Generic Hardware Description for Embedded Platforms

Clément Parisot

June 9th 2015
Abstract

On the area of microcontrollers, a firmware is traditionally built for a very specific hardware configuration. Without special design, there is little chance that the same firmware will run on several hardware platforms with a different sets of peripherals.

But running the same firmware on different hardware configurations could have some benefits. It could allow a manufacturer or a sensor network manager to deploy the exact same firmware on all its nodes regardless of their hardware. It would greatly simplify the firmware management, and thus the update process.

We know that such a system is possible on larger architectures, such as x86 or even ARM, but in this thesis we target smaller architectures. The typical target here is a sensor network node, running on a very low-power microcontroller. No generic system currently exists to allow a firmware to run on several hardware configurations of this type.

In this thesis we present a new generic hardware description system that specifically targets small devices. This system can be integrated with existing frameworks or operating systems for embedded systems so that the firmware can adapt to the hardware it is running on. We show that it is possible by presenting a demonstration prototype using our hardware description system.
Contents

List of Acronyms vii

Introduction 1
  Goals ......................................................... 1
  Methodology .................................................. 2
  Structure ..................................................... 2

1 Generic hardware description 3
  1.1 Principle .................................................. 3
  1.2 On non-embedded platforms ................................. 3
    1.2.1 x86 PCs .................................................. 3
    1.2.2 Linux on non-x68: the Device Tree .................... 5
  1.3 On targeted platforms ...................................... 6
    1.3.1 Contiki ................................................ 6
    1.3.2 Arduino ............................................... 8
  1.4 More precise description of the Device Tree .............. 10
    1.4.1 The reason to study the Device Tree ................ 10
    1.4.2 High-level view ....................................... 10
    1.4.3 Bootloader and extension boards ..................... 11
    1.4.4 DTS overview ......................................... 12
    1.4.5 DTB basics ........................................... 14

2 Adaptation to embedded devices 17
  2.1 Specific constraint for small embedded devices ........ 17
  2.2 Analysis of the existing format .......................... 17
    2.2.1 Sample Device Tree file .............................. 17
    2.2.2 What can be changed .................................. 18
  2.3 Other software constraints ................................ 20
    2.3.1 Dynamic driver loading ............................... 20
    2.3.2 Same microcontroller architecture ................. 20
    2.3.3 Separation from the firmware ....................... 20
  2.4 Format specification ...................................... 21
    2.4.1 The Light DTB specification ......................... 21
    2.4.2 The Light DTS specification ......................... 22

3 Prototype 23
  3.1 Goal and required hardware ................................ 23
  3.2 Compiler implementation .................................. 24
    3.2.1 The DTC as a base ................................... 24
CONTENTS

3.2.2 Frontend and backend implementations .................................. 24
3.3 Arduino toolchain ................................................................. 25
  3.3.1 Arduino IDE ................................................................. 25
  3.3.2 Makefile automation ......................................................... 26
3.4 Firmware Implementation ........................................................ 27
  3.4.1 Code structure ............................................................... 27
  3.4.2 Sample Programs behavior ............................................... 28
  3.4.3 Development program ..................................................... 28
  3.4.4 Callback system ............................................................ 29
  3.4.5 Arduino dynamic driver management ................................... 30
  3.4.6 LDT adaptations for multiple drivers .................................. 31
  3.4.7 Advanced callback system ................................................ 34
3.5 Results and measurements ....................................................... 34
  3.5.1 Results ....................................................................... 34
  3.5.2 Measurements ............................................................... 35
3.6 Conclusion of the prototype experiment ...................................... 37

4 Going further ............................................................................ 39
  4.1 Other platforms: Contiki ......................................................... 39
  4.2 Device Tree header file generator ........................................... 41
  4.3 Shield/Cape/Hat-like system ? ............................................... 42

Conclusion ................................................................................. 43

Appendix A .................................................................................. 45

Appendix B .................................................................................. 51
  Simple callback Hello World ...................................................... 51
  Advanced Callback with sensors ................................................ 51
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The Contiki sensors definition</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>The Contiki sensors declaration</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Arduino initialisation code</td>
<td>9</td>
</tr>
<tr>
<td>1.4</td>
<td>Arduino conditional initialisation</td>
<td>9</td>
</tr>
<tr>
<td>1.5</td>
<td>Example Cortex-A8 board DTS file</td>
<td>12</td>
</tr>
<tr>
<td>1.6</td>
<td>The structure of a DTS file</td>
<td>13</td>
</tr>
<tr>
<td>1.7</td>
<td>Extract of the Linux Device Tree documentation</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>DTS file extract with multiple DHT sensors</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Two examples of class drivers</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Extract of a debug boot log</td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>Flash size results summary</td>
<td>37</td>
</tr>
<tr>
<td>1</td>
<td>A fake Device Tree describing the peripherals of a test Arduino board configuration</td>
<td>46</td>
</tr>
<tr>
<td>2</td>
<td>The DTB file corresponding to the former DTS file</td>
<td>47</td>
</tr>
<tr>
<td>3</td>
<td>A Light Device Tree describing the same configuration</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>The LDTB file corresponding to the former LDTS file</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>Binary data presented as as C string</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>Hello World Arduino example with basic callback</td>
<td>52</td>
</tr>
<tr>
<td>7</td>
<td>Arduino Callback with sensors - Variables</td>
<td>53</td>
</tr>
<tr>
<td>8</td>
<td>Arduino Callback with sensors - Setup</td>
<td>54</td>
</tr>
<tr>
<td>9</td>
<td>Arduino Callback with sensors - loop function</td>
<td>55</td>
</tr>
</tbody>
</table>
List of Acronyms

DT  Device Tree .................................................. 11
GPIO General Purpose Input/Output ......................... 1
LDT Light Device Tree ......................................... 27
OS  Operating System .......................................... 3
Introduction

Background

The market for embedded devices is growing fast. We see sensor networks emerging in different fields: home automation, building or city-wide smart grids. There are also many standalone devices like body health sensors, connected clothes, etc ...

This growing number of peripherals creates new challenges for the manufacturer. Let’s imagine a home automation network composed of many different wireless sensors or actuators: a temperature sensor in the living room, a temperature and humidity sensor in the bathroom, a presence sensor, a smoke detector in the kitchen, ... All of these may communicate with a home automation box.

One of the problems for the manufacturer is the firmware management. Even if the sensors all share the same base platform (like a low-power microcontroller, a radio and a generic way to connect a sensor using a bus or General Purpose Input/Output (GPIO)), the manufacturer has to make one firmware for each device, because the sensors attached to the platform behave differently. There may even be one firmware for each hardware revision of a same product.

All those different firmware versions make it harder to update the firmwares of the devices in the network. The updater has to detect the hardware version correctly before pushing the update. The manufacturer has to keep track of all the possible firmware versions corresponding to all the hardware configuration.

It would be helpful to make the firmware management of a fleet of small IoT devices easier. The solution that is explored in this thesis is to find a way to run a single firmware on multiple hardware configurations. Obviously this single firmware should support more functionalities (that is: the functionalities of all the hardware configurations), so it will be bigger. But the size of the firmware, as we will see, is not the only challenge, and may not even be a problem.

Goals

The goal of this project is to explore the ways to make a single firmware run on multiple hardware configurations. Our target platforms are the microcontrollers, for example those dedicated to low-power sensors (using a Cortex-M3 core or smaller). We restrain the study to this area because on bigger platforms, big enough to run Linux, some solutions already exist, as we will see.
INTRODUCTION

The solution should also integrate with some existing platforms or framework, as the goal is to find a practical solution, that can be implemented by vendors, and that doesn’t need to reimplement half an OS.

The practical aspect of the project is also important, so the solutions that are found should be validated with a prototype. On a high level point of view, it should consist of two (or more) boards with different hardware configurations. We should show that when the same firmware is pushed to every board, each one behaves as if it was programmed with its own firmware.

Methodology

The project is structured in the following way.

First we study existing solutions that exist outside of our perimeter. By finding solutions that have already been tested, we can find ideas that may be applied to our specific case: small embedded systems.

Then we must study the specific constraints of those systems, and find why the found solutions for bigger systems can not be applied as-is.

Based on those ideas and constraints, a new solution can be imagined and designed.

Once the solution is defined well-enough, it is implemented and a prototype is built to validate the idea.

The prototype is used to check if the solution works, and to find obvious problems. A series of measurement are taken to check the impact of code changes on various parameters like code size.

Structure

The structure of the thesis reflects the methodology:

- Chapter 1 explains the base principle and studies existing solutions for other devices
- Chapter 2 explores the constraints of embedded devices and create the specification of the format to be implemented.
- Chapter 3 describes the implementation of the prototype and the results of the measurements
- Chapter 4 presents ideas to improve the system and to implement new features.
Chapter 1

Generic hardware description

1.1 Principle

As explained in the introduction, our goal is to adapt some microcontroller-based platform in order to be able to run the same firmware on two different hardware configurations.

This could be useful for example in the case of a home automation network. We can imagine a home automation box to which many wireless sensors are connected, directly or using a mesh network. It is not absurd to suppose that many of those sensors share a common base, both in hardware and in software.

For example, if we compare a temperature sensor and an humidity sensor, the same microprocessor can be used in both. The networking capabilities are also similar: both the hardware radio and the software network stack can be shared. The only real difference between those two devices is that one will have a temperature sensor attached while the other will detect humidity variations.

A lot of code is shared between the two devices, only the sensor-specific part of the software will vary, and that part is not so big. It makes sense to try to make a single firmware that will run on both devices. The problem is: how does the firmware detect whether a temperature sensor, a humidity sensor or a smoke detector is connected? How can existing software platforms be adapted to achieve this goal?

But first, a study of existing solutions in other similar domains should be undertaken. It is necessary to understand the pitfalls of this kind of setup and understand how they have been avoided in other contexts. Then a study of the specific constraints of embedded microcontrollers is necessary, to understand why existing solutions can’t always apply and if they can be adapted.

1.2 On non-embedded platforms

1.2.1 x86 PCs

Our first subject of study is the regular x86 computer. It is very far from an embedded microcontroller, but it is nevertheless interesting because it is a very evolved platform that has found various solutions to old problems.

Translated to a PC world, our problem is stated this way: can the same Operating System (OS) run on different hardware configurations? We know
the answer is yes. For example, Microsoft Windows or Linux can run on many different configurations. Nowadays it even works out-of-the-box in most cases: the user doesn’t need to install or configure specific drivers, everything is detected by the operating system: the amount of available RAM, a networking card, a USB key, ... 

It should be noted that there are actually two problems here: the first is to discover what hardware peripherals are present, and the second is to know how to use them. In modern operating systems the "how to use it" problem is solved by embedding many drivers in the OS. The problem we are interested in is the first one: how to actually detect what hardware is present to know which driver to load (and how to configure it)? It is solved on x86 platforms thanks to two main concepts: discoverable hardware and ACPI tables.

1.2.1.1 Discoverability

On modern x86 PCs, most hardware buses are discoverable. That means that the OS can ask the bus to list all the hardware devices connected to the bus as well as various properties of those devices (like the manufacturer, an unique ID and/or a requested memory range to map to memory). For example, the PCI bus is discoverable: if I plug a new graphic card or a network card, the OS can detect it. It is the same for the USB bus. This discoverability notion is important because the OS doesn’t need to know in advance the hardware that is connected. It can detect it at run-time, or even allow hot-plugging of devices (depending on the bus).

1.2.1.2 ACPI tables

But unfortunately everything is not connected to a discoverable bus. For example, to be able to enumerate the PCI devices, the OS must first know the address of the PCI controller. There are also many peripherals that are not on any bus (like a serial RS232 port). There are two solutions to this problem: the address of such devices can be a convention (it will be the same fixed address on every x86 PC) or the address can vary between models and must be registered somewhere for the OS to know.

This last possibility is implemented on x86 PCs as ACPI tables. ACPI is a standard that is mainly known to manage the power states of the hardware (suspended or running, the screen brightness on laptops, ...), but it is not its only use. ACPI is composed of two parts: static tables containing information about the hardware and a "dynamic" part composed of code that is used to manage various hardware capabilities (including power management). Here we are more interested in the static part.

In the static ACPI tables the manufacturer puts a description of all the hardware that is not discoverable. Because of that, the OS now has a way to discover those informations and can initialize the corresponding devices properly.

1.2.1.3 Summary

To sum up, on x86 platforms there are three way the OS knows how to discover hardware:
1. fixed memory addresses (x86 convention) mainly for legacy hardware or early boot-time configuration

2. discoverable buses (PCI, USB, ...)

3. ACPI tables for everything that is not discoverable

1.2.2 Linux on non-x68: the Device Tree

The x86 platform may be very common, it is however very far from the kind of device that we are ultimately interested in. In this section we study the case of Linux on smaller platforms, and more specifically ARM (but most of what is developed here also holds for MIPS and PowerPC).

1.2.2.1 The non-discoverability problem

The difference between ARM and x86 platforms is that in the ARM world most of the hardware is not discoverable. The ARM device comes as a System on Chip, where everything is defined in silicon and is not going to change, which is different of the PCs for which the CPU can be plugged into many different motherboards, with different peripherals and amounts of RAM.

The ARM software is supposed to be designed for a specific platform and it should thus know what hardware is present and how it is configured (for memory-mapped controllers for example).

This non-discoverability is a problem, for example in the case of a Linux system. Initially, in order to support ARM platforms, the Linux kernel had to have one source (.c) file for each possible supported board. This board file took care of all the boot-time board-specific configuration, like initialisation code or hard-coded memory addresses. The supported board file had to be selected at compile time, which means that a kernel compiled for a specific board could absolutely not run on a board with a different configuration. This way to organize code had many consequences:

- One .c file for each supported board means a lot of files
- If two boards are similar, but not exactly the same, either the same file is used with #ifdef statements, or the code is duplicated in two different board files. That means that the code is either duplicated or unreadable (too many #ifdefs)
- The driver code itself could be duplicated. Indeed, if the same silicon IP (requiring a specific driver) is used in two different chips, it doesn’t mean that they are configured the same way (memory mapping, interrupt lines, clock frequency, ...) so the driver code for this IP could end up duplicated in board files (once for each configuration) whereas it should be centralized in one single place for easier maintainability.

All those problems forced the Linux community to find a solution. What was adopted was a port of a system that already existed for MIPS and PowerPC architectures: the Device Tree.
1.2.2.2 The Linux Device Tree

The Device Tree idea is quite straightforward: the idea is to have a generic kernel, with generic non-duplicated drivers. Then, next to the kernel there is a file, called the Device Tree Blob, that contains a description of the hardware. It is important: it is only a description, it contains no code! The description should convey sufficient information to tell the drivers the correct way to initialize.

At boot-time, the kernel reads the Device Tree Blob, decodes the information, and uses it to load the correct drivers who initialize the hardware. As the Device Tree is a description of the hardware, it is tied to this particular hardware. Two different boards will have two different Device Trees, which will not be modified unless there is a hardware change.

It solves the problem because now the kernel is entirely generic: the hardware configuration that was previously hard-coded in the kernel is now independent of it. The same kernel (and thus the same firmware image) is able to run on different boards as long as each board has its own Device Tree file.

This solution is interesting to us and is described in more details on section 1.4.

1.3 On targeted platforms

The x86 solution is powerful, but it is not usable on small microcontrollers: there are no discoverable buses at this level. Moreover, Linux-based solutions are too complex to fit in such devices.

However, the Linux example demonstrates that, starting from a platform with no discoverable hardware, it is still possible to achieve more flexible kernel/firmware management. But it was possible to integrate such a solution because Linux itself was quite modular: it is possible to load a driver at runtime, for example. It is also possible to include many drivers in the kernel, for many different boards, and the unused drivers (for hardware that is not connected) will not interfere with loaded ones.

It seems possible to achieve the same goal on a microcontroller if the driver management is flexible enough on existing platforms. To anticipate the creation of a prototype, let’s start by studying very quickly two platforms that we might use to build it.

1.3.1 Contiki

Contiki is an operating system that specifically targets small embedded devices for the Internet of Things. It is written in C and provides many functions from simple GPIO management to a full mesh networking stack.

The code that is looked at here is more specifically the reference example code for the CC2530 module (manufactured by TI), but as Contiki is designed to be adapted to many platforms, the code is generic enough so that we can draw conclusions for Contiki in general.

1.3.1.1 Driver management

The part we are interested in is mainly the hardware drivers management. How are the hardware peripherals accessed by the software, and are there some
drivers that can be loaded at run-time depending on a hardware configuration?

In Contiki, a sensor is a generic structure that represents any hardware from which data can be read. For example, the CC2530 has one ADC and one button onboard, and its brother the CC2531 has one ADC and two buttons.

The definition of a sensor structure is wrapped in a macro (where value, configure and status are functions that manage the sensor), as in figure 1.1. The code to initialize the sensor list is on figure 1.2.

**Figure 1.1:** The Contiki sensors definition

```
//platform/cc2530dk/dev/button-sensor.c
3 SENSORS_SENSOR(button_1_sensor, BUTTON_SENSOR,
4   value_b1, configure_b1, status_b1);
5
6 #if MODELS_CONF_CC2531_USB_STICK
7   SENSORS_SENSOR(button_2_sensor, BUTTON_SENSOR,
8     value_b2, configure_b2, status_b2);
9 #endif
```

**Figure 1.2:** The Contiki sensors declaration

```
//platform/cc2530dk/dev/smartrf-sensors.c
2 const struct sensors_sensor *sensors[] = {
3   #if ADC_SENSOR_ON
4     &adc_sensor,
5   #endif
6   #if BUTTON_SENSOR_ON
7     &button_1_sensor,
8     #if MODELS_CONF_CC2531_USB_STICK
9       &button_2_sensor,
10    #endif
11   #endif
12  0
13};
14
15 unsigned char sensors_flags[(sizeof(sensors) / sizeof(struct sensors_sensor *))];
```

We first notice that, depending on the model (CC2530 or CC2531) different sensors are defined: the CC2531 model has one more button (button_2_sensor). But they are overall quite similar, so it would make sense to make a single firmware for the two of them. The firmware could make use of the second button only if it detects that it is running on the CC2531.

It is also worth noting that both the ADC and the button sensors can be activated or not at compile time. This piece of code makes extensive use of #if macros. The main consequence of this is that it seems impossible to do dynamic loading, as everything is statically defined at compile time.
CHAPTER 1. GENERIC HARDWARE DESCRIPTION

The last thing is that, even if we somehow replaced the #if macros with more dynamic if() statements, there would be a problem to add a sensor to the sensor list because this list is declared in a C array, which can’t be extended at runtime. We can see on the last line that at least some part of the code depends on the size of the array at compile time (use of sizeof(sensors)).

We could try to define a bigger array and use only the first items, and add items when new sensors are defined, but we would need to study the rest of the OS to evaluate the impact of such a dynamic sensor array. Some code may not be compatible, and some may create some subtle bugs.

1.3.1.2 Contiki conclusion

Overall, even if using a hardware description file could prove useful given the similarities between those two platforms, it seems that it won’t be possible without studying closely and possibly re-writing parts of the OS. Indeed those parts may rely on the fact that the sizeof(sensors) array if constant and can only be defined at compile time.

1.3.2 Arduino

Arduino, unlike Contiki is not really an operating system but more like a framework. It can run on targets as small as an AVR 8-bits microcontroller.

The difference with Contiki is that Arduino mainly consists of a set of drivers for the hardware (as well as a few more libraries), and it is the user code that "manually" instantiates the drivers the user needs (here "user" is to be understood as "user of the Arduino framework", who is actually the developer, and not "user of the device")

A typical code is composed of a setup() function in which the system is initialized, and a loop() function which contains the actual program logic.

The Arduino framework is designed to run on several Arduino boards with different capabilities. For our study we use the low-end board: the Arduino Uno, running on an Atmega328P (8-bits, 2kB of RAM, 32kB of Flash), but the code itself (from the user point of view) is basically the same for other boards.

1.3.2.1 Driver flexibility

Let’s look at one of the basic examples provided by the framework: how to use a servomotor (a small actuator controlled with PWM). A sample basic initialization code is presented in figure 1.3

This is only the setup() part of the code. Afterwards (in the loop() function), the programmer can use myservo.write(pos) to send a position order to the actuator.

There are several observations we can make by looking at this piece of code. First the driver is represented as a Servo C++ class. It is a global variable but the constructor doesn’t actually activate the driver. The driver is only active when myservo.attach(servo_pin) is called in setup(). That means that a driver initialisation can depend on a condition (Figure 1.4)

We also notice that the parameter passed to the servo initialisation function is not static: we could dynamically (but still at start-up time) setup the
1.3. ON TARGETED PLATFORMS

Figure 1.3: Arduino initialisation code

```c
#include <Servo.h>

// create servo object to control a servo
Servo myservo;

int servo_pin = 9;

void setup()
{
    // attaches the servo on pin 9 to the servo object
    myservo.attach(servo_pin);
}
```

Figure 1.4: Arduino conditional initialisation

```c
void setup()
{
    servo_is_present = is_servo_present();
    if(servo_is_present)
        myservo.attach(servo_pin);
}
```

servomotor driver on pin 12 if we detect that the actuator is actually on that pin.

As Arduino is only a framework, compared to Contiki which is a full OS, it is more flexible and lets the user do things the way he wants. The counterpart is that Arduino provides fewer facilities (like process or threads, unified sensors management, file system,...) and the user must reimplement them from scratch if he needs them.

1.3.2.2 Arduino conclusion

Arduino seems to me more flexible than Contiki when it comes to driver management, because everything is done "manually" and is not constrained by a framework. As such, it looks easier as a prototyping platform to develop new paradigms, because we can focus on what we are building and not on how to work around certain OS limitations.

However, Contiki probably carries more value for a company, because it is more likely to be used in a product than Arduino. In that perspective, it is important to think about Contiki even when developing an Arduino prototype, because the prototype could be ported later.
1.4 More precise description of the Device Tree

1.4.1 The reason to study the Device Tree

At that point, we have seen three solutions to generic firmwares, and all the three come from non-embedded or "big" embedded systems:

- Discoverable/enumerable hardware/busses
- ACPI tables on x86
- the Linux Device Tree (ARM / MIPS / PPC Architectures)

In our search for a solution for small devices, we can eliminate the "discoverable bus" option, because introducing new buses require changing the hardware, and our goal is to find a solution that can be used on existing platforms.

Two solutions remain, and they are basically two implementations of the same idea: let the OS access a software description of the hardware and initialize its driver according to this description. The ACPI norm also has a dynamic "firmware code" part in which we are not interested here.

Of the two solutions, one is more interesting: the Device Tree. It has many advantages over ACPI for our purpose:

- It is more 'open'. Even if ACPI is a norm, in practice every manufacturer tries to respect the Microsoft implementation of ACPI (to be able to run Windows), and Microsoft’s implementation doesn’t always respect the norm. Even if we can find free documentation on ACPI, it is harder to find one about Microsoft’s specific implementation. On the other hand, all Linux documentation is freely available.

- It is easier. ACPI is a complex norm that was created to tackle a bigger problem that what we want to solve: it also contains things about power management, for example. The Device Tree, however, precisely aims at solving our problem (albeit for other kinds of platforms).

- Some free code is available. The Linux Device Tree code is available freely, as well as the tools that are necessary to manipulate it. We can base some of our research on them.

Because of all these advantages, the solution that we are going to implement later will probably be based on the Device Tree more than on ACPI. Therefore it is necessary to do a deeper study of the Device Tree to understand what is usable on microcontrollers and what is not.

1.4.2 High-level view

On a high-level view, as explained in section 1.2.2.2, the device tree is a file that describes the board hardware and that is read by the kernel at boot time.

Ideally, the Device Tree file is written by the ARM board manufacturer (and could even be burned into a read-only memory tied to the board), while the kernel is written by the software team and can change between firmware versions.
1.4. MORE PRECISE DESCRIPTION OF THE DEVICE TREE

Ideally, the same generic kernel can also boot on multiple boards (provided that the needed drivers are still built-in) just by changing the driver configuration (at boot time) according to the Device Tree of each board.

An important fact is that the device tree only ever contains a description of the hardware. It isn’t designed to be a generic configuration interface (like the IP address of the device) nor does it contain code.

For readability and maintenance reasons, the device tree files must be text-based (for easy modifications by developers). But on the other hand, for performance reasons, it is better not to include a whole text parser in the kernel. The compromise was to use two file formats: the developer edits a text-based source format, which is then compiled to a binary format to be put on the board.

The source format is called the Device Tree Source, or DTS. The binary format is the Device Tree Blob (or DTB) and the compiler that outputs binary format is the DTC: Device Tree Compiler.

As there are often few differences between two boards of the same family, there is an include system that allows to centralize the common part of a description in an include file, while the board-specific differences are in their own files. The include files have a .dtsi extension (like DTS Include) and the ‘child’ files have a regular .dts extension.

1.4.3 Bootloader and extension boards

1.4.3.1 Role of the bootloader

On the last section, we explained that "the device tree is [...] read by the kernel at boot time". That is actually incorrect: the kernel doesn’t ‘read’ the file, it expects that the file is already in memory when it boots and that the memory address of the Device Tree (DT) is passed as a boot parameter.

That means that it is not the kernel’s job to read the Device Tree, it is actually the bootloader’s. And the bootloader does more than to do a raw copy of the file to memory: it actually understands the DT format and adds information to it. For example, the kernel boot parameters are now passed using the Device Tree.

And there is also a pretty interesting mechanism that uses this fact to allow the board hardware to be even more modular.

1.4.3.2 Extension boards

Many prototyping cards have a motherboard/extension boards mechanism. The Raspberry PI has 'hats' and the BeagleBone has 'capes', which are names for generic extension boards formats.

The problem is that plugging a new capes induces a hardware change from the point of view of the main processor. How is this change reflected on the device tree, which is supposed to represent the hardware?

To avoid flashing a new Device Tree on the main board after each extension board change, there is a clever mechanism. Basically, the Device Tree itself also has an extension mechanism. Each extension board carries, on an EEPROM, a small Device Tree that describes the board.

At boot time, the bootloader reads those DT fragments and "plugs" them into the DT of the main board. The resulting DT describes the whole {board +
extension boards] hardware in a single tree. Then the Linux kernel is loaded, and it only has to read this combined DT without worrying about extension boards.

1.4.4 DTS overview

Let’s see how the information is presented in those DTS files, and what information is actually present.

Here is a basic example of an incomplete DTS file for a single-core Cortex-A8 CPU board (Figure 1.5)

```plaintext
/* */ {
  cpus {
    cpu0 {
      compatible = "arm,cortex-a8";
      operating-points = <
        /* kHz uV */
        720000 1285000
        275000 1125000
      >;
      Voltage-tolerance = <2>; /* 2% */
      clock-latency = <300000>;
    }
  }
  [...]}
};
```

It is now apparent why the format is called the Device Tree. Each device is represented in a hierarchy of devices: the core `cpu0` is inside the `cpu` block (which would group all the CPUs if there were more than one), itself within the root "/" node.

Each block contains informations relevant to its behaviour. For example here the CPU has an operating point at 720MHz with a voltage of 1.285V. This kind of low-level information is useful to configure the voltage regulator depending on the CPU frequency.

Let’s look at another piece of DTS file for the same board (Figure 1.6)

1.4.4.1 The tree structure

This piece of dts file is very interesting because its structure is the base of what is used in every other DTS file.

Let’s start with the second part: we have an I²C bus controller of type "ti,omap4-i2c". On this bus, as a child node, there is an EEPROM of type "at,24c256". This is our first demonstration that a tree structure is relevant: a device (the EEPROM) can “depend” on another (the I²C bus).
1.4. MORE PRECISE DESCRIPTION OF THE DEVICE TREE

Figure 1.6: The structure of a DTS file

```plaintext
/* */{
    
    [...] intc: interrupt-controller@48200000 {
        compatible = "ti,omap2-intc";
        interrupt-controller;
        #interrupt-cells = <1>;
        ti,intc-size = <128>;
        reg = <0x48200000 0x1000>;
    }

    [...] i2c0: i2c@44e0b000 {
        compatible = "ti,omap4-i2c";
        reg = <0x44e0b000 0x1000>;
        clock-frequency = <4000000>;
        interrupt-parent = <&intc>;
        interrupts = <70>;
        status = "okay";
        baseboard_eeprom: baseboard_eeprom@50 {
            compatible = "at,24c256";
            reg = <0x50>;
        }
    }

    [...] };
```

It is worth noting that the I\(^2\)C bus is not discoverable. There is a trick to scan the 256 addresses on the bus to see if something is responding, but depending on the connected devices it can cause problems so is not recommended. Moreover, the only information that we can learn with this kind of scan is the presence or not of a device at each address, but for example noting can tell us if it is an EEPROM or a temperature sensor. Such information has to come from elsewhere: here from the Device Tree.

1.4.4.2 Device properties

In this DTS example there are many other pieces of information. For example node-dependant information like `clock-frequency = <4000000>` for the frequency of the I\(^2\)C bus. But many other properties are standardized.

The `reg` property is very important: it is mandatory for each node. It describes the registers used to control the current node. For example the I\(^2\)C
bus controller is mapped at address $0x44e0b000$ and the mapped address has a size of $0x1000$ bytes.

The EEPROM also has a `reg` property, but it doesn’t describe a register. Instead, it states that the EEPROM is at address $0x40$ on the I²C bus (it is still an address, albeit not in the memory address space).

A property that we find in many devices is `interrupt-parent`, and it shows how to do a reference to another node. Its value must be a reference to the interrupt controller for the device. Here, the controller is named `intc`, and it will receive interrupts for the I²C bus (the interrupt number is 70 according to the `interrupts` property).

1.4.4.3 The `compatible` property

And the last property that is common to all nodes and close to mandatory (or at least very normalized) is the `compatible` property. This one is very important: it describes the type of the peripheral represented by the node, and thus determines which kernel driver will be used to control it.

For example, there may be a lot of I²C drivers in the kernel: one for the TI controllers, one for the NXP controllers, one that is generic, etc. Each of those drivers has to declare a list of `compatible` properties that it supports, and this list is used by the kernel to choose the driver to use.

The `compatible` property is made of two parts separated by a comma: the first one is a short manufacturer identifier ('`ti`' is Texas Instruments, `'fsl`' is Freescale, ...) and the second part is a model identifier. This 'manufacturer,model' template is designed to prevent name collisions between products of different manufacturers.

There is also a mechanism of generic driver loading which uses the `compatible` property. For example let’s imagine a specific GPU. This piece of hardware can be controlled by a generic Linux driver, which supports a lot of drivers and provides basic support, but lacks certain capabilities. And there is another driver, made by the GPU manufacturer, which only supports this GPU but also has more capabilities. We still want to have a basic support by loading the generic driver if the vendor driver is missing. The way this could be achieved is to advertise multiple compatible strings (this, unlike previous DTS extracts, is a made-up example):

```
compatible = "fsl,imx6dl-gpu", "linux,generic-gpu";
```

The `compatible` strings are advertised in the order of more specific to less specific strings. This way the generic driver doesn’t have to know all the possible `compatible` strings of all devices, and the more specific driver can still be loaded if it is present.

1.4.5 DTB basics

After the introduction to the DTS format, it is necessary to have an overview of its binary counterpart. Indeed, this DTB file is what is ultimately stored in the device and it is also parsed by our embedded system. Because of this, we must study its size and complexity: will it take a lot of memory or CPU cycles to parse?
1.4. MORE PRECISE DESCRIPTION OF THE DEVICE TREE

1.4.5.1 Format specifications

The device tree format is composed of 4 major parts:

1. A header

The header contains information like the size of each section, the version of the DT format (there are several revisions) or the CPU on which the system is booting (in case of multiprocessor systems).

2. A "reserved" section

The reserved section contains a list of memory ranges that are reserved and that the kernel should not use to allocate memory.

3. The actual tree content section. It is described in more details later.

4. A strings database section. It is used to centralize the strings corresponding to property names. As the same properties are often reused, it makes sense to centralize everything in one place to save space.

The interesting section is of course the third, where data is actually encoded. The encoding keeps the tree structure: basically the root item is a node structure. A node can contain properties followed by other nodes (the children of the first node).

Here is the definition of a node structure (for the last version of the DT format) as described in the Linux documentation (figure 1.7).

**Figure 1.7: Extract of the Linux Device Tree documentation**

* token OF_DT_BEGIN_NODE (that is 0x00000001)
* node unit name (or an empty string for the root node)
* [align gap to next 4 bytes boundary]
* for each property:
  * token OF_DT_PROP (that is 0x00000003)
  * 32-bit value of property value size in bytes (or 0 if no value)
  * 32-bit value of offset in string block of property name
  * property value data if any
* [align gap to next 4 bytes boundary]
* [child nodes if any]
* token OF_DT_END_NODE (that is 0x00000002)

It is a pretty straightforward way to build a tree in a binary format.

One thing that we have to notice is that everything is aligned on 32 bits boundaries. Even if a property value is only 1 byte, it will take a minimum of 4 bytes in the blob. The tokens that indicate a node or property boundary also have a 32 bits size, even if the last byte is the only one really used.

1.4.5.2 Size of the blob

In order to have an idea of the magnitude of the size of a typical Device Tree Blob, let’s take an example.
On the BeagleBone Black, a common Cortex A8 development board, the DTB size is of approximately 18kB. The sources DTS files that compile to that 18kB DTB are (everything combined, including comments) 47kB in size.

Immediately we see that this format is not usable as-is because an Arduino Uno (for example) only has 2kB of RAM and 32kB of flash memory. A CC2530 Contiki-powered device has a bit more resources (8kB of RAM and from 32 to 256kB of flash) but the raw DTB still seems to be too big for this kind of devices.

Having realised this, there are two solutions:

- Abandon the Device Tree idea and find another mechanism, or
- Adapt the DTB format to fit it to our needs.

In this study we persist in the idea of using the Device Tree, but with some modifications. To understand what changes will be needed, let’s study the specific constraints of our targeted devices.
Chapter 2

Adaptation to embedded devices

In this chapter we study the specific constraints of microcontrollers related to a future Device Tree port or adaptation. We also try to deduce and plan the needed modifications for both the Device Tree system and the operating system or framework running on the microcontroller.

2.1 Specific constraint for small embedded devices

Small embedded devices have more constraints than those capable of running Linux. The most obvious constraints that come to mind are the code size, memory size.

The devices we target only have about 2kB to 8kB of RAM and about 32kB to 256kB of Flash to store the code. They often contain some EEPROM memory, but only a few kilobytes as well.

The available memory is small, so we must make sure that an implementation of the DT doesn’t use more than a few kilobytes of code, and ideally no more than 1kB for the hardware description.

Code complexity is another point to take care of. As the computing power is low, we can’t afford to waste time on the treatment of the DT. If the format can be simplified in order to reduce its complexity, it should be done. On the other hand, a bit of added complexity is acceptable at boot time, as it doesn’t happen very often in the lifetime of the device (if it is on most of the time anyway).

2.2 Analysis of the existing format

As we already know, the current Device Tree format is too big to fit on an Arduino. But we still want to apply this concept to the platform. What can we change in the DT format to make this possible?

2.2.1 Sample Device Tree file

To make a study on a fair basis, we first created a sample Device Tree that could represent an Arduino board. Indeed, it is not fair to compare the 18kB DTB of the BeagleBone Black (which is a big device) to a smaller Arduino.
Our sample tree will represent a board with an Arduino Uno, an Ethernet port, a temperature sensor on an I2C port and a GPS receiver communicating via a serial port. This sample tree is presented in Appendix 4.3.

The sample DTS file is compiled by the regular DTC, and the result is a 690 bytes binary blob. It is much smaller than the 18kB of the BeagleBone, but it is still close to 1kB, and surely we don’t need that much space to represent the few devices that are present on the board.

To confirm this fact, let’s try to measure the amount of information that is really in this binary file. A simple compression in the GZip format has an output size of barely 328 bytes. It shows that the DTB format has a lot of useless bytes (for instance, there are a lot of zero bytes due to padding) and we can expect to find a format that holds the same amount of information in approximately 350 bytes.

2.2.2 What can be changed

To go deeper, let’s see how we can improve the DTB format to fit our specific purpose.

2.2.2.1 8 bits processors

The DTB format is initially designed for 32 bits processors with no specific size constraints. As a consequence, all the words in the file are:

- 4 bytes long (or a multiple of 4 bytes)
- aligned on 32 bits boundaries (for strings that are not of a specific length)

In the DTB vocabulary, those 4-byte integers are called "cells". A cell is the basic object storage block: An integer fits in a cell, a string has to fit in multiple cells, and the tokens describing the tree structure are also cells.

We can gain a lot of space by suppressing these constraints. On 8-bits processors it doesn’t make sense to align everything on 32 bit boundaries: there is no speed increase. Thus we can remove the zero padding that occurs every time a string or another piece of data is not aligned.

Most of the integers that we have to store will be small: only 1 or 2 bytes. Thus it makes sense to set the default integer size to 1 byte and not to store 3 zero bytes for each one.

To set the default cell size to 1 byte has another consequence: all the tree tokens like OF_DT_BEGIN_NODE (whose value was 0x00000001) will also be stored on 1 byte. This thing only saves at least 6 bytes per node (begin and end tokens) plus 3 bytes per property inside the node.

2.2.2.2 Removing strings

After the removal of unnecessary zeros, what takes most of the place in the binary file will be the various strings. There are three types of strings:

- The 'data' strings, those who are properties data. That can be things like a board name or, more importantly, the nearly mandatory compatible property.
2.2. ANALYSIS OF THE EXISTING FORMAT

• The property names, which are indeed written to the file. As the properties are often the same, there is a special mechanism. All those strings are stored together in a specific 'string' section of the DTB, and what is stored in the tree is actually pointers to those strings. It allows to have two properties with the same name (in different nodes) but to write the string only once. It is a basic space-saving mechanism.

• The node names (or "labels"). Those are meant to be unique to identify the nodes, so they are not grouped in a specific section (like property names) but instead written directly in the node definitions.

In order to save space, we must try to reduce the amount of strings used in the tree. But it is not only a size constraint. The problem with strings is that they introduce more complexity in the DT parsing. They are of variable length, and thus complicate memory allocations.

At a price of a small change in the organization of board development, we propose a solution. Instead of using a globally unique compatible string identifying a certain node, we propose to use company-unique identifiers, and to use fixed-size integers instead of strings as identifiers.

The justification is that a device-tree for IoT devices is meant (in our research) to be used "inside" a certain company to ensure an easier management of its own devices, with firmware developed internally. Thus there is no need that an identifier be globally unique, because a firmware by company A will not run on a board designed by company B (unless they are partners, in which case they can cooperate to avoid problems).

We propose to use a 16-bits identifier to replace the compatible string. By using a single identifier we lose a bit of flexibility, but gain a lot of ease with string parsing (as well as a lot of space).

There are two other kinds of strings left: node identifiers and property identifiers.

We can simply abandon node names and replace them with the compatible identifier. In fact it is the opposite: remove the compatible property and use the node name as a compatible 16-bits identifier. There is a problem: those node names are meant to be unique in the hierarchy (to be able to reference a certain node, for example), so how to represent two nodes with the same compatible identifier?

A solution is to tweak the identifiers to make them unique (using a counter) or we can also remove the constraint of uniqueness. In the implementation we explore the two possibilities and their consequences.

And finally we can replace property identifiers with fixed-size integers without too much concern. The uniqueness of the property is only required within a certain node type. The source file will lose some readability, but it is not a big concern, as we implement a solution later.

2.2.2.3 Simplified structure

A last change to the format is to simplify the overall structure. In the last section we decided to remove the property strings, so the strings section of the DTB is unused: we can delete it.

The reserved section is also of little use: it is meant to be populated by the bootloader for an OS with dynamic memory allocation, to specify memory
addresses which are not to be used. In our case it is unlikely that such a situation will appear: we can remove this section too, and save a few cells which were full of zeros anyway.

As we removed two sections of the DTB format, we can also simplify the header quite a lot: references to those sections are now useless. We can also remove some of the header fields that are not particularly useful, like the id of the CPU used for the boot. Should those properties be needed, we can re-implement them later as properties of the root node.

2.3 Other software constraints

All those possible improvements were only about the DTB file. But the Device Tree is not simply a file, it also has to be parsed and interpreted at runtime. Here are a few other software constraints that must be fulfilled in order to make it work.

2.3.1 Dynamic driver loading

A description of the hardware is great, but when the system boots, it must be able to use it. In particular, it must be able to adapt its configuration, or to load specific drivers, according to the DT information.

This dynamic driver loading can be the normal way of working (Linux), easy to implement (Arduino) or hard to do without changing part of the OS (Contiki).

2.3.2 Same microcontroller architecture

Something that was briefly mentioned but not discussed clearly is that the base platform must be (nearly) the same (at the architecture level) between different sensors. In order to make a common firmware, a base requirement is that the microcontrollers share the same instruction set, otherwise the problems will be far bigger than the gains of the Device Tree.

The ideal situation is that all the devices share the same microcontroller, or at least the same microcontroller core (a few internal peripherals may be different). We have to be careful, because the manufacturer part number can be close between two modules even if they don’t share the same core. For example, the CC2530 and CC2531 are based on a 8051 CPU Core, whereas the CC2538 (only the last digit changes) is a Cortex-M3, which is very different.

Our research will restrict to the case where they are the same, and only the external peripherals vary. We see the microcontroller as a base to connect different sets of sensors with their own appropriate DT.

2.3.3 Separation from the firmware

Another important point is that, in order to benefit from the Device Tree concept, the DTB should be stored in a memory that is independent of the firmware memory.

Otherwise, a firmware update will also update the DTB. And if the DTB is also updated, the updated firmware must contain the right DTB for the device,
which contradict the whole point: a firmware update should be the same for all devices!

The independent memory can be of several types: an external EEPROM or ROM, for example. Or a region of the program flash memory that is untouched by firmware updates (it supposes that the data on this region is readable by the program at runtime).

The constraint with this independent memory is that it must be present and accessible in the exact same way on each device of the same family, because the firmware relies on it to detect variations of the hardware (so the DTB memory itself can’t vary).

2.4 Format specification

In this section we describe the new format specification whose implementation is described in the Prototype chapter.

The new format is based on the regular Device Tree format, but it is used in more constrained environments. For this reason, it is called the Light Device Tree (or LDT for short). It is also sometimes referred to as the "light format" in the rest of the document.

The specification is made of two parts: the source format and the binary format. They are also called the LDTS and LDTB (like the regular "Source" and "Blob" acronyms but with a "L" for "Light").

2.4.1 The Light DTB specification

A complete device tree blob is formed as such:

1. A header
2. The actual tree data
3. Token OF_END_DT = 0x09
4. An optional footer

When stored on multiple bytes, an identifier or number always uses the big-endian convention (most significant byte first).

The header has two 16-bits fields:

1. A "magic identifier" whose value is always 0xFEED
2. A size identifier, which contains the number of bytes in the "tree data" section.

The optional footer is the same but with a "magic" value of 0xDEEF. Its purpose is to be able to place the device tree at the very end of a flash block, and to be able to find the beginning of the tree without troubles if only the end address is known. The 0xFEED value comes from the 0xD00DFEED 32-bits magic field of the regular DTB format. The 0xDEEF value is just a "reversed" value to differentiate the header and footer.

Now let’s describe the actual content, namely the tree data section. The format is very similar to the Linux DTB format except that everything is
CHAPTER 2. ADAPTATION TO EMBEDDED DEVICES

downsized to adapt to 8 bits. The data section actually contains only the root node (which contains its own children, as it is a tree).

The node specification is:

- token OF_BEGIN_NODE = 0x01
- identifier of the node on 16 bits.
- for each property:
  - token OF_LDT_PROP = 0x03
  - 8-bit value of property value size in bytes (or 0 if no value)
  - 16-bit property identifier
  - property value data if any (integers are in big-endian)
- child nodes if any, using the same format recursively
- token OF_END_NODE = 0x02

And finally a few things on node identifiers:

- The node identifiers must be unique (to identify a particular node).
- The root node has the 0x00 identifier.
- The node identifiers as well as property identifiers must have their most significant bit set to 0. We reserve this bit for a possible future extension of the format. It means that their value must be lower than 0x7FFF.

2.4.2 The Light DTS specification

The source format syntax is mainly the same as the regular DTS syntax. Please refer to the DT compiler documentation (in Documentation/dts-format.txt) or the examples above for more details.

There are however a few changes. The header of the file is now /ldts-v1/; (a 'l' is added).

All the node identifiers and property identifiers are now numbers instead of strings. As explained in the binary format section, they are 16 bits integers and the most significant bit must be set to 0. It is interesting to note (and it will be used later) that (0x10 + 1) or (0x7FF0 | 0x03) are valid numbers.

All the labels and references are now ignored (but could be reused in a future version).

It is not in the format specification itself, but it is advised to define the numerical constants in a separate .h header file, to #include it at the beginning of the file and to pass the source file through a C preprocessor before it is processed by the compiler. The purpose of that is explained in 3.3.2.1.
Chapter 3

Prototype

After the initial study, we now apply our ideas to a real prototype which demonstrates the feasibility of the concept.

3.1 Goal and required hardware

We now want to implement the ideas that have been developed in Chapter 2. The goal is to have a working prototype with the following properties:

- based on an existing platform
- two separate boards with different sensors
- each board has its own Device Tree, which is independent of the firmware
- a single firmware, loaded on both boards, must be able to adapt to each configuration and use the sensors correctly

To achieve this goal, there are multiple steps:

- Choose and assemble the hardware
- Develop a Device Tree Compiler (DTC) supporting our new format.
- Develop a library to parse the DTB on the device and possibly adapt the OS
- Develop the demo firmware itself

To minimize the impact of the steps not directly related to the Device Tree, we chose to use an Arduino platform for the first prototype. The hardware is easy to handle and, as we saw in section 1.3.2, the framework modifications should be very small. The firmware itself is also simple to develop, because Arduino was designed for beginners and easy prototyping.

We will need two Arduino Uno boards, a few I2C sensors (like temperature sensors, humidity sensors, but an I2C-controlled LCD screen will also be tested), some SPI devices (a SD card reader).

We will also test analog and GPIO inputs (like a light sensor or a push button). The more diversity in our tested peripherals, the more we can prove our prototype is robust and flexible.
3.2 Compiler implementation

The first step that was completed is to get a compiler working for our Light DT format.

3.2.1 The DTC as a base

As implementing a whole compiler is not an easy task, the implementation was based on the existing DTC coming from the Linux tools. It is free software so the source code is available and we can make changes. The fact that the device tree already had some tools available was a criteria of choice compared to an ACPI-like solution (see section 1.4.1)

The constraint of the licence used for this software (the GPLv2 Licence) is that our modified source code must be published (with the same license) if we decide to distribute our version of the compiler.

After a first test version that worked for the Light format but broke a lot of features of the original DTC (most of the unit tests failed), a cleaner version was developed. The goal was to integrate our light format modifications with the DTC while remaining 100% compatible with the original tool. This way our changes could be integrated in the original DTC project.

At the tool user level, our modifications are simple. The user just needs to add a \texttt{--light} (or \texttt{-l} for short) on the compiler command line to switch to the light format. Otherwise the tool expect a regular DTS file and outputs a regular DTB. The compiler can also switch automatically to the light format if it detects a correct header in the input file: \texttt{/ldts-v1/} instead of \texttt{/dts-v1/} for a regular DTS.

3.2.2 Frontend and backend implementations

The compiler itself was quite well structured and a new format was easy to implement. As a regular compiler, there is a frontend, that parses input files and produces an intermediate representation, and a backend, that produces the output file given the intermediate representation.

For a regular DT, the input can be a DTS file but also a DTB file or a special Linux folder representing the Device Tree of the running kernel. The output can be a simplified DTS file (to debug the DTB given at input), a DTB file, or a special assembly (ASM) file to be included in a kernel build.

We only implemented a Light DTS front-end and a Light DTB output. This was most of the work done on the compiler, although there are a few changes in the middle. Some tests were done on the intermediate representation (called the "live" tree because it lies in memory). We had to adapt those tests to the new format, because of course some of these didn’t pass anymore and made the compiler abort.

The front-end required a few changes in the parser (to accept numbers where strings were needed in the first place, for example). The intermediate representation also had to be slightly adapted to store those numbers.

The backend was also modified to comply with the Light DTB specification.
The tree structure itself has not changed that much (except for numbers instead of strings), so the changes were more about changing the output integer size and removing padding. Of course the whole structure has changed a bit too: a smaller header, some sections were removed, ...

There is one additional output format, that can prove useful in some cases. In the early firmware development, when the DT was not yet separate from the firmware, the DT was included in the code, and recompiled with every change. The way it was done was to store the binary data of the DTB in a C string in its own .c file. For this reason, the "C string format" was also implemented in the compiler, to generate the C file automatically. It was only a different representation of the DTB data.

One additional thing is that the C string is split in chunks that make it easier to debug the binary output (it was useful to find some bugs in the compiler or the firmware parser). Those chunks are even structured visually like a tree to make the reading even easier.

3.3  Arduino toolchain

To develop for Arduino, a toolchain and an IDE are required. But the official IDE is not great, so let’s see how a better toolchain can be setup.

3.3.1  Arduino IDE

The Arduino IDE is not the best there is. Its interface is basically a text editor with a bit of syntax highlighting, but it lacks features like code completion. Another problem is that when there is a compiler error, it is not easy to see where the error is. It is mainly designed for beginners and simple projects, not for developing additional libraries.

In order to be more comfortable with the environment I found a documentation on how to use the Code::Blocks IDE (which is free and open-source) with the Arduino environment. After a bit of setup it was working.

I also found another (more Linux-like) way to develop for Arduino: a project providing a Makefile for Arduino. A Makefile is like a recipe that tells the command-line tool make how to build a binary. This Makefile had two very interesting features: it could upload the freshly built firmware to the device, and open a serial console on the device (two things that the Arduino IDE could do, but not Code::Blocks, which was harder to setup).

And, even more interesting, the build and upload could be automated, for example for tests scripts (which the official IDE couldn’t do), and new features could be added easily. In our case, building a DTB or sending it to the device, for example.

In the end, for the development I used Code::Blocks for the code editing and checking, and the Makefile for firmware building and upload and also to communicate with the Arduino using a serial port.

However, the code is designed to be compiled on any of the three platforms: Makefile, Code::Blocks and Arduino IDE.
3.3.2 Makefile automation

The advantage of the Makefile is that it could be modified easily to add features like the Device Tree compiling and upload to the device.

3.3.2.1 Constants file separation

As mentioned briefly in 2.4.2, the file format with only numbers as node and property identifiers is not really readable by humans. To make it easier, we define a .h file with constant definition macros, so that each identifier can be represented by a human-readable constant.

The source file containing constants is processed by the C preprocessor before feeding it to the DTC. This way, the file processed by the DTC only contains numbers.

The advantage of separating the source in two files (the .dts actual source and the .h macro definition file) is that the .h include file can also be reused in the firmware source code. As the device tree constants must be reused at some point in the code, it is good to be able to reuse the constants there too.

The constraint with this constants file is that the same file must be used by both the hardware designer (who produces the device tree) and the software developer (who uses that device tree). For this reason, every change in the .h file must be avoided if it impacts already deployed devices (or future updates won’t recognize the hardware).

What is great with the Makefile build solution is that we can add new build rules easily. The DTB generation, which is split into preprocessing and actual DTB compiling, has been integrated into it and rebuilding a DTB file is as easy as typing `make dt`. The firmware itself is build by simply typing `make` and both use the same .h include file which is common to all test or development projects.

3.3.2.2 Firmware upload

In the same way that it is possible to build a light DT as easily as a firmware, the upload of a DT file to the board has been automated based on the firmware upload mechanism that is already present in the Makefile.

We mentioned earlier that it was necessary to have a separated storage, to update the firmware independently of the DT, but we haven’t searched a solution for Arduino yet.

We saw that the Arduino had two persistent memory devices: the flash memory (32 kB) and the EEPROM, that is smaller (1kB) and slower.

The EEPROM is meant to store data that can vary during the program execution and that must be retained in case of a power cut. It is writeable by the program but is quite slow.

The flash memory is meant to store the program itself. However, on Arduino there is a way to access it as a read-only memory. This mechanism is often used to store constant values that would be too big to fit in RAM (like long strings of computing tables). This way the variables are read directly from the flash without being copied to the RAM. This kind of read-only storage is great for our use case: the DTB will not change.

An obstacle is that, usually, the data stored in the flash memory is included in the firmware at compile time. What we want is actually a way to include
data outside of any firmware compilation. Is it possible in the Arduino model? Actually yes. The Arduino bootloader, that is used to flash devices, can flash data at arbitrary address location, without erasing the whole flash at each firmware change.

What we can do is to store the DTB in a memory location that is never overridden by the firmware upload. The Arduino memory layout is simple: the firmware is flashed in one block starting from address 0x0000. It means that the end of the flash memory is not used unless the firmware is so big that it needs the whole memory. We can use this fact to store the DTB at the end of the memory. There is no problem as long as the firmware is small enough, and this implementation reduces the maximum firmware size.

This mechanism has been implemented as a Makefile rule: to upload a freshly-built DTB to the board, the command make dt-upload is enough (and it is similar to a regular firmware upload: make upload).

3.4 Firmware Implementation

In this section we describe the implementation of the Device Tree usage in the board firmware itself.

3.4.1 Code structure

The code we implement must be as generic as possible, because even if we first focus on Arduino, we want to be able to port it to other platforms later.

It means that, even if most of the Arduino code is C++, we can’t use this language because many framework for embedded platforms only support C. As Arduino also supports C, it is the language of choice to implement our prototype.

However, many Arduino idioms will be violated by using C. For example, most of the Arduino libraries are made so that the end user never has to write a code with a pointer: C++ references and other C++ constructs are used instead. But it is very unlikely that such a low-level code can be written in C without any pointer.

We also want our Device Tree code to be as independent of the Arduino environment as possible, again to be able to reuse the code. The code will be written as a standalone library, to be modular and to suppress dependencies to a particular end-user program. However we can’t avoid a few dependencies to Arduino or hardware-specific constructs (low-level functions to access the DT data, for example). Those parts containing dependencies should be separated in functions (some sort of a light hardware abstraction layer) that can easily be reimplemented for other systems.

To avoid potential conflicts with other libraries, all functions and data types we implement for this library will have the same ldt_ prefix: for instance some functions will be named ldt_init(), ldt_parse, ...

To be clear and avoid confusion, only the Light Device Tree (LDT) parser library will be written in C. All the Arduino drivers are written in C++ so the end-user Arduino program will still be in C++, but using our C library, which itself will be portable.
3.4.2 Sample Programs behavior

Before we go further, let’s describe what our test program will do. We already know that it will need to adapt to different sensors, but to do what?

Actually, during the development, there were multiple program/firmwares developed to test various things with our Device Tree implementation. Let’s describe them here.

3.4.2.1 Simple programs

First there is a simple debug program. Its goal is simply to make a dump of all the detected nodes in the device tree. It is useful for debug purposes, to make sure that the data on the device is actually what we think it is.

The second simplest program is the mandatory ‘Hello World’ program. It is meant to be an example of how to use the device tree library for Arduino programmers that want to learn it. It is very basic: if it detects a certain property in the root node of the tree, it prints it to the serial port and stops. It can also detect a property in a child node and print it too.

3.4.3 Development program

The next program is the one that was used during most of the development. It is called the ‘Sensors’ program.

Its behaviour is quite simple. It has many sensors connected as well as a few outputs: a hardware serial interface, another serial interface emulated with software bit-banging, a LCD screen, ...

The goal is to select one sensor and print this sensor data to all outputs at the same time. The selected sensor can be chosen using a push button, or changes automatically every 5 seconds if no button is detected (using the device tree, of course).

The connected sensors are of many different types, and the more types tested, the better the library is guaranteed to perform. The sensors are things like temperature sensors (both I^2C and analog), humidity sensors (a custom interface using one signal wire), a pressure sensor (an I^2C barometer that can also sense temperature), a light sensor (with an analog interface), a push button.

There were also some output devices like actuators (a stepper motor, a relay) and of course the I^2C LCD screen and the two serial inputs/outputs.

There are actually two versions of the "Sensors" firmware. One that only does what is described above, and one other that includes code to perform various measurements.

The measurements code include things like debug output (to verify a certain node has indeed been detected), timing measurement (to measure the overhead caused by the device tree), or free RAM measurement (to ensure that the few dynamic allocations don’t fill up the memory).

The last feature of that the development code is that, using some #ifdef statements, we can decide not to compile certain parts of the code to evaluate the impact of each section (or each "feature") on the total code size. The code size is important because it helps evaluate the complexity of the program and it is a limited resource (limited by both the flash memory size and the device tree, which is also stored on the flash memory).
Most of the figures that are presented in the result section come from this prototype.

3.4.3.1 Realistic prototype

And finally there is the prototype we were presenting at the beginning as the objective: to have two different boards run the same firmware.

This prototype’s main functionality will be to log the data coming from different sensors. The sensors are very similar to what is presented above. An additional GPS sensor is also used on one board, its data is available on a serial bus. The logging part can be done on an SD card (which uses the SPI bus) or more simply on a serial bus if no SD card is available.

The exact sensors or the presence of a SD card are the things that will vary between prototypes and that will therefore be listed in the Device Tree.

3.4.4 Callback system

Now let’s focus on the library code itself. The library will mainly consist of a Device Tree Blob parser. The role of the library is to allow the user to access this data in a structured and efficient way. How to organize the API to achieve that goal?

The "user", again, is actually the Arduino programmer, it is the user in the sense of 'user of our library', or writer of 'user-mode' code as opposed to drivers code (even if there is no "user" or "kernel" mode in Arduino).

The first solution that has been implemented is to use a callback system. The code flow is the following:

- The user initializes the DT parser that checks that the format is correct.
- The user starts the parsing by supplying a callback function, let’s call it user_callback
- For each node that the parser finds, the user_callback function is called with a pointer to the current node as a parameter.
- inside user_callback, the user checks what is the node type and instantiates the correct driver that corresponds to this device. Actions like finding the node type or reading properties of the node can be done using helper functions that are also provided by our library.

With this approach, the DT library is a pure parser. It provides data to the user, but the user is still responsible of the driver instantiation or device initialisation.

It provides a lot of flexibility to the user, who can do anything he wishes with the data. On the other hand, the user has the responsibility to handle dynamic driver instantiation, which is quite repetitive.

First let’s see how to handle dynamic "drivers" or dynamic input sensors in Arduino, and what are the challenges that the user has to solve to implement it efficiently. Then the section after that will present a second way to handle the device tree. That second way was implemented later to reduce the driver managment hassle.
3.4.5 Arduino dynamic driver management

3.4.5.1 Different kinds of drivers

There are multiple cases of Arduino peripherals when it comes to "driver" management.

First there are some simple peripherals whose value can be read with a simple call to the Arduino library, any time, without special initialisation. It is mostly the case for GPIOs: they don’t really have a "driver" as the calls are basically wrappers around registers read/writes (although they still must be setup by another simple call to the library). The analog input or PWM outputs are also in the same case: they are very simple and do not really require a "driver" initialisation beyond what is already in the Arduino core library.

Then there are the more complex peripherals. They can be peripherals included in the Arduino but not part of the core (like the I2C driver) or third-party peripherals (like our LCD screen). These drivers are libraries that are separate from the core and that the user chooses to include or not at compile time.

They almost always consist of a C++ class, but the usage vary from class to class. Some of them take their parameters in the constructor, while others prefer to have a default constructor and a `begin(parameters params)` method to perform initialisation tasks. For most of them one class represents one device and you can have multiple instances of the same type (if multiple devices are connected). But sometimes a class can’t be instantiated multiple times, for various reasons (because the hardware can’t be present more than once, for instance).

In the next sections we study how to adapt to each case to allow dynamic driver loading. But in order to do that, we need to identify the problems caused by the memory management.

3.4.5.2 Memory management

On embedded devices such as Arduinos, the RAM memory is a scarce resource which should be managed with care. More specifically, we should avoid dynamic memory allocation on the heap, because they are not easily predictable, can lead to memory fragmentation, etc... Memory fragmentation is especially bad on microcontrollers which have no virtual memory support, such as our Arduinos.

Most of the memory should be allocated either statically (at compile time, using global variables) or on the stack (functions local variables), because those two mechanisms no not lead to fragmentation and the compiler can (to a certain extent when it comes to stack memory) predict the amount of used memory and make sure that the firmware has enough free space.

However, our goal is to instantiate drivers dynamically, so it will be hard to make everything static. There are two solutions to do dynamic instantiation: pre-allocate empty slots in global memory, and use them only when they are really needed, or do dynamic allocation anyway, using `malloc` or `new`.

Let’s imagine a case where we have peripherals of two types, A and B. We can either do static allocation (for example reserve slots for a maximum of 3 A and 3 B devices) of dynamic allocation.
Obviously dynamic allocation is more flexible, because it only uses memory when required and also covers the case of 4 A devices and no B, as long as there is enough memory.

Usually dynamic allocation is bad for embedded systems without virtual memory due to fragmentation. In our case, however, we can make an exception on certain conditions. As the memory is allocated at boot-time for drivers that will always be in use, the memory is never de-allocated, so there is no risk of fragmentation (only of memory exhaustion). All the drivers allocations are made in one block at boot time. Allocations that are made at runtime may fragment the memory, but they are not related to the DT.

To sum up, we allow certain drivers to be loaded using `malloc` or `new`, as long as it is done once, at boot time.

### 3.4.6 LDT adaptations for multiple drivers

In our previous specification of the LDT, we removed the compatible string to replace the device identifier by the node identifier. In theory we have a mapping: 1 node identifier $\leftrightarrow$ 1 node type. But what if we want to include multiple devices of the same type? For example, multiple LEDs on different GPIOs, or multiple bus of the same type (the Arduino Mega has 4 serial ports for example)?

We addressed that point on 2.2.2.2 and concluded that we would discuss it in detail here, in the implementation.

There were two solutions: remove the constraint of uniqueness of the node ids, or keep it but introduce a way to differentiate two nodes of the same type.

The problem with removing the uniqueness is that, when using sub-nodes (of a bus, for example), we have to keep a reference to the parent node to be able to identify it. If two parents have the same node id, it is harder to identify them. We could re-introduce a one-byte property in the identical nodes to act as a counter or identifier, but it adds complexity to the LDT where we previously tried to remove it.

The other way was to re-introduce a difference in the 16-bits identifier. It is unlikely that a family of devices will have to support 65535 types of peripherals, so we can reserve a few bits in the identifier to do something else than identify the peripheral type. For example, with a 12 bits for the identifier, we still have 4096 different types of sensors, and we now have 4 bits to uniquely identify 16 devices of the same type using a counter.

In fact, some drivers don’t need to have identifying bits, for example if the peripheral can only be instantiated once. For this reason, and to avoid wasting identifiers, we decided that each driver could define a variable number of bits containing the identifier counter.

Practically, each driver has an identifier (the type of the node that it handles) and a bit mask (which will be applied before comparison to determine the driver).

How to use this in the device tree? Well the DTS format already supports basic bitwise operators, which we can use to manipulate the counter. As an example, on figure 3.1 is an extract of a (non-pre-processed) DTS using two DHT temperature sensors.

One final precaution: as we reduce the number of potential node types, and we don’t want to be blocked if one day they are all used, we keep a way to
extend the node type system. We reserve the most significant bit for a future use: for now it must always be zero. This way, in the future, we can define a new node type, for example on 32 bits with the first bit at '1', if we need more space.

This property was implemented in the compiler, which for now simply issues a warning if the condition is violated.

### 3.4.6.1 Single or multiple drivers

We already mentioned that some drivers can be instantiated multiple times, while some can only be loaded once. Let’s see how both types are managed.

The single drivers can be represented by a variable. That variable is either not initialized (no peripheral is present) or initialized properly (a driver has been loaded). If it is not possible to distinguish between an uninitialized and a loaded state, for example if even 0 or -1 can have a significant value to the driver, a second boolean variable can be used to store the initialization state of the driver.

The single drivers are straightforward, but the multiple drivers can be more challenging. If we want to be able to load as many drivers as we want, we need a way to allocate dynamically a list of loaded drivers. That could be done using a linked list or another dynamic structure.

But a linked list requires a lot of dynamic allocation and adds to the code complexity. Even if we saw that we could use dynamic allocation, it is better to use as few of them as possible. Instead, we simply used a fixed-size array and a counter stating how many items in the array are actually initialized. For example for a light sensor, the variables look like this:

```c
struct light_sensor light_sensors[LDT_MAX_LIGHT_SENSORS];
uint8_t light_sensor_count = 0;
```

The `LDT_MAX_LIGHT_SENSORS` actually limits the number of light sensors that can be loaded, but it is a trade-off between flexibility and memory usage/code simplicity.

The typical driver instantiation for a multiple drivers system looks like this:

1. Check that the driver count is below the maximum defined driver number.
2. Initialize the driver in the `count`-th slot in the array
3. If initialization was successful, increment the count and return success

4. Otherwise, clean up and return failure

Now that we have an overview of how drivers are managed, let’s see in more details how the C++ drivers are instantiated. Indeed, originally they are only C++ classes designed (by the Arduino team or third-party vendors) to be declared statically. Do they all handle dynamic initialization well?

### 3.4.6.2 C++ class drivers

There are two kind of driver classes for Arduino. The ones that take initialisation parameters in the constructors and the ones that have only a generic constructor and a `begin()` function that takes those parameters. For example, see figure 3.2.

**Figure 3.2: Two examples of class drivers**

```cpp
1 Servo myservo; // create servo object to control a servo
2 //create a stepper motor on pins 8,9,10,11
3 Stepper stepper(STEPS, 8, 9, 10, 11);
4
5 void setup()
6 {
7     // attaches the servo on pin 9 to the servo object
8     myservo.attach(9);
9     stepper.setSpeed(30);
10 }
```

In this example, the Stepper class takes the device pins in the C++ constructor, and the Servo can wait the `setup()` function at runtime to initialize them.

With this mechanism, in C++, there is no way that we can declare a global array of Stepper objects without initializing them, that is without knowing in advance the parameters to their constructors.

To fit in the array model we described, there are two possibilities: in the array we can store classes or pointers to classes. In the case of classes, we already saw that we need to know parameters of the constructors at compile time, which is not very dynamic. So the only remaining solution is to store pointers to a class, and to dynamically instantiate those classes. This is also the reason we couldn’t have used only static allocation.

To sum up, for drivers represented by C++ classes:

- If the given parameters are correct, we allocate a new driver with `new`.
- We check that the returned pointer is non-null, and add the class to the table (or, if it is a single driver, there is no array but one global pointer)
- The object is used normally through the pointer.
3.4.7 Advanced callback system

The concept described here was not implemented in the first prototype, but was tested afterwards.

The concepts described above, with the class arrays, the LDT parser and the user callback system, are enough to develop a working prototype. The strength of such a design is that it can use nearly any Arduino driver (core or third-party) with no modification to its code. The driver designer doesn’t need to know about the DT for the driver to work with it eventually.

But during the design of the programs, it appeared clearly that the tasks of checking the properties, instantiating the drivers, managing the array lengths, ... was very repetitive for the programmer, and could be improved.

We came up with a second, more elaborate system. The problem with the current system is that the user has to manage the callback function, which for each node:

1. has to determine its type
2. must know which driver is related
3. must instantiate it by hand, which means checking the array lengths, managing error cases, ...

The idea is that instead of having the user manage and register every single device, we could let the driver do it. The user will only need to load the driver once before starting the DT parsing. The callback function itself is no longer written by the user, but it is moved in the DT library. The initialisation, which was previously done by the user, is now handled by the driver itself. The array which stores the driver instances is also declared by the driver files.

As a consequence, the user has much less to worry about: one line is enough to load the driver. The use of the peripherals, however, doesn’t change compared to the previous system.

On the other hand, the driver now has to be LDT-aware: it effectively transfers the LDT initialisation workload to the driver developer instead of the driver user. The good news is that, if a driver has not been modified by the author, most of the time it is still possible for the user to use it with the old LDT callback system.

On the other hand, it is also possible for the developer to develop a driver so that it will work with non-LDT systems. With this kind of low-coupling, the driver compatibility should not be a problem preventing the LDT from being adopted.

But before trying to get it adopted, let’s see how it performs on the prototypes.

3.5 Results and measurements

3.5.1 Results

Finally, the different prototypes are working. On a functional level, the sample programs described in 3.4.2 have been implemented and work as expected.
3.5. RESULTS AND MEASUREMENTS

3.5.1.1 Prototypes
In particular, the 'two boards' prototype logger has been tested with on one board: a GPS and a SD card slot (the data is logged on both the SD card and the debug serial port) and on the second board: two I^2C sensors (air pressure and humidity) and a Servomotor (the data output is only on the debug serial port).

The other important prototype, the 'Sensors' project, is working too, and interesting measurements have been taken. They will be described on the next section.

3.5.1.2 Code and DT integration
The prototype code show promising results.

The first thing to notice is that there was no need to modify the Arduino code in depth. It is sufficient to develop the LDT code as a library and to do the needed changes at the user code level.

The only changes that have been made to the Arduino project are in fact in a few driver libraries (like the Stepper library), that were not mandatory but helped save some memory (de-duplication of some variables) and simplify the user code.

Of course, as we mentioned in the 'advanced callback system', the driver libraries must be adapted to work with the LDT, so in this model the Arduino base project will be modified, but the changes only concern the optional drivers, not the 'core' Arduino library.

3.5.2 Measurements
The measurement results are encouraging.

3.5.2.1 DTB size
The first parameter is the size of the light DTB describing our real prototype. Remember: a fake non-light DTB for Arduino was about 700 bytes big. Now with the same content, the Light format file is about 70 bytes. In all our experiments, various LDTB files have been used, and none was bigger than 200 bytes (for a realistic board anyway). Average files (for the 'Logger' prototype for example) are smaller than 90 bytes, so the size reduction is quite good.

3.5.2.2 Parsing speed
Then there is the runtime impact: how does the DT influence the boot time? Actually there is nearly no difference. The DT is stored in flash memory, which is a bit slower than the SRAM memory, but still much faster than an external EEPROM for example. The total added boot time has been measured to be about 500µs, which is negligible when a single I^2C peripheral sometimes takes 1ms or 10ms to initialize. The boot time really isn’t a concern with this implementation. Raw numbers for a single measurement are presented on figure 3.3.

To go a bit further, we can study the case of a DTB file stored on an external memory. In that case, the parsing time will be longer and dominated
Initialisation total time = 176 ms
Including:
38 ms for serial out printing
137 ms for driver init
=> 0.476 ms for LDT parsing

by the reading delays. Here is another raw figure: for a DTB size of 95 bytes, one particular firmware read a total of 197 bytes in the file during the boot. According to this, on average, one byte in the DTB is accessed twice during a full parsing, and it is easy to estimate the induced delay.

3.5.2.3 RAM usage

The RAM impact of the LDT library itself is basically zero. No variable is allocated in the heap during parsing, the only variables are temporary (because located in the stack) so after the parsing has ended there is no impact at all. This is possible because the DTB content is accessible anytime (using special instructions and a separate address space, but with basically no delay on the Arduino model that we used), so variables to store the DT values are unnecessary.

What has not been measured, and it is certainly non-null, is the impact of regrouping multiple functionalities in a single firmware. It is hard to evaluate in general, because it depends a lot on what those functionalities are. In general, a good approximation can be obtained by counting the amount of global variables (and more importantly the number and size of global classes). When using some dynamic allocation for the drivers, however, the memory allocation will depend on the actual devices that are present, and not really on the maximum possible number of devices.

3.5.2.4 Program size

The next point to be measured is the size of the code to flash on the microcontroller.

In general, the Arduino code is not extremely optimized for size, even inside the framework core. When using Arduinos it is often easier to use a more powerful Arduino than to try to decrease the code size. But it is not always the best choice, both for cost reasons and because, for other platforms, it may not be as easy to use another microcontroller.

The measurements for the size of the code were done using a special 'modular' program containing many #ifdef allowing most parts of the code to be conditionally compiled. The program is based on the 'Sensors' example, with some added conditional debug output.

A lot of tests have been done but there are 4 main different firmwares that allow us to draw conclusions:
1. A program with all sensor libraries included, but no "dynamic" instantiation. All drivers are static global variables and can’t be changed at runtime.

2. A program with the same functionalities but with "dynamic" instantiation. Drivers are loaded at boot time by a function that doesn’t include the device tree library yet (so the loaded drivers are determined at compile time).

3. A program similar to the last one, but with the Device Tree library included. However the DT data is not actually used: this program is used to determine the impact of the library itself, not of the code/structure changes that the rest of the firmware needs to include to actually use the DT data.

4. A program with dynamic instantiation which uses the device tree to choose the drivers to load: this is the full Device Tree firmware functionality.

Note that the supported drivers/peripherals across all those programs are the same. The only thing that changes is the dynamic loading and the DT usage.

The results of the measurements are (for a total flash size of 32kB) on figure 3.4.

<table>
<thead>
<tr>
<th>Firmware number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (Bytes)</td>
<td>11706</td>
<td>12946</td>
<td>13920</td>
<td>14736</td>
</tr>
</tbody>
</table>

Here are some interesting conclusions we can draw from these numbers:

- The LDT library itself adds approximately 1kB.
- Making the driver loading dynamic also adds 1.2kB, for no real added functionality.
- The whole process of the LDT integration (library size and code structure changes) add a total of +3kB on a 11.7 kB program. The size change is a little over +25%, which is not negligible.

Of all those measurements, the code size is the one that shows the most changes. It is certainly the main constraint to take into account when considering modifying some existing code to use the device tree on a given microcontroller. And this series of measurement doesn’t even address the size increase due to the merging of multiple firmware functionalities into a single firmware, which will also add to the total size.

3.6 Conclusion of the prototype experiment

The prototype demonstrated that the Device Tree concept can be successfully applied to small devices such as microcontrollers, at least those using the
Arduino framework. Multiple peripheral drivers can be instantiated at runtime using the information contained in a DTB file that is separate from the firmware, and tied to the hardware.

That firmware must be adapted, and the measurements show that the principal constraint of the first callback mechanism is the increased size of the firmware.

To decrease the workload of a developer wishing to implement a Device Tree system in his program, the second "advanced" callback system has been introduced to shift part of the device tree integration towards the driver developers. This system has not been measured because it has been developed after the series of measurements and it would require even more changes to each used driver to make it possible. Besides, a lot of the code has simply been moved to the driver files without significant change, so the overall size should not vary too much (even if a small increase is probable).

Now that we have developed a new format, a tool to work with it and a working prototype, let's see what can be done to go further.
Chapter 4

Going further

4.1 Other platforms: Contiki

We already mentioned that Arduino was a nice and simple prototyping and
development platform, but it is not as professional as, for instance, Contiki
which is a real operating system (as opposed to Arduino which is more like a
framework).

The problem is now to know whether the Device Tree can be ported to
Contiki and used effectively in this environment. We initially developed an
Arduino prototype because Contiki seemed harder to modify, and we preferred
to focus on the DT itself rather than on the Contiki-specific problems.

However it doesn’t mean that Contiki can’t be modified to use the device
tree. In this section we describe a study of what is needed to port the light
device tree to Contiki.

First we have to mention that the LDT library itself was designed to be
portable: written in C instead of C++ precisely to be ported to other devices,
also has a modular design that tries to separate the OS-specific memory
access primitives from the rest of the code.

As a proof of this modularity, we demonstrated a quick prototype of a Con-
tiki node compiled with the LDT library displaying some DT information on
boot. This demonstration, performed on a TI CC2538 (Cortex-M3) develop-
ment kit, was very basic:

- the DTB was included in a binary string inside the program, so not sepa-
rated from the firmware
- the DTB data was not used to load any driver, just to print a string to
  the serial output

But nevertheless it is a proof that the LDT library itself is very easy to port
and compile on another platform.

The two main challenges to make the DT work on Contiki are thus related
to the listed demonstration limitations: separate the DTB from the firmware,
and (the biggest one) actually instantiate drivers according to the DT content.
Let’s study on a high level how to solve these problems.

The first one, the device tree blob separation from the firmware, should not
be very difficult. The used CC2538 microcontroller has more flash memory
than an Arduino (256kB on the device used here), and it supports In System Programming (ISP).

It doesn’t seem to be implemented in Contiki, but the ISP capability should allow the firmware to read arbitrary data on the flash memory. It should be possible to implement the same DT separation as for Arduino: a DT blob stored in flash at a fixed address (at the end of the memory for example).

A point that needs to be addressed is to see if the bootloader (or the programming system if the bootloader is unused) allows to flash only a specific location of the memory without erasing the whole memory. It should not be the biggest problem as this kind of capability is quite widespread.

The second one is the hardest. How to modify the code so that the drivers can be loaded at boot time, depending on the DT content?

We already saw that there is a global static variable containing a sensor list. We need to modify it so that the list is dynamic. Whether the best implementation is a linked-list or an array with a maximum size is not the hardest problem.

What must really be studied is to know how much code in the OS depend on this variable being static and defined at compile time. And of course this is not the only variable that needs to be changed. A lot of the code is using #ifdef statements depending on the sensors support being compiled in or not. These static statements will have to be replaced by dynamic if() statements testing variables that are populated by the DT library.

On a code structure level, the DT library will have to follow the Contiki structure:

- in core/lib/ lies the main LDT library code. This code is common to all platforms.
- in cpu/ lies the code that is specific to an architecture. For example the ARM-specific methods to access the DTB stored in flash memory should be implemented here.
- in platform/, a file contains the main() function that is specific to each board. The DT initialisation code should be placed here, as early as possible in the main() function, before most drivers are initialized.
- in core/dev/, cpu/dev/ and platform/dev/, the drivers are present and separated in global, CPU-specific and board-specific code. Each driver that we want to work with the DT will have to be modified at those locations. Most of the code changes should be in the global file, because we want the DT to work as broadly as possible, but depending on the driver some platform code may need to be adapted too.

Of course in this model, we will need to adapt the drivers we need to make them dynamically loadable. As they will be modified anyway, the cost of adding some DT-specific code in each driver will be low, so it is better to use the 'Advanced' callback system, as opposed to the first callback system that needed no driver change (for Arduino code) but was more complex for the end user.

Contiki (being an OS) is more complex and more structured than Arduino anyway, so the additional complexity in each driver is acceptable.
4.2. DEVICE TREE HEADER FILE GENERATOR

As a conclusion, porting the Device Tree to Contiki seems feasible now that a modular implementation exist for Arduino. In the context of this project, only a pre-study has been done, because most of the work would have been redundant with the work already done for the Arduino prototype, and the academic value of implementing the same thing once more was low (although there are some interesting new challenges). Instead of developing a Contiki port, the Arduino system was improved, for example with the "advanced" callback system which was actually developed after this Contiki study.

4.2 Device Tree header file generator

Another concept that could be studied is at the toolchain level. We mentioned that the LDT compiler used a .h definition file to convert readable constants into 16 bits integers before compiling the file to the DTB format. The same .h file is then used in the firmware code, to reuse the exact same constants and make sure that everything is consistent.

The advantage is that the constants are centralized and they are guaranteed to be the same between the DTB and the firmware. Well, as long as the file used to make the DTB is the same as the one used for the firmware.

The problem is that the file will have a tendency to grow when new devices are added, and it will be easier to make a mistake and use twice the same device identifier for two different devices. To avoid that, there are multiple solutions. We could decide of a certain process to make sure that the identifiers are unique, such as simply increasing the identifier number by one for each new device. But then there are problems because a certain number of peripherals need to reserve a range of identifiers (to differentiate two sensors of the same type).

One solution to explore could be to create a small software that will manage those constants for the user. To add a device, adding a string constant and the number of expected devices of the same type would be enough and the program would generate the rest of the file. No numerical constant would be typed in by the user, so there is no identifier reuse problem.

One other problem is how to manage the DT constants if this Light DT becomes widespread (in a Contiki implementation for example). Initially, the DT was meant to be used inside a single company (to manage its own device fleet) so there is only one entity that maintains the DT constants file. But how could separate entities collaborate to maintain a coherent naming space ?

This problem, while it does exist in the Linux (non-light) version of the device tree, is less apparent in that case because the name system uses strings that can have entity-specific prefixes. But in our case, with only 16 bits of name space, the problem is harder. We could re-introduce strings, but the whole point of the light DT was to save space by removing them. We could also use the (now reserved) first bit to introduce a new naming scheme. But which one, in that case ?

Such identifier system is fine while a low number of peripherals are used inside a single entity, but interesting problems are raised if we get out of this scope, and more study must be done, should the Light DT become popular among multiple entities.
4.3 Shield/Cape/Hat-like system?

Finally, a last possible lead to improve the system is to study a possible extension boards system similar to what is described in 1.4.3.2.

On platforms such as the BeagleBone Black or the Raspberry Pi, external extension cards can be plugged (they are called 'capes' or 'hats' depending on the platform). Such board contain a small piece of DTB that is merged with the motherboard main DTB by the bootloader, and the result is passed to the Linux kernel.

But the Arduino platform also support extension cards, called "Shields", and there is currently no support for those in the LDT.

Of course a Shield can still be used: in fact the SD card reader used in the Logger demo was on a Shield. But it means that when we change the shield on a board, we currently have to re-flash the DTB on that board to indicate that the hardware has changed.

That way of operation is not optimal, so an improvement could be to embed a small DTB part on the shield. That small part could be read on boot and merged with the main DTB, so that a Shield change automatically implies a device tree change, with no flash required.

For Contiki there is no such normalized extension board right now (because Contiki runs on a lot of heterogeneous hardware), but it doesn’t mean that the concept can’t be used for something else.

For example let’s imagine a commercial wireless sensor product with a common wireless platform, but no actual sensor. The sensor itself can be a small part to insert into the device, so that the same device can sense different parameters and be reconfigured at the customer’s will (if he buys multiple sensor extensions). A single product could even support 2 or 3 sensors extensions of different types. In that context, the extension part could contain both the sensor and a small EEPROM with a partial DTB so that the main platform automatically recognizes the sensor and knows how to access it. No further configuration is needed for the customer.
Conclusion

In this project we introduced a flexible way to adapt an embedded system so that a single firmware can run on several hardware configurations based on the same core.

This mechanism was already present on other platforms such as the x86 computer architecture or the ARM architecture for Linux running on powerful embedded platforms. However no such system existed for small devices running sensor networks or other mobile devices of the same size.

We solved the problem for the small devices by creating a new and lighter Device Tree format, based on the original Device Tree used on Linux. This new format is suitable for microcontrollers due to a massive size reduction and format simplification.

To use this new format, a compiler was implemented and integrated into a toolchain. A device-side library was also developed with portability in mind. A prototype has been developed to prove that the library was working and also to detect problems or things that could be improved.

The prototype was based on the Arduino platform because, of the few we studied, it was the easiest to adapt. The prototype was successful and measurements showed that the runtime impact of the Device Tree was negligible. However, the main constraint is the code size, that significantly increased with the integration of the Device Tree.

After a few refinements on the Arduino prototype, like the new callback system that simplifies the use of the DT for the final developer, we tried to prepare the port of the Device Tree to another platform. The Contiki OS was analysed and the main tasks have been identified in order to port the device tree to this operating system.

Finally we presented a few development axis that could be explored to improve the system in order to make it more complete and more flexible.
Appendix A

This appendix presents the evolution of the LDT format in terms of syntax, usability and resulting blob size.

The first figure (Figure 1) presents a valid DTS file that could be used to represent an Arduino test board with a few sensors if that format were used with an Arduino.

The next figure (Figure 2) is the resulting DTB binary file, using the regular non-light format. We observe a lot of zero bytes, many strings and a data format that is not really dense.

The third figure is the light-format file corresponding to the first DTS example. The content is basically the same but the syntax is a bit different. All strings are replaced with numeric constants which are defined in a separate C header file.

The fourth figure is the light DT blob corresponding to the last source example. Notice how it is much more compact than the regular DTB file (there are not many 00 bytes remaining), but contains approximately the same information. The `compatible = "some,string"` information has been replaced with the node identifiers.

And finally the last figure presents the exact same binary data but presented using the new `-O c-dtb` compiler output format. This format is a C string (which can be compiled into a firmware if included in the right file). But it can be used for debugging, as the output is formatted so that the initial tree structure is still visible.
### APPENDIX A

**Figure 1:** A fake Device Tree describing the peripherals of a test Arduino board configuration

```dts-v1;
{
compatible="ar,mega2560";

spi0 {
  compatible = "spi";
  #reserved-pins=/bits/8
    < 0 /*clk*/
    1 /*master out*/
    2 /*master in*/
    3 /*"chip select" (on arduino, must be kept as output even if unused)*/
};
sdcard {
  compatible = "ar,sdcard";
  #select-pin = /bits/8 <4>;
};
eternet {
  compatible = "arshld,eth";
  #select-pin = /bits/8 <10>;
  mac = /bits/8 <0x01 0x23 0x45 0x67 0x89 0xab>;
};

tart0 {
  compatible = "ar,hw-uart";
  #uart-num = /bits/8 <0>;
  #speed = /bits/16 <9600>;
  debug@debug0{
    compatible = "ar,pc-debug";
  };
}

tart1 {
  compatible = "ar,soft-uart";
  #uart-pins = /bits/8 <2 3>;
  #speed = /bits/16 <9600>;
  gps@gps0{
    compatible = "gps,nmea";
  };
};
i2c0 {
  compatible="i2c";
  htu21D-F {
    compatible = "HTU21D-F";
    /*Adresse fixe pour ce capteur*/
    #adress = /bits/8 <0x40>;
  };
};
};
```
Figure 2: The DTB file corresponding to the former DTS file

This is a binary file, so what is presented is the output of the "hexdump -C" command. The data is displayed byte by byte in hexadecimal (middle data), while the addresses are on the left and corresponding ascii data is on the right.

```
00000000  d0 0d fe ed 00 00 01 e6 00 00 00 38 00 00 01 98 |...........8....|
00000010  00 00 00 28 00 00 00 11 00 00 00 10 00 00 00 00 |................|
00000020  00 00 00 4e 00 00 01 60 00 00 00 00 00 00 00 00 |...N................|
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000040  00 00 00 03 00 00 00 0c 00 00 00 00 01 73 70 69 |................|
00000050  00 01 02 03 00 00 00 01 61 72 5f 73 64 63 61 72 |....ar_sdcar|
00000060  64 00 00 00 00 00 00 03 00 00 00 01 00 00 00 1a |d..............|
00000070  04 00 00 00 00 00 00 00 02 00 00 00 01 61 72 74 |............uart|
00000080  31 00 00 00 00 00 00 03 00 00 00 02 00 00 00 3b |1..............;|
00000090  25 80 00 00 00 00 00 00 01 67 70 73 5f 6e 6d 61 |%...gps_nmea|
000000a0  69 32 63 30 00 00 00 00 00 00 00 00 00 00 00 00 |i2c0.hurstu2|
000000b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
000000c0  01 23 45 67 89 ab 00 00 00 00 00 00 00 00 00 02 |.Eg............|
000000d0  00 00 00 01 75 61 72 74 74 30 00 00 00 00 00 00 |uart0...........
000000e0  00 00 00 01 01 75 61 72 74 7a 2a 00 00 00 00 00 |................|
000000f0  00 00 00 02 00 00 00 03 25 80 00 00 00 00 00 01 |............4%
00000100  73 65 72 69 61 6c 23 72 65 73 65 72 76 65 64 00 |serial-debug.|
00000110  00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 02 |uart...........
00000120  31 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 |1................|
00000130  02 03 00 00 00 00 00 00 00 00 00 00 00 00 00 03 |2..............|
00000140  25 80 00 00 00 00 00 00 01 67 70 73 5f 6e 6d 61 |%...gps_nmea|
00000150  69 32 63 30 00 00 00 00 00 00 00 00 00 00 00 00 |i2c0.hurstu2|
00000160  00 00 00 00 00 00 00 02 00 00 00 02 00 00 00 01 |................|
00000170  69 32 63 65 72 69 61 6c 23 72 65 73 65 72 76 65 |serial-debug.|
00000180  00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 02 |uart...........
00000190  00 00 00 02 00 00 00 09 63 6f 6d 70 61 74 73 00 |compatib.|
000001a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
000001b0  6c 65 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |le.#reserved-pin|
000001c0  63 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |c.#uart-num.#spe|
000001d0  65 64 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |ed.#uart-pins.#a|
000001e0  64 72 65 73 73 73 00 |dress.|
000001e6
```
APPENDIX A

Figure 3: A Light Device Tree describing the same configuration

```
#include "dts-values.h"

SPI {
    COMPATIBLE="ar,mega2560";
    
    RESERVED_PINS = /bits/ 8
        < 0 /*clk*/
        1 /*master out*/
        2 /*master sn*/
        3 /* "chip select" (on arduino, must be kept as output even if unused ) */
    >;

    SCARD {
        SELECT_PINS = /bits/ 8 <4>;
    }

    ETHERNET {
        SELECT_PINS = /bits/ 8 <10>;
        MAC = /bits/ 8 < 0x01 0x23 0x45 0x67 0x89 0xab >;
    }

    UART_HARD {
        UART_NUM = /bits/ 8 <0>;
        SPEED = /bits/ 16 <9600>;
        SERIAL_DEBUG{
        
        }

    }

    UART_SOFT {
        UART_PINS = /bits/ 8 < 2 3 >;
        SPEED = /bits/ 16 <9600>;
        GPS_NMEA{
        
        }

    }

    I2C {
        HTU21D_F {
            /*Adresse fixe pour ce capteur*/
            ADDRESS = /bits/ 8 < 0x40 >;
        }
    }
```

Figure 4: The LDTB file corresponding to the former LDTS file

00000000  fe ed 00 71 01 00 00 03 0c ff 01 61 72 2c 6d 65 |
00000010  67 61 32 35 36 30 00 01 aa 01 03 04 ff 02 00 01 |
00000020  02 03 01 aa 06 03 01 ff 03 04 02 01 aa 07 03 01 |
00000030  ff 03 0a 03 06 ff 07 01 23 45 67 89 ab 02 02 01 |
00000040  02 02 01 aa 03 03 02 ff 05 02 03 03 02 ff 06 25 |
00000050  80 01 aa 08 02 02 01 aa 04 01 aa 09 03 01 ff 08 |
00000060  40 02 02 02 09 de ef 00 71 |
00000070  02 02 02 09 de ef 00 71 |
00000079  02 02 02 09 de ef 00 71 |
The modified compiler can also output a C string containing the same data. The string is presented so that the initial tree structure is still visible.

```
\x02 \x01 \xAA \x02
    \x03 \x02 \xFF \x05 \x02 \x03
        \x02 \x02
    \x03 \x02 \xFF \x06 \x25 \x80
```

```
\x02 \x01 \xAA \x03
    \x03 \x02 \xFF \x05 \x02 \x02 \x03
```

```
\x02 \x01 \xAA \x04
    \x01 \xAA \x09
        \x03 \x01 \xFF \x08 \x40
```

```
\x02 \x01 \xAA \x01
```

```
\x01 \x00 \x00
\x03 \x0C \xFF \x01 \x61 \x72 \x2C \x6D \x65 \x67 \x61 \x32 \x35 \x36 \x30 \x00
```

```
\x01 \xAA \x01
```

```
\x03 \x04 \xFF \x02 \x00 \x01 \x02 \x03
```

```
\x01 \xAA \x06
```

```
\x03 \x01 \xFF \x03 \x04
```

```
\x02
```

```
\x01 \xAA \x07
```

```
\x03 \x01 \xFF \x03 \x0A
```

```
\x03 \x06 \xFF \x07 \x01 \x23 \x45 \x67 \x89 \xAB
```

```
\x02
```

```
\x02
```

```
\x02
```

```
\x02
```

```
\x09
```

```
\xDE \xEF \x00 \x71
```
Appendix B

In this Appendix we present some Arduino code using the Device Tree library.

Simple callback Hello World

The first extract is a simple HelloWorld example. The DT has the root node and one child node, and each one can contain a string message. The purpose of this example is to print the messages (if they are present) on the debug console.

This example uses the first callback model: it is the responsibility of the user to define the callback function that is called every time a node is found in the DT. Typically what this callback does is retrieving the type of the node and then doing an action accordingly.

Here the action is to print either "Root node" or "Subnode" (to show the node was found), then check if the string property is present (using ldt_get_property_*_by_name) and print it.

Advanced Callback with sensors

The second example (split in three parts because the code is longer) uses the new callback system.

The purpose of the example is to read a few sensors (a GPS, a humidity sensor, a pressure sensor, ...) and store the result of the measurements on a SD card every 10 seconds. If the SD card is not present, the results are only printed to the debug console.

There are three parts here, and some irrelevant code has been removed for the sake of clarity and shortness.

The first part is the headers include parts and variables declarations. We can see that there are not as many global variables as we would expect to represent each driver. This is because, when using the new callback system, they are all declared in their own library (whose include files are the "ldt_*.h" files).

The second part is the setup part. The setup() function is part of the Arduino framework and it is the first user function that is called. Here the setup() function is quite simple: it registers some node types with their callbacks (there are a lot of them so it is a bit repetitive) and then it calls the LDT library which takes care of all the initialisation. If all the initialisation code was in the setup(), the function would be a lot longer. Then at the end, a few non-LDT actions can be done such as opening a file on the SD card if that card has been detected.
Figure 6: Hello World Arduino example with basic callback

```c
#include <Arduino.h>

// This is the callback function that is called for each node
// found by the parser in the ldt library.
static void ldt_init_callback(struct dt_node* node, struct dt_node* parent) {
    uint16_t node_type=ldt_get_node_type(node);
    const __FlashStringHelper* message=NULL;

    switch(node_type){
    case NODE_TYPE_ROOT: {
        Serial.println("Root node");
        if(!ldt_get_property_raw_by_name(node, PROP_ROOT_MESSAGE,(const char**)&message)){
            Serial.print("\tMessage: ");
            Serial.println(message);
        }
        break;
    } 
    case NODE_TYPE_SUBNODE: {
        Serial.println("Subnode");
        if(!ldt_get_property_raw_by_name(node, PROP_SUBNODE_MESSAGE,(const char**)&message)){
            Serial.print("\tMessage: ");
            Serial.println(message);
        }
        break;
    } 
    }

    void setup() {
        int err;
        Serial.begin(115200);
        err = ldt_parse(ldt_init_callback,NULL);
        if(err != 0){
            Serial.print(F("[E] Error while DT parsing: "));
            Serial.println(err);
        }
    }

    void loop() {
    }
```
And finally the third part is the core of the program. It is called `loop()` because it is called in a loop by the Arduino framework: it will run indefinitely. It is also quite simple: for each sensor type, check if there is a sensor. If yes, read the value and print it. The `print_*()` functions are helper functions (omitted in this document) which print to both the debug console and the SD card, if it is present. The first two sensors are shown (the GPS and the BMP pressure sensor) and the code for the others has also been omitted, because it is quite similar.
APPENDIX B

Figure 8: Arduino Callback with sensors - Setup

```c
/* Some setup function which is not defined in a library,
   code removed for shortness */

static void setup_feedback_servo(
    struct dt_node *node, struct dt_node *parent){
    ...
}

/* Setup function: register all the drivers and start
   the ldt_init_drivers() function which initializes
   all the found peripherals. */

void setup()
{
    int err;

    Serial.begin(INIT_SERIAL_SPEED);

    ldt_register_driver_callback(NODE_TYPE_SOFTSERIAL,
        NODE_MASK_SOFTSERIAL, setup_serial);
    ldt_register_driver_callback(NODE_TYPE_HWSERIAL,
        NODE_MASK_HWSERIAL, setup_serial);
    ldt_register_driver_callback(NODE_TYPE_GPS,
        NODE_MASK_GPS, setup_gps);
    ldt_register_driver_callback(NODE_TYPE_BMP085,
        NODE_MASK_BMP085, setup_bmp_sensor);
    ldt_register_driver_callback(NODE_TYPE_SPI,
        NODE_MASK_SPI, setup_spi);
    ldt_register_driver_callback(NODE_TYPE_SDCARD,
        NODE_MASK_SDCARD, setup_sdcard);
    ldt_register_driver_callback(NODE_TYPE_TMP36,
        NODE_MASK_TMP36,TMP36::ldt_init_callback);
    ldt_register_driver_callback(NODE_TYPE_FEEDBACKSERVO,
        NODE_MASK_FEEDBACKSERVO, setup_feedback_servo);

    err = ldt_init_drivers(NULL);
    if(err != 0){
        Serial.print(F("[E] Error while parsing DT: "));
        Serial.println(err);
        while(true);
    }

    if(sd_initialized){
        logfile = SD.open("datalog.txt", FILE_WRITE);
    }

    print_all(F("[D] ------END OF SETUP------\r\n"));
}
```
void loop()
{
  float flat, flon;
  unsigned long age;
  sensors_event_t event;

  if(gps){
    print_all(F("DATE: "));
    print_date(gps);
    print_all(F("\r\n"));
    gps->f_get_position(&flat, &flon, &age);
    print_all(F("POS: "));
    print_float(flat, TinyGPS::GPS_INVALID_F_ANGLE);
    print_all(F(" | "));
    print_float(flon, TinyGPS::GPS_INVALID_F_ANGLE);
    print_all(F("\r\n"));
  } else {
    //Some uninteresting printing functions have been removed
  }

  if(bmp){
    /* Get a new sensor event */
    bmp->getEvent(&event);
    if (event.pressure) {
      print_all(F("PRESSURE: "));
      print_all(event.pressure,2);
      print_all(F("\r\n"));
    }
  }

  if(htu){
    [...] //Code removed for shortness
  }

  if(TMP36::tmp36_count)
  {
    [...] //Code removed for shortness
  }

  if(servo)
  {
    [...] //Code removed for shortness
  }

  smartdelay(10000);
}