented for the board running MQX RTOS on the Cortex-M4 and Linux on the Cortex-A9, in which the MCU collects data from an IMU, calculates the board's orientation and sends data to the MPU for visualization.

Contents

1 Introduction **1**
1.1 Asymmetric Multiprocessing 1
1.2 Drone navigation system 2

2 Linux Kernel **3**
2.1 Kernel Source Structure 4
2.2 Yocto 8
2.3 U-Boot Bootloader 12
2.4 Preliminary work on AMBER board 13
2.4.1 U-Boot testing 14
2.4.2 Linux 4.1.13 Kernel Porting 18

3 NXP i.MX 6SoloX SABRE board **23**
3.1 i.MX 6SoloX Processor 24
3.2 MQX RTOS 26
3.2.1 MQX source structure 26
3.2.2 MCC - MultiCore Communication 28
3.2.3 MCC pingpong demo application 30
3.2.4 Firmware and demo compilation 32
3.3 Working on the i.MX 6SoloX SABRE board 33
3.3.1 Linux 4.1.13 Kernel on the SABRE board 33
3.3.2 Inter-core communication testing 35

4 Application in a drone navigation system **39**
4.1 Mahony's algorithm 45
4.2 Madgwick's algorithm 46
4.3 Demo application overview 47
4.4 MQX application 49
4.4.1 The MAG3110 sensor 49
4.4.2 The MPU6050 sensor 50
4.4.3 Main task 51
4.4.4 Statistics task 54
4.5 Linux application 56

4.5.1 Sysfs 56
4.5.2 Kernel-space character device driver 57
4.5.3 User-space application 59
4.5.4 Application usage and testing 60

5 Conclusions and future developments 65

References 67

List of Figures

2.1 Basic process to create a new Yocto recipe. 9
2.2 Layers of the Yocto project. 10
2.3 Front image of the Amber board. 13
2.4 U-Boot prompt screen. 15

3.1 Front view of the i.MX 6SoloX SABRE board. 23
3.2 Block diagram of the i.MX 6SoloX processor. 24
3.3 Overview of core and optional MQX components. 27
3.4 Ping-pong demo application. 31
3.5 Ping-pong demo application on Cortex-M4. 37
3.6 Sender task on Cortex-A9. 37
3.7 Receiver task on Cortex-M4. 38
3.8 Led lights on the board showing that both cores are running. 38

4.1 Diagram illustrating a quadcopter's flight control system. . . 40
4.2 Common sensors used in a IMU. 40
4.3 Diagram of a quadcopter's flight system involving both a MCU and a MPU. 42
4.4 Illustration of roll, pitch and yaw rotations. 43
4.5 Body's position in which gimbal lock occurs. 44
4.6 Inertial frame and body frame. 44
4.7 Structure of the demo application. 48
4.8 Front and back view of the GY-87 IMU board. 50
4.9 Overview of the I^{2}C bus. 52
4.10 Example of data samples printed on the M4 console. 53
4.11 Demo screen on Cortex-M4 startup. 61
4.12 User space application's start-up screen. 61
4.13 Example of data received by the Cortex-A9. 62
4.14 Cortex-M4 console screen (on top) and Cortex-A9 console screen (on bottom). A sent message and the correspondent received message are highlighted. 63

Chapter 1

Introduction

1.1 Asymmetric Multiprocessing

As the need for computational performance keeps growing it's becoming more common for embedded systems to include more than one CPU, to keep pace with increasingly demanding applications, such as those for real-time processing of media streams. Multicore processors meet the need for higher performances, speed and better power consumption, thus increasing the potential for embedded devices that are all growing in complexity. Two main options are available to handle the cooperation between multiple cores: **symmetric multiprocessing mode** (SMP) and **asymmetric multiprocessing mode** (AMP). The choice relies on the level of parallelism required by the application and on how easily the tasks can be distributed within the cores.

In SMP mode a single OS manages all the cores simultaneously. The parallelism in the application is extracted by the single OS, which dynamically schedules tasks across all cores while allowing full utilization of them. It is also responsible for handling the sharing of all resources and the inter-communication between the cores.

With AMP mode separate OS images can reside in memory; each core may or may not run an OS and may have a different architecture. AMP can be homogeneous or heterogeneus: in the first case each core runs the same OS, while in the latter each core runs a different OS (or a different version of the same one). Resource sharing between different OSs running simultaneously has to be managed more carefully than in SMP mode and a proper inter-core communication protocol has to be implemented, in order for the cores to communicate transparently between each other.

An interesting usage for AMP is possible, where one core runs a real-time OS (such as FreeRTOS or NXP's MQX) and handles all the computationally-demanding operations, while another core runs a Linux OS and manages the higher-level applications. Another scenario may include a Linux OS

alongside a wireless-oriented distribution such as OpenWRT.
An AMP environment allows also to make the best of multiple cores with different architectures optimized for specific activities, like having a MCU for real-time tasks and a MPU managing the UI. The ability for an embedded system to run on AMP mode highly increases the potential of the board and allows to fully exploit its capabilities.
The aim of this thesis is to study the asymmetric multiprocessing on the NXP's i.MX 6SoloX SABRE board. The board is equipped with a i.MX 6SoloX processor, which couples a Cortex-A9 core with a Cortex-M4 core. For the purpose of this thesis the A9 core runs Linux, while MQX RTOS is booted on the M4 core.

1.2 Drone navigation system

The capability to run real-time tasks on one core alongside a Linux OS on another core is the strongest feature of an asymmetric multiprocessing environment. To study and test the inter-core communication and to illustrate the potential of this heterogeneous approach, the i.MX 6SoloX board has been used as a quadcopter stabilization system.
In order for a drone to be balanced, all sensors of the Inertial Measurement Unit (IMU) are polled and the collected data are used to determine the body's attitude and orientation. Several approaches for sensor data fusing have been proposed, among which the complementary filter, the Kalman filter and the Mahony&Madgwick filter. As a representative example of the cooperation between the two cores, the i.MX 6SoloX SABRE board has been used to test and compare two algorithms for data fusing. In the developed application the M4 core is used to poll the sensors and determine the board's orientation. The results are then sent to the A9 core, responsible for visualization. The board is already equipped with a magnetometer and an accelerometer, while a compatible gyroscope module has been connected externally through the PCI-Express interface.
This solution combines the functional indipendence of a microcontroller, handling low-latency operations with real-time constraints (such as sensor polling and data fusing), with the flexibilty and the processing power of a MPU, supervising all higher level operations. Thus an asymmetric approach can improve the efficiency and the potentiality of a drone control system, compared with the traditional single-core configuration commonly used for quadcopters.

Chapter 2

Linux Kernel

Linux is the first free Unix-like operating system. Born as a student project, it has grown to become one of the most popular and loved operating system worldwide.
The history of Linux can be traced back to Minix, a simple non-free UNIX-based operating system. In 1991 Linus Torvalds, a young finnish student at the University of Helsinki, started to develop his own kernel, inspired by Minix and driven by the need of an less-limited OS while being unable to afford one. He started by developing drivers and hard-drive access and soon he had a first kernel version ready to be tested. Meanwhile the GNU Project was launched by Richard Stallman, with the intent of developing a free Unix-compatible OS called GNU. By 1991 most of the components of the system (an assembler, a C compiler, an editor and other Unix utilities) were ready, except for the kernel.
The Linux kernel developed by Torvalds was afterwards combined with the GNU underlying system to create a fully operational and free operating system. The Linux OS was released under the open source GPL licence, that enabled anyone to freely download and edit the source code. After that Linux kept drawing attention from a growing group of programmers and enthusiasts all over the world, that volounteered to debug, develop and improve the source code.
One of the earliest debate on Linux kernel was the one between Torvalds and Tanenbaum (Minix's creator), regarding Torvald's choice of designing it as a monolithic kernel[1]. In a monolithic kernel the whole operating system is a binary file (containing process management, memory management, file system, etc.) that runs in reserved RAM kernel space. This model differs from the microkernel architecture used by Minix, in which most of the OS runs as separate processes principally in user-space.

2.1 Kernel Source Structure

All latest kernel versions are available and downloadable for free from the Linux Kernel Archives at https://www.kernel.org/. The kernel sources have a simple numbering system. The first number denotes the kernel version; it's the least frequently changed number and it's incremented only when major changes regarding the concept and the code of the kernel occur. The second number indicates the major revision of the kernel version. Originally an even number denoted a stable release whereas an odd number indicated an unstable development release, but this convention has been dropped since the 2.6 release. The third number indicates the minor revision of the kernel and it's updated when new features or new drivers are added. The fourth number indicates further security patches or bug fixes.
Being one of the most flexible and adaptable operating system, the Linux kernel includes over 20 million lines of code and the structure of the source files can be rather confusing to look at as a first step[10]. All source files are accurately organized in a tree structure and each supported architecture has its own subdirectory, containing the architecture-specific kernel code. The kernel can be tweaked and built for a wide range of different environments, allowing the user to customize every aspect of a Linux distribution. An understanding of the structure of the source tree provides a good starting point for porting the kernel on a new device. The root of the source tree is organized in subdirectories as follows[7]:

- **arch**: this directory contains one subdirectory per supported architecture, containing all of the architecture-specific kernel code.
- **block**: this directory contains code for block device drivers (block devices exchange data in blocks, unlike character devices which exchange data in single bytes).
- **crypto**: this directory contains the source code for several encryption algorithms, such as the SHA1 secure hash algorithm and the AES cipher algorithm.
- **Documentation**: this directory contains documentation files in txt format. A large number of documents can be generated from the code and compiled in various formats (e.g. html documentation can be generated by running `$ make htmldocs` from the kernel source directory)[1].
- **drivers**: it includes all Linux device drivers source code. The directory is further divided in subdirectories, one for each device type (e.g. subfolder `drivers/char` is for character device drivers, `drivers/bluetooth` is for bluetooth device drivers, etc.).

[1]Up-to-date documentation is also available at https://www.kernel.org/doc/Documentation/

- **firmware**: this directory contains firmware images from old device drivers, prior to the introduction of the `request_firmware()` function. This function allows to obtain the image from user-space and it's currently considered the proper solution to deal with firmwares.
- **fs**: contains all the code to handle supported filesystems. The directory includes one subdirectory for each filesystem (e.g. fat filesystem's code is in the `fs/fat` subfolder).
- **include**: this directory represents the global header space and it contains all header files defining the interfaces between components of the kernel and between kernel and user space. The include files needed for a specific architecture are contained in the `arch/*/include` folder.
- **init**: this directory contains the kernel initialization code.
- **ipc**: contains the code that handles the communication layer between the kernel and processes (i.e. Inter-Process Communication).
- **kernel**: this directory contains the main kernel code. For instance, the subdirectory `kernel/locking` includes the code for semaphores and mutexes.
- **lib**: directory containing the code for kernel's library.
- **mm**: contains the code for memory management (e.g. code for page allocation and workingset detection).
- **net**: this folder contains the code for network protocols (e.g. protocols for ethernet, ipv6, bluetooth, etc.).
- **samples**: collects the code for examples and started (but unfinished) modules. This directory is a good starting point for everyone who wants to help developing useful features not yet fully implemented.
- **scripts**: this folder contains the scripts needed to configure and build the kernel.
- **security**: contains the code implementing the kernel's security modules.
- **sound**: this directory contains the sound/audio cards driver's code.
- **tools**: contains various tools interacting with the kernel. It includes, for instance, the code for Lguest, a minimal hypervisor that allows to experiment with virtualization.
- **usr**: this is the directory where compiled kernels are placed when the building process is complete.

- **virt**: contains the code for virtualization (i.e. providing the capability to run multiple operating systems at once).

Understanding this top level structuring of the source folder helps navigating the tree when looking for the code that implements specific features. For the purpose of this thesis, the following folders have been a matter of concern as well[2]:

- `$KERNEL_ROOT/arch/arm/boot`: after the kernel has been successfully built, the compiled binary image is put in this folder. Kernel images generated by the build process are either uncompressed `Image` files, compressed self-extracting `zImage` files, or `uImage` files (i.e. including an U-Boot wrapper). The compressed format is usually preferred, since decompressing is much faster than reading from external memory.

- `$KERNEL_ROOT/arch/arm/boot/dts`: this folder contains the device tree data files. The device tree is a data structure for describing hardware, in a way that is readable by an operating system. Rather than hard coding the details of a device, the OS reads the hardware topology from the device tree file provided at boot time. The device tree was originally created by the Open Firmware[3], as a method to exchange data between Open Firmware and the OS. The DT support was then introducted in Linux in 2005 for all powerpc architectures, regardless of whether or not they used Open Firmware. The DT structure was then condensed in the Flattened Device Tree (FDT) representation: a binary blob that can be passed to the kernel without requiring an Open Firmware implementation. Later, the FDT infrastructure was adopted for all the other mainline architectures (arm, microblaze, mips, powerpc, sparc, and x86) as well. All device tree source files, include files and compiled blobs are located in this source folder.

Changes to the kernel sources are distributed in the form of patch files, as generated by the `diff` command[2]. This saves from downloading whole source trees whenever a new update is available; applying a patch to the previous mainline kernel is much faster than downloading the latest tree from scratch. Anyone can contribute and several mailing lists are available to discuss proposed kernel patches[4]. All submitted patches are reviewed by kernel developers and must be accepted by Linus Torvalds before being integrated in the mainline. In order for Torvalds and the other reviewers

[2]Since the work is focused on an ARM-based board, this description refers to the ARM architecture

[3]Open Firmware is an hardware technology by Sun Microsystems, used to develop firmware independent from the operating system.

[4]A list of the main kernel mailing lists is available at http://vger.kernel.org/vger-lists.html

to be able to read and comment the submitted changes using standard e-mail tools, all patches must be issued as plain text files. It is advisable to use Git to create patches since the output file fullfills most of the format requirements as it is.

2.2 Yocto

With Linux becoming increasingly popular on embedded devices and the variety of commercial and non-commercial distros available, the industry needed a common build system and core technology. The Yocto project, born to fullfill this need, provides an open-source infrastructure to help developers create a custom distribution for any hardware architecture. Yocto is not itself a Linux distribution, it is a set of tools and components that allows to create ad-hoc distributions for specific devices. It provides a powerful customization architecture that allows to choose all aspects of the system, such as graphic subsystems, visualization middleware and services. Yocto supports all major embedded architectures (x86, x86-64, ARM,PPC, MIPS) and changing the target is as easy as changing a line in the configuration file. The project includes:

- A complete Linux OS, equipped with a validated set of packages (toolchain, kernel, user space). The included package versions are known to work well together, accurately selected to be up-to-date while ensuring robustness. Releases are distributed every 6 months, including the latest stable kernel, toolchain and package versions.

- A set of BSPs for the supported architectures, supplied and mantained by Yocto. Commercial linux vendors, OSVs, silicon suppliers, and board vendors may supply other BSPs, to add support for other architectures. A Board Support Package (BSP) is a collection of information that defines how to support a particular hardware device, set of devices, or hardware platform. It includes information about: hardware features present on the device, kernel configuration and additional hardware drivers or software components required.

- Support for target-device software developers through Yocto ADT (Application Developer Tools), including Eclipse plug-ins, the Quick EMUlator (QEMU, useful to simulate target hardware) and several user-space tools (e.g. for application profiling or latency measurements).

- Full documentation and strong supporting community.

Moreover, the Yocto Project hosts other projects and resources specifically intended for facilitating development with Linux on embedded devices. These include the Poky build system, the Autobuilder automated build and test system and the Embedded GLIBC library (i.e. a variant of the GNU C Library, designed to work well on embedded systems).
One of the benefits of choosing Yocto over other build systems, based on makefiles or shell scripts, is that it automates the source fetching process

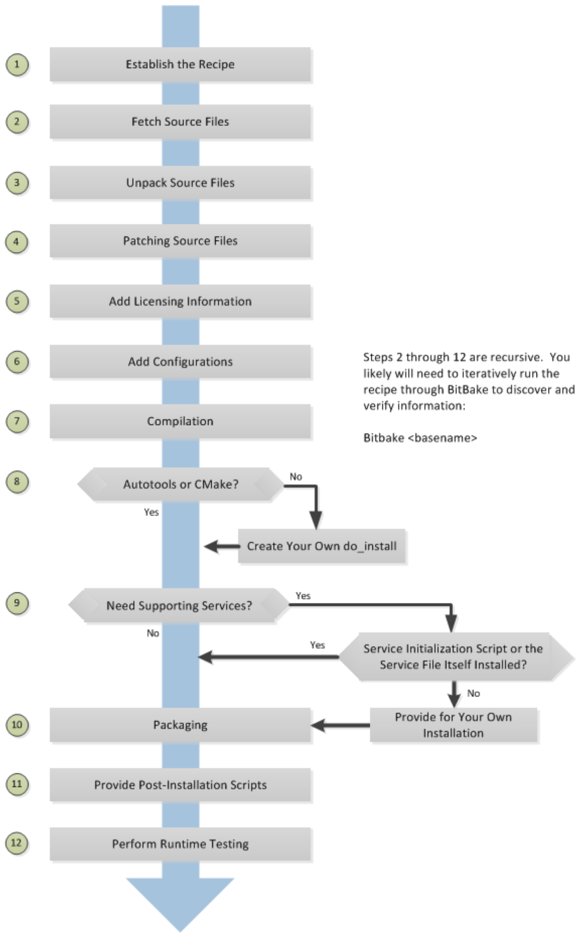

1
Establish the Recipe
2
Fetch Source Files
3
Unpack Source Files
4
Patching Source Files
5
Add Licensing Information
6
Add Configurations
Steps 2 through 12 are recursive. You likely will need to iteratively run the recipe through BitBake to discover and verify information:
Bitbake <basename>
7
Compilation
8
Autotools or CMake?
No
Yes
Create Your Own do_install
9
Need Supporting Services?
Yes
No
Yes
Service Initialization Script or the Service File Itself Installed?
No
10
Packaging
Provide for Your Own Installation
11
Provide Post-Installation Scripts
12
Perform Runtime Testing

Figure 2.1: Basic process to create a new Yocto recipe.

from different upstream sources or from local project repositories. Another strong point is that it makes easy to update packages, by using **recipes**. A recipe is a file containing information about a single package such as the location from which to download source, source patches (if needed), how to compile the source code and how to wrap the output (Fig. 2.1).
Yocto allows to build a whole Linux system in about an hour, therefore it is a helpful starting point for developers. In the preparatory phase of this study, Yocto has been used to quickly create a basic Linux distribution for the AMBER board, to become familiar with the build system and to test the board features. The basic steps to get started with Yocto are here briefly described[5]:

- Set up the Linux host system with the right packages. Most Linux

[5]A detailed quick-start guide is available at
http://www.yoctoproject.org/docs/latest/yocto-project-qs/yocto-project-qs.html.

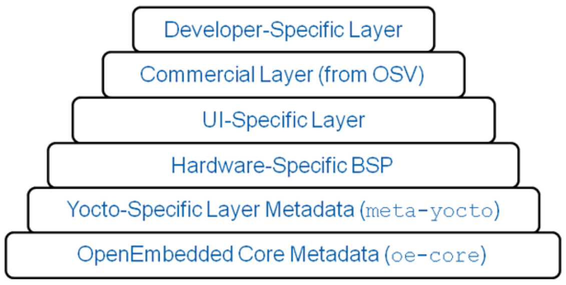

Developer-Specific Layer
Commercial Layer (from OSV)
UI-Specific Layer
Hardware-Specific BSP
Yocto-Specific Layer Metadata (meta-yocto)
OpenEmbedded Core Metadata (oe-core)

Figure 2.2: Layers of the Yocto project.

distributions are supported; the distro used in this case is Linux Mint 17.2.

- Download a Yocto release and add the BSP layers if needed, by editing the `conf/bblayers.conf` file.
- Edit the `conf/local.conf` file and set build parameters to fully utilize the host machine's resources.
- Initialize the build environment by sourcing the oe-init-build-env script. The script defines the OpenEmbedded build environment on the host.
- Run Bitbake, the task executor and scheduler used by the OpenEmbedded[6] build system to create the Linux image. The bitbake command is followed by the OS image for the target. A lot of different choices are available, based on the desired features: from lightweight minimal images allowing the device just to boot, to more complex images including graphic interface and applications.

When the building is done, the following images are located in the deploy folder:

- A `.sdcard` image, that can be flashed as-is on an SD Card. It already includes the kernel, the bootloader and a filesystem. Once flashed, the card is ready to be plugged in the board for booting.
- The rootfs `ext3` filesystem image
- The Linuk kernel image (`uImage` format)
- U-Boot bootloader images, both for SD Card and NAND Flash.

[6]The Yocto Project uses a build host based on the OpenEmbedded (OE) project, which includes the BitBake tool, to create complete Linux images. The BitBake and OE components are combined together to form the Poky build host.

- SPL images, both for SD Card and NAND Flash. The SPL (Secondary Program Loader) is a small binary that fits in the SRAM, used to load the main U-Boot into the system RAM.
- Other filesystem images, such as `UBIFS` and `NFS`.

The user can either flash the complete `.sdcard` image, that automatically takes care of partitioning, or manually partition the card and flash images separately. In both cases any image can be flashed to the SD Card with the `dd` Unix shell command, by specifying the proper operands (e.g. data block size and starting data block number).

2.3 U-Boot Bootloader

U-Boot is a multi-platform, open-source, universal bootloader with support for loading and managing boot images, such as the Linux kernel. U-Boot provides a combination of bootloader, system firmware and features such as booting from IDE or SCSI disk, booting from flash memories and via network. These aspects, along with the support for several platforms (including 68k, ARM, AVR32, Blackfin, MicroBlaze, MIPS, Nios, SuperH, PPC and x86), makes U-Boot the most flexible and the most widely used bootloader for embedded systems.
On most desktop and server systems, information on hardware devices' configuration, interrupt routing specifics and other information needed by the Linux OS are provided by extensive system firmwares such as BIOS, UEFI or OpenFirmware. Embedded systems usually don't have such detailed firmwares and need to perform these tasks through the bootloader. As a consequence, on embedded devices U-Boot is preferred over more desktop/server oriented bootloaders such as GRUB or LILO. The main features of U-Boot are the following:

- Network download via TFTP, BOOTP, DHCP, NFS protocols
- Serial download via terminal emulation program, such as Kermit or Minicom. S-record and binary formats are supported.
- Flash memory management: copy, erase, protect, cramfs, jffs2
- Support for CFI NOR-Flash and NAND-Flash
- Various memory utilities (e,g. copy, dump. crc, check, mtest)
- Support for IDE, SATA, USB mass storage devices
- Boot from disk: raw block, ext2, fat, reiserfs
- Interactive shell: choice of simple shell or "busybox" shell with many scripting features.

U-boot provides a powerful command line interface, accessible through a terminal emulator connected to the target board's serial port, and a large set of commands. User configuration is handled by means of environment variables, which can be saved to flash memory. Command scripts can be saved in environment variables as well and launched by the run command. This highly interactive and programmable approach makes U-Boot a useful tool for initial development and debugging on the board, as well as handling the loading of the kernel image[7].

[7]Extensive documentation for U-Boot is available at http://www.denx.de/wiki/DULG/Manual

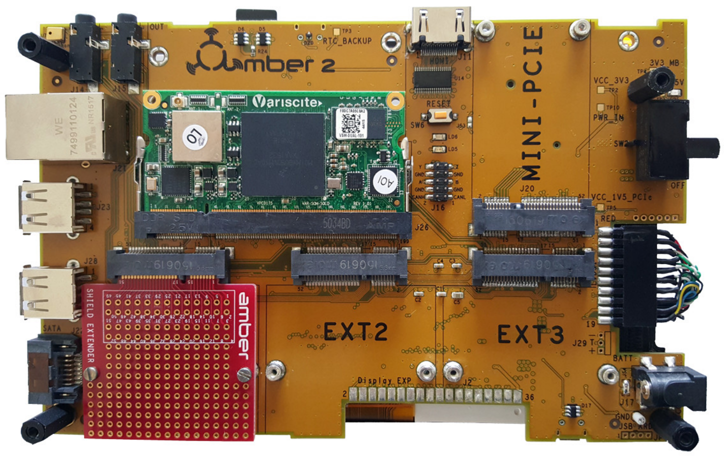


Figure 2.3: Front image of the Amber board.

2.4 Preliminary work on AMBER board

To become familiar with U-Boot, the Linux Kernel and development tools, some preparatory work has been done on the Amber board (Fig.2.3). Amber (Advanced Mother Board for Embedded systems pRototyping) is an open platform, designed to be a scalable and flexible general purpose test board for any IoT activity. The board can be easily customized, by changing both communication modules and processor. Amber can natively host all Variscite SoMs (System On Module) among the MX6 and SOLO/DUAL family. The board used for testing is equipped with a VAR-SOM-MX6 module with a quad-core Cortex-A9.
The Amber board can be connected to a PC through the USB serial port. Interaction with the board can be handled through serial communication softwares. Two open-source alternatives for Linux have been tested:

- **Minicom**: menu-driven serial communication program. Minicom can be launched on a terminal by typing:
  ```
  $ sudo minicom -D /dev/ttyXXX
  ```
  replacing /dev/ttyXXX with the name of the serial device. The device number can be identified by checking the kernel messages with the unix command:
  ```
  $ dmesg
  ```
- **C-Kermit**: C-Kermit is a portable Scriptable Network and Serial Communication Software for Unix, similar to Minicom. C-Kermit can be launched by typing:
  ```
  $ kermit -l /dev/ttyXXX
  ```
  replacing /dev/ttyXXX with the name of the serial device. Image

or file transfering over serial port by using Minicom sometimes failes; C-Kermit has been verified to be more robust on that.

In both cases, in order to successfully interact with the board, the baud rate and the flow control have to be set.
The activity on the Amber board included:

- Booting Linux on the board, by using a Yocto-based image (Fido release) including U-Boot, Linux 3.14 Kernel and Yocto filesystem. Variscite provides patches to extend the framework to support i.MX6 System On Modules[8].
- U-Boot recompilation and feature-testing
- Linux 4.1.13 Kernel porting

While U-Boot and the Linux kernel have been recompiled, the Yocto ext3 rootfs has been kept as filesystem.

U-Boot and the Linux Kernel have been build by using the **Linaro** toolchain. In order to cross-compile for the arm architecture the following shell environment variables must be set:

```
$ export PATH=/opt/linaro/gcc-linaro-4.9-2015.02-3-x86_64_arm-linux-gnueabihf/bin
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabihf-
```

Where `/opt/linaro` is the folder where the toolchain source have been extracted.

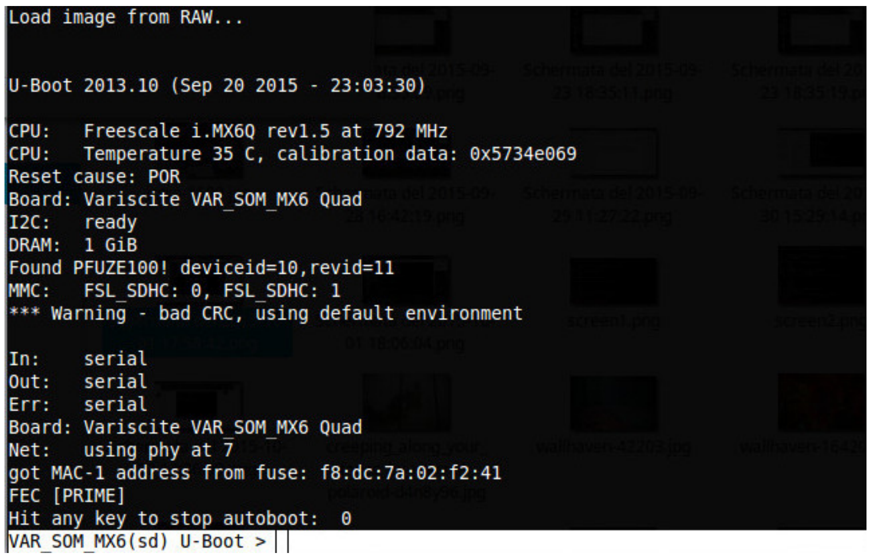
2.4.1 U-Boot testing

Compilation

Board vendors and processor manufacturers may supply U-Boot customized images. Three main repositories hosts U-Boot source code suitable for the Amber board's hardware: the repository of Variscite (i.e. the SoM manufacturer), the repository of Freescale (i.e. the SoC manufacturer) and the repository of Freescale's community. Here Variscite's repository has been chosen. Here are the steps taken to build and run U-Boot on the board:

- Checkout of the latest repository branch:
  `$ git clone https://github.com/varigit/uboot-imx.git`

[8]Instructions on how to patch Yocto to support the SoM are available at: http://variwiki.com/index.php?title=VAR-SOM-MX6_Yocto_Fido_R2_Build_Yocto_release



```
Load image from RAW...


U-Boot 2013.10 (Sep 20 2015 - 23:03:30)

CPU:   Freescale i.MX6Q rev1.5 at 792 MHz
CPU:   Temperature 35 C, calibration data: 0x5734e069
Reset cause: POR
Board: Variscite VAR_SOM_MX6 Quad
I2C:   ready
DRAM:  1 GiB
Found PFUZE100! deviceid=10,revid=11
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Board: Variscite VAR_SOM_MX6 Quad
Net:   using phy at 7
got MAC-1 address from fuse: f8:dc:7a:02:f2:41
FEC [PRIME]
Hit any key to stop autoboot:  0
VAR_SOM_MX6(sd) U-Boot > |
```

Figure 2.4: U-Boot prompt screen.

- U-Boot cross-compilation for arm-based device:
  `$ cd uboot-imx/`
  `$ export ARCH=arm`
  `$ export CROSS_COMPILE=arm-linux-gnueabihf-`
  `$ make mrproper`

- SD-Card image building:
  `$ make mx6var_som_sd_config`
  `$ make`

- SD-Card (/dev/sdxxx) flashing:
  `$ sudo dd if=SPL of=/dev/sdxxx bs=1K seek=1; sync`
  `$ sudo dd if=u-boot.img of=/dev/sdxxx bs=1K seek=69; sync` **NB**: the sync command is essential to commit buffer cache to memory.

The U-Boot prompt can be accessed by pressing a key right before the kernel is loaded, to stop the booting process and interact with the bootloader (Fig.2.4).

Demo testing

U-Boot allows to dynamically load standalone applications and run them directly in the environment provided by the bootloader. Standalone applications can use some resources of U-Boot such as console I/O functions or interrupt services and they are designed to work out-of-the-box on different architectures (see `$UBOOT_ROOTDIR/doc/README.standalone`). Two demos are provided and automatically compiled with U-boot: the Hello

world application will print the arguments provided by the user on the console, while the CPM timer application will generate an interruption every second (this last one is designed to work only on MPC8xx CPUs). Sources and generated binaries of the demos can be found in the folder `$UBOOT_ROOTDIR/examples/standalone/`.
Trying to run the Hello world app as it is on the Amber board will cause a bootloader freeze, due to some inconsistencies in the configuration files for the ARM architecture. By investigating the configuration files two main discrepancies have been found: the first one is relative to the register holding the pointer to the jump table for exported functions and the second one is relative to start and load addresses for the standalone applications.
The sequence of steps to fix them and to be able to successfully run the Hello world app on the Amber board is here described. The description refers to U-boot 2013.10 (branch imx_v2013.10_v4 on the Variscite repository).

Loading the Hello World app The binary file of the demo apps on U-Boot can be loaded dynamically on a specific address and with a specified baudrate with the prompt command `loadb [ off ] [ baud ]`. Once the app has been loaded at the specified address on memory it can be started with the prompt command `go addr [arg ...]`, where "arg" are the function's arguments. These are the commands to correctly load and run the Hello world demo app using C-Kermit:

```
$ loadb
$ CTRL+c
$ send UBOOT\_ROOTDIR/examples/standalone/hello\_world.bin
$ connect
$ go 12000000 Hello world!}
```

The Hello world demo contains calls to the `printf` U-Boot function. U-Boot functions need to be exported in order to be used in standalone applications; all exported functions are listed in the `$UBOOT_PKG/include/_exports.h` file.
The functions are exported by U-Boot via a jump table and the pointer to that jump table is machine-dependent. According to the definition in `$UBOOT_PKG/arch/arm/include/asm/global_data.h`, the dedicated register that holds the pointer to the jump table on ARM is r9.

Setting the right register for the pointer to the jump table In the application folder, along with the sources and the compiled binaries of the demos, there is a `stubs.c` file that define some crucial macros for exported functions. Each macro defines, for each architecture, the register holding the pointer to the jump table and sets the offset corresponding to the exported function in the table. In particular, for the ARM architecture:

stubs.c

```
/*
 * r8 holds the pointer to the global_data, ip is a
    call-clobbered
 * register
 */
#define EXPORT_FUNC(x) \
        asm volatile (                    \
"       .globl " #x "\n"                  \
#x ":\n"                                  \
"       ldr     ip, [r8, %0]\n"           \
"       ldr     pc, [ip, %1]\n"           \
        : : "i"(offsetof(gd_t, jt)), "i"(XF_ ## x *
            sizeof(void *)) : "ip");
```

The first `ldr` instruction identifies the position of the jump table containing all the addresses for the exported functions within the global data section, while the second `ldr` instruction identifies the specific function to jump to within the jump table. For example, a `printf` call will be resolved by the compiler in:

```
.globl printf
printf:
ldr     ip, [r8, \#84]
ldr     pc, [ip, \#20]
```

The `stubs.c` file refers to r8 as the register holding the pointer to the jump table, although the proper register for the ARM architecture should be r9 as described earlier. In order for the processor to jump to the proper function, r9 needs to be set instead of r8 in the macro definition.

Setting the right addresses for the standalone application The default load and start addresses of the applications for ARM-based devices are specified in the `$UBOOT\_ROOTDIR/arch/arm/config.mk` file. The default load address is: `CONFIG_STANDALONE_LOAD_ADDR = 0x0c100000`. Even if 0x0c100000 is set as the default address, calling `loadb` on the U-Boot prompt will return 0x12000000 as the default load and start address:

```
VAR_SOM_MX6 on Amber board (sd)> loadb
## Ready for binary (kermit) download to 0x12000000 at 115200
   bps...
## Total Size      = 0x0000024e = 590 Bytes
## Start Addr      = 0x12000000
```

The address 0x12000000 is configured in `$UBOOT_ROOTDIR/include/configs/mx6var_common.h` as:

```
#define CONFIG_LOADADDR 0x12000000
```

This inconsistency between the load address used by `loadb` command and the load address set by the standalone application's linker cause bad memory accesses. Setting `CONFIG_STANDALONE_LOAD_ADDR = 0x12000000` in the `$UBOOT\_ROOTDIR/arch/arm/config.mk` allows the Hello World application to run successfully.
A patch, applicable with the `patch` Unix command, has been developed to automatically fix the issues described earlier.

2.4.2 Linux 4.1.13 Kernel Porting

The porting of the Linux 4.1.13 Kernel on the Amber board has been realized using the existing porting of the Linux 3.14.38 Kernel by Variscite as a starting point. Variscite's custom version of the Linux 3.14.38 Kernel has been developed as a branch of the official Freescale's Linux 3.14.38 Kernel, adding support for various SoM such as VAR-SOM-MX6, VAR-SOM-SOLO and DART-iMX6. At the time when the porting was done, the latest Kernel released by Variscite was 3.14.38 and the latest vanilla kernel available was 4.1.13. The decision on working on the latest release, and therefore to realize the porting, has been made to ensure a more stable starting base and to benefit from the performance and power efficiency improvements of the 4.1 Kernel.
The Vanilla 4.1 Kernel (i.e. the Linux kernel without additional patches) already integrates the support for Freescale's iMX6 processor, only the compatibility for the SoM needs to be added. The Amber on which the porting has been realized features the VAR-SOM-MX6; the steps to port the Kernel on Amber boards equipped with a different Variscite SoM are analogous to the ones described.
The comparison between Variscite's 3.14 kernel and Freescale's 3.14 kernel (i.e. the branch from which Variscite's kernel has been developed from), gives a hint on the changes that have to be made in order to support the VAR-SOM-MX6 on the latest 4.1 kernel . The changes on the Variscite's kernel have been analyzed by running the `diff` Unix command against Freescale's kernel. `diff` analyzes two files and prints the lines that are different: essentially, it outputs a set of instructions for how to change one file in order to make it identical to the second file. It can be run recursively, to compare sub-directories as well. The major changes have been replicated on the new 4.1 Vanilla Kernel, while some minor changes have been neglected as not relevant to the purpose of the thesis.

Kernel configuration

Since the kernel offers several features and supports a lot of different hardwares, configuration is a required step before building it. The Linux kernel configuration is located in the file: `$KERNEL_ROOT/usr/src/linux/.config`. It is not recommended to edit this file directly but to use one of these commands:

- **make config** goes through each option and starts a character-based questions and answer session
- **make menuconfig** starts a terminal-oriented configuration tool
- **make xconfig** starts a X-based configuration tool

Kernel configuration is controlled by configuration options, which are prefixed by `CONFIG` in the form `CONFIG_FEATURE`. An option will either indicate some driver is built into the kernel (=y), will be built as a module (=m) or is not selected. The unselected state can either be indicated by a line starting with "#" (e.g. `"# CONFIG_SCSI` is not set") or by the absence of the relevant line from the `.config` file. To get off and running quickly it's also possible to use default configuration files (`.defconfig` extension), located in the `$KERNEL\_ROOT/arch/<target-architecture>/config/` folder.
The default configuration file for the VAR-SOM-MX6 module has been directly ported from the Variscite Linux 3.14.38 Kernel. None of the Kernel options has been modified except for the `CONFIG_PROC_DEVICETREE` flag, which has been enabled in order to allow device tree debugging after booting. The defconfig file is located in `$KERNEL_ROOT/arch/arm/configs/imx_v7_var_defconfig`.

Porting Device Tree files

Device Tree data files specific for the VAR-SOM-MX6 module are located in the `$KERNEL_ROOT/arch/arm/boot/dts` folder of the Variscite Kernel. All .dts and .dtsi files regarding the module must be included in order for the hardware of the SoM to be correctly detected at boot. The following files have been ported to the 4.1.13 Vanilla Kernel:

- `$KERNEL_ROOT/arch/arm/boot/dts/imx6qdl-var-som.dtsi` device tree source include file has been added to the 4.1 Kernel. The file contains references to the `hdmi_audio`, `hdmi_cec`, `hdmi_core`, `hdmi_video` devices, which have been unified in a single `hdmi` device in the latest 4.1 Kernel. Therefore all references to those nodes have been changed accordingly in the .dtsi file. The label for the `flexcan` device has been changed in the latest Kernel release, so all references to that device have been corrected as well.
- `$KERNEL_ROOT/arch/arm/boot/dts/imx6q-var-som.dts` device tree source file has been copied among the Kernel 4.1 .dts files. It includes the `imx6qdl-var-som.dtsi` file and encompasses the description of all physical devices on the VAR-SOM-MX6 module.

- `$KERNEL_ROOT/arch/arm/boot/dts/imx6q.dtsi` and `$KERNEL_ROOT/arch/arm/boot/dts/imx6dl.dtsi` have been modified. A label has been added to the `cpu@0` device, in order for the references on the `imx6qdl-var-som.dtsi` file to be satisfied. In fact this file makes use of the label to add two properties regarding voltage supplies for the device.

- `$KERNEL_ROOT/arch/arm/boot/dts/Makefile` has been modified to add a make rule to build the dtb for the board.

Making the audio codec work

The VAR-SOM-MX6 features the Texas Instrument's TLV320AIC3106 low power stereo audio codec, to process analog audio signals. In order for the codec to be recognized, the following changes have to be made:

- `$KERNEL_ROOT/sound/soc/fsl/imx-tlv320aic3x.c` codec driver has been copied in the same folder of the 4.1 Kernel. The `$KERNEL_ROOT/sound/soc/fsl/` folder groups all the Freescale SoC drivers.

- `$KERNEL_ROOT/sound/soc/fsl/Makefile` has been consequently modified, to indicate that the added codec driver source file has to be built.

- `$KERNEL_ROOT/sound/soc/fsl/Kconfig` file has been modified to add a new kernel configuration option to support SoC audio with the tlv320aic3x codec (including the description for the option itself).

Making Wi-Fi and Bluetooth work

The VAR-SOM-MX6 includes the Texas Instruments' WL183xMOD WiLink, a module for both Wi-Fi and Bluetooth support. The driver source file for the module added by Variscite in the 3.14 Kernel has been ported to the 4.1 Kernel as follows:

- `$KERNEL_ROOT/drivers/misc/ti-st/tty_hci.c` driver has been copied in the same folder of the latest Kernel.

- A rule to build the driver source file just added has been added to `$KERNEL_ROOT/drivers/misc/ti-st/Makefile`.

- A new kernel configuration option to support the WiLink module has been added to the `$KERNEL_ROOT/drivers/misc/ti-st/Kconfig` file.

- `$KERNEL_ROOT/drivers/bluetooth/btwilink.c` has been modified, according to some changes apported by Variscite in the old 3.14 Kernel.

Build and test the Kernel

The kernel has been built with the Linaro toolchain as follows:

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ make -j6 LOADADDR=0x10008000 uImage
$ make -j6 LOADADDR=0x10008000 modules
$ make -j6 dtbs
```

The kernel image and the device tree blob are then copied to the SD-Card and modules are installed:

```
$ sudo make ARCH=arm modules\_install
    INSTALL\_MOD\_PATH=/media/rootfs/
$ cp arch/arm/boot/zImage /media/laura/BOT\_varsomi/
$ cp arch/arm/boot/dts/imx6q-var-som.dtb /media/BOT\_varsomi/
```

Once the board is booted from the SD and the kernel is loaded, it's possible to change the current `console_loglevel` (to be able to see all debug printed messages) by typing on the kernel prompt:

```
$ echo 8 > /proc/sys/kernel/printk
```

The patch for the 4.1 Kernel, including all changes made so far, has been generated with the `git diff` command. At the time of writing, the latest stable vanilla kernel is 4.4.6. The patch has been tested and verified to be working on this latest version as well.

Chapter 3

NXP i.MX 6SoloX SABRE board

Once familiar with the build tools and the kernel development environment, the work was focused on the NXP i.MX 6SoloX SABRE (Smart Application Blueprint for Rapid Engineering) board (Fig.3.1). The reason for choosing this board relies on its heterogeneous asymmetric architecture: the board features the i.MX 6SoloX processor, which couples a Cortex-M4 and a Cortex-A9 in a single chip. The hybrid MPU/MCU system-on-chip design allows to run advanced operating systems (e.g. Linux, Android) while providing realtime responsiveness.

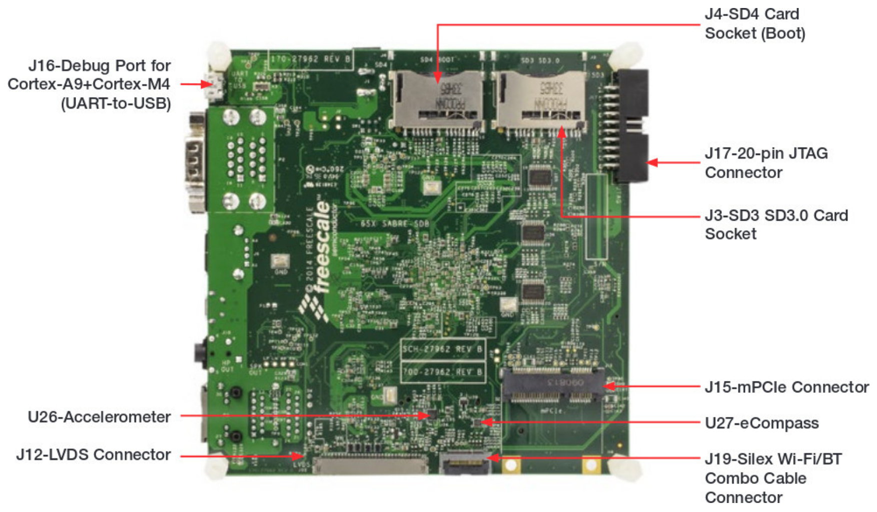

J4-SD4 Card Socket (Boot)
J16-Debug Port for Cortex-A9+Cortex-M4 (UART-to-USB)
J17-20-pin JTAG Connector
J3-SD3 SD3.0 Card Socket
J15-mPCIe Connector
U26-Accelerometer
U27-eCompass
J12-LVDS Connector
J19-Silex Wi-Fi/BT Combo Cable Connector

Figure 3.1: Front view of the i.MX 6SoloX SABRE board.

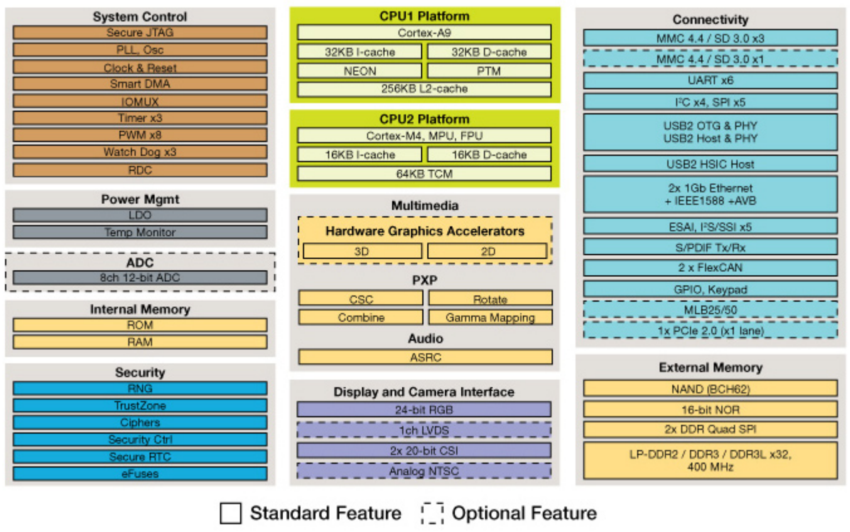


Figure 3.2: Block diagram of the i.MX 6SoloX processor.

3.1 i.MX 6SoloX Processor

As said before, the i.MX6 SoloX processor has two cores: one ARM Cortex-A9 (1 GHz with 512 KB L2 cache) and one ARM Cortex-M4 (200 MHz with 16 KB instruction and data caches). The Cortex-A family of processors provides a range of solutions for devices that feature a rich operating system such as Linux or Android. The Cortex-A9, based on the ARMv7-A architecture, is a popular choice in smartphones, low-cost handsets, tablets, digital TV and applications enabling the IoT (Internet Of Things). The Cortex-M4 is built on the ARMv7-M architecture and includes instructions specifically optimized to handle DSP (Digital Signal Processing) algorithms. It has also the option to get single precision FPU (floating point unit). Therefore, the Cortex-M4 is a better choice (in terms of performance and power consumption) for applications involving floating point math, in comparison to the other processors of the Cortex-M series. This comes in handy in the case of this study since the developed application manipulates sensor data, which are usually delivered as floating points to ensure accuracy. Moreover, IMU data fusing algorithms outputs floating point data as well.

Developing an application that uses both cores requires handling the inter-core communication and deciding which core is responsible for each device, since both can access peripheral blocks on the processor. Safe access to the SoC resources (peripherals, shared memory) is ensured by a Resource Domain Controller (RDC); if a peripheral is assigned to one core, the other one won't try to access it. The belonging of a device's control to one core or the other can be specified by enabling/disabling the peripheral in the device

tree file.
The Cortex-A9 is the primary core and booting i.MX 6SoloX requires U-Boot (or another bootloader) to be present. As a primary core, the Cortex-A9 is responsible for loading the bootloader, initiating the Cortex-M4 firmware and launching the M4. Once the Cortex-A9 is up and running, the Cortex-M4 can be brought up either by the U-Boot command `m4boot` or after the Cortex-A9 starts loading the Linux kernel. The Cortex-M4 does not have a boot ROM, so in either case the application code and the firmware have to be made available to the M4 as a unified binary image via on-board QSPI flash. The Cortex-M4 is brought up by the Cortex-A9 by a reset following the standard M4 start-up process of booting from a vector table at address zero. Once it's up, the Cortex-M4 can interact with most of the on-board peripherals.
Communication between the cores is possible, by exchanging messages through buffers in shared memory space; a multi-core communication solution will be described in the next section.

3.2 MQX RTOS

MQX (Message Queue Executive) is a proprietary Real Time operating system distributed by NXP, designed for uniprocessor, multiprocessor, and distributed-processor embedded real-time systems. The RTOS is integrated with a full range of leading 32-bit embedded processors (including Cortex-M4) and natively includes the commonly used device drivers. Crucial portions of the OS (e.g. context switching and interrupt handling) are optimized for NXP architectures. The boot sequence have been optimized as well, to guarantee minimum delay time between the hardware reset and the application startup.
MQX is based on a microkernel architecture and it's designed to take as little space as possible, accordingly to memory constraints of embedded systems, thus leaving more space to the applications. The OS can be configured to occupy as little as 6 KB of ROM, including kernel, interrupts, semaphores, queues and memory manager.
Real-time constraints are satisfied by priority-based preemptive scheduling; this ensure that deadlines of high-priority threads are consistently met, irrespective of the number of competing threads. MQX RTOS allows three task-scheduling policies: FIFO (which is the default one), Round Robin and explicit scheduling through task queues. A combination of the three is also possible, by setting the policy for each task separately.
The RTOS provides a run-time library of functions (with complete API), which can be used to realize real-time multitasking applications. Among others, there are MCC (Multi-Core Communication) functions that implement a simple message passing protocol, allowing communication between tasks running on the same CPU or on different processors.
The key features of MQX are the high degree of personalization and scalable size. The highly customizable design is achieved by adopting a component-based approach. MQX consists of a set of core (mandatory) and optional components. Functions of core components are the only ones called by the RTOS itself and can be called by applications as well. Furthermore, applications can be extended by adding additional components. In the diagram of Fig.3.3 core components are represented in the inner circle, while optional components are in the outer circle. Components are linked only if required, to keep the system as minimal, light and fast as possible. The version of MQX used is 4.1.0 (the latest available release at the time of the thesis development).

3.2.1 MQX source structure

Full source code for MQX RTOS and communication stacks is available with a commercial-friendly license, that enables developers to edit the sources but to distribute only the resulting binary code. The structure of the source code

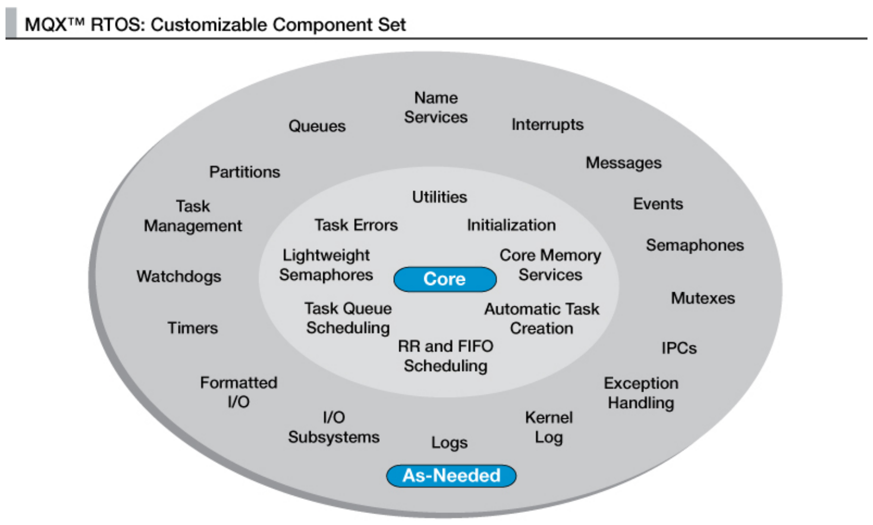


Figure 3.3: Overview of core and optional MQX components.

is rather simple and it's here described:

- **build**: this folder contains the build project for all supported toolchains.
- **config**: this folder holds the configuration files for the OS. Compile-time configuration options for each supported board are in the `config/<board>/user\_config.h` subfolder.
- **doc**: this folder contains complete documentation for the RTOS, including exhaustive explanation of the MCC library.
- **lib**: this folder contains the built libraries for the target, in the corresponding subfolder `lib/<board>`. Built libraries for debug mode (in which compiler optimization are set low to simplify debugging) and for release mode (in which compiler optimization are maximum) are respectively in the `lib/<board>/debug` and `lib/<board>/release` folders.
- **mcc**: this directory contains the source code for MCC libraries and related examples.
- **mqx**: this directory contains the source code for the MQX kernel. The `mqx/examples` folder holds useful examples.
- **tools**: contains additional tools, such as U-Boot images.

MQX for the SABRE board can be built with the Linaro toolchain. Specific scripts to ease the building process are available in the `$MQX_ROOTDIR/build/<board>/make` folder.

3.2.2 MCC - MultiCore Communication

The communication between MQX applications running on the M4 core and Linux applications running on the A9 core is enabled by the MCC (Multi-Core Communication) subsystem. MCC provides a software mechanism to exchange messages between tasks running either on the same core or on different cores in a heterogeneous AMP environment, by using shared RAM and interrupts. The MCC subsystem is designed to be lightweight and fast and all settings (e.g. buffer sizes, number of buffers and maximum number of endpoints) are configurable by the user at build time. It also provides a set of well-documented API functions that allows different send/receive modalities, including copy-less mechanisms[1]. The working principles of MCC are briefly described below.

Endpoints User applications communicate with each other by sending messages to endpoints, which are receive buffer queues in shared RAM. Similarly to the IP protocol each endpoint is addressed by a triplet:

- **Core**: number of the core within the processor (i.e. in the i.MX 6SoloX processor the Cortex-A9 is core 0 while the Cortex-M4 is core 1).

- **Node**: arbitrary number associated to each user process involved in MCC.
  NB: MQX has only one node.

- **Port**: arbitrary value up to a configurable maximum (except for the port 0, which is reserved).

Unlike the IP protocol, there is no procedure to find out the [core, node, port] triplet since it's part of the user's system design and implementation. Here's an example of pseudo-code for the endpoint creation on both sides:

Linux side

```
mcc_create_endpoint([0,0,1], 1) //Cortex-A9 is core 0.
   Linux will send on [1,0,2] and receive on [0,0,1].
```

MQX side

```
mcc_create_endpoint([1,0,2], 2) //Cortex-M4 is core 1.
   MQX is arbitrarly set to be node 0, receiving on
   port 2.
```

[1] A complete documentation (including API reference) for MCC is available at http://cache.freescale.com/files/32bit/doc/user_guide/MQX_MCC_User_Guide.pdf

Send	Receive
Copy mode: data is copied from the sending application's buffer to the MCC buffer in the shared memory.	**Copy mode**: data is copied from the MCC buffer in shared memory to the receiving application's buffer.
Copy-less mode: a pointer to the pre-allocated MCC buffer in shared memory is provided to the sending application. The MCC buffer is directly filled with the data to be sent. The message is then sent to the destination endpoint.	**Copy-less mode**: a pointer to the MCC buffer containing the received data is provided to the receiving application. Received data is directly read from shared memory. Finally, MCC buffer is released.

Table 3.1: MCC send/receive options.

Buffer management Data is transferred between endpoints in fixed size buffers, allocated in one common buffer pool in shared RAM. All buffers in the pool are allocated in the initialization step.
The API supports both send/receive with and without copy. In the first case data is copied from the user application to the buffer and viceversa, respectively in the sending and receiving phase. In the latter case, a pointer to the MCC buffer is provided to the application, both during sending and receiving. If copy-less send/receive mechanism is used, the user must release the MCC buffer once the operation is complete. Table 3.1 summarizes the available send/receive options.
Example of a data transfer between Cortex-A9 and Cortex-M4:

Linux side

```
mcc_send([0,0,1], [1,0,2], "hello", 5, 50) //A message
   "hello" of 5 bytes is sent (with copy) from core 0
   to core 1. A timeout of 50 milliseconds is set, to
   wait for a free buffer.\\
mcc_recv_copy(&src_ep, [0,0,1], &buf, sizeof(buf),
   length, 0xffffffff) //Core 0 receives from endpoint
   '&src_ep'. The received message of 'length' bytes is
   stored in '&buf' of size 'sizeof(buf)'. The last
   parameter indicates to wait forever (blocking call).
```

MQX side

```
mcc_recv_nocopy(&src_ep, [1,0,2], &buf_p, length,
   0xffffffff)//Core 1 waits (forever) for a message of
   'length' bytes, coming from endpoint '&src_ep'. Data
   is directly read from MCC buffer pointed by
   '&buf\_p'.\\
mcc_send([1,0,2], [0,0,1], "hello", 5, 50) //A message
```

```
    "hello" of 5 bytes is sent from core 1 to core 0,
    with a timeout of 50 milliseconds.
```

Shared RAM Shared RAM contains, along with buffers for transferring data between endpoints, internal buffer management data structures such as endpoint and signal queue head and tail pointers. All accesses to the shared memory are synchronized by hardware semaphores. The maximum number of endpoints, the number of data buffers and their size in bytes can be adjusted in the `$MQX_ROOT/mcc/source/include/mcc_config.h` header file.

Signaling Signaling between communicating endpoints is implemented through interrupts. An interrupt is sent from one core to the other in either one of the following cases:

- A buffer has been queued to an endpoint on the interrupted core
- A buffer has been freed by the interrupting core

Each core has its own signal queue and each signal indicates the type of interrupt. The maximum number of outstanding signals queued and waiting to be processed can be configured in the `mcc_config.h` file.

3.2.3 MCC pingpong demo application

The pingpong demo application is a simple demo that exemplify the communication between two endpoints on different cores. In the example, one endpoint is created on the Cortex-A9 core and the other one on the Cortex-M4 core. The application consists of two communicating tasks, a sender on the Cortex-A9 and a receiver on the Cortex-M4. Two initial steps are required on both communication ends: MCC subsystem initialization and MCC version verification (the major version number of the library must be the same on both cores) . Endpoints are then created on each core, by initializing the proper [core, node, port] triplet. The sender task sends a message to the responder, containing an int counter value. The responder task reads the received data, increments the counter and sends the result back to the sender. The two tasks keep exchanging messages in a loop fashion. The schema in Fig.3.4 illustrates the demo's flow of execution.
The MQX application runs on the M4 core, therefore only the responder task is active. The A9 core is responsible for implementing and running the sender task. The code to initialize the mcc communication and launch the main task from Linux on the A9 is located in `$KERNEL_ROOT/drivers/char/imx_amp/imx_mcc_test.c`. The Linux driver will be discussed later in the Kernel porting section.

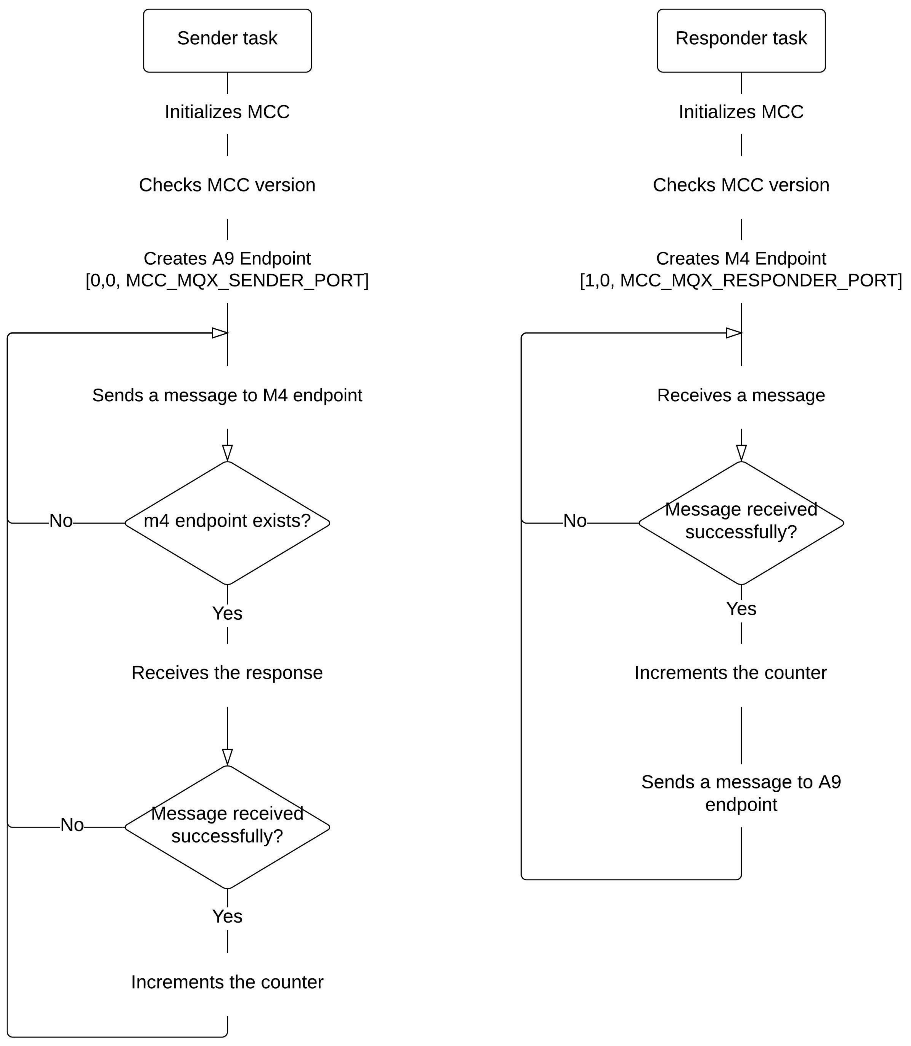

Sender task
Initializes MCC
Checks MCC version
Creates A9 Endpoint [0,0, MCC_MQX_SENDER_PORT]
Sends a message to M4 endpoint
m4 endpoint exists?
No
Yes
Receives the response
Message received successfully?
No
Yes
Increments the counter
Responder task
Initializes MCC
Checks MCC version
Creates M4 Endpoint [1,0, MCC_MQX_RESPONDER_PORT]
Receives a message
Message received successfully?
No
Yes
Increments the counter
Sends a message to A9 endpoint

Figure 3.4: Ping-pong demo application.

3.2.4 Firmware and demo compilation

The MQX firmware and the demo application are built together in a single image, that must be copied in QSPI flash in order for the Cortex-M4 to execute it at boot. Scripts to facilitate the compilation of applications are provided. The building process results in a ELF file that includes both MQX firmware and the application. The file can be converted to a binary file ready to be flashed, by using the Linaro toolchain:

```
$ TOOLCHAIN_ROOTDIR/bin/arm-none-eabi-objcopy \
./gcc_arm/ram_release/pingpong_example_imx6sx_sdb_m4
.elf \
-O binary m4_app.bin
```

The binary file can then be transferred on the SD-card and copied in the QSPI flash by the bootloader.

3.3 Working on the i.MX 6SoloX SABRE board

The work on the SABRE board has been built around the 4.1.13 Linux kernel and MQX 4.1.0 (which were the latest versions at the time of developing this thesis). The steps to port all necessary libraries, to boot the Linux kernel and to test the inter-communication between the two cores of the i.MX 6SoloX processor are here described.

3.3.1 Linux 4.1.13 Kernel on the SABRE board

Support for the i.MX 6SoloX SABRE board is already included in the official Linux 4.1.13 Kernel release. However, the latest vanilla kernel lacks support for the MCC subsystem, even though the proper libraries were included in the older 3.14.38 release. The latest release also does not include some relevant adjustments on the board's device tree file, required to boot the M4 core alongside the A9 core and ensure a peaceful coexistance of the two. The MCC libraries, their dependencies and the required device tree modifications have been integrated in the 4.1.13 kernel.
As with the previous case with the Amber board, a patch for the 4.1.13 Kernel that includes all changes made has been created with the `git diff` command.

Kernel configuration

The default kernel configuration file used to build for the SABRE board is `$KERNEL_ROOT/arch/arm/config/imx_v6_v7_defconfig`. The following flags have been enabled:

- `IMX_MCC_TEST`: this flag enables the support for the MCC ping pong test.
- `IMX_SEMA4`: needed by the `IMX_MCC_TEST` flag.
- `INV_MPU6050_IIO`: this flag enables the support for the Invensense MPU6050 sensor, a gyroscope/accelerometer combo device. The device, together with the integrated MAG3110 sensor, has been used as IMU (Inertial Measurement Unit) and its functioning will be described on the next chapter.

Device Tree

As said before, both the A9 and the M4 core might have access to peripheral blocks in the processor. The decision on which core holds control on a specific device affects the device tree file, used by Linux during the drivers' initialization. A separate dts file has been added to allow the two cores to boot simultaneously, and some additional devices have been added on the main board's dts:

- The `imx6sx-sdb.dts` file has been edited to add support for the PCI EXPRESS interface and for the I^{2}C3 bus. These additions are required for the kernel to recognize and correctly initialize the GY-87 board, which contains the MPU6050 sensor. The GY-87 module has been connected to the SABRE board through the PCIe slot and communication takes place over the I^{2}C3 bus. The correct device driver initialization has been observed from within the Linux kernel, to check whether the GY-87 was correctly switching on and working. These additions on the device tree have been made only to quickly check for the correct functioning of the module; control is then handed over the M4 core as explained on the next point.

- The `imx6sx-sdb-m4.dts` file has been added. The file includes the main `imx6sx-sdb` file, but it disables some peripherals conflicting with the Cortex-M4 core. Among others, the I^{2}C3 device has been disabled since sensors' polling will be ultimately managed by the M4 core and not by the A9 core. The user is free to choose whether to use only the A9 core, thus using the `imx6sx-sdb` dtb, or both the A9 and M4, by using the `imx6sx-sdb-m4` dtb.

MCC libraries

The following files have been ported from the old 3.14.38 kernel to the latest release:

- **MCC header files** The following files contain all shared function declarations and macro definitions for the MCC subsystem and they have been copied to the `$KERNEL_ROOT/include/linux` folder: `imx_sema4.h`, `mcc_api.h`, `mcc_common.h`, `mcc_config_linux.h`, `mcc_imx6sx.h`, `mcc_linux.h`.

- **MCC source files** The following source files contain the implementation of the MCC core functions and APIs and they have been copied in the `$KERNEL_ROOT/arch/arm/mach-imx/` folder: `mcc_api.c`, `mcc_common.c`,`mcc_config.h`,`mcc_imx6sx.c`, `mcc_linux.c`. The `mach-imx` subdirectory contains all source code for cpus of the i.MX family.

- **MCC auxiliary files** The `mu.c` source file, containing auxiliary functions required for the multi-core communication, has been copied to the `$KERNEL_ROOT/arch/arm/mach-imx/` folder. The following files, contained in the same folder, have been modified to add additional functions needed by the MCC subsystem: `clk-imx6sx.c`, `common.h`, `gpc.c`. The added functions mainly concern the M4 core management and status monitoring (e.g. check if M4 is sleeping, set the bus frequency).

- **Makefile** The `$KERNEL_ROOT/arch/arm/mach-imx/Makefile` has been edited: rules to compile the newly added files have been added.

AMP driver porting

The MCC ping pong demo application can be tested through a character device driver, that exchanges messages with the M4 core and prints the result on the screen. A character device driver for this purpose is included within the Linux 3.14.38 kernel files, in the `$KERNEL_ROOT/drivers/char/imx_amp` folder. The directory contains the source code for the ping pong demo, the source code for the MCC tty demo (i.e. same as the ping pong application, with terminal interface) and the source code for the imx_sema4 driver (i.e. mutex implementation used by the MCC subsystem). The `Makefile` to build the source code and the `Kconfig` file to add the proper kernel configuration options are included in the folder as well. These drivers have been ported to the 4.1 Kernel, by copying the entire `imx_amp` folder and by adjusting the following files:

- `$KERNEL_ROOT/drivers/char/Makefile`: a rule to build the `imx_amp` subfolder has been added.

- `$KERNEL_ROOT/drivers/char/Kconfig`: an instruction has been added to source the `imx_amp/Kconfig` file.

In particular, the `imx_mcc_test.c` file contains the implementation of the sender task, responsible for initiating the communication with the responder task on the M4 core in the ping pong demo. The code has been used as a starting point to develop the character device driver for the sensors' data receiving and visualization, as will be described in the next chapter.

3.3.2 Inter-core communication testing

The output of both A9 core and M4 core can be analyzed by connecting the board to the PC through the J-16 Debug port (see Fig.3.1) and by using Kermit (or any other serial communication software). When connecting to the board, two tty devices are shown: the first one handles serial communication with the A9 core, the second one manages the serial communication with the M4 core. By opening one instance of Kermit for each tty device, its possible to interact with both cores independently.

U-Boot configuration

In order to boot the M4 core along with the primary core, some U-Boot environment variables need to be updated. First of all the dtb file to be used by Linux need to be changed to the `imx6sx-sdb-m4.dtb`. Then, the name of the M4 application file has to be set. Finally, the boot command has to be edited to allow the M4 core to be booted right before the A9 core. The following commands show how to properly set the U-Boot environment:

```
=> setenv fdt_file imx6sx-sdb-m4.dtb
=> setenv m4image m4_app.bin
=> setenv mmcargs "${mmcargs} uart_from_osc"
=> setenv bootcmd "run m4boot;${bootcmd}"
=> saveenv
```

The following commands copy the Cortex-M4 application from the SD-Card to the QSPI flash and eventually boot the board:

```
=> run update_m4_from_sd
=> boot
```

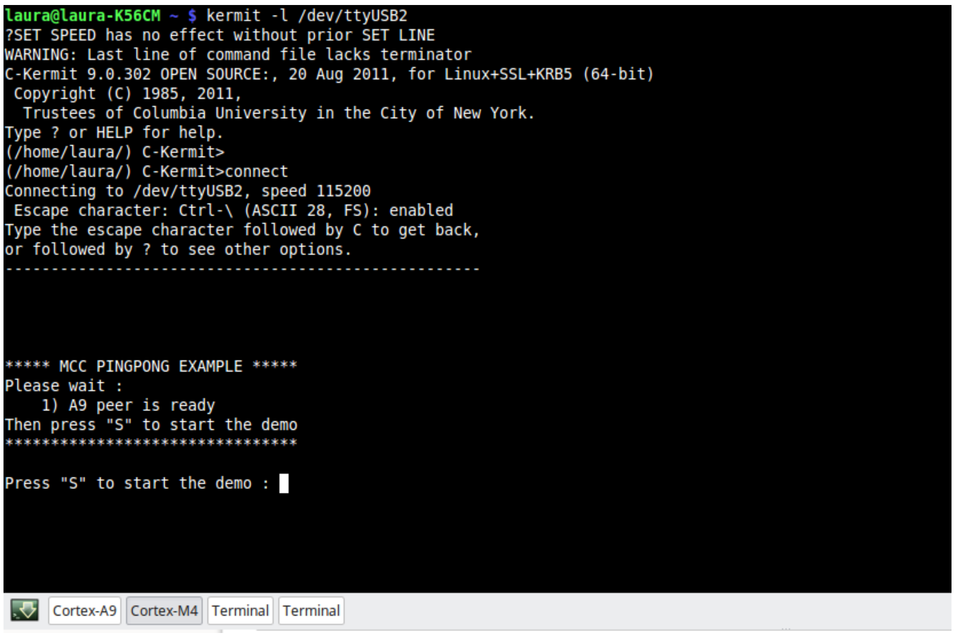
Ping-pong demo testing

As soon as the last U-Boot command is entered, the Cortex-M4 application is loaded and the Linux kernel is simultaneously booted in the Cortex-A9 core. A message is printed on the M4 serial console, suggesting that the demo can be started by pressing ”S” once the A9 peer is ready (Fig.3.5)
Once the Linux boot process on the A9 core finishes, the ping-pong sender task can be started by writing in the file descriptor of the character device that handles the multi-core communication. In fact, the function responsible for initializing MCC and starting the sender task is triggered whenever the user-space writes something in the `pingpong_en` file (`sysfs` is used in the character device driver as interface with the user-space). This can be done by typing the following command on the A9 serial console (after logging in as root):

```
$ echo 1 > /sys/bus/platform/drivers/imx6sx-mcc-test/mcctest.15/pingpong_en
```

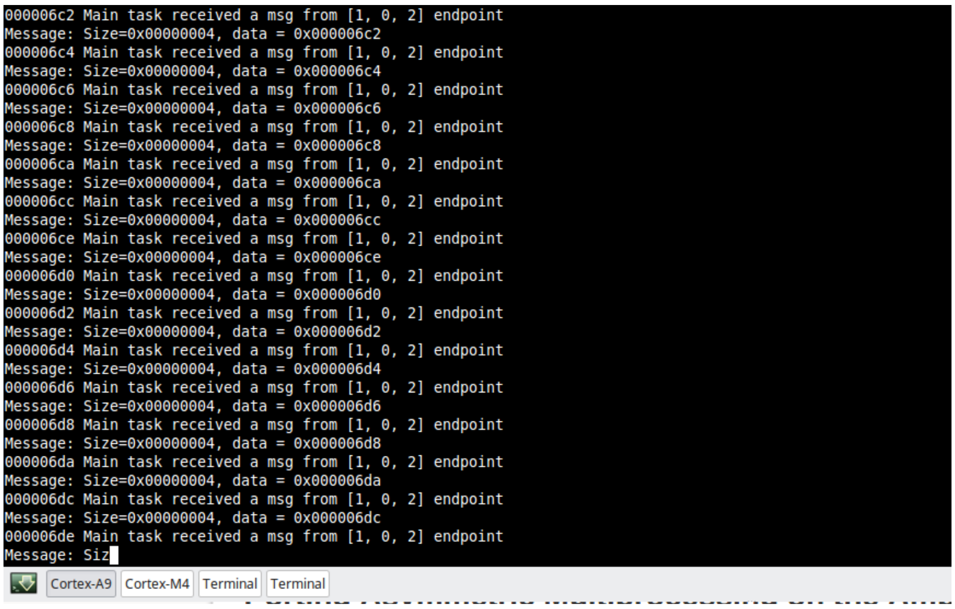
Once the sender task is running and the demo application on the M4 is started, the two tasks begin to exchange data and log messages are printed on both consoles. The endpoint on the A9 (sender task) receives data from the [1,0,2] endpoint on the M4 (Fig.3.6), whereas the endpoint on the M4 (responder task) receives data from the [0,0,1] endpoint on the A9 (Fig.3.7).

While the communication is active, the red led lights on the board associated respectively to the M4 and the A9 are both blinking, showing that both cores are up and running (Fig.3.8).



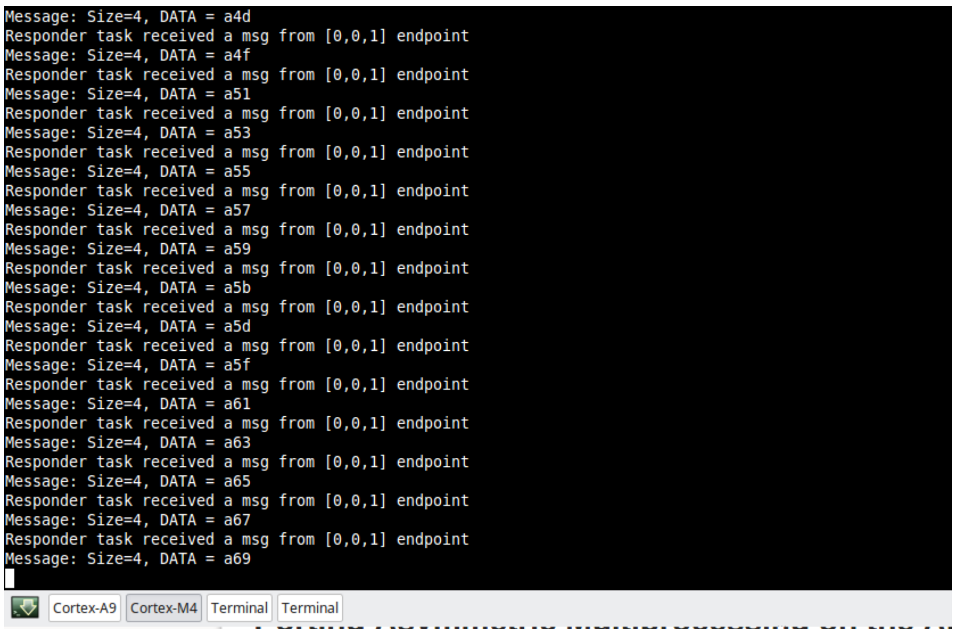
```
laura@laura-K56CM ~ $ kermit -l /dev/ttyUSB2
?SET SPEED has no effect without prior SET LINE
WARNING: Last line of command file lacks terminator
C-Kermit 9.0.302 OPEN SOURCE:, 20 Aug 2011, for Linux+SSL+KRB5 (64-bit)
 Copyright (C) 1985, 2011,
  Trustees of Columbia University in the City of New York.
Type ? or HELP for help.
(/home/laura/) C-Kermit>
(/home/laura/) C-Kermit>connect
Connecting to /dev/ttyUSB2, speed 115200
 Escape character: Ctrl-\ (ASCII 28, FS): enabled
Type the escape character followed by C to get back,
or followed by ? to see other options.
----------------------------------------------------



***** MCC PINGPONG EXAMPLE *****
Please wait :
    1) A9 peer is ready
Then press "S" to start the demo
********************************

Press "S" to start the demo :
```

Figure 3.5: Ping-pong demo application on Cortex-M4.



```
000006c2 Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006c2
000006c4 Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006c4
000006c6 Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006c6
000006c8 Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006c8
000006ca Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006ca
000006cc Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006cc
000006ce Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006ce
000006d0 Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006d0
000006d2 Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006d2
000006d4 Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006d4
000006d6 Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006d6
000006d8 Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006d8
000006da Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006da
000006dc Main task received a msg from [1, 0, 2] endpoint
Message: Size=0x00000004, data = 0x000006dc
000006de Main task received a msg from [1, 0, 2] endpoint
Message: Siz
```

Figure 3.6: Sender task on Cortex-A9.



```
Message: Size=4, DATA = a4d
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a4f
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a51
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a53
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a55
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a57
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a59
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a5b
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a5d
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a5f
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a61
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a63
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a65
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a67
Responder task received a msg from [0,0,1] endpoint
Message: Size=4, DATA = a69
```

Figure 3.7: Receiver task on Cortex-M4.

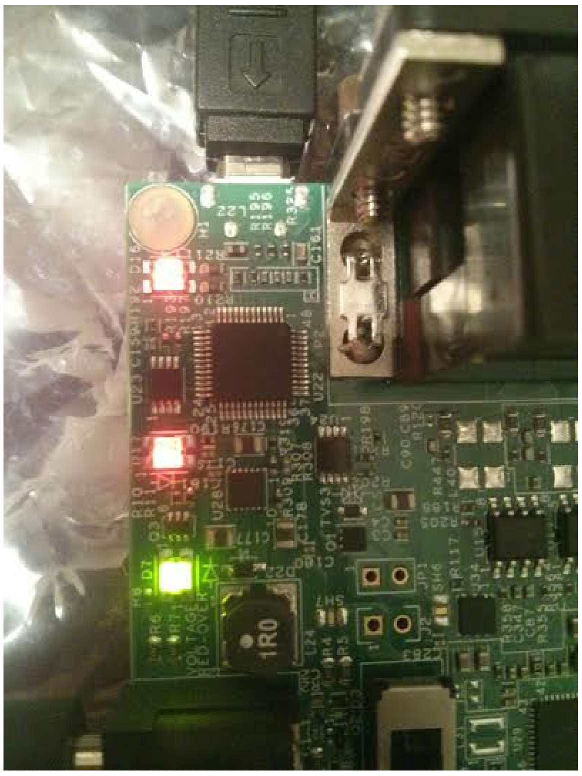


Figure 3.8: Led lights on the board showing that both cores are running.

Chapter 4

Application in a drone navigation system

Unmanned Aerial Vehicles Over the last few years there have been a significant growth in the manufacture and sales of Unmanned Aerial Vehicles (UAV), often reffered to as drones, quadcopters or multicopters. This new breed of radio controlled vehicles is rapidly gaining in popularity and has applications in the military, civil and commercial fields.
Military use of drones mainly involves the ISR field (i.e. Intelligence, Survaillance, Reconaissance). By being lightweight and flexible, quadcopters are excellent ISR assets that can wirelessly transmit critical live video and data from multiple sensors directly to a ground control unit. The versatility, the ease of use and the broad variety of models, ranging from minimal low cost solutions to fully equipped high-end options, makes drones affordable and interesting from a civilian perspective too. The quadcopter has a simple structure with very few moving parts and it's relatively easy to assemble. In its simplest form it is made up by four motors and four propellers on a lightweight frame (usually in light wood, carbon fibre or fibreglass) equipped with a small control board connected to a Lipo battery. Various experimentations and the improvement over the years of microprocessors, batteries and accelerometer/gyroscope technologies have led to a wide spectre of designs and variations with different amounts of arms, including tricopters, hexacopters and octocopters.

IMU The control board is equipped with an embedded stabilization system, made up of sensor components that generates information about the different mechanical phenomenon involving the drone, such as acceleration, tilt, orientation in space, angular velocity, pitch and rotation. A device that hosts multiple types of sensors is reffered to as an Inertial Measurement Unit (IMU). An IMU measures accelerations, rotation rates and possibly earth's magnetic field to determine the quadcopter's orientation. Usually the term

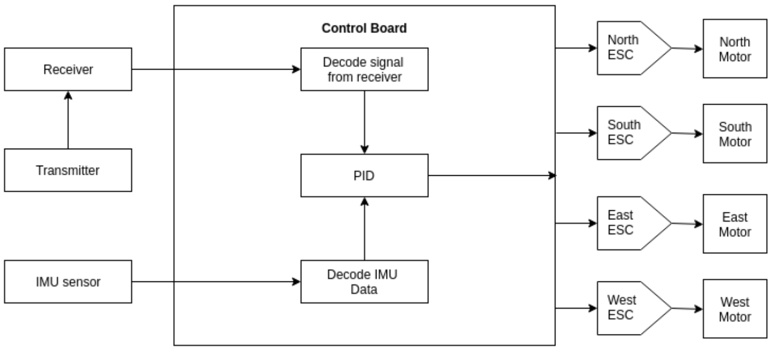


Figure 4.1: Diagram illustrating a quadcopter's flight control system.

IMU refers to a sensor that incorporates a tri-axis accelerometer and a tri-axis gyroscope; the term MARG (Magnetic, Angular Rate, and Gravity) refers to a sensor device that includes also a tri-axis magnetometer. An IMU alone can measure the body's attitude relative to the direction of gravity; MARG systems, also known as AHRS (Attitude and Heading Reference Systems) can measure the attitude relative to both the direction of gravity and the earth's magnetic field, thus computing a complete and more precise evaluation. The control board continuously takes readings from the IMU, decodes the commands coming from the pilot, processes the information and computes the PID values to control the motors' speed and mantain the body's balance. A quadcopter's flight controller system is shown in Fig.4.1; the output signals generated by the control board are fed to the ESCs (i.e. Electronic Speed Controller), that provide signals to the motors.
Three sensors are at the root of a drone's control and autonomous flight: the accelerometer, which measures acceleration and tilt, the gyroscope, which measures angular velocity and orientation and the magnetometer, which measures gravitational forces (Fig.4.2).

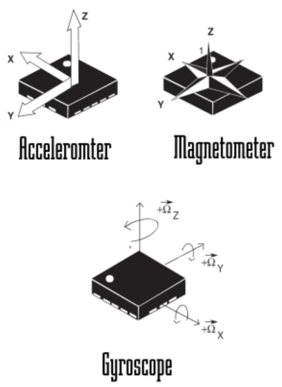


Figure 4.2: Common sensors used in a IMU.

The control software is responsible for:

- Balancing the drone by using IMU sensor's data
- Interpreting the pilot's commands and converting them in signals for the motors
- Verifying the correct functioning of the components
- Sending data to the ground control station

The integration of data coming from the different sensor sources is known as **multisensor data fusing** and it is a wide ranging subject with many areas of application[9]. Several approaches have been proposed, among which the Madgwick algorithm and the Mahony algorithm that will be described later.

Flight control board A flight controller for a multi-rotor UAV is usually made up of a microcontroller, sensors and input/output pins. The microcontroller either runs an RTOS or some low-level code with interrupts to guarantee low latency on critical sections of the code, such as IMU data gathering and PID running.
Even if the most common tendency so far is to use a single microcontroller unit, multiprocessor solutions are recently being taken into consideration. Furthermore, there has been growing interest in the use of Linux SoCs as flight control boards[8]. Running Linux on a drone highly improves its capability to run autonomous operations and higher-level computations. Moreover, involving Linux opens up to new possibilities for ground-to-drone communication, by allowing wireless control over SSH instead of traditional radio-control. The drawback of using a SoC running an OS such as Linux is that it needs to be properly tweaked and optimized to ensure that real-time needs of critical operations are consistently met. Linux can be turned into a real-time system by applying the PREEMPT_RT patch, that removes all unbounded latencies from the kernel. An heterogeneous asymmetric multiprocessing approach combines the best of both microcontroller and microprocessor solutions, by joining a MCU for dealing with low-latency tasks (e.g. IMU interfacing, PID management, interpreting manual flight controls) and a MPU for higher-level tasks (e.g. navigation, computer vision, sending data to the ground station) (Fig.4.3). Having the two combined on the same chip highly increases the drone's ability of running fully autonomous tasks. Finally, splitting the tasks across multiple dedicated cores and delegating all repetitive operations to a microcontroller running an RTOS is indeed a much powerful and appealing solution if compared to the use of patched Linux alone. Inter-core communication completes the picture by enabling cooperation and message exchanging between the cores. The aim of this thesis is to study the possibility to use the i.MX 6SoloX SABRE

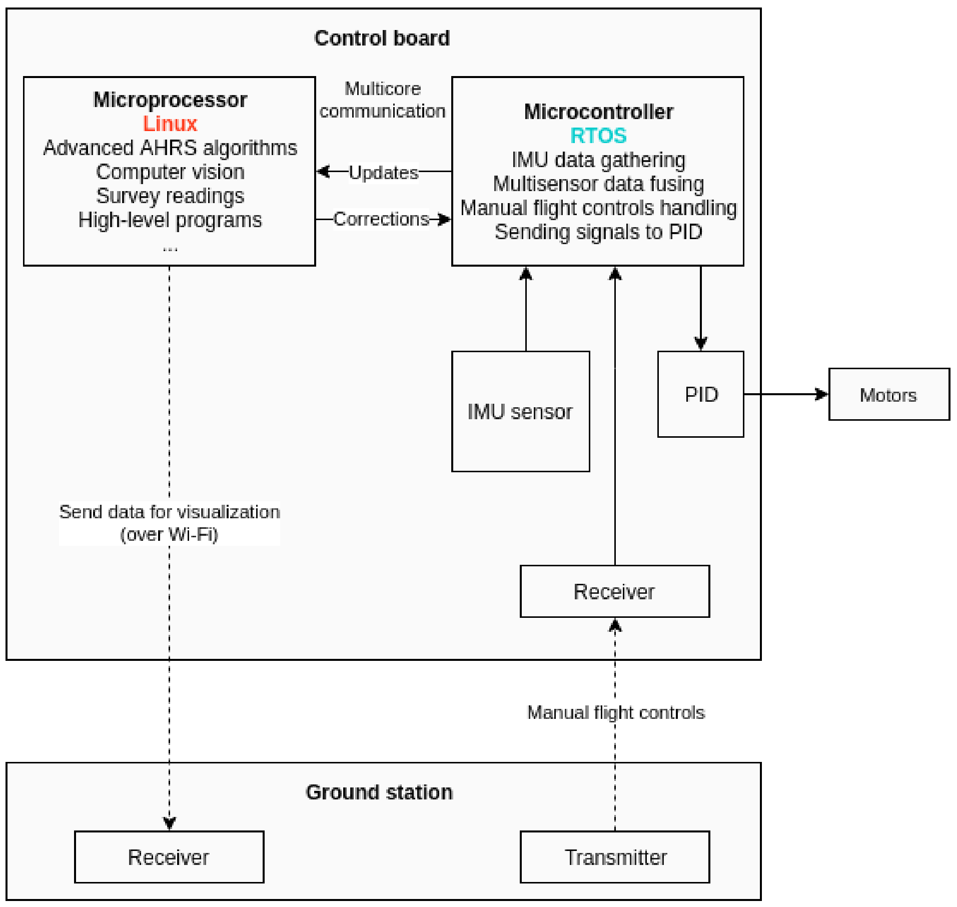


Figure 4.3: Diagram of a quadcopter's flight system involving both a MCU and a MPU.

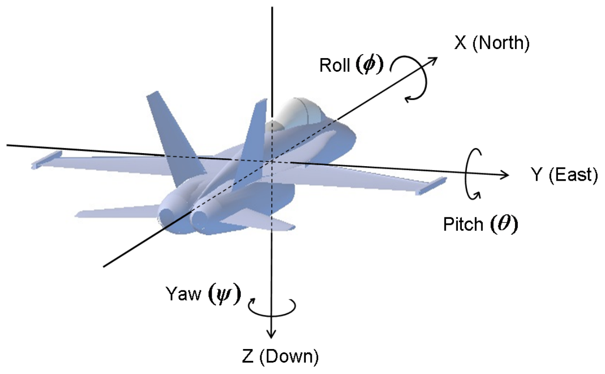


Figure 4.4: Illustration of roll, pitch and yaw rotations.

as a flight control board and to prove the exploitability of its asymmetric multi-processing architecture from a drone-oriented perspective.

Quadcopter's dynamics The dynamics of a UAV, assumed as a rigid body, can be described either through the Euler parametrization or by means of quaternions. Although Euler angles are more intuitive and easier to develop and visualize, quaternions represent a more efficient method for aircraft dynamics modelling since the orientation in space can be characterized by a single rotation around an axis.

Euler angles Euler angles provide a way to describe the 3D orientation of a body using a combination of three rotations about different axes: yaw, pitch and roll (Fig.4.4)[6].
Multiple coordinate frames are used to describe the orientation of the sensor. The Euler parametrization suffers from a phenomenon called "gimbal lock", which prevents the orientation of the body to be uniquely represented by Euler angles in certain circumstances. In fact when the pitch angle approaches 90 degrees (i.e. when the body is vertical) the movement of the sensor caused by yaw and roll is exactly the same. Therefore, one degree of freedom is lost and it is impossible to determine unambiguously the body's orientation (Fig.4.5).

Quaternions Quaternions provide an alternative method that avoids gimbal lock and allows better performance in numerical implementation. A quaternion is a four-dimensional complex number, composed of a real part and three imaginary parts and it can be used to represent any rotation in 3D space. A quaternion can be formalized as:

$$q = s + xi + yj + zk \quad s, x, y, z \in \mathbb{R}$$

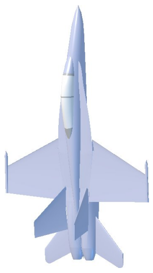


Figure 4.5: Body's position in which gimbal lock occurs.

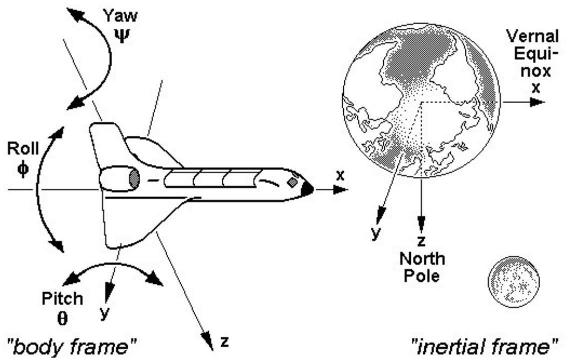


Figure 4.6: Inertial frame and body frame.

where

$$i^2 = j^2 = k^2 = ijk = -1$$

and

$$ij = k \quad jk = i \quad ki = j$$

$$ji = -k \quad kj = -i \quad ik = -j$$

A quaternion represents the rotation from a Earth-fixed reference coordinate frame, the so-called "inertial frame", to the "body frame" aligned with the sensor (Fig.4.6). Even if this representation is less intuitive, one rotation is enough to describe the body's orientation, unlike with Euler representation. The first element of a quaternion is the "scalar part" and specifies the amount of rotation that should be performed. The other elements represent a vector about which rotation should be performed. Both sensor fusing algorithms described below use quaternions to determine the drones orientation relative to the reference coordinate frame.

4.1 Mahony's algorithm

The purpose of an orientation filter is to optimally join gyroscope, accelerometer and magnetometer measurements to estimate a body's orientation. Robert Mahony theorized a solution to obtain good estimates from low-cost IMUs, affected by high noise levels and distortion[3]. The complementary filter proposed by Mahony addresses the problem of merging gyroscope and accelerometer data, in spite of the different nature of the rotation rate vector and the acceleration vector. It is also an alternative to the non-linear Kalman filters, that are more power-demanding and therefore less suitable for small low-cost UAVs. The idea behind Mahony's approach is to correct the rotation rate vector, computed from the gyro data, by an amount determined by a Proportional-Integral controller. The correction vector is calculated by means of the previously estimated attitude and acceleration. The main steps of the algorithms can be simplified as:

- compute the rotation rate vector and the acceleration vector, by gathering data respectively from the gyroscope and the accelerometer.
- compute the gravity vector, starting from the current orientation estimation
- calculate the error vector
- calculate the rotation rate correction, by using the previous attitude estimation and acceleration vector
- apply the correction and recalculate the rotation rate vector
- integrate the rate of change to evaluate the orientation
- repeat the procedure

Mahony's filter, originally evolving on the special orthogonal group SO(3), has been implemented in quaternion form by Sebastian Madgwick.

4.2 Madgwick's algorithm

Madgwick presented an alternative solution that also employs a quaternion representation of orientation and that can be applied to both IMU and MARG sensor devices[5]. The algorithm first computes the orientation from angular rate, measured by the gyroscope, then it computes orientation from gravity and earth's magnetic field, measured respectively by the magnetometer and the accelerometer. By fusing the two computed quaternions, an estimated orientation of the body's frame relative to the inertial frame is obtained.
Starting from the tri-axis gyroscope output, that measures the angular rate about the x,y and z axes of the sensor frame, the quaternion derivative that represents the rate of change of orientation of the inertial frame relative to the sensor frame can be computed. The quaternion describing the orientation of the inertial frame relative to the sensor's frame can be obtained by numerically integrating the quaternion derivative and by taking into account the previous estimate of orientation. The tri-axis accelerometer outputs the magnitude and the direction of the field of gravity in the sensor frame, combined with linear accelerations caused by the motion of the sensor. The tri-axis magnetometer outputs the magnitude and direction of the earth's magnetic field in the sensor frame, together with the local magnetic field and distortions. The measurement of the earth's magnetic field alone do not provide a unique evaluation of the orientation of the sensor. The measures, along with the reference directions of both fields, must be combined to obtain a unique solution. Finally, the estimated orientation of the sensor is obtained by merging the orientation measures computed so far.
The presence of ferromagnetic elements in proximity of the magnetometer can affect the earth's magnetic field measure. Madgwick's algorithm includes a normalization step to compensate for magnetic distortion. An orientation filter must also account for gyroscope zero bias drifting, due to temperature variations and motion. In Madgwick's approach, based on the filter theorized by Mahony, gyroscope bias drift is balanced through the integral feedback of the error in the rate of change of orientation.
The open source code for Madgwick's implementation of Mahony's complementary filter in C is available and downloadable for free[1].

[1]The code is hosted at http://www.x-io.co.uk/open-source-imu-and-ahrs-algorithms/

4.3 Demo application overview

A demo application for the i.MX 6SoloX SABRE has been developed to prove the usability of the board as a flight control system. The application consists of a MQX application running on the Cortex-M4 and a Linux user-space application running on the Cortex-A9; inter-core communication is handled by the MCC subsystem. The application on the Cortex-M4 side polls the IMU sensor and feeds the data to the Mahony and Madgwick filters. Samples of the quaternions obtained are then periodically sent to the Cortex-A9 for visualization. A second task on the M4 application, distinct from the one polling the sensors, keeps track of the elapsed time and periodically sends an average evaluation of the quaternions obtained so far to the Cortex-A9. The Linux counterpart on the Cortex-A9 is responsible for receiving the messages from the Cortex-M4 and displaying the results.
Ideally the Cortex-M4 would use the information on the sensor's orientation expressed by the quaternions to keep the quadcopter stable and balanced (by interpreting manual flight commands and by sending signal to a PID controlling the motors), whereas the Cortex-A9 would use data in higher-level programs for navigation, send corrections back to the M4 and wirelessly send data to the ground station for visualization. The demo partially implements this approach and shows a possible method for exploiting the asymmetric architecture of the board.
The developing of the application involved the following aspects:

- Writing the MQX driver for the MPU6050 sensor
- Integrating the Mahony and Madgwick filter implementations
- Using MCC for inter-core communication
- Writing a character device driver in Linux kernel space
- Writing a Linux user space application interfacing with the character device

Fig.4.7 illustrates the structure of the application on both M4 and A9 side.

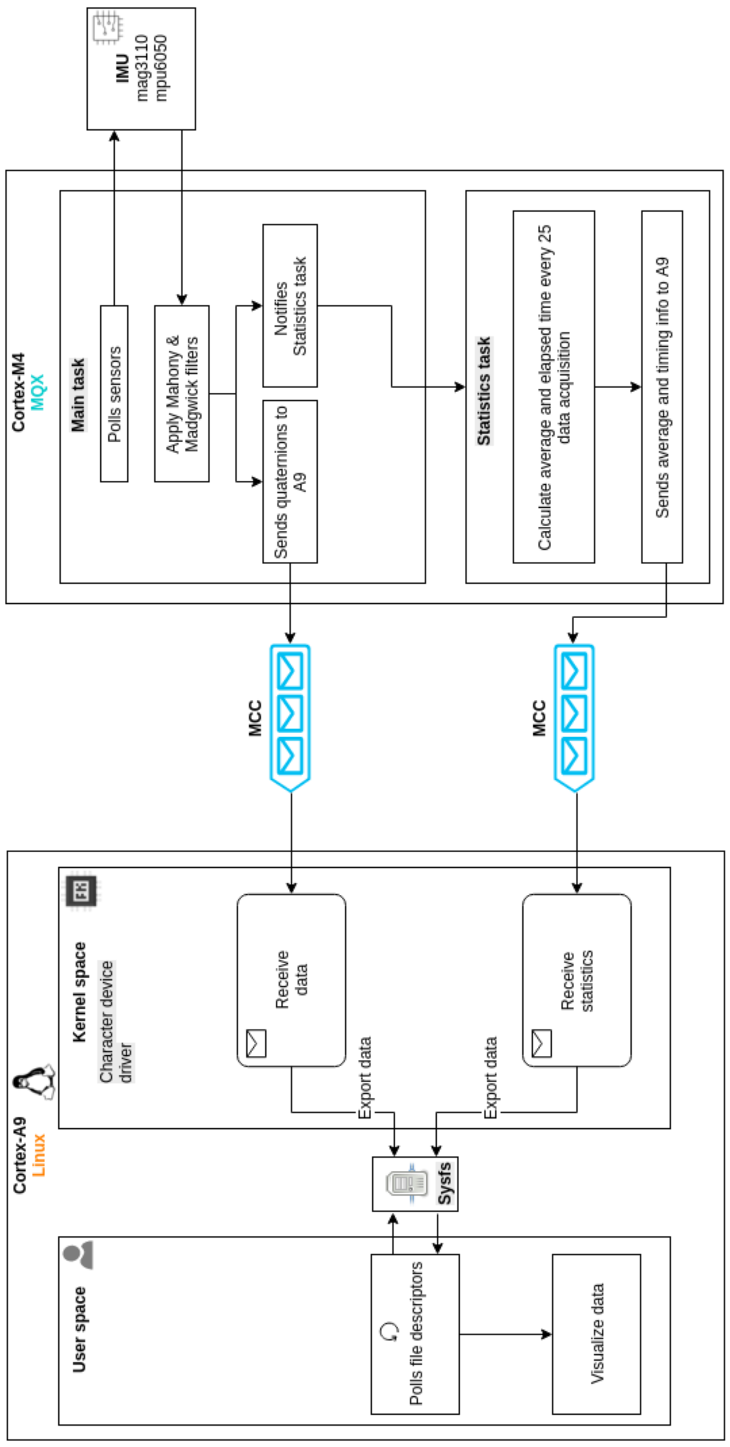


Figure 4.7: Structure of the demo application.

4.4 MQX application

The MQX application on the Cortex-M4 side is responsible for gathering data from sensors, applying the orientation filters to obtain the quaternions and sending periodic updates to the Cortex-A9, along with occasional statistics (e.g. the mean of the quaternions obtained so far). In the hypothesis of using the board as a flight control system integrated in a quadcopter, the board would be connected to a PID controlling the motors. In this scenario, the Cortex-M4 handles the drone's stabilization and therefore needs samples from the sensors as frequently as possible, in order to continuously evaluate the orientation and keep it balanced. The MPU can process the received data for further analysis on the orientation or it can graphically visualize the drone's attitude. In both cases the Cortex-A9 doesn't need to get all single data acquisitions; receiving periodic samples is enough for the purpose. The average of quaternion values can be received with a lower rate too and it can be used, for example, as input for autonomous navigation algorithms.
MQX guarantees for real-time responsiveness and the sensors' polling phase has been kept as straightforward as possible: raw data is read directly from sensors over I^2C without interrupts. As said before, the typical sensors of an inertial measurement unit are a gyroscope, an accelerometer and, optionally, a magnetometer. The i.MX 6SoloX SABRE board encompasses the MAG3110 sensor, a tri-axis magnetic sensor that features a standard I^2C serial interface. The board natively includes an accelerometer sensor as well (the MMA8451q) but doesn't have a gyroscope. The GY-87 IMU board has been chosen instead. The breakout board features the MPU6050 sensor, that includes a 3-axis gyroscope and a 3-axis accelerometer. MQX already includes the driver for the MAG3110; the driver for the MPU6050 has been developed separately, as described below.

4.4.1 The MAG3110 sensor

The MAG3110 is a low-power digital 3D magnetic sensor. It measures the components of the local magnetic field, the sum of the geomagnetic field and the magnetic field created by components on the circuit board. The sensor features a standard I^2C serial interface output and smart embedded functions. Being the i.MX 6SoloX SABRE supported by MQX and being the MAG3110 integrated on the board, the RTOS already includes the driver and a demo application testing the sensor functionalities. The `$MQX_ROOT/mqx/source/io/sensor/mag3110` folder contains the driver's source code, structured as follows:

- `mag3110_reg.h`: header file containing macro definitions for all MAG3110 registers
- `mag3110_prv.h`: header file containing private struct definitions

- `mag3110_basic.h` and `mag3110_basic.c`: declaration and implementation of basic IO functions, such as single register read/write, sensor initialization/de-initialization and slave address setting.

- `mag3110_fun.h` and `mag3110_fun.h`: declaration and implementation of functional level IO functions, such as sensor status retrieval, data acquisition and sensor operating mode setting.

An example application that shows how to work with the MAG3110 component and how to use API functions is included in the `$MQX_ROOT/mqx/examples/sensor/mag3110` folder. This demo has been run on the board to verify the correct functioning of the magnetometer and the source code has been used as reference for the MPU6050 driver development.

4.4.2 The MPU6050 sensor

The GY-87 is a common 10 degree of freedom IMU board (Fig.4.8), featuring the Invensense MPU6050 motion sensor module, the HMC5883L three-axis magnetic field module and the BMP180 barometer pressure sensor. All sensors support I^{2}C connectivity. The MPU6050, in particular, combines a 3-axis gyroscope ranging from +250 to 2000 °/s with a 3-axis accelerometer with range from ±2g to ±16g.

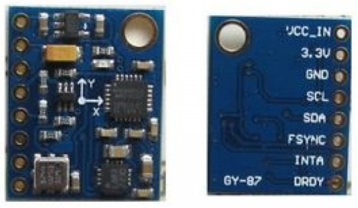


Figure 4.8: Front and back view of the GY-87 IMU board.

By observing the PCI-Express connector pin-out table on the i.MX 6SoloX SABRE board's datasheet, the pin numbers for 3.3V voltage, ground, I^{2}C clock and data have been individuated. The GY-87 module has been soldered to a prototyping PCIe extender board, according to the pin numbers previously detected on the SABRE board. The extender has then been connected to the SABRE board through the PCI-Express socket. The correct functioning of the GY-87 module has been tested by booting Linux on the SABRE and verifying on the kernel log the proper sensor's detection and initialization.

MPU6050 MQX driver

The MQX driver for the MPU6050 sensor has been developed from scratch, by using the existing drivers for Arduino[2] and for ERIKA RTOS as reference. The MQX driver's code has been structured in the wake of the MAG3110 driver:

- `mpu6050_reg.h`: header file containing macro definitions for all sensor's registers.
- `mpu6050_prv.h`: header file containing a private struct definition. The struct here defined collects the MPU6050's device information, limited to a I^{2}C device handler pointer and the slave address.
- `mpu6050_basic.h` and `mag3110_basic.c`: declaration and definition of basic IO functions, such as single register read/write, sensor's initialization/de-initialization, slave address setting. A function for writing a single bit has been added, to semplify bit flag operations on certain registers.
- `mag6050_fun.h` and `mag6050_fun.h`: declaration and definition of higher-level IO functions, such as gyroscope's and accelerometer's range setting and data reading.

By observing the MPU6050 datasheet and register map, the relevant registers for the initialization of the device and for data acquisition have been identified. The device's initialization phase involves enabling bit flags of specific registers to execute the following actions: resetting the sensor, switching on the sensor, setting the gyroscope on "continuous update" mode (i.e. the sample rate divider is set to zero, in order to get data at the gyroscope's maximum output rate), disabling interrupts.

4.4.3 Main task

The MQX demo application consists of two asynchronous tasks. The main task is responsible for collecting data from the sensors, aggregating and fusing data through the orientation filters and sending samples of the computed quaternions to the Cortex-A9 core. The main task also notifies the other task whenever a new sample of quaternions has been computed. The flow of execution of the main task is here described.

Main task initialization The task must be listed in the specific `TASK_TEMPLATE_STRUCT` structure, along with required initialization parameters. The main task is declared as an auto-start task, meaning that it is

[2]The MPU6050 driver for Arduino is available at http://www.control.aau.dk/ jdn/edu/doc/datasheets/MPU6050/

created as soon as MQX starts. Task priority, stack size, name of the task, creation parameter and time slice are also set in the template.

I^2C driver initialization The I^2C driver is initialized and the file descriptor for the I^2C device is opened. The microcontroller is set as the I^2C master, whereas the sensors are the slaves (Fig.4.9). The file descriptor of the I^2C device is passed as a parameter to the functions responsible for the sensors' initialization.

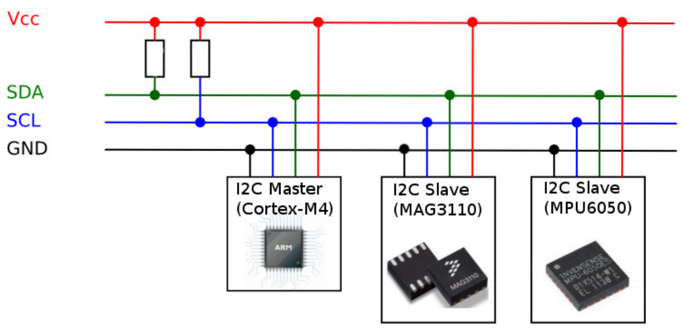


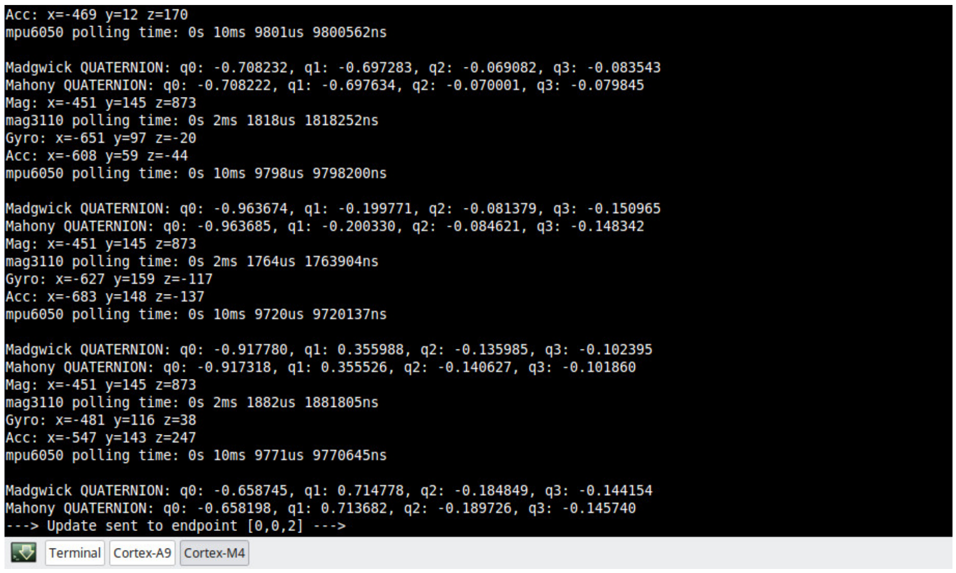
Figure 4.9: Overview of the I^2C bus.

MAG3110 and MPU6050 initialization Both devices are initialized, according to the drivers' init functions. MPU6050 initialization struct sets the device's slave address only, while MAG3110 init struct includes also some additional settings such the sample rate, the sample ratio, the burst read mode.

MCC initialization The MCC subsystem is initialized and two endpoints are created: the first one is used by the main task, the latter is used by the statistics task (described later). The two endpoints are used to send respectively quaternion samples and statistics to the Cortex-A9 core; they share the same core number and node but they use different ports.

Data acquisition Once the initialization phase is complete, the statistics task is created and the MAG3110 and MPU6050 sensor are constantly polled in a loop. If the `TIME_DEBUG` option is enabled, the polling time of each sensor is measured and printed on the console for debugging (Fig.4.10). Polling time measurements have been verified to be consistent with what expected from a real-time system getting raw data from the sensors. The average time to get data from the magnetometer is approximately of 2 ms, while the average time to get the data from gyroscope and accelerometer together is approximately of 10 ms.

Mahony's and Madgwick's filter The C source code for both Mahony's and Madgwick's algorithm has been integrated within the application's folder. The open-source code provided by Madgwick is already optimised to reduce the number of arithmetic operations and therefore to suit low power and low cost hardware.
Sensors' measurements are fed to the Mahony's and Madgwick's filters, which output the quaternions currently representing the sensor's orientation. The quaternion values are printed on the console, for debugging and for later consistency checking with the data received by the Cortex-A9 (Fig.4.10). Quaternion values computed by the two filters are very similar and they change consistently with the motion of the sensor. The Madgwick's algorithm takes up more memory space and requires more arithmetic operations, due to the magnetic distortion and gyroscope's bias drift compensation features[5]. However, the algorithm's overhead doesn't affects the overall performance of the application.



```
Acc: x=-469 y=12 z=170
mpu6050 polling time: 0s 10ms 9801us 9800562ns

Madgwick QUATERNION: q0: -0.708232, q1: -0.697283, q2: -0.069082, q3: -0.083543
Mahony QUATERNION: q0: -0.708222, q1: -0.697634, q2: -0.070001, q3: -0.079845
Mag: x=-451 y=145 z=873
mag3110 polling time: 0s 2ms 1818us 1818252ns
Gyro: x=-651 y=97 z=-20
Acc: x=-608 y=59 z=-44
mpu6050 polling time: 0s 10ms 9798us 9798200ns

Madgwick QUATERNION: q0: -0.963674, q1: -0.199771, q2: -0.081379, q3: -0.150965
Mahony QUATERNION: q0: -0.963685, q1: -0.200330, q2: -0.084621, q3: -0.148342
Mag: x=-451 y=145 z=873
mag3110 polling time: 0s 2ms 1764us 1763904ns
Gyro: x=-627 y=159 z=-117
Acc: x=-683 y=148 z=-137
mpu6050 polling time: 0s 10ms 9720us 9720137ns

Madgwick QUATERNION: q0: -0.917780, q1: 0.355988, q2: -0.135985, q3: -0.102395
Mahony QUATERNION: q0: -0.917318, q1: 0.355526, q2: -0.140627, q3: -0.101860
Mag: x=-451 y=145 z=873
mag3110 polling time: 0s 2ms 1882us 1881805ns
Gyro: x=-481 y=116 z=38
Acc: x=-547 y=143 z=247
mpu6050 polling time: 0s 10ms 9771us 9770645ns

Madgwick QUATERNION: q0: -0.658745, q1: 0.714778, q2: -0.184849, q3: -0.144154
Mahony QUATERNION: q0: -0.658198, q1: 0.713682, q2: -0.189726, q3: -0.145740
---> Update sent to endpoint [0,0,2] --->
```

Figure 4.10: Example of data samples printed on the M4 console.

The quaternion values are stored in two matrices, for further processing from the statistics task. Whenever a new pair of quaternions is stored in the matrices, the main task sends an event to the statistics task. MQX implements events as optional components and provide simple APIs to employ them for task synchronizing and conveying simple information in form of bit-state changes.

Sending data to Cortex-A9 The main task periodically sends updates to the Cortex-A9 core through the MCC subsystem; the sending frequency is set to 1 update every 10 acquisitions but it can be adjusted according to needs. The quaternion's components are computed by the orientation filters as floating point numbers. Not all processors include a FPU and, in general, it is not recommended to deal with floating point calculations within the kernel space. To avoid declaring floating points in the Linux device driver receiving the data from the Cortex-M4, the quaternion values have been stored as unsigned 32-bit integers by using the `union` C data type. This special data type allows to store different data types in the same memory location, providing an efficient way of using a same location for multiple purposes. Therefore, the C structure that represents the message to be sent consists of eight unsigned 32-bit integers, each representing a component of the quaternions computed by the two orientation filters. The message is sent to the proper endpoint on the Cortex-A9; a specific option in the header file of the application specifies whether to use non-copy send mode or copy send mode.

4.4.4 Statistics task

The statistics task is responsible for keeping track of the Cortex-M4 uptime, calculating the average quaternion values in a certain time window and sending information to the Cortex-A9. The functioning of the task is here described.

Statistics task initialization The statistics task is listed in the `TASK_TEMPLATE_STRUCT` structure, along with the main task. The statistics task is created by the main task previous to starting the data acquisition, so it's not declared as an auto-start task. All the task characteristics are set in the template as well.

Event loop The statistics task waits for a certain event bit in a specific event group to be enabled. In order to do so, it sets up an event group and it waits in a spin loop for the main task to set the event bit. As described before, the main task notifies the statistics task whenever a new pair of quaternions is calculated and stored, by opening the connection to the event group and setting the event bit. The statistics task uses an integer counter to keep track of the number of events received.

Time tracking and average calculation Everytime that the statistics task receives a specified number of events (frequency can be easily changed), the elapsed time since the Cortex-M4 started is measured. The average value of the quaternions computed so far, stored in the specific matrices, is also calculated.

Sending data to Cortex-A9 The statistics are then sent to the Cortex-A9 through the MCC subsystem. Each message contains the elapsed time (seconds and milliseconds) and the quaternions' components, stored as unsigned 32-bit integers.

4.5 Linux application

A character device driver and a user space application have been developed to handle the MCC communication on the Linux side and visualize the data received from the Cortex-M4.

4.5.1 Sysfs

Sysfs is a virtual ram-based file system, used to export system information from the kernel space to the user space for specific devices[4]. It is part of the kernel infrastructure and it provides a clean, well-documented programming interface. By providing virtual files, sysfs makes information about kernel sub-systems, hardware devices and device drivers accessible to the user space. The filesystem provides two interfaces: a kernel interface to export the kernel objects, their attributes and the relationships between them and a user interface to view and manipulate these items. User space programs can also send values to the kernel subsystem through sysfs, to control the internal settings. The hierarchical structure of the filesystem follows the internal organization of kernel data structures.
A relation between the kernel constructs and their external user space representation exists: kernel objects are mapped to directories, object attributes are mapped to regular files and object relationships are mapped to symbolic links. The core components of sysfs are here described.

Kobjects Sysfs is intrinsecally tied to the kobject abstraction. A kobject (kernel object) is an object of type `struct kobject`. For each kernel object registered within the system, a directory is created in sysfs. The directory exposes internal object hierarchies to the user space. Kobjects have a name, which corresponds to the name of the folders matching that object in sysfs, a reference count, incremented whenever a kernel module refers to that object, and a parent pointer, that allows the object to be arranged in a hierarchical structure.

Attributes and groups Attributes of kernel objects are exposed in form of regular files in the filesystem. Sysfs maps file read/write operations to functions defined for the attributes, providing the user space a way to interact with kernel space and read/write kernel attributes. To ensure that the exported information is as easily accessible as possible, it is recommended to create attributes in form of simple ASCII text files. Each attribute should consist of only one value per file or, at most of an array of homogeneous values.
When declaring an attribute, the `show()` and `store()` functions must be specified. Each read operation on a file is forwarded to the `show()` method of the corresponding attribute, whereas every write operation on the file is

mapped to the `store()` attribute's method. These functions usually consist of only three parameters: an object, an attribute and a buffer. When a file is opened, sysfs allocates a buffer for transferring data from kernel space to user space. Whenever a read is invoked, the buffer is passed to the `show()` function, which fills it up with the formatted data. Data is then copied from the buffer to user space. When a write operation is invoked, the user space data is copied to the kernel buffer. The buffer and its size are then passed to the `store()` method, which parses the data.
The driver model facilitates the definition of device attributes by defining the following helper:

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
struct device_attribute dev_attr_##_name = __ATTR(_name,
    _mode, _show, _store)
```

The `_mode` field refers to the type of permission associated with the attribute (e.g. `S_IWUSR` simbolizes the write permission, `S_IRUSR` stands for the read permission).
If a kernel object has multiple attributes, they can be gathered in an array. The attribute group interface simplifies handling multiple attributes, by adding/removing the whole set instead of the functions for each individual item.

Notify user space If a user space process is waiting for updates on a certain file, it needs to be notified by the kernel space whenever the data for the attribute changes. The `sysfs_notify()` call releases all pending processes waiting for changes on a file of interest. The user space program can indicate which attributes it's interest in and block in a `poll()` call, until it's notified by the kernel. The call must be explicitly made by any subsystem implementing a pollable attribute, otherwise sysfs has no way of knowing when the value of a specific attribute has changed.

4.5.2 Kernel-space character device driver

The character device interfacing with the MCC subsystem is classified as a platform device. The platform category encompasses all those devices that are not inheritably discoverable by the CPU, thus appearing as autonomous entities in the system. The platform interface provides a way for the present hardware to be recognized by the kernel. A virtual platform bus groups all devices under a common bus and allows them to be discovered. Drivers of platform devices must register themselves as such with the platform bus code. The platform driver follows the standard driver model convention, in which discovery is handled outside the driver, and the driver itself provides `probe()` and `remove()` functions. Driver registration is done by way of a `platform_driver` structure, that must include at minimum the `probe()`

and `remove()` functions. The driver must also specify a name in the `driver` field, to provide a way for the platform bus driver to actually bind the device to its driver.
The `probe()` function gets a `platform_device` pointer as argument, describing the characteristics of the device to be allocated, and registers it. In the developed driver, the function is also responsible for initializing the MCC subsystem and creating the group of attributes that will be exposed to the user space via sysfs. The sysfs attribute group is removed in the driver's `remove()` function.
The driver has been developed by using the one for the MCC pingpong demo as reference. The device's main task is to collect the messages coming from the Cortex-M4 endpoints and make them accessible to the user-space. The driver's workload is distributed among the following functions:

- **Initialization function**: the function creates two endpoints, one for each of the sending endpoints on the Cortex-M4 counterpart and initializes the necessary data structures (e.g. FIFO message queues).
- **Data receiving functions**: two functions are responsible for receiving the incoming messages containing respectively the quaternion update values and the statistics on the elapsed time together with the average quaternion values. Each function have a FIFO queue available, in which messages can be progressively pushed as they are received. Every time that a message is received and queued, the user space is notified so it can read and visualize the data. Congruence on the order in which messages are received and later displayed by the user application is ensured by the FIFO policy of the message queues.
- **Data reading functions**: two functions are responsible for returning the message's content to the user space: one function retrieves the update messages coming from the main task, the other retrieves statistics messages coming from the statistics task. Whenever one of this two functions is called, a message is popped from the proper FIFO queue and the content is formatted and copied on a provided buffer. The content of the buffer is then copied to user space.
- **De-initialization function**: when called, the function destroys the endpoints.

Interaction between kernel space and user space is handled through the `/sys` filesystem. The device driver developed creates a single kernel object, with six attributes (i.e. one directory in the filesystem, containing six files):

- `trigger_init` (Write-only): used by the user space to express the will to initiate the MCC communication. Whenever the user space program writes a value on the file, the initialization function is triggered (i.e. MCC is initialized and the endpoints are created).

- `trigger_read` and `trigger_read_stats` (Write-only): used by the user space to trigger the data receiving functions (respectively for update and statistics messages). When a value is written on the file, the driver starts receiving incoming messages and pushing them in the FIFO queues. Whenever a new message is received by the data receiving functions, a call to `sysfs_notify()` is made to inform the user space that new data is available.

- `orientation_data` and `stat_data` (Read-only): attributes representing the received data. When a read operation is invoked on these files, the driver pops the respective message from the queue and passes the formatted content to the user space.

- `trigger_close` (Write-only): by writing a value on this file, the user triggers the function responsible for closing the communication and destroying the endpoints.

The driver's code has been copied in the `$KERNEL_ROOT/drivers/char/imx_amp/` folder, along with the device driver for the pingpong MCC demo. The `Makefile` and `Kconfig` files have been edited, to allow the driver's code to be compiled. The proper functioning of the device can be tested by using the shell commands `cat` and `echo` respectively to read and write a value on the sysfs files. The files for platform devices are located in the `/sys/bus/platform/devices/<name-of-the-device>` folder.

4.5.3 User-space application

The user space application is responsible for accessing the sysfs files and for displaying the quaternion and statistics values received from the Cortex-M4 core. The application interacts with the platform driver in kernel space via sysfs, to collect and display the data. The application consists of three asynchronous threads: the main thread coordinating all the tasks, a thread initiating the reception of update messages and a thread initiating the reception of statistics messages. The application's workflow is here described.

Initialization Initially, all necessary data structures are initialized, including the buffers in which data is copied from kernel space when a read operation on a file occurs. The MCC subsystem is then initialized and endpoints are created, by opening the `trigger_init` file and by writing a "1" value on it. As said before, this action triggers the device's initialization function. The `orientation_data` and `stat_data` files are also opened and a dummy-read operation is invoked on both files. The reason for these preliminary read operations, apparently useless as previous to the actual data reception, will be clarified below.

Message receiving Two asynchronous threads are then started. The first one starts the reception of update messages by writing on the proper file, while the second one starts the reception of statistics messages. From now on, messages sent by the Cortex-M4 core are received by the Cortex-A9 endpoints.

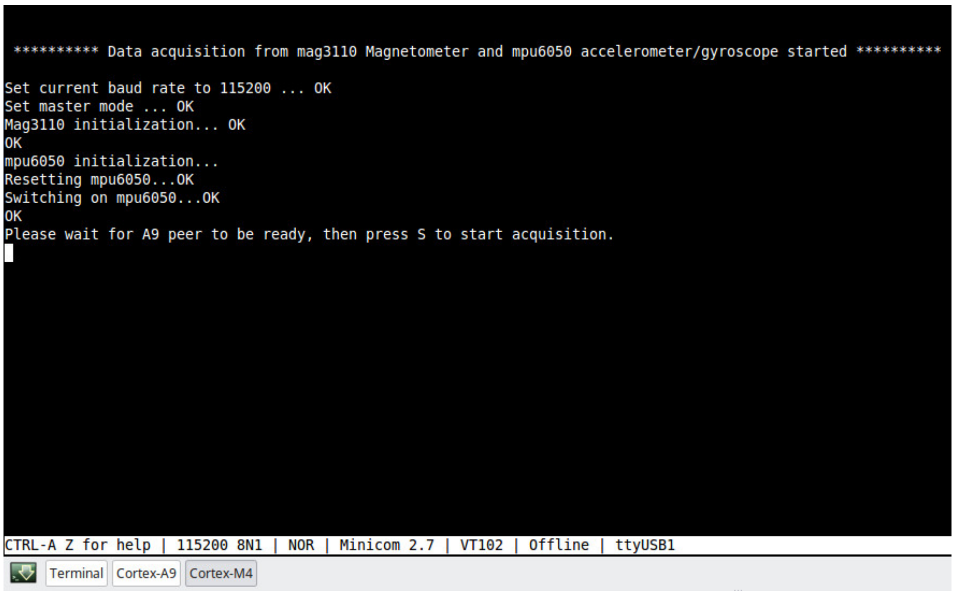
File polling Concurrently with the starting of the two asynchronous threads, the main thread starts polling the `orientation_data` and `stat_data` files. The `poll()` function waits for one or more file descriptors to become ready to perform I/O operations. Each file descriptor to be monitored is specified in a specific `pollfd` structure. The structure specifies an `event` field as an input parameter, containing a bit mask that specifies the events the program is interested in for the respective file descriptor. The structure includes also a `revents` field as an output parameter, which is filled in by the kernel with the events that actually occur (e.g. a `POLLPRI` value indicates that there is urgent data to read). This field allows to distinguish which file has changed within the observed set. Any newly opened file is considered changed; the previous dummy-reads of the files of interest makes the `poll()` function to block until actual data is received. Any `sysfs_notify()` call from kernel space releases the `poll()` function, meaning that a new message has been received.
When the polling function returns, the file in which a `POLLPRI` event occurred is read and data is displayed on the console. As said before, quaternion floating point values are sent in form of unsigned 32-bit integers; the values are converted back to floating point data before displaying them on the console. The `union` C datatype allows to convert between the binary representations of integers and float, avoiding the possible loss of precision of casting.

De-initialization When all data has been received, endpoints on the Cortex-A9 are destroyed by calling the driver's de-initialization function. This is achieved by writing on the dedicated `trigger_close` file.

The quaternion and time values are printed on both the A9 and M4 consoles, to allow consistency check between the transmitted and received data.

4.5.4 Application usage and testing

Once the MQX application has been compiled, the binary file is transferred on the SD-Card and it's copied to the flash memory. The MQX application is executed on the Cortex-M4 as soon as the board is switched on. After the sensors' initialization is completed the application hangs, waiting for the user to signal whenever the A9 peer is ready (Fig.4.11).

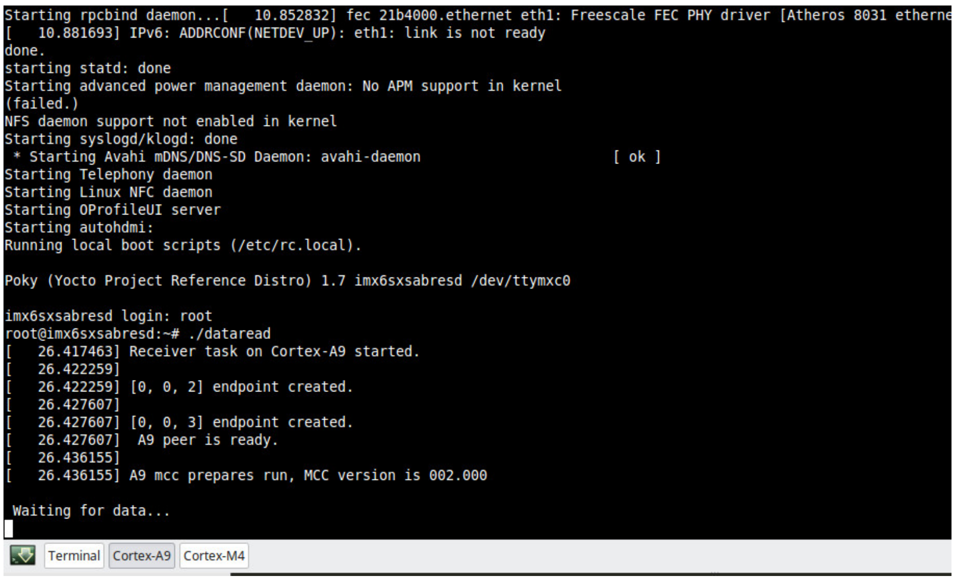


```
 ********** Data acquisition from mag3110 Magnetometer and mpu6050 accelerometer/gyroscope started **********

Set current baud rate to 115200 ... OK
Set master mode ... OK
Mag3110 initialization... OK
OK
mpu6050 initialization...
Resetting mpu6050...OK
Switching on mpu6050...OK
OK
Please wait for A9 peer to be ready, then press S to start acquisition.
```

```
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Offline | ttyUSB1
```

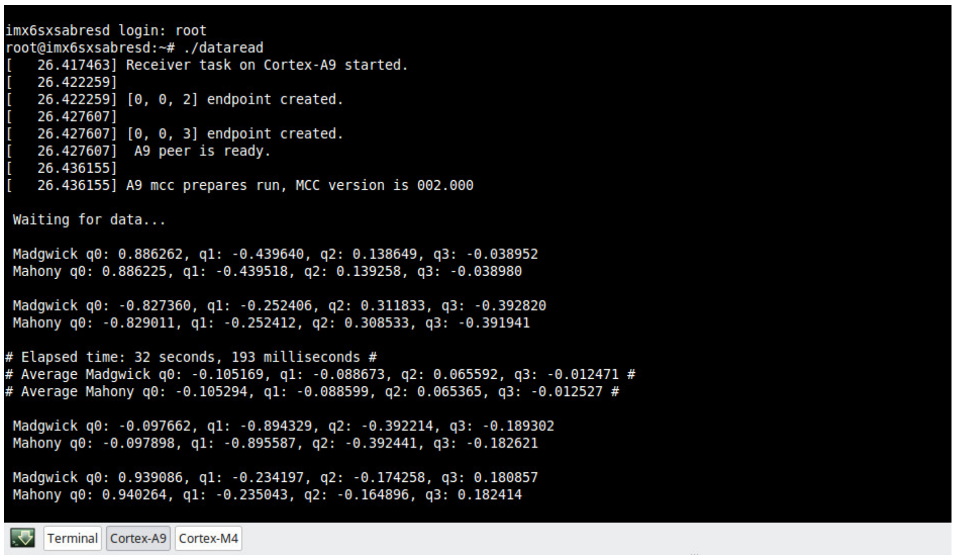
Figure 4.11: Demo screen on Cortex-M4 startup.

On the Cortex-A9 side when the kernel finishes loading and the prompt is shown, the user space application's binary can be launched. The MCC subsystem is initialized and endpoints are created. A message is printed on the console to notify the user that the application is ready to receive data (Fig.4.12).



```
Starting rpcbind daemon...[   10.852832] fec 21b4000.ethernet eth1: Freescale FEC PHY driver [Atheros 8031 etherne
[   10.881693] IPv6: ADDRCONF(NETDEV_UP): eth1: link is not ready
done.
starting statd: done
Starting advanced power management daemon: No APM support in kernel
(failed.)
NFS daemon support not enabled in kernel
Starting syslogd/klogd: done
 * Starting Avahi mDNS/DNS-SD Daemon: avahi-daemon                       [ ok ]
Starting Telephony daemon
Starting Linux NFC daemon
Starting OProfileUI server
Starting autohdmi:
Running local boot scripts (/etc/rc.local).

Poky (Yocto Project Reference Distro) 1.7 imx6sxsabresd /dev/ttymxc0

imx6sxsabresd login: root
root@imx6sxsabresd:~# ./dataread
[   26.417463] Receiver task on Cortex-A9 started.
[   26.422259]
[   26.422259] [0, 0, 2] endpoint created.
[   26.427607]
[   26.427607] [0, 0, 3] endpoint created.
[   26.427607]  A9 peer is ready.
[   26.436155]
[   26.436155] A9 mcc prepares run, MCC version is 002.000

 Waiting for data...
```

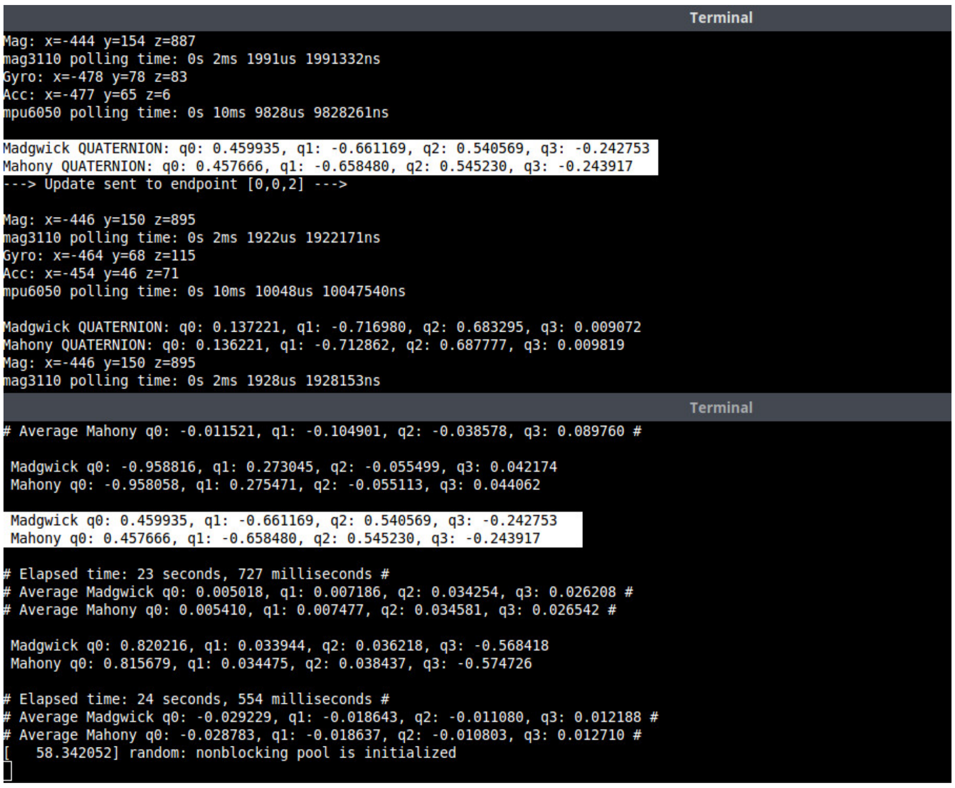
Figure 4.12: User space application's start-up screen.



```
imx6sxsabresd login: root
root@imx6sxsabresd:~# ./dataread
[   26.417463] Receiver task on Cortex-A9 started.
[   26.422259]
[   26.422259] [0, 0, 2] endpoint created.
[   26.427607]
[   26.427607] [0, 0, 3] endpoint created.
[   26.427607]  A9 peer is ready.
[   26.436155]
[   26.436155] A9 mcc prepares run, MCC version is 002.000

 Waiting for data...

 Madgwick q0: 0.886262, q1: -0.439640, q2: 0.138649, q3: -0.038952
 Mahony q0: 0.886225, q1: -0.439518, q2: 0.139258, q3: -0.038980

 Madgwick q0: -0.827360, q1: -0.252406, q2: 0.311833, q3: -0.392820
 Mahony q0: -0.829011, q1: -0.252412, q2: 0.308533, q3: -0.391941

# Elapsed time: 32 seconds, 193 milliseconds #
# Average Madgwick q0: -0.105169, q1: -0.088673, q2: 0.065592, q3: -0.012471 #
# Average Mahony q0: -0.105294, q1: -0.088599, q2: 0.065365, q3: -0.012527 #

 Madgwick q0: -0.097662, q1: -0.894329, q2: -0.392214, q3: -0.189302
 Mahony q0: -0.097898, q1: -0.895587, q2: -0.392441, q3: -0.182621

 Madgwick q0: 0.939086, q1: -0.234197, q2: -0.174258, q3: 0.180857
 Mahony q0: 0.940264, q1: -0.235043, q2: -0.164896, q3: 0.182414
```

Figure 4.13: Example of data received by the Cortex-A9.

Data acquisition and transferring over MCC can then be initiated, by typing "S" on the Cortex-M4 prompt. Sensor data acquisition is started, quaternions are calculated and messages are sent from the M4 endpoints. As soon as messages are received on the Cortex-A9, the content is displayed on the screen (Fig.4.13). Statistics messages are framed by # symbols to be recognized from the update messages, which occur more frequently. The computed values are shown on both sides and they can be compared, to verify the conformity between sent and received data.
Fig.4.14 shows the correspondence between an update message sent by the M4 core and the relative message received by the A9 core.



```
Mag: x=-444 y=154 z=887
mag3110 polling time: 0s 2ms 1991us 1991332ns
Gyro: x=-478 y=78 z=83
Acc: x=-477 y=65 z=6
mpu6050 polling time: 0s 10ms 9828us 9828261ns

Madgwick QUATERNION: q0: 0.459935, q1: -0.661169, q2: 0.540569, q3: -0.242753
Mahony QUATERNION: q0: 0.457666, q1: -0.658480, q2: 0.545230, q3: -0.243917
---> Update sent to endpoint [0,0,2] --->

Mag: x=-446 y=150 z=895
mag3110 polling time: 0s 2ms 1922us 1922171ns
Gyro: x=-464 y=68 z=115
Acc: x=-454 y=46 z=71
mpu6050 polling time: 0s 10ms 10048us 10047540ns

Madgwick QUATERNION: q0: 0.137221, q1: -0.716980, q2: 0.683295, q3: 0.009072
Mahony QUATERNION: q0: 0.136221, q1: -0.712862, q2: 0.687777, q3: 0.009819
Mag: x=-446 y=150 z=895
mag3110 polling time: 0s 2ms 1928us 1928153ns
```

```
# Average Mahony q0: -0.011521, q1: -0.104901, q2: -0.038578, q3: 0.089760 #

 Madgwick q0: -0.958816, q1: 0.273045, q2: -0.055499, q3: 0.042174
 Mahony q0: -0.958058, q1: 0.275471, q2: -0.055113, q3: 0.044062

 Madgwick q0: 0.459935, q1: -0.661169, q2: 0.540569, q3: -0.242753
 Mahony q0: 0.457666, q1: -0.658480, q2: 0.545230, q3: -0.243917

# Elapsed time: 23 seconds, 727 milliseconds #
# Average Madgwick q0: 0.005018, q1: 0.007186, q2: 0.034254, q3: 0.026208 #
# Average Mahony q0: 0.005410, q1: 0.007477, q2: 0.034581, q3: 0.026542 #

 Madgwick q0: 0.820216, q1: 0.033944, q2: 0.036218, q3: -0.568418
 Mahony q0: 0.815679, q1: 0.034475, q2: 0.038437, q3: -0.574726

# Elapsed time: 24 seconds, 554 milliseconds #
# Average Madgwick q0: -0.029229, q1: -0.018643, q2: -0.011080, q3: 0.012188 #
# Average Mahony q0: -0.028783, q1: -0.018637, q2: -0.010803, q3: 0.012710 #
[   58.342052] random: nonblocking pool is initialized
```

Figure 4.14: Cortex-M4 console screen (on top) and Cortex-A9 console screen (on bottom). A sent message and the correspondent received message are highlighted.

Chapter 5

Conclusions and future developments

The results obtained by running the application on the i.MX 6SoloX SABRE board illustrate the prospect of adopting an asymmetric architecture as the core infrastructure for a quadcopter's on-board flight control system. The intrinsic heterogeneity of an AMP system satisfies the need for both real-time responsiveness and processing power, while fully exploiting the potential of microcontroller and microprocessor architectures.
Nowadays, predictions about the future of drones tend to focus on the capability of executing fully autonomous tasks and performing more and more complex operations. Executing MQX and Linux side by side allows to run sophisticated algorithms and take advantage of the standard Linux features without affecting the drone's stabilization and maneuvering.
MQX RTOS has proven to be a valid choice for efficiently handling sensor data and implementing data fusing algorithms. The upside of using the RTOS on an asymmetric environment is the availability of simple and well-documented multi core communication libraries. The developed demo application shows that inter-core communication works seamlessly and that it's possible to exchange messages of different nature and size, carrying all kind of useful information on the drone's state.
The implemented application lays the foundations for a future integration of the board with the drone's infrastructure, by adding a propulsion system (including motors, propellers, ESCs and battery). Moreover, the user-space application favors the implementation of more complex algorithms on the Linux side. Computed quaternion data is made available to user space, thus encouraging the buildup of the current application and the processing of the available information for more sophisticated purposes, besides plain visualization. Potential high-level applications that could employ the available data on the drone's orientation include: tracking and plotting of the drone's position and orientation, running more advanced AHRS algorithms to cor-

rect the drone's attitude, using the indication on the drone's direction for autonomous navigation, performing computer vision tasks.

References

[1] Chris DiBona, Sam Ockman, *Open Sources - Voices from the Open Source Revolution*, O'Reilly Media, 1st edition, 1999.
`http://www.oreilly.com/openbook/opensources/book/`

[2] Greg Kroah-Hartman, *Linux Kernel in a Nutshell*, O'Reilly Media, 1st edition, 2006.
`http://www.kroah.com/lkn/`

[3] Robert Mahony, *Nonlinear Complementary Filters on the Special Orthogonal Group*, 2008.
`http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4608934&tag=1`

[4] Patrick Mochel, *The sysfs Filesystem*, 2005.
`https://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf`

[5] Sebastian O.H. Madgwick, *An efficient orientation filter for inertial and inertial/magnetic sensor arrays*, 2010.
`http://www.x-io.co.uk/res/doc/madgwick_internal_report.pdf`

[6] chrobotics.com, *Understanding Euler Angles.*
`http://www.chrobotics.com/library/understanding-euler-angles`

[7] linux.org, *The Linux Kernel: The Source Code*, 2013.
`http://www.linux.org/threads/the-linux-kernel-the-source-code.4204/`

[8] lwn.net, Nathan Willis, *Linux and the future of drones*, 2015.
`https://lwn.net/Articles/659687/`

[9] olliw.eu, *IMU Data Fusing: Complementary, Kalman, and Mahony Filter*, 2013.
`http://www.olliw.eu/2013/imu-data-fusing/`

[10] tldp.org, *The Linux Kernel*, 1999.
`http://en.tldp.org/LDP/tlk/tlk.html`