Conception d'une architecture complète pour l'interopérabilité des objets connectés hétérogènes et des services de l'Internet des Objets

Nahit Pawar

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2021IPPAS009

Thèse de doctorat

On Interoperability and Network Architecture Bottom-Up Heterogeneity Control in Internet of Things

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Réseaux, informations et communications

Thèse présentée et soutenue à Palaiseau, le 2 juillet 2021, par

NAHIT PAWAR

Composition du Jury :

Adlen KSENTINI Professor, Eurecom, France	Président
Laurent GEORGE Professor, ESIEE Paris, France	Rapporteur
Andreia CATHELIN Docteure, STMicroelectronics, France	Rapporteur
Adlen KSENTINI Professor, Eurecom, France	Examinateur
Anelise MUNARETTO Associate Professor, Federal University of Paraná (UFPR), Brazil	Examinateur
Hakima CHAOUCHI Professeure, Institute Polytechnique de Paris - Télécom SudParis	Directeur de thèse
Thomas BOURGEAU Docteur, KBDigital AG, Switzerland	Co-directeur de thèse
Marco DI FELICE Associate Professor, University of Bologna, Italy	Invité
Xavier CARCELLE Engineer, Asahi Net, Japan	Invité

1 Introduction

L'un des attributs largement connus de l'Internet des objets (IoT) est qu'il englobe un vaste réseau de "Things" (objets) hétérogènes, allant des appareils à ressources limitées utilisés dans les batteries à faible puissance Réseaux d'actionneurs de capteurs sans fil vers des appareils plus puissants utilisés dans les passerelles et autres applications gourmandes en calcul comme les véhicules autonomes, les robots connectés, les drones, etc. De plus, la présence de l'IoT dans divers domaines d'application énergie intelligente, santé intelligente, bâtiments intelligents, intelligent transports, industrie intelligente et ville intelligente - en fait un domaine diversifié et multidisciplinaire, où chaque domaine d'application a ses propres caractéristiques et exigences diverses.
L'hétérogénéité de l'IoT peut être observée dans de nombreuses disciplines de l'IoT telles que l'application, l'architecture de l'appareil, le réseau et le protocole de communication. Cette hétérogénéité pose de sérieux problèmes pour le contrôle de l'interopérabilité et aussi pour l'uniformité dans le développement de solutions logicielles et matérielles sur le large gamme d'appareils IoT hétérogènes - unités de traitement, capteurs, actionneurs, émetteurs-récepteurs, etc.

Prototypage de l'Internet des objets : perspective logicielle - Le succès de l'IoT est attribué à l'avancement de diverses technologies telles que - la communication, Internet, réseau, microélectronique, capteur, sécurité, systèmes d'exploitation, stockage de données, etc. ainsi qu'aux applications pilotes éprouvées déployées dans divers domaines d'application susmentionnés. Mais l'implication de diverses technologies nologies et la relation complexe entre eux et la présence inhérente de dispositifs hétérogènes ont fait du développement, du déploiement et de la maintenance des applications IoT une tâche fastidieuse et chronophage. En conséquence c'est très difficile à atteindre pour les experts du domaine (agriculteur, médecin, ingénieur, etc.) connaissance de toutes les technologies derrière l'IoT. Construire une solution IoT de bout en bout réussie implique une approche systématique pour gérer différentes vies IoT phases du cycle - phase de développement, phase de déploiement et phase de maintenance.
Dans la phase de développement, la logique applicative est cadrée et séparée en un grand nombre de tâches distribuées pour un réseau de terminaux IoT hétérogènes, puis la logique applicative est implémentée, vérifiée et validée

sur de nombreuses plates-formes matérielles composées d'un grand nombre d'IoT hétérogènes dispositifs. Dans la phase de déploiement - le micrologiciel de l'appareil IoT qui remplit les le scénario d'application ciblé est chargé sur un réseau de système final IoT et est vérifié et validé à nouveau en environnement réel. En phase d'entretien, les périphériques hérités sont remplacés en raison, par exemple, d'un changement dans les exigences des applications imposant de nouvelles fonctionnalités matérielles ou une mise à niveau du système. Le principal la préoccupation dans toutes ces trois phases est l'hétérogénéité des appareils IoT qui fait de l'interopérabilité, de l'évolutivité et de la flexibilité un défi en particulier pour les grands déploiement à grande échelle de systèmes basés sur l'IoT. Les industries sont obligées d'embaucher des développeurs de systèmes embarqués expérimentés pour mettre en œuvre et maintenir base de code logiciel pour divers fournisseurs de matériel faisant du prototypage et Preuve de concept (PoC) tâches difficiles et chronophages. Cela fait aussi la solution IoT déployée est difficile à mettre à jour en cas de remplacement des appareils IoT d'un autre fabricant de matériel. Le développement d'applications embarquées dépend du fournisseur de matériel, de sorte que tout changement de fabricant d'appareils IoT déclenchera le redéveloppement de la même application déployée sur les appareils IoT précédents. De plus, il n'y a toujours pas de commune adoptée standards pour ces appareils hétérogènes et la plupart des différents fournisseurs de matériel proposent leurs propres outils de développement. Cela rend l'industrialisation des services IoT difficile car la phase PoC est coûteuse en raison de cette l'hétérogénéité et l'absence de normes communes partagées. Par conséquent, pour une adoption généralisée des systèmes et services basés sur l'IoT, un une couche intermédiaire de logiciel/service est nécessaire pour masquer les détails de divers technologies hétérogènes sous-jacentes à l'écosystème des appareils IoT.

Prototypage de l'Internet des objets : point de vue matériel - Du point de vue de la conception de systèmes embarqués, un objet IoT est une fusion de plusieurs technologies habilitantes en évolution (processeurs embarqués basse consommation, capteurs, actionneurs, émetteurs-récepteurs, récupérateurs d'énergie, etc.), ce qui nécessite concepteurs de systèmes pour tester et évaluer en permanence les nouvelles technologies et mettre à niveau le système final de manière appropriée. De plus, il est difficile de généraliser l'architecture matérielle d'un objet IoT en raison de diverses applications

IoT exigences telles que le débit de données, la mémoire, le traitement, la fiabilité, la puissance, la portée, coût, sécurité, évolutivité, etc.
Néanmoins, une architecture typique d'un objet IoT et est composé de divers blocs fonctionnels matériels hétérogènes - l'unité de traitement qui représente le "cerveau" sous la forme d'un microcontrôleur ultra-basse consommation, d'un microprocesseur, d'un système sur puce (SoC), etc. ressources de calcul nécessaires (CPU, mémoire et accélérateurs algorithmiques, etc.) pour exécuter des applications IoT. La connectivité sans fil telle que RFID, BLE, WiFi, LTE, ZigBee et LoRa pour n'en nommer que quelques-uns, fournissent le lien de communication nécessaire pour établir un réseau distribué d'objets IoT.
L'unité de traitement (PU) intègre une grande variété de périphériques hétérogènes (interfaces de communication). Cette large intégration périphérique hétérogène est destinée à faciliter la communication avec différents blocs fonctionnels extérieurs à l'unité de traitement. Les périphériques sont utiles car ils permettent aux unités de traitement de se décharger des tâches de calcul (échange de données entre Unité de traitement et ressources IoT) à eux, économisant ainsi de précieux calculs ressources pour d'autres tâches importantes. Le nombre et le type de périphériques pris en charge par le PU et leur mappage de broches varient d'un PU à l'autre, donc cette hétérogénéité d'interface périphérique peut conduire à différents et incompatibles conception matérielle. De plus, le manque d'interface cohérente oblige souvent les intégrateurs de systèmes à créer les connexions électriques requises qui dépendent du type de PU utilisé. De plus, en raison de l'avancement des technologies matérielles IoT il faut reconcevoir ou remplacer le système existant pendant le cycle de vie d'un Objet IoT, même si seule une sous-partie doit être remplacée.
En conséquence, il existe un fort besoin et une forte demande pour une interface périphérique standard homogène pour gérer la configuration système avancée des blocs fonctionnels susmentionnés et la fonctionnalité plug-and-play pour la conception de systèmes embarqués adaptés au prototypage d'applications Internet des objets.

Declaration of Authorship

I, Nahit PAWAR, declare that this thesis titled, "On Interoperability and Network Architecture Bottom-Up Heterogeneity Control in Internet of Things" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

Abstract

Nahit PAWAR

On Interoperability and Network Architecture Bottom-Up Heterogeneity Control in Internet of Things

Internet of Things (IoT) combines many technologies and it has spanned across diverse and multidisciplinary application domains. Each domain has its own set of application requirements in terms of hardware, communication, software, source of energy, etc. This inhibits the use of conventional *programming models* of distributed computing which assumes that the systems are always connected, having abundant computational resources and access to infinite electric energy. Additionally, IoT encompasses a wide range of heterogeneous embedded IoT devices (processing units, sensors, actuators, transceivers, etc.) provided by various manufacturers each with different device architecture, as a result the application software developed for these devices are not compatible with each other. This device heterogeneity poses serious problems for device interoperability and also for harmonized IoT development tools over a wide range of heterogeneous IoT devices. An important challenge not only for domain experts but also for professionals is to realize proof-of-concept (PoC) during industrialization of IoT services, that involves - development, deployment and maintenance of end-to-end IoT application services requiring different types and levels of expertise.
The main contribution of this thesis is to introduce a new framework named PrIoT (Prototyping Internet of Things) that allows easy and rapid IoT device programming from design to deployment that better handles the heterogeneity of IoT device architecture. More specifically, the PrIoT framework is based on the concept that IoT applications possess various invariant characteristics that we studied and gathered from various IoT architectures and applications presented in the literature. We then developed a minimalist high level programming language and APIs to show the easy composability of our invariant functionalities in the development of IoT applications. We validate our

PrIoT framework through reference implementation and also the development of prototype implementation of various IoT scenarios using our framework and comparing it against various existing solutions.
From a hardware perspective, in order to control better the device heterogeneity, we propose two novel modular systems named R-Bus and P-Bus for designing embedded systems as a set of hardware modules that can be mounted and dismounted based on the IoT applications needs. This resolves device interface heterogeneity and accommodates various classes of constraint devices along with advance system configuration and plug-&-play functionalities to ease IoT hardware prototyping. We validate our modular system using two metrics - *suitability* and *coverage ratio* that measure the compatibility of embedded modular systems with respect to processing units. We used these metrics to compare our solution with existing modular systems.
This approach complements our proposed PrIoT framework as it offers a new way to build end-to-end IoT application prototypes with flexibility in both hardware and software of the IoT devices.
In fact, our objective is to enable rapid end-to-end IoT prototyping by implementing a high level abstraction layer that hides the details of various technologies underlying IoT and implementing modular systems for flexible device integration targeted for IoT system design. This work has provided a step forward in controlling device heterogeneity from both hardware and software perspective, but it still lacks the standardization among the IoT community to foster its continuous development.

Acknowledgements

First of all, I would like to thank Telecom SudParis - Institute Polytechnique de Paris (TSP-IPP), for accepting my application and incubating me in it's doctoral program. Without the resources of TSP-IPP this work would not have been realized.
I would like to express my sincere gratitude to my thesis director Prof. Hakima Chaouchi for her continuous support of my PhD research and study. Her guidance helped me in all the time of the research especially while preparing research papers and writing of this thesis.
A big thanks to Dr. Thomas Bourgeau for his guidance, recommendations, follow-up and his constant support throughout my PhD journey. I am grateful to have you as my co-director of my thesis. Our weekly brainstorming, feedback and continuous discussions were a source of motivation. Thank you for always challenging my research ideas & directions and helping me to develop strong proposals without losing sight of research objectives.
My deep gratitude is also addressed to the member of jury - Prof. Laurent George and Dr. Andreia Cathelin for reviewing my dissertation and providing valuable comments. Thanks to Prof. Adlen Ksentini, Prof. Anelise Munaretto, Prof. Marco di Felice and Xavier Carcelle for accepting being part of my PhD committee members.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Context and Problem Statement

One of the widely known attribute of Internet of Things (IoT) is that it encompasses a large network of *heterogeneous "Things"* (objects) [1, 2, 3, 4, 5], ranging from resource constrained devices used in battery operated low power Wireless Sensor Actuator Networks (WSANs) to more powerful devices used in gateways and other compute intensive applications like autonomous vehicles, connected robots, drones, etc. Moreover, the presence of IoT in various application domains [1, 3] *smart energy, smart health, smart buildings, smart transport, smart industry and smart city* - makes it a diverse and multidisciplinary field, where each application domain has its own diverse characteristics & requirements [2].
Heterogeneity in IoT can be seen in many disciplines of IoT like application, device architecture, network and communication protocol. This heterogeneity poses serious problems for interoperability control and also for the uniformity in the development of *software* and *hardware* solutions over the wide range of heterogeneous IoT devices - processing units, sensors, actuators, transceivers, etc.

Prototyping the Internet of Things : Software Perspective - The success of IoT is credited to the advancement in various technologies [4] (depicted in Figure 1.1 as infinite branches of a tree) like - communication, Internet, network, microelectronics, sensor, security, operating systems, data storage, etc. and also to the proven deployed pilot applications [6, 7, 8] in various aforementioned application domains. But the involvement of various technologies & intricate relationship between them and the inherent presence of heterogeneous devices has made IoT application development, deployment

and maintenance a cumbersome and time consuming task. As a result it is very difficult for the domain experts (farmer, doctor, engineer, etc.) to attain knowledge in all the technologies behind IoT. Building a successful end-to-end IoT solution involves systematic approach to handle different IoT life cycle phases - *development phase*, *deployment phase* and *maintenance phase*.

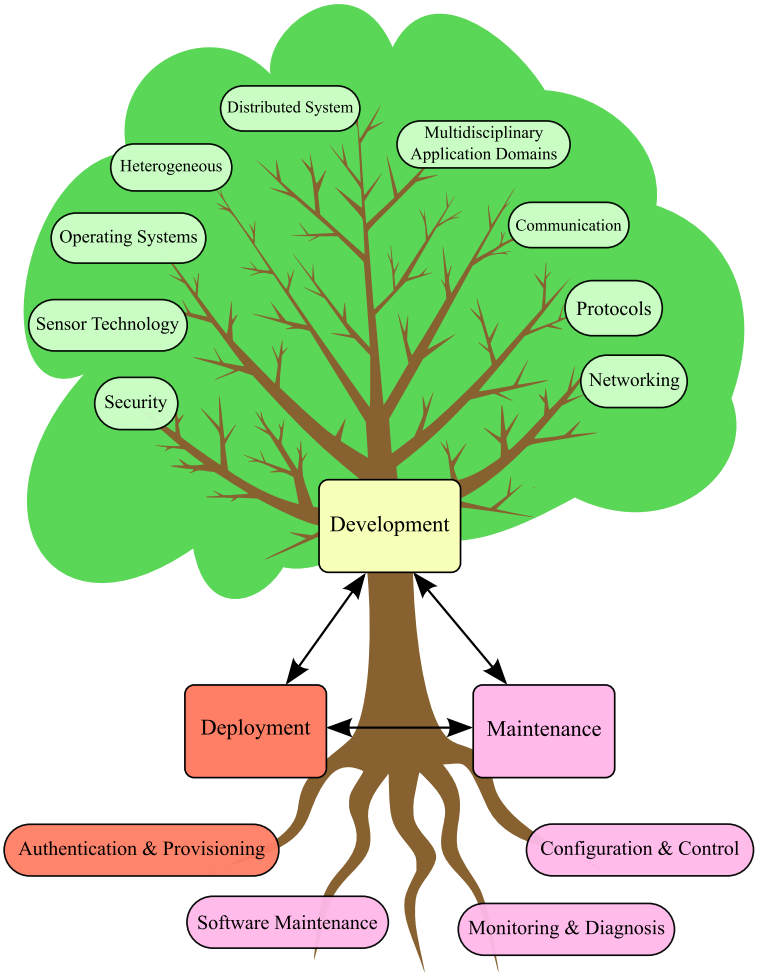


FIGURE 1.1: IoT Ecosystem

In the ***development phase***, the application logic is framed and separated into a large number of distributed tasks for a network of heterogeneous IoT end-devices, then the application logic is implemented, verified & validated on many hardware platforms consisting of large number of heterogeneous IoT devices. In the ***deployment phase*** - the IoT device firmware that fulfills the targeted application scenario is loaded on a network of IoT end-system and is verified & validated again in real environment. In the ***maintenance phase***, the legacy devices are replaced due to, for instance, change in application requirements imposing new hardware features or system upgrade. The major concern in all these three phases is IoT device heterogeneity which makes

interoperability, scalability and flexibility a challenge in particular for large scale deployment of IoT-based systems. Industries are forced to hire experienced embedded systems developers to implement and maintain embedded software codebase for various hardware vendors making prototyping and Proof-of-Concepts (PoC) difficult and time consuming tasks. It also makes the deployed IoT solution hard to update in case the IoT devices are replaced from a different hardware manufacturer. The development of embedded applications are hardware vendor dependent, so any change in IoT device manufacturer will trigger the re-development of the same application deployed on the previous IoT devices. Moreover, there is still no adopted common standards for these heterogeneous devices and most of the different hardware vendors offer their own development tools. This makes the industrialization of the IoT services difficult as the PoC phase is expensive due to this heterogeneity and the lack of common shared standards.
As a result, for widespread adoption of IoT based systems and services an intermediate software/service layer is needed to hide the details of various heterogeneous technologies underlying the IoT device ecosystem [9, 10, 11, 12].

Prototyping the Internet of Things : Hardware Perspective - From an embedded system design perspective, an IoT object is a fusion of various ever evolving enabling technologies (low-power embedded processors, sensors, actuators, transceivers, energy harvesters, etc.) [4, 2, 9, 13], which requires systems designers to constantly test and evaluate new technologies and upgrade the final system appropriately. Moreover, it is difficult to generalize the hardware architecture of an IoT object because of diverse IoT application requirements such as data rate, memory, processing, reliability, power, range, cost, security, scalability, etc.
Nevertheless, Figure 1.2 shows a typical architecture of an IoT object and is composed of various heterogeneous hardware functional blocks - the *processing unit* [2] that represent the "brain" in the form of ultra-low power microcontroller, microprocessor, system-on-chip (SoC), etc. and provides the necessary compute resources (CPU, memory and algorithmic accelerators, etc.) to execute IoT applications. The *wireless* connectivity such as RFID, BLE, WiFi, LTE, ZigBee and LoRa to name a few, provides the necessary communication link to establish a distributed network of IoT objects. Depending on the application's power budget, the *power unit* provides various options such as energy harvester, battery management system (BMS) for battery powered

devices, voltage regulators (Low-dropout (LDO)) & converters (DC-to-DC), etc. On the other hand an IoT object can also have various connectors for off-board sensors & actuators - for example environment sensors which require their placement outside the IoT object.

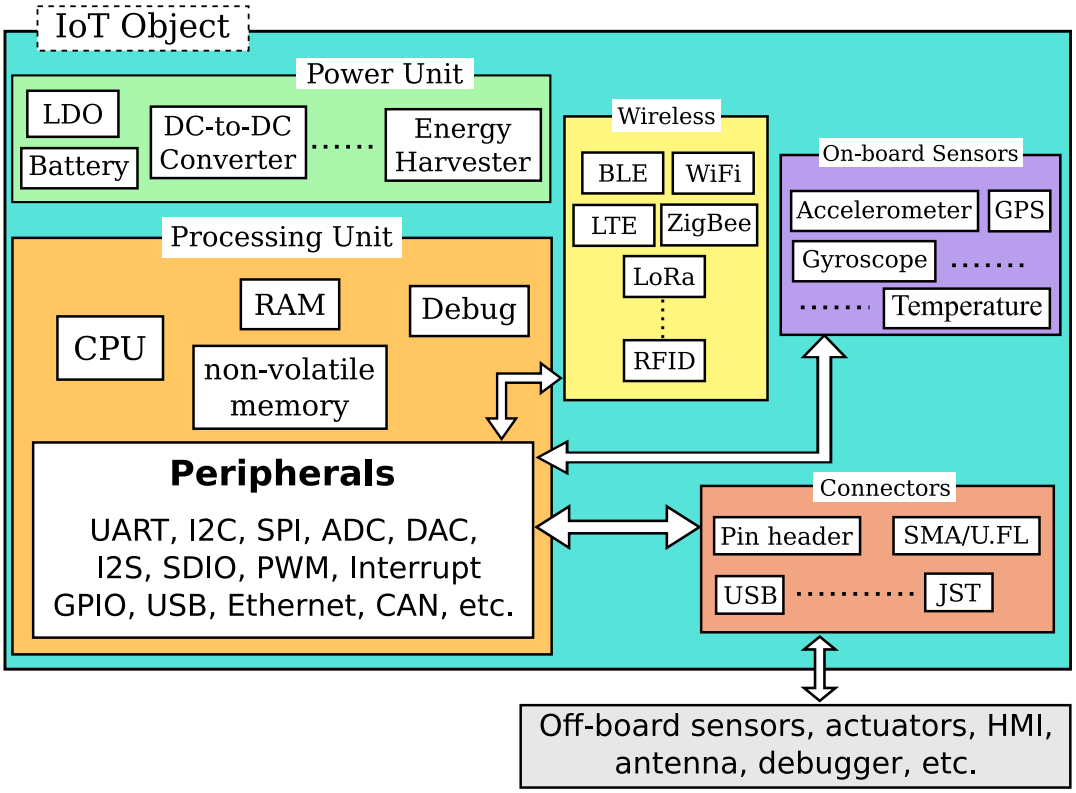


FIGURE 1.2: IoT Object Architecture

The processing unit (PU) integrates a wide variety of heterogeneous *peripherals* (communication interfaces) [14, 15, 16]. This wide heterogeneous peripheral integration is intended to facilitate communication with various functional blocks outside of the processing unit. Peripherals are useful as they allow processing units to offload computational tasks (data exchange between Processing Unit & IoT resources) to them, thereby saving valuable compute resources for other important tasks. The number and type of peripherals supported by PU and their pin mapping vary from one PU to another, therefore this *peripheral interface heterogeneity* can lead to *different and incompatible hardware design*. Also the lack of consistent interface often requires system integrators to create the required electrical connections that depend on the type of PU used. Moreover, due to the advancement in IoT hardware technologies one has to redesign or replace the existing system during the life cycle of an IoT object, even though only sub-part needs to be replaced.

As a result, there is a strong need and demand for a homogeneous standard

peripheral interface to address advanced system configuration of aforementioned functional blocks and plug-&-play functionality for designing embedded systems that are suitable for prototyping internet of things applications.

1.2 Aims and objectives

In the context of heterogeneous IoT device heterogeneity, different research problems are addressed. First, how to maintain the IoT service continuity when the IoT devices have to be replaced with different ones? Then, how to ensure that the IoT devices components replacement such as storage, processing, connectivity and energy modules is optimal? and how to reduce the complexity of re-configuring IoT devices built by different manufacturers with different hardware and software components? and most importantly how to ensure the end to end interoperability of an IoT application and service that crosses different layers from the IoT devices, the gateway to the IoT cloud platform.
The aim of this thesis is to address the aforementioned research problems and provide the necessary solutions and their evaluation and compare them to the related research state of the art. The research contributions presented in this thesis provides also a set of tools both in terms of software and hardware for easy and rapid prototyping Internet of Things (IoT) applications. In order to accomplish this, many challenges have to be addressed including - diverse IoT application characteristics, heterogeneous IoT system architecture, device and protocol heterogeneity, energy constraints and more importantly interoperability due to these heterogeneity. To address these challenges, this thesis provides three main contributions that are :

1. **IoT application invariant functionality and programming patterns :** To achieve this we did systematic study and understanding of IoT architecture and IoT application characteristics is needed. We proposed a 4-layer (Device Layer, Edge Layer, Cloud Layer and Cross-Layer) architecture to systematically study the various heterogeneity problems and IoT application characteristics at each layer. This allowed us to extract various functionalities and programming patterns that are IoT applications invariant that forms the underlying basis of our proposed framework in Chapter 4.

2. **IoT application lifecycle management on heterogeneous devices :** To achieve this we proposed and developed a framework in Chapter 4 that

exposes an intermediate abstraction layer to hide various technologies underlying IoT devices in terms of device architecture and communication protocols. The main building block of the intermediate abstraction layer is based on the invariant functionality and programming patterns proposed in 1st contribution (Chapter 3). This framework recommends an optimal bill of material and allows easy integration and configuration of IoT device components from various manufacturers without affecting the IoT service continuity.

3. **Modular system for designing interoperable embedded systems :** To achieve this we proposed two new modular systems named R-Bus (Resource Bus) and P-Bus (Power Bus) for controlling IoT device peripheral heterogeneity. The two modular systems presented in this thesis (Chapter 5) reduce the complexity of integrating, replacing and reconfiguring IoT devices from various manufacturers that require different hardware and software components.

1.3 Research Methodology

In order to tackle the research problem an iterative research methodology was adopted to systematically reach the aims and objectives of the thesis. The iterative research methodology has 4 main steps. 1) Identify the requirements through literature survey 2) Research and development of the proposed solution 3) Design and implementation 4) Test, Evaluate and Validate. We followed these steps in both our proposed software and hardware based research contributions.

A comprehensive literature review was carried out in order to identify the various heterogeneous technologies underlying the IoT ecosystem that hinders the prototyping of Internet of Things (IoT) both from software and hardware perspective. The outcome came to the conclusion that new approaches were needed to address this problem. Based on this, a new framework for prototyping IoT applications is proposed that better handles the IoT device heterogeneity control problem. This framework is based on our high level abstraction in terms of application invariant functionalities and programming patterns that we found by systematic study of various IoT architecture and IoT application characteristics. To validate our work, the proposed framework is implemented using the available open source tools & languages. To this end, different IoT application scenario use cases are implemented using

our framework. The framework is iteratively improved and updated during the thesis.
From a hardware perspective, a new modular system for designing interoperable embedded systems is proposed that better handles the IoT device peripheral interface heterogeneity. The requirements of the proposed solution are identified based on the limitations of existing solutions to cater diverse IoT application requirements. The proposed solution is implemented and evaluated both qualitatively & quantitatively against existing solutions. We validated the system by implementing various IoT application use cases.

1.4 Thesis Scientific Contributions and Thesis Outline

The structure of this thesis is depicted in Figure 1.3 and the major contributions of the thesis are presented as follows:

- **Chapter** 2: We present the state of the art of interoperability and heterogeneity in the IoT ecosystem. This serves as a background of our research and also for common understanding of proposed solutions.
  We briefly discuss the various types of heterogeneity in the IoT ecosystem: 1) device heterogeneity. 2) network heterogeneity. 3) protocol heterogeneity. 4) IoT system architecture and 5) IoT platform. This allows us to better understand the various interoperability issues. We aim to expose various technologies that are responsible for interoperability in IoT devices networks. We also highlight the presence of heterogeneous domain dependent system architecture in IoT and review various IoT architectures from academia, industry and standards organizations. In addition, we also state the requirement of various IoT platforms in the IoT for implementing solutions.
  We introduce the notion of interoperability in IoT and briefly define various types of interoperability present in the IoT ecosystem, starting with 1) Semantic Interoperability. 2) Syntactic Interoperability. 3) Network Interoperability. 4) Platform Interoperability. 5) Device Interoperability. We present and review the state of the art of interoperability solutions from academia, standards organizations and industrial alliances.
  In addition, we introduce the notion of IoT device modularity and propose a complementary hardware based approach to handle some parts

of the IoT device heterogeneity, thus easing the interoperability. We describe the modular architecture in general and discuss the role of modular systems in hardware heterogeneity control when designing embedded systems for IoT. We present various existing modular systems and their limitations. We also state the importance of energy aware embedded systems in IoT and present the importance of modular systems in dealing with heterogeneous energy sources and energy constrained IoT devices.

- **Chapter** 3: We introduce our 4-layer IoT architecture that allows us to systematically map various IoT scenarios and identify various IoT application characteristics and associated research problems at each layer. This helps us to extract and identify high level abstraction in the form of invariant functionalities and programming patterns to cover most IoT application scenarios. This method provides a lightweight approach when designing software abstraction layers with minimalistic programming functionalities that are IoT application invariant. We used this design philosophy in implementing our proposed framework in Chapter 4. The results of this contribution are presented in [17] and [18].

- **Chapter** 4: We present our framework named PrIoT for easy and fast IoT prototyping. We describe the main components of PrIoT that aims to leverage IoT adoption and usage from design to deployment and better handle the heterogeneity property of IoT devices, services and applications. This framework introduces the design philosophy of "code once and port anywhere". In addition, we validate our framework by implementing PrIoT using various open source software & tools. We also showcase the advantage of PrIoT by implementing an example IoT scenario use case. The results of this contribution is presented in [19].

- **Chapter** 5: We present our two modular systems named R-Bus (Resource Bus) and P-Bus (Power Bus) for easy and fast prototyping of embedded systems that enhances system integration by providing sufficient resource integration. The modular approach for implementing hardware systems for IoT objects allows for better handling of IoT hardware requirements, addresses advanced system configuration, features for plug-n-play functionalities, can handle various power sources, provides features for optimizing energy consumption and is usable across diverse IoT applications. In addition, we analytically evaluate our system with existing modular systems using two metrics - *coverage* and

suitability ratios. We also showcase the advantage of our modular system with example use cases. The results of this contribution are presented in [20], [21], [22] and [23].

- **Chapter** 6: We conclude our dissertation by summarizing all the work achieved during this thesis and we present the future directions that will follow this research work.

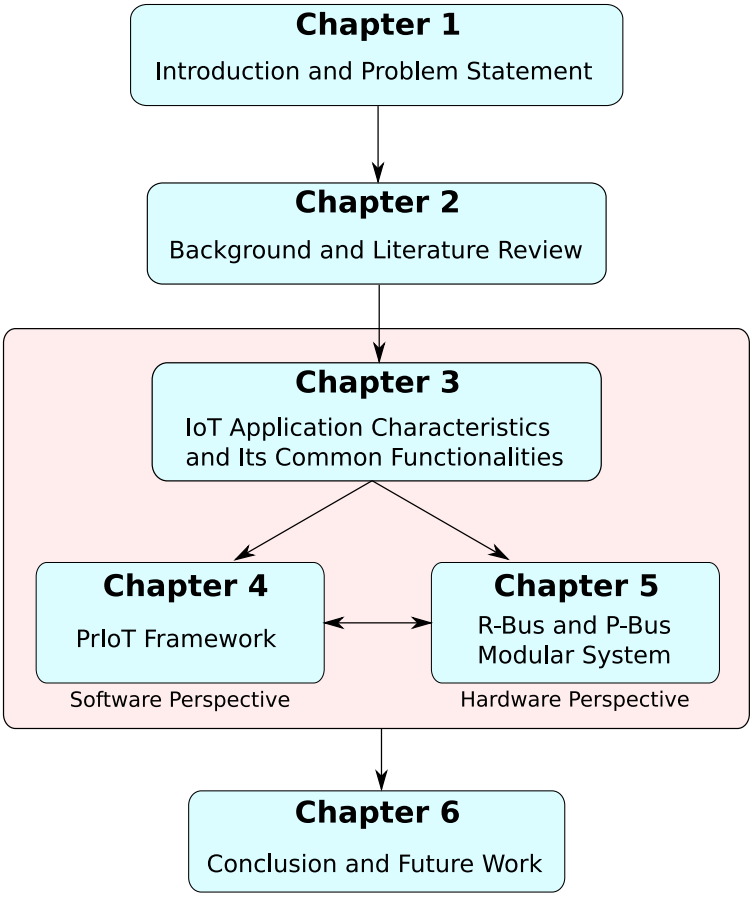


FIGURE 1.3: Thesis Structure

Chapter 2

Literature Review

2.1 IoT Heterogeneity and Interoperability

Interoperability issues in IoT is a consequence of various heterogeneities in the IoT ecosystem [24]. Therefore in this section, we begin by understanding and listing the various heterogeneities that are present in the IoT ecosystem (Section 2.1.1). Then in Section 2.1.2, we describe various interoperability levels in the IoT ecosystem to better understand various interoperability solutions as discussed in Section 2.2. This section lays the groundwork for Section 2.2 and Section 2.3.

2.1.1 Understanding Heterogeneous IoT Ecosystem

Interoperability related issues in IoT arise due to a lack of common standards and vast heterogeneity in the IoT ecosystem which is mainly a consequence of the variety of IoT applications requirements. In this section, we describe various types of heterogeneity categories present in the IoT ecosystem such as - device heterogeneity (Section 2.1.1.1), network heterogeneity (Section 2.1.1.2), protocol heterogeneity (Section 2.1.1.3).
We then describe the existing IoT system architectures (Section 2.1.1.4) and IoT platforms (Section 2.1.1.5) that are adopted without consensus and common IoT standards by different IoT services and applications, but a joint industrial framework for each IoT vertical domain.

2.1.1.1 IoT Device Heterogeneity : Hardware and Power Constraints

Hardware devices that are used in the Internet of Things (IoT) come with certain requirements and constraints. It is well known as shown in Figure 1.2 that IoT encompasses a wide range of heterogeneous hardware devices

[1] ranging from 8-bit microcontrollers used in low power wireless sensor nodes to 64-bit processors used in gateways of other compute intensive applications. These devices are available from various vendors that differ in CPU (Central Processing Unit) architecture, power consumption, memory (RAM & ROM), peripherals, documentations, development tools and software. Also there is a huge price difference between lower end 8-bit microcontrollers and higher end 64-bit processors with cheaper devices being inferior to their more expensive counterparts in terms of processing speed, memory, power, etc. Moreover, the devices that are used in wireless sensor networks are expected to operate on a limited energy budget for a very long time without human intervention. This requires devices to be powered by battery or extract energy from the environment (energy harvesting). More details on power requirements are presented in Section 2.3.1.2.
Bormann et al. [25] (IETF RFC7228) categorizes energy constrained devices into various classes (Table 2.1) according to energy limitations. As shown in Table 2.1, class E9 represents devices in the extreme end of the spectrum where the device has no limitations with respect to available energy, whereas E0 represents devices in the lower end of the spectrum where the device has no storage element and relies on external events to extract energy for doing useful work.

Class	Type of energy limitation	Example Power Source
E0	Event energy-limited	Event-based harvesting
E1	Period energy-limited	Battery that is periodically recharged or replaced
E2	Lifetime energy-limited	Non-replaceable primary battery
E9	No direct quantitative limitations to available energy	Mains-powered

TABLE 2.1: Class of Energy Limitations (Source - [25])

Based on these economical, physical and power constraints, the IoT devices exhibit certain limitations in terms of slow processing speed, small memory and lack of constant energy source. Such devices are referred to as constrained IoT devices [25, 26].
Also, Bormann et al. [25] categorizes the constrained devices into different hardware classes (Table 2.2) based on available resources in terms of maximum code complexity and processing capabilities. As shown in Table 2.2,

Class	Data Size (e.g. RAM)	Code Size (e.g. Flash)
C0	$\ll$ 10KB	$\ll$ 100KB
C1	$\sim$ 10KB	$\sim$ 100KB
C2	$\sim$ 50KB	$\sim$ 250KB
KB - Kilobyte = 1024 bytes		

TABLE 2.2: Class of Constrained Devices (Source - [25])

Class 0 devices are very constrained sensor-like motes and do not have resources to communicate directly with the Internet in a secure manner. Class 1 devices on the other hand do not have enough resources to support full IETF Internet protocol suite [27] but can support constrained node protocols for example CoAP (Constrained Application Protocol) over UDP (User Datagram Protocol). Whereas Class 2 devices are less constrained and can support most protocol stack that are otherwise not possible in Class 0 and Class 1 devices. The boundaries of these classes (Table 2.1 and 2.2) are expected to shift over time based on the improvement in technology to reduce cost, power and the development of new constrained protocol stack.
The presence of so many IoT devices[1] with different capabilities in terms of processing, memory and power requirements, creates a problem when developing IoT applications using various software components such as communication protocol libraries, embedded operating systems, etc. that requires considerable resources making it difficult to cater full range of constrained devices from class 0 to class 2 without optimizing existing software components. In order to tackle this problem in Chapter 4 we propose our new development framework named PrIoT that provides a lightweight and minimalistic approach to IoT application development on heterogeneous IoT devices. Moreover in Chapter 5 we propose a new hardware approach named P-Bus (Power Bus) that helps in the development of power aware IoT applications that better caters the power requirements of IoT devices.

2.1.1.2 IoT Network Heterogeneity: Network Constraints

According to Bormann et al. [25], the *constrained network* has the following attributes - low bit rate and throughput with limits on both duty cycle and transmission power, high packet loss with unpredictable packet loss rate, asymmetrical up-link and down-link characteristics, limits on reachability and lack of advanced services.

[1] 29.3 billion networked devices by 2023, Cisco Annual Internet Report (2018–2023) White Paper, [Online]

On the other hand, the network formed by constrained nodes is defined as a *constrained node network* because the network constraints are coming from the constrained nodes rather than the network itself. A typical example of a constrained node network as defined by IETF is Low-Power Lossy-Network (LLN) that consists of many constrained nodes over constrained networks like IEEE 802.15.4 or low power Wifi.
Whereas, Low Power Wide Area Networks (LPWANs) [28] such as LoRa, Sigfox, Weightless, etc. are considered as constrained networks that operate on ISM bands. Another example of a constrained network as defined by IEEE 802.15 working group is LoWPAN (Low-Power Wireless Personal Area Network) described in the IETF RFC 4919 [29] that consist of devices using IEEE 802.15.4 standard radios.

2.1.1.3 IoT Protocol Heterogeneity

IoT communication protocols and technologies enable heterogeneous devices to communicate together over lossy and noisy networks. There are a number of heterogeneous protocols that exist in IoT due to different hardware & power requirements of constrained devices (Section 2.1.1.1) and also to cater various requirements of the IoT market. This section presents various protocols and communication technologies that exist in the physical & link layer, network & transport layer and application layer.

Physical and Link Layer : Physical and link layer of the network protocol stack defines the network interface between IoT objects. The physical layer defines the way in which data is transmitted on the network medium (wired or wireless) and the link layer determines how the streams of bits are put into manageable frames of data [30]. There are a number of protocols and technologies defined by various organizations and standards for both physical and link layer protocols. The reason for such a large variety of protocols is due to different requirements (bandwidth, range, energy consumption, coverage, reliability, spectrum availability, etc.) of the IoT market [31]. For example, LPWAN (Low-Power Wide-Area Network) protocols such as LoRa, Sigfox, NB-IoT, Ingenu, etc. are designed for wireless sensor networks that require long range communication at a low bit rate. Bluetooth and BLE that are based on IEEE 802.15.1 standard are used in smartphones, wearable & medical devices, etc. LR-WPAN (Low-Rate Wireless

Personal Area Network) protocols like Zigbee, Thread, 6LoWPAN, WirelessHART, etc. are based on IEEE 802.15.4 standard and are used for ad-hoc wireless sensor actuator nodes, mesh network, machine to machine, etc. Wifi (based on IEEE 802.15.11) is used in smartTVs, home appliances, smartphones, edge devices, etc. Whereas, Near-field-Communication (NFC) and Radio-Frequency Identification (RFID) uses a short range communication technology and are mostly used in logistics & supply chain, inventory and access tracking, access control, etc. On the other hand technologies like 2G, 3G, LTE (Long-Term Evolution), 4G, 5G, future 6G [32] etc. and that are based on the standards defined by 3GPP are used in applications that require long range and high bandwidth such as video surveillance, edge devices, autonomous vehicles, etc. In addition, to respond to the IoT needs, cellular networks also introduced new standards such as 4G NB-IoT, 5G mMTC [33] that are covering constraint devices and networks.

Network and Transport Layer : In Internet applications end-to-end connectivity is made available using TCP/UDP (Transmission Control Protocol)/(User Datagram Protocol) and IP (Internet Protocol - IPv4 and IPv6) as the fundamental protocols in the Internet protocol suite [27]. The device connected to the Internet is capable of processing IP packets irrespective of the physical and link layer protocol used. In the context of constrained devices, to allow end-to-end IP connectivity an adaptation of IPv6 is proposed (6LoWPAN RFC4919 [34]) over IEEE 802.15.4 that allows communication with other devices on an IP network like Wifi. Zigbee protocol defines its own networking layer on top of IEEE 802.15.4 standard and cannot easily communicate with other protocols. On the other hand, their are other proprietary protocols (Z-Wave, ANT+, enoCEAN,etc.) [35] that also defines its own networking layer for its mesh network over the ISM (Industrial, Scientific and Medical) frequency band.

Application Layer : The application layer is the final layer of the OSI model of computer networks and it ensures effective communication between two or more application programs in a network [30]. There are several application protocols that can be used in IoT applications and each one of them uses different technology and has its own benefits and drawbacks [36]. The most popular application layer protocols in the IoT community are RESTful HTTP (Hypertext Transfer Protocol) [37], Constrained Application Protocol (CoAP) [38], Message Queuing Telemetry Transport (MQTT) [39], Extensible

Messaging and Presence Protocol (XMPP) [40], Advanced Message Queuing Protocol (AMQP) [41], etc.
Selecting the most appropriate application protocols depends on specific use cases and application requirements such as security, reliability, performance, etc. For example, Nastase et al. [42] compares various application layer protocols based on security requirements, Safaei et al. [43] compares the reliability of both MQTT and CoAP. Whereas, Pohl et al. [44] evaluates the performance of AMQP, MQTT and AMQP based on bandwidth usage, latency, throughput and reliability.
Finally, the application layer protocols require TCP-UDP/IP stack to communicate and therefore are only effective between IoT cloud servers up to IoT devices that run full IP stack. The devices that do not run IP stack are required to go through gateways to communicate with IoT applications in the cloud.

2.1.1.4 IoT System Architectures

Architecture of Internet of Things (IoT) is defined as a layered structure in which each layer represents a well defined set of services. It allows mapping of IoT application domains onto more structured and well defined building blocks that share the same terminologies.
The presence of IoT in multiple heterogeneous domains and its various domain requirements has resulted in multiple IoT architectures making it difficult to select which architectural style or protocol suit is suitable for realizing applications. Moreover, no one can predict which architectural style will prevail, the chances are that they all will be used. The presence of multiple IoT architectures has resulted in limited interoperability, where IoT solutions are operating in silos. An ideal IoT architecture that is agnostic to a particular domain is far from reality but research efforts in academia, industry and standards organization continue to define a wide variety of architectures.
In the context of architecture design guidelines, IoT-A (IoT-Architecture) [45, 46] is a EU FP7-ICT project worth mentioning, although IoT-A is not an architecture but provides the architectural reference model in the form of building blocks, design guidelines for protocols, interface, algorithm and models of interoperability for designing future IoT architectures. The IoT-A does not impose any restrictions on the underlying technology. The architecture of IoT6 [47] is one such example that is designed using IoT-A architectural reference model that implements 6LoWPAN and CoAP as its underlying protocol stack.

oneM2M [48] standard for M2M & IoT defines a horizontal architecture that provides common services (IoT service layer) that enables applications in multiple domains. This IoT Service Layer is like a distributed operating system for IoT providing uniform APIs to IoT applications that hides the underlying network technologies, transport protocols and data serialization.
The Open Connectivity Foundation (OCF) [49] is an industry organization that develops standards, interoperability guidelines and provides certification guidelines. It defines an architecture that is based on Resource Oriented Architecture (RSO) design principles, where each entity (e.g. wired or wireless sensor actuator devices) is represented as Resources. Interactions with an entity are achieved through its Resource representations using operations that adhere to REST (Representational State Transfer) architectural style. The OCF architecture defines the framework of the information system and the interrelationship between entities. IoTivity [50] is an open source implementation of OCF specification developed by various members of OCF.
While each industrial alliance such as, Thread Group [51], LoRa [52], Zigbee [53], Z-Wave [54], Wi-SUN [55], NFC Forum [56], etc. - promotes their own "in-alliance industrial standard" with their protocol (Section 2.1.1.3) & architecture and there are no real successful attempts to cover a cross domain standard for the IoT ecosystem as a whole. Meanwhile, international standardization bodies such as International Standardization Union (ITU), European Telecommunications Standards Institute (ETSI), Internet Engineering Task Force (IETF), World Wide Web Consortium (W3C) are all providing their specifications for standardized connected IoT architecture and protocols. The architecture from IETF is based on their protocol suite [57] - CoAP, RPL (Routing Protocol for Low Power and Lossy Networks), 6LoWPAN, IEEE 802.15.4 - to allow easy integration of resource constrained IoT devices in IP networks.
Research community also actively participates in defining and comparing the requirements for the future IoT architecture. Wu et al. [58] introduces a 5-layer (business management layer, service management layer, network management layer, element management layer and network element layer) IoT architecture by improving the traditional 3-layer (application layer, network layer and perception layer) IoT architecture that lacks network management and business models. The 5-layer architecture includes the qualities of both the architectures - Internet and Telecommunication management network.
The architecture of most publicly available IoT experimental testbeds follow

either a two-tier (IoT device tier and Server tier) or three-tier (IoT device tier, gateway device tier and server tier) structure [59], where the structure refers to the organization of testbed hardware components. Although these architectural structures are suitable for experimentation and prototyping but do not cover IoT system management features.
Peña et al. [60] defines an IoT architecture centered around Fog/Edge computing and proposes an extension to IoT-A architectural reference model in terms of computing location transparency & topology management and integration & automation of IoT visualization systems.
Misra et al. [61] proposes a 4-layer (things layer, sensor/network as a service layer, data management layer, analytics layer) architecture by extracting the requirements and characteristics from various IoT application domains that are desired in practical architecture. Yelamarthi et al.
[62] proposes a simpler device oriented modular view of IoT where boundaries between various architectural layers are defined between sensor & actuator, low power embedded processor, wireless transceivers, Internet gateway and cloud.
Yashiro et al. [63] also proposes a device oriented architecture based on two existing technologies - CoAP and uID (Ubiquitous ID) along with RESTful IoT services to implement practical IoT applications.
In summary, the study and research for new and existing architecture and reference models for IoT will continue to grow as we are still far away from an ideal architecture. Our analysis of this state of the art showed a convergence of the approaches to control the heterogeneity from the IoT gateways to the IoT cloud servers using the standard Internet protocol based suite. However the heterogeneity due to the IoT end device technologies is hidden by the IoT gateways and there is no standard framework to control this part which makes IoT end to end interoperability difficult to achieve. In our work, we addressed exactly this IoT device related heterogeneity control, and we propose to adopt a 4-layer architecture (Chapter 3) to study the various IoT characteristics for extracting invariant functionalities and programming patterns at each layer.

2.1.1.5 IoT Platforms

Mineraud et al. [64] defines the IoT platform as the middleware and the infrastructure that enables the end-users to interact with smart objects. Pliatsios et al. [65] defines an IoT platform as a comprehensive suite of services

that facilitates services, such as development, maintenance, analysis, visualization and intelligent decision-making capabilities in an IoT application. There are a number of diverse IoT platforms that exist in the IoT ecosystem that we propose to gather in two major categories: IoT cloud service platforms and IoT development platforms & Operating systems.
First, in IoT cloud service platform there are a number of IoT service providers - *Amazon AWS IoT, Microsoft Azure IoT, Google IoT, IBM Watson IoT, Oracle IoT Cloud*, to name a few - that provide cloud based scalable commercial IoT platform across wide range of application domains, with ultimate goal to help businesses in - making critical decisions, improving efficiency, resource planning, customer interaction, etc. From our study all service providers implements a 3-layer (device - edge - cloud) architecture and expose more or less similar functionalities in terms of cloud *micro services* [66]. They also provide Software Development Kit (SDK) for easy integration of devices into the platform, but the connectivity is limited to IP enabled devices with non-IP devices enabled using protocol adapter. Ray et al. [67] provide a detailed survey of various IoT cloud platforms and compare their suitability in the wide context of heterogeneity management, device management, deployment, visualization, system management, tool & analysis to name a few.
Second, the IoT development platforms and operating systems are required for the application development on heterogeneous devices. There are a number of IoT development platforms such as Arduino [68], ARM mbed [69] and Energia [70] to name a few and also vendor specific platforms that targets the vendor specific device architecture. IoT applications built using one platform are not easily portable to another architecture. Moreover, there are currently many operating systems (OSes) for constrained devices such as RIoT [71], Zephyr [72], Contiki [73] and TinyOS [74] to name a few. These OSes are equipped with the necessary building blocks - communication protocols, network stack, device drivers and hardware abstraction layer - for developing embedded IoT applications and support multiple device architectures.
Our analysis of the IoT platforms state of the art shows a clear need to understand the IoT heterogeneity due to the IoT device hardware and Operating systems which accounts for a lot of industrial non open interfaces, but also open source based interfaces that are mainly used for fast prototyping such as Arduino, RIOT OS, Zephyr OS, etc.
Regarding our analysis of the IoT cloud service platforms, the existing industrial solutions show a common standard based on Internet protocol suite that solves the heterogeneity related to the protocol communication between

the edge/gateway to the IoT service in the cloud, however there is no joint standard on the IoT data management and processing used in each IoT cloud platform, thus making it complex to move from one IoT service cloud platform to another. This type of high level heterogeneity related to the IoT data processing and management is not handled in this thesis. Later, in Section 2.2.2 we review in detail the various IoT development platforms and operating system platforms.

2.1.2 Interoperability in the IoT

There are a number of interoperability definitions available in the literature in various contexts. The IEEE defines interoperability as *"the ability of two or more systems or components to exchange information and to use the information that has been exchanged"* [75]. The ISO/IEC 2382:2015 information technical - vocabulary defines interoperability as *the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units* [76]. According to these definitions, interoperability is needed to enable platforms that are communicable, operable, and programmable across devices, regardless of their make, model, manufacturer, industry or organization [77]. In the context of IoT, Noura et al. [24] classified IoT interoperability into four categories. 1) Network Interoperability. 2) Syntactic Interoperability. 3) Semantic Interoperability. and 4) Device Interoperability.
In this section, we briefly explain each of the four interoperability dimensions to better understand the research and development efforts in IoT heterogeneity control and the various interoperability solutions presented in Section 2.2.

2.1.2.1 Syntactic Interoperability

Syntactic Interoperability in IoT is associated with the data formats, encoding and decoding schemes used in the exchange of information or services between heterogeneous systems [78]. The syntactic interoperability issues arise from the great variety of formats that are used to encode information such as XML, JSON, RDF, etc. In case of format incompatibility the receiver will not have the syntactic rules (defined in the format's grammar) to decode the information which hinders the exchange of data or services between systems. The use of common standard formats can provide syntactic structure to the exchanged information that allows syntactic interoperability between systems.

2.1.2.2 Semantic Interoperability

The World Wide Web Consortium (W3C) defines semantic interoperability as *"enabling different agents, services, and applications to exchange information, data and knowledge in a meaningful way, on and off the web"* [79]. Semantic interoperability is achieved when the exchange of data is made consistent between the interacting systems irrespective of the original data format [80]. This consistency is achieved by use of existing standards, formats, metadata or it can be achieved dynamically using shared vocabularies either in schema form or ontology driven approach.
The semantic technologies - RDF Schema, Web Ontology Language (OWL), Web Services Description Language (WSDL), etc. - that are used for the web services to enable interoperability are also commonly adopted in IoT [81]. In this context, Murdock et al. [80] provides a comprehensive survey on enabling technologies for semantic interoperability for the web of things.
Towards this goal, the Web of Things (WoT) working group [82] from W3C are trying to resolve fragmentation in IoT and enable interoperable IoT devices and services through APIs. On the other hand, the Semantic Sensor Network (SSN) ontology [83] by W3C is a set of ontologies for describing observed properties, features of interest, sampling strategies for both sensors and actuators. It allows for proper interpretation and utilization of data generated by sensors.
Finally, the syntactic and semantic interoperability is related to systematic exchange of information between various IoT cloud service platforms thereby enabling cross domain IoT applications. This type of interoperability is not handled in this thesis.

2.1.2.3 Network Interoperability

According to Noura et al. [24] Network interoperability allows seamless message exchange between IoT devices operating on heterogeneous networks. Due to the variety and heterogeneity of IoT communication protocols (Section 2.1.1.3) the end-to-end communication link (M2M and between IoT devices & IoT cloud server) is not compatible and hinders the direct exchange of messages between IoT devices. Network interopability is associated with the network, transport, session and application layer of the OSI model and addresses issues related to addressing, routing, resource optimization, security, QoS and mobility support [84, 85].

2.1.2.4 Platform Interoperability

According to Bröring et al. [86] platform interoperability enables the emergence of cross-platform, cross-standard, and cross domain IoT services and applications. This will allow developers to create applications by combining data from multiple platforms and also platforms from multiple domains.
In this section, we describe IoT platforms related to IoT cloud service platforms, IoT device hardware and operating system platforms. Platform interoperability issues, in IoT arises due to the availability of diverse IoT platforms (Section 2.1.1.5) that are rarely interconnected. All platforms are vertically oriented and closed system that provide their own domain specific APIs, services, programming language, operating systems, protocol suite, and target hardware architecture. This results in fragmentation of the IoT ecosystem into many vertical IoT silos that hinders application developers to develop cross-platform and cross-domain IoT applications.
In our work (Chapter 4), we propose a framework that combines various hardware platforms and allows application developers to develop applications in a hardware agnostic way. Finally, in Chapter 5 we propose a new modular hardware platform for designing interoperable embedded systems.

2.1.2.5 Device Interoperability

Noura Et al. [24] defines device interoperability as enabling the integration and interoperability of heterogeneous devices with various communication protocols and standards supported by heterogeneous IoT devices. It allows the exchange of information between heterogeneous devices (Section 2.1.1.1) that uses heterogeneous IoT communication protocols (Section 2.1.1.3). In the absence of shared common communication protocols and also the continuous evolution of wireless technologies is hindering the advancement of interoperable communication between devices. Also due to the presence of heterogeneous IoT devices there are still no adopted common standards for programming these heterogeneous devices and most of the different hardware vendors offer their own development tools and solutions. This makes the industrialization of the IoT services difficult as the Proof-of-Concept (PoC) phase is expensive due to this heterogeneity and lack of common shared standards. In our work (Chapter 4), we propose a framework that aims to leverage IoT adoption and usage from design to deployment and better handle the heterogeneity property of the IoT device, services and applications.

2.2 Interoperability and IoT Device Management Solutions

In this section, we review various interoperability and device management solutions from Industrial Alliances, Standards Organizations, Industries and Academia.

2.2.1 Industrial Alliances and IoT Standards

There are many industrial alliances and standards organizations in the IoT and Machine-to-Machine/M2M community whose sole purpose is to define and deliver specifications for IoT and M2M so that the connected devices are able to communicate with one another and with IoT application server in the cloud regardless of manufacturer, operating systems, silicon vendor and physical transport.

These specifications are defined in the form framework which provides interoperability guidelines for the industries to develop platforms or tools that complies with the specifications defined by the framework. In the context of IoT heterogeneity control, different industrial alliances define different frameworks. Additionally international standardization bodies are actively working on the best standard framework for IoT communication protocols and APIs. In this section we review a selection of most used frameworks defined by standards organization and industrial alliances such as - oneM2M, OCF, IETF, ETSI, Thread group and Zigbee Alliances.
Industrial alliances such as - oneM2M [87] and OCF [88], define an IoT architecture in the form of reference framework to overcome the interoperability problem.
oneM2M [87] defines a horizontal architecture using a common framework and uniform APIs that enables application across multiple domains. It defines an IoT service layer in the form of software middleware between IoT applications and sensor nodes. This IoT service layer is like a distributed operating system that provides uniform APIs to IoT applications. The uniform APIs help to cope with complex and heterogeneous connectivity choices and abstracts the details of the underlying network technologies, transport protocols and data serialization. The list of common services defined by the IoT service layer includes - secure end-to-end data/control exchange between IoT devices, authentication, authorization, encryption, remote provisioning

& activation, connectivity setup, buffering, scheduling, synchronization, aggregation, group communication, device management, etc. It also defines service layer messages as oneM2M primitives that are generic with respect to underlying network transport protocols. These primitives are binded to underlying transport layer protocol such as HTTP, CoAP, MQTT or Websocket and reuses existing IP-based protocols and non-IP nodes are supported via interworking proxies.

On the other hand the architecture defined by OCF [88] is based on Resource Oriented Architecture design principle, where each entity (e.g. wired or wireless sensor actuator device) are represented as Resources. Interaction between entities are achieved using operations that adhere to REST architectural style. The OCF architecture defines the framework of information system and the interrelationship between entities. The entities expose themselves as Resources with unique identifiers (URIs) and support RESTful operations on Resources. Any device acting as a client can initiate a RESTful operation on a device acting as a server. The architecture is organized using three aspects - *Resource model*, *RESTful operation* and *Abstraction*. The *Resource model* is similar to oneM2M primitives and provides the abstractions and concepts required to model the entity. The entity is represented by a Resource and encapsulates the state of an entity. The resource model is independent of any specific application domain and provides communication protocol interoperability by mapping the resource to the transport protocol to enable communication between the entities. *RESTful operations* provides five generic operations - CREATE, RETRIEVE, UPDATE, DELETE and NOTIFY (CRUDN) - defined using RESTful paradigm to model the interaction with a Resource. These operations are independent of any protocol & technology, like oneM2M these operations are also mapped onto the underlying transport layer thus using existing IP-based protocol. *Abstraction* contains two abstraction primitives. The first is the entity handler which maps an entity to a Resource and second is the connectivity abstraction primitives used to map CRUDN RESTful operations to data connectivity protocols or technology.

Similar to oneM2M, OCF also defines semantics for common service/functions needed across multiple domains and leaves the development, network and application specifics to others thus creating an IoT ecosystem which is open to collaboration and provides interoperability among various IoT organizations and industries.

Both oneM2M and OCF framework handle the interoperability up to the

nodes that are IP enabled, which mainly includes IoT gateways like devices. There are a number of IoT end devices that are not IP-capable and are widely used in IoT related industrial and home automation applications.
There are number of implementations that exist for both oneM2M and OCF, for example OpenMTC [89] is a backend and gateway side implementation of oneM2M with limited number of supported protocol adapters, whereas OS-IoT [90] is device-side library that implements oneM2M defined network & protocol functions and therefore reduces the effort needed to hook IoT devices into oneM2M ecosystem but requires considerable device resources to run effectively. On the other hand IoTivity-Lite [91] is a light-weight implementation of OCF but still requires resources at least that of class 2 devices (Table 2.2).
Furthermore, some industrial alliances are pushing dedicated frameworks based on their protocol stack to respond to specific industrial vertical needs. In this regard, the *Thread Group* alliance [51] promotes specific standards for the home automation industry that uses 6LoWPAN over IEEE 802.15.4 wireless protocol with mesh network for communication. Zigbee Alliance [53] on the other hand uses its own networking protocol over IEEE 802.15.4 and supports star, tree and mesh networking that is not compatible with IP enabled devices. IP connectivity on the other hand is supported by *Zigbee IP* that provides support for full IPv6-based mesh networking. Zigbee protocol is widely used in many verticals such as home automation, industrial control and building automation to name a few. Wi-SUN (Wireless Smart Ubiquitous Network) [55] alliance promote technology that are based on UDP/TCP and IPv6/RPL/6LoWPAN over IEEE 802.15.4g/802.15.4e with mesh network topology for wireless smart utility and smart city applications such as advance metering, distributed automation, municipal lighting, smart parking, environmental sensing, etc.
While each industrial alliance promotes their own protocol, there are no real successful attempts to cover a cross domain standard for the IoT ecosystem. Meanwhile, international standardization bodies such as ITU, ETSI, IETF are all providing their specifications for standardized connected IoT architecture and protocols. The IETF focuses mainly around IP compliant approach for interoperability. For this, it has developed a protocol suite and open standards for accessing applications and services for resources constrained devices and networks. Sheng et al. [57] provides a detailed survey on IETF protocol suite for the IoT. Despite the IETF IoT standards such as CoAP, 6LOWPAN, RPL,

6Tisch, etc. the real IoT services deployment is following more industrial alliances as mentioned above than a universal standard. This is due mainly to the lack of memory, processing and energy resources in most of the IoT electronic devices such as monitoring sensors that are lacking resources to run for instance the IETF 6LoWPAN. On the other hand, ETSI-M2M [92] framework by ETSI provides a RESTful horizontal functional architecture for applications to share common infrastructure, environments and network elements. The ETSI-M2M architecture has three domains - device & gateway domain, network domain and application domain - with a generic set of service capabilities for M2M on top of connectivity layers deployed in M2M networks, gateways, and devices. The standards defined by ETSI-M2M are adopted by oneM2M as discussed above.

2.2.2 IoT Frameworks and Platforms Survey

In the context of IoT application lifecycle management (development, deployment and maintenance), Atzori et al. [1] explains the need of the software layer between application and technology to simplify application development and integration of new technologies with legacy ones. Also for widespread adoption of IoT-based system and services an intermediate *software abstraction layer* is required to hide the details of various technologies underlying IoT [93, 9, 11]. Towards this goal as described in Section 2.1.1.5 there are several solutions in the market in the form of - development platforms, operating systems, middleware, etc. which results in a heterogeneous IoT ecosystem. All these solutions try to hide the vast variety of the underlying IoT hardware technologies and enable easy application development by utilizing hardware abstraction layers, APIs, libraries, runtime systems, etc. These solutions lack IoT friendly interface and functionalities for scenarios description, device deployment and maintenance. In this section, we briefly describes some of the published and commercial solutions for IoT application life cycle management and their limiting factor.

2.2.2.1 Programming Frameworks

In the context of IoT application development on embedded IoT devices one can mention the Wiring programming-framework [94]. The Wiring framework is the underlying framework for embedded IDEs (Integrated Development Environment(s)) such as Arduino [68] for AVR, ARM, ESP, stc based device architecture and Energia [70] for TI (Texas Instruments) MSP based device architecture. The Wiring framework allows easy interaction with hardware by exposing a set of high level functionalities in the form of Wiring-APIs, which abstracts various hardware peripherals like - UART, SPI, I2C, PWM, GPIO, etc. The device drivers are offered as libraries that are built using Wiring-APIs. Commercial embedded IDEs (listed in Table 2.4) are also available from various semiconductor vendors and software companies which offer similar kinds of hardware abstraction. Although embedded IDEs are famous for developing device software and firmware, they still require knowledge of various wireless technologies, hardware & peripheral configuration, electronic systems, etc.

Persson et al. [95] propose a framework named Calvin for application development and a platform for deploying and managing applications. The framework inherits the idea from Actor model and Flow Based Computing (IFTTT [96], NoFlo [97], etc.) that divides the application into four well defined parts - *describe*, *connect*, *deploy* and *manage*. *Describe* - Actors communicate through in/out ports, data is consumed on in port and the results are sent on out port. Calvin defines the concept of Actor as a usable software component representing anything from a device, a service or a computation The actors run on a Calvin runtime environment which provides an agnostic view of the underlying system to the developers. *Connect* - it forms a directed graph between actors in/out port and CalvinScript is used to implement this connection. *Deploy* - For deployment it is assumed that the Calvin runtime is running on all the devices in the scenario and have access to this runtime. Calvin runtime forms a mesh network and the application script is passed to one runtime and it migrates to the destination node. The distribution of application script depends on the deployment algorithm. *Manage* - once the application is running it enters the managed phase and the execution environment monitors various states of actors, resources and does update and error recovery. The Calvin runtime architecture - starting from bottom there is a hardware followed by an OS layer than a platform dependent runtime environment and finally on top there is one more platform independent runtime

environment on which actors of the applications run. The platform dependent runtime handles the communication between runtime and other high level abstraction. The proposed solution requires an OS and two runtime environments that make it difficult to run on constrained devices.
Soursos et al. [98] takes on platform interoperability and describes the problems associated with highly specialized IoT vertical solutions that are affecting the sustainability of IoT. The author presented his views towards an ecosystem where IoT platforms cooperate to jointly address cross-domain challenges. They presented their approach in the H2020 project - symbIoTe. It highlights the requirement of cross-domain solutions as IoT platform federation, domain enabler, cooperation and collaboration by sharing of resources and new business models in the IoT value chain. The project enables various IoT platforms to collaborate at various levels - device domain, smart space domain, cloud domain and applications domain. It uses the existing IoT reference architecture (IoT-A ARM, oneM2M, WoT, etc.) and defines the compatibility of symbolIoTe requirements with them, thereby creating an ecosystem of existing IoT platforms that is interoperable. The symbIoTe architecture is divided into four layers - 1) Device Domain - where devices will have symbolIoTe clients which helps in the initial introduction of the devices within a smart space and other things. 2) Smart Space Domain - It defines symbIoTe middleware that exposes a standardized API for resource discovery and configuration and implements sensor discovery protocol for a simplified integration of sensors with platforms. 3) Cloud Domain - It defines an interworking interface to exchange information between IoT platforms at cloud level apart from platform specific building blocks. 4) Application Domain - it offers an high level API for a unified view on different platforms to enable cooperation and support cross-platform discovery and management of IoT resources.
Blackstock et al. [99] presents an hub based approach for interoperability that uses web technologies (HTTP, JSON, REST architecture) to expose things to the web. The hub provides facilities for search, (meta) data storage and interaction between things and applications. They provide the specification for exposing a diverse set of IoT resources - real time sensor data feeds, metadata, static data sets for describing things. The specification is based on a lightweight hypermedia catalogue for querying and representing IoT resources (URIs) on the web. The exposed resources are described by a list of RDF-like triple statements. Every hub exposes a top level catalogue to applications, this catalogue represents an unordered list of items that refers to

a single URI. It specifies how to insert, update and delete these catalogues. They used their own web centric IoT toolkit - WoTKit for managing real-time data and external open source tool CKAN to support static data and metadata. They also build a proxy that provides an application with a unified interface to access these tools, a third party search engine - Apache Solr - to both store and search catalogues. To deal with heterogeneous data from a set of disparate sources they developed a tool called Harvester which is based on CKAN Harvester plug-in. Their approach is similar to the hub based projects like - Xively and ThingsSpeak.

Chatzigiannakis et al. [100] introduces the term "true self-configuration" that refers to the capability of a node to automatically identify nearby devices, query information from the web, combine local and remote data to generate meaningful new information. They extended the Semantic Web technology to form a semantic IoT, such as for the machine-understandable and domain-independent representation of data they used Resource Description Format (RDF), the data are linked with Linked Open Data (LOD) and to query the IoT and data sources from the internet, SPARQL and data from LOD cloud are used. Their system is built around their two core contributions - "Semantic Data" and "Smart Self-Annotation". For semantic data they used existing semantic web technologies such as RDF and Linked Data (LD). For smart self-annotation instead of using existing work such as SensorML [101], Transducer Electronic Data Sheets (TEDS) [102] and Semantic Sensor Network (SSN) Ontology [83], they compared the output of new sensor to already deployed sensor with known metadata and with the assumption that sensors with similar metadata will output similar stream of sensor data. They used fuzzy-based methods to associate membership with already deployed sensors. Another feature of their project is "Semantic Actuation" - which uses "Semantic Web Rule Language" (SWRL) [103] from "W3C" to formulate complex rules over RDF data. These rules are evaluated to trigger actuation and is similar to IFTTT.

The existing approaches presented in this section still require considerable resources to run effectively on constrained devices or at least require knowledge of various underlying technologies such as communication protocols, device architecture, etc. In Chapter 4, we propose our development framework named PrIoT that provides a lightweight approach for implementing IoT applications on heterogeneous IoT devices, including resource constrained (Class 0) IoT devices. In addition, it also provides a high level application interface (PrIoT-API) and programming language (PrIoT-Lang) that are IoT

devices and communication protocol technology agnostic.

2.2.2.2 Development Platforms and Tools for IoT Devices

In IoT platforms, one can refer to the well known platforms - Arduino [68], Energia [70] and mbed [69] that combine the embedded system boards and components with their programmation language and design tools. Arduino platform is based on an open-source wiring programming-framework [94] and supports a variety of microcontroller architectures such as AVR, ARM, ESP, etc. that can be programmed using a C-like programming language with easy to use integrated development environment (IDE). In terms of hardware, Arduino boards have an interface to connect "Shields" - add-on circuit boards to enhance the capability of Arduino boards (See Section 2.3.1.1 for more details). Arduino boards together with Shields are designed for easy to use and include all the necessary circuitry for quick and rapid prototyping. On the software side, Arduino provides an extensive set of libraries and a lot of example codes for effortless learning. Furthermore, Arduino is backed by a large community of hobbyists, engineers and professionals to create, share and support libraries and examples online. Although Arduino is the most favorite platform for IoT proof-of-concept it has many disadvantages making it not suitable for commercial and industrial use. In fact Arduino is restricted to a small number of architecture and is not portable to other microcontrollers, its libraries are very inefficient in terms of resource usage (RAM, ROM, CPU cycles, etc) and therefore has low performance as compared to bare-metal implementation. In addition, Arduino boards are expensive and are not suitable for large scale deployment. In addition, Arduino boards are expensive and are not suitable for large scale deployment.
Inspired by Arduino, Energia platform is also based on the Wiring programming-framework and provides an IDE similar to Arduino. Energia supports only Texas Instruments (TI) development boards and device architecture. Since Energia is based on the same wiring framework as Arduino, it has the same limitations - programs written in Energia are not portable to other microcontrollers. On the other hand, The mbed platform [69] is an initiative from ARM and provides an online IDE and operating systems (mbed OS) for the development of embedded applications. The platform supports microcontrollers based on ARM Cortex-M architecture. In ARM mbed the development and contribution is supported at two levels. The first level - *Core Platform*, includes all the generic software components and the HAL which allows mbed

to transparently run on different ARM cortex-M based microcontroller manufacturers. At second level - *Component Database*, the open community is involved in the development of libraries to support peripheral components, sensors and protocols that are needed to build applications on the IoT-end devices. As of now, mbed IDE is restricted to online code editor & compiler and is not available for offline use, which brings in the question of code privacy and security.
Also depending on the amount of resources available on embedded hardware such as processing power, memory size, etc, only a limited set of functionalities can be realized. Table 2.3 shows the list of various hardware architecture and the type of functionality they can support based on available resources. For example a gateway needs to be powerful enough to provide functionalities like - encryption, device orchestration, protocol translation, database hosting, sensor data post processing and high level application packages. These functionalities require an embedded OS which can only be run on more powerful embedded systems (MPUs) compared to simple connected objects that rely only on lightweight MCUs. On the other hand, small devices can't host an embedded OS and need to be programmed by vendor specific language that is not portable to other devices thus reducing code reuse. This requires a common platform to integrate the development of hardware applications for both MPUs and MCUs.

	Architecture		**Functionalities**		
Name	**MCU**	**MPU**	**IPv6**	**AES**	**Gateway**
ARM Cortex-Mx	✓	✗	✓	✗	✗
Atmel AVR	✓	✗	✗	✓	✗
TI-MSP430	✓	✗	✗	✗	✗
Intel x86	✗	✓	✓	✓	✓
ARC	✗	✓	✓	✗	✓
NIOS II	✗	✓	✓	✗	✓
Tensilica Xtensa	✗	✓	✗	✗	✓
RISC-V	✗	✓	✗	✗	✓
Nordic	✓	✗	✗	✓	✗

TABLE 2.3: Embedded System Functionalities

In the context of tools for device programming, there are numerous semiconductor vendors that provide microcontrollers for IoT hardware development and different vendors have different development tools as shown in

Table 2.4. Embedded firmware developers are forced to procure, learn and install new tools for every time they switch to different vendors. Also finding proper libraries and sample code for sensor, actuator and communication devices is time consuming which makes hardware testing and prototyping more difficult. PlatformIO [104] is one such tool that overcomes this problem. It is an open source IDE with a collection of cross platform tools for IoT device programming and provides a unified ecosystem for embedded code development for various hardware platforms and is equipped with a code editor, library manager, toolchain and debugger that supports more than 400 development boards.

On the other hand, Eclpise IoT [105] from Eclipse foundation provides a collection of open source software that is divided into three independent IoT software stack namely - IoT device stack, IoT gateway stack and IoT cloud stack. IoT device stack includes projects like - *Eclipse edje* [106] (Hardware asbtraction layer, JAVA APIs for MCUs - "Android for IoT"), *Eclipse paho* [107] (C implementation of MQTT), *Eclipse Wakaama* [108] (C implementation of OMA LWM2M). IoT gateway stack includes *Eclipse Kura* [109] project and IoT cloud stack include *Eclipse KAPUA* [110] project.

Bröring et al. [86] presents the architecture of platform interoperability through the H2020 project "BIG IoT". The aim of the project is to enable cross-platform, cross-standard and cross-domain IoT service and application by building an IoT ecosystem of interoperable IoT platforms. It provides key functionalities like - advertising, dynamic discovery, automated orchestration and negotiation of services. They explained their architecture of interoperability through "BIG IoT" APIs that offer seven well defined functionalities - 1) Identity Management - for resource registration. 2) Discovery of resources. 3) Access to (meta) data. 4) Tasking - to forward commands to things. 5) Vocabulary management - for semantic description of concept 6) Security management and 7) Charging - to allow monetization of assets. They also suggest the five interoperability patterns that need to be supported by IoT platforms for interoperability among platforms.These patterns are based on a well defined syntax and semantics of the interfaces and categorized as - 1) Cross-platform access. 2) Cross-application domain access. 3) Platform independence. 4) Platform scale independence. 5) Higher-level service facades. The BIG IoT APIs are designed to enable these patterns. Although the interoperability architecture introduces the concept to enable platform interoperability but fails to explain the implementation details and how it can be adapted to constrained IoT devices.

In our work (Chapter 4) we propose our development tool that implements the features provided by our PrIoT framework. It provides a lightweight development of embedded application across diverse hardware platforms.

2.2.2.3 Embedded Operating Systems

There are currently many embedded operating systems (OSes) such as Riot [71], Zephyr [72], TinyOS [74] and Contiki [73]. They are designed to hide the heterogeneity of underlying IoT hardware platforms and hence provide a solution for device interoperability. They clearly need more hardware resources than an application built using platforms such as Arduino, Energia and mbed. OSes are designed to provide much higher level of hardware abstraction by dividing software components into two parts - hardware-dependent (peripheral drivers, CPU & Board related codes, etc.) and hardware-independent (kernel, libraries & network code, etc.). Each embedded OSes have their own APIs, data structure, libraries and supported list of hardware platforms thus creating many vertical silos of operating systems that are not compatible with each other. We focus on three open source OSes - RIOT [111], Zephyr [72] and contiki [73] - to show the advantages and disadvantages of using embedded OSes for prototyping connected-devices.
RIOT [71] is a free and open source real-time microkernel-based OS developed to adapt to the network of constrained IoT devices. It leverages the capabilities of hardware with both constrained as well as abundant resources and provides support for network protocols like the standard IETF 6LoWPAN and RPL for constrained devices and also full support for IPv6, UDP and TCP. Because of the RIOT hardware abstraction layer (RIOT-HAL) between RIOT kernel and hardware, it is possible to build complete embedded software which can be easily ported to any hardware supported by RIOT. The RIOT-HAL avoids redundant code development and reduced application maintenance cost. Although RIOT provides hardware independent approach for application development which is beneficial for rapid and flexible prototyping, as of now it supports a certain number of hardware boards but provides the necessary resources to add new boards.
Zephyr [72] is a linux foundation hosted collaboration project. Like RIOT, Zephyr is a small, scalable and open source real-time operating system which is driven by community to support new hardware, developer tools, sensor and device drivers. It supports standards like IETF 6LoWPAN, CoAP, IPv4, IPv6, NFC, bluetooth, bluetooth low energy (BLE), Wi-Fi and IEEE-802.15.4.
Like RIOT and Zephyr, Contiki is another tiny operating system for resource

constrained microcontrollers. Unlike RIOT and Zephyr, the contiki kernel is by default designed as event driven, that means kernel processes are implemented as event handlers which run to completion. The event-driven kernel design simplifies the design of OS. Contiki implements preemptive multithreading as an application library and is not a part of OS-kernel. Contiki supports fully standard IPv6 and IPv4 along with low power wireless standards like - IETF 6LoWPAN, RPL, CoAP, UDP, TCP and HTTP. RIOT, Zephyr and Contiki support multiple hardware architectures as shown in Table 2.4 and covers an extensive set of hardware architectures for gateway and constrained devices. Although these embedded OSes are equipped with the necessary building blocks (communication protocol, network stack, device drivers and hardware abstraction layer) for developing embedded applications, they still lack the IoT friendly interface to develop applications on distributed networks IoT nodes. Also there is a plethora of embedded devices (Class 0, see Table 2.2) that do not have enough resources to support embedded OSes but they are still widely used. In Chapter 4 we propose high level interface in the form of PrIoT-Lang and PrIoT-API that not only hides the underlying device heterogeneity but also ease IoT device application development.

	OSes			Non-OS			
Name	**RIOT**	**Zephyr**	**contiki**	**mbed [69]**	**Arduino [68]**	**Energia [70]**	**Vendor Specific Tools**
ARM Cortex-Mx	✓	✓	✗	✓	✓	✗	ARM Mbed
Atmel AVR	✓	✗	✗	✗	✓	✗	Atmel Studio
TI-MSP430	✓	✗	✗	✗	✗	✓	Code Composer Studio
Intel x86	✓	✓	✓	✗	✗	✗	Intel System Studio
ARC	✗	✓	✗	✗	✗	✗	DesignWare ARC
NIOS II	✗	✓	✗	✗	✗	✗	Nios II Embedded Design Suite
Tensilica Xtensa	✗	✓	✗	✗	✗	✗	Xtensa Xplorer IDE
RISC-V	✗	✓	✗	✗	✗	✗	RISC-V GCC
Nordic	✓	✓	✓	✗	✗	✗	nRF5 SDK

TABLE 2.4: Embedded system programming diversity from bare metal (non-OS) to OS approaches

2.2.2.4 Language-based Approaches

IoT application development complexity is also regarded by language-based approaches that are gaining popularity, for example [112] proposes a language-based approach for interoperability between heterogeneous isolated IoT platforms (IoT islands) at both transport and application level. Their approach is based on the service-oriented language - *Jolie* [113]. It allows the reuse of the same logic over disparate communication stacks (HTTP-TCP, CoAP-UDP, MQTT, etc.) thereby maintaining interoperability among protocols. This interoperability is achieved by dividing the description of collectors (gateway

interface that send and receives data to and from devices of different IoT islands) into two parts : behaviour (logic) and deployment (how communication is performed), as a result the behaviour of the program can take any communication method described in the deployment part. As of now their approach is limited to hardware (Class 2 devices - See Table 2.2) that can support JVM (Java Virtual Machine). Eclipse mita [114] is another programming based approach that allows easier programming of IoT applications for developers without embedded development background. Mita syntax and APIs provide hardware platform agnostic view to developers allowing easy programming of connectivity modules, sensor, actuators, etc. The Eclipse mita is new project and only support limited number of hardware platforms. It is also important to mention the role of Domain Specific Language (DSL) in projects like - IoTLink [115] and IoTSuit [116]. In IoTLink inherits the concept of IoT-A. It allows developers to compose software representation of physical objects through model-driven approach. The application is defined in a platform-independent model through visual notations, which is then transformed in a platform specific model based on Java programming language. The implementation of IoTLink is based on various eclipse plugins - Eclipse Modeling Framework (EMF), Eclipse Graphical Modeling Framework (GMF), Extended Editing Framework (EEF) and Acceleo. But the tool assumes that the IoT hardware (sensor node) is available off the shelf and that it can be accessed through any one of the popular communication protocols.

On the other hand, the IoTSuit toolkit divides an application into different concerns and integrates a set of high level languages to specify them. IoTSuit still requires experts or at least expert knowledge at various stages of IoT application development. The domain expert generates a domain vocabulary which contains domain-specific concepts specific to target IoT application. The domain vocabulary is then made available to experts at various stages of IoT application development, like - network manager, software designer, device developer and application developer. The tool generates a vocabulary framework to aid device developers in developing embedded device drivers, although they integrated an open-source sensing frameworks for android devices leaving device developers to only implement interfaces. But this will greatly restrict IoT scenario designers to limited number of sensors, actuators and communication devices.

In our approach we also follow similar language based approach not only for IoT application logic development but also for IoT scenario configuration as

explained in Chapter 4.

2.2.3 IoT Device Management

IoT device management (DM) plays an important role in IoT lifecycle management. Provisioning for DM helps in the *deployment* and *maintenance* of IoT end-systems as well as edge-systems. Sheng et al. [117] defines DM as integration of network, system and application managements. It includes functionalities, but not limited to, device & service provisioning, configuration & control of device behavior and network parameters, firmware update, registration & management of services, performance monitoring, etc.
It is clear that in case if all IoT devices were IP enabled, they will inherit from Internet based device management frameworks such as SNMP [118], TR-069 [119] and other cloud based internet management frameworks. IoT architecture as explained in previous section having the Internet protocol stack available only between the IoT gateways to the IoT application servers in the cloud, the Internet based management protocols can be used up to the IoT gateways that are running IP protocol stack and this can be extended and adapted to the IP-like IoT end devices.
Due to this IoT device heterogeneity there are number of device management solutions and standards available for both constrained and non-constrained devices with IP and non-IP networks. DM solutions over non-IP networks such as LoRaWAN, Sigfox, WirelessHART, Zigbee, Z-Wave, etc. are isolated within their own vertical silos that hinders their usability across heterogeneous communication network. On the other hand DM solutions and their associated problems over IP networks are extensively studied and proposed in the literature, standards and industrial organizations. Some of these solutions - [48, 49, 89, 90, 91, 95] - have already been covered in Section 2.2.1 and 2.2.2.
One of the promising device management architecture is OMA (Open Mobile Alliance) LWM2M [120]. It defines a lightweight weight M2M device management protocol for managing constrained (memory and power) devices. The specification includes both client and server side architecture. The LWM2M protocol is based on REST architecture that is built on top of CoAP and follows protocol & security standards from IETF. There are a number of implementations and performance evaluation of OMA LWM2M available in the literature for example, Rao et al. [121] showcase the implementation of client side architecture of OMA LWM2M specification on contiki based IoT

nodes and evaluated the performance in terms of memory footprint.
Similar to OMA LWM2M, Sheng et al. [117] also proposed a framework for CoAP-based DM over IPv6 network and includes five DM functions - Registeration, Provisioning, Management Services, Observing and Application data transmission. These functionalities share common resources on sensor devices and are abstracted as parameters, status and data. Interaction with IoT client (cloud application) can be directly triggered with these resources via CoAP methods.
On the other hand, there are number of application specific DM solutions such as, *TR-069* (CPE WAN Management Protocol) [119] that defines specification for remote management of CPE (customer-premises equipment) such as, set-top box, router, etc. over IP network, *Field Device Integration* (FDI) for the management of field devices in control system applications [122]. *SNMP* (Simple Network Management Protocol) that defines a network management protocol architecture for IP-based networks [118]. Similar to SNMP, *NETCONF* (Network Configuration Protocol) [123] is another network management protocol maintained and standarized by IETF is used for monitoring and configuration of IP based networks. All these application specific solutions require considerable device resources making them unsuitable for resource constrained devices used in the IoT.
IoT cloud platforms (Section 2.1.1.5) also support proprietary DM solutions mostly over IP based networks and includes services like registration, organization, monitoring, status, over-the-air (OTA) update, etc. all accessible via web applications. DM solutions from cloud platforms are vendor specific and operates independently within their own verticals.

2.3 IoT Hardware Heterogeneity Control in Embedded Systems

In this section, we investigate the various hardware approaches for heterogeneity control in embedded systems especially for designing Internet of Things (IoT) based systems. The hardware solutions presented in this section tries to control the IoT device peripheral interface heterogeneity as discussed in Section 2.1.1.1. To tackle the device heterogeneity, in our research we decided to investigate the hardware approach to control IoT device heterogeneity due to the diversity of its hardware components. In our work we argued that solving the IoT device heterogeneity control and building end to

end interoperability also requires another design approach in the IoT hardware part. In Chapter 5 we propose our contribution based on a modular approach for designing embedded systems that provides a uniform homogeneous standard interface to better meet IoT device requirements.
The section is organized as follows, in subsection 2.3.1, we briefly introduce the concept of modular architecture and systems from engineering design perspective and its importance. Then in subsection 2.3.1.1, various existing modular systems are explained in details along with their limitations. Finally, in subsection 2.3.1.2 we review various power optimization techniques in IoT and state the importance of energy aware IoT applications. In addition, we state the requirement of intelligent modular systems that can cater to various energy limitations of IoT device and applications.

2.3.1 Modular Architecture and Systems

Modularity in [124] is described as the use of common units to create product variants. The use of modular architecture [125, 126] requires identification of independent, standardized, or interchangeable units to create a variety of functions. There are three categories of modularity as described in [125] - *Component-swapping modularity*, *Component-sharing modularity* and *Bus modularity*. For our work, we focused on *Bus modularity* because existing embedded systems uses this approach to partition the system into independent units and is suitable for designing practical modular embedded systems. In *bus modularity*, all modules are connected to a single common module through an interface bus. For example, in embedded systems various modules such as sensors, actuators, transceivers, memory, etc. are connected to a common processing unit through various peripheral interface buses [15, 14]
The concept of modular architecture [126] is not new in the embedded world, for example modular smartphones [127, 128] where the essential elements like camera, display, battery and processor can be replaced easily due to damage or an improvement in current technologies without replacing the entire phone.
In the research community the modular architecture been studied in many contexts for example, modular sensor network [129, 130, 131, 132], application specific modular IoT systems [133, 134, 62], modular hardware platform for edge computing [135], educational learning & teaching [136], plug & play interface for modular systems [137, 138].

Another advantage of modular systems apart from controlling device peripheral heterogeneity is that it allows to replace part of the system without replacing or removing the complete system. This is important because electronic devices are not designed to last, this makes electronic waste (e-waste) one of the biggest waste stream contributors [139] and IoT related hardware will become one of the main sources of e-waste.

2.3.1.1 Existing Modular Systems and their Limitations

In this subsection we investigate some of the popular design choices for building modular embedded systems for prototyping IoT applications and also their shortcomings.
Before presenting the rest of the section, we define few terminologies to maintain uniformity in understanding the existing work and our contribution that follows in Chapter 5.

- *IoT Resources :* With respect to processing unit an IoT resource is an auxiliary circuitry that provides capabilities in the form of sensors, actuators, wired or wireless transceivers, HMIs, or any other similar capabilities that help in the realization of IoT based nodes.
- *Main-Board :* An embedded board that contains at least one primary processing unit, companion circuits such as - power unit, debug interface, etc. and *optional* IoT resources.
- *Auxiliary-Board* - An embedded board that contains one or more IoT resource or optional secondary processing unit.

In the context of modular system design there are a number of design choices. Each of them differs in form factor, number of supported peripherals along simultaneously access to these peripherals, demo boards and the way the main-board is connected with auxiliary-board(s).

M2.COM : The M2.COM [140] platform module is a main-board standard based on 2230 M.2 form factor with 75 pin host interface connector, the module measures $30 \times 20\ mm$. The platform specification combines wireless connectivity with processing unit onto a single host module (main-board) and the auxiliary board is developed separately from main-module that carries an equivalent 2230 M.2 key-E socket for connecting the main board. Figure 2.1 depicts an M2.COM system along with example board in Figure 2.2. The

idea behind M2.COM modular design is to allow sensor integrators and sensor makers to select most efficient way to transmit data. The 75-pin connector exposes various embedded peripherals listed in Table 2.5.

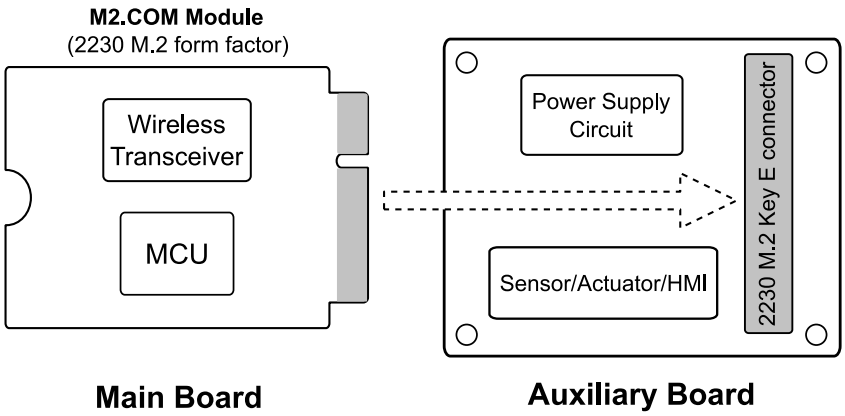


FIGURE 2.1: M2.COM : Main-Board and Auxiliary-Board

FIGURE 2.2: M2.COM - Main Board

Micro Bit : Micro Bit [141] (main-board) is an open source ARM-based embedded system designed by the BBC (British Broadcasting Corporation) for its use in computer education. The size of the board is 43×52 *mm* and carries an ARM Cortex-M0 based processor, sensors, buttons, LEDs, Bluetooth, USB and external battery connector. Micro Bit uses 25 pin edge connector to interface with external auxiliary-board that carries either a 90° or 180° compatible female edge socket. The 25 pin edge connector exposes various peripherals listed in Table 2.5. The main disadvantage of Micro Bit is that it is designed for ARM based processor and is mainly used for computer education.

mikroBUS^{TM} : The mikroBUS^{TM} [142] is an auxiliary-board standard created by MikroElectronika [143] that defines the specification for sockets on the main-board and add-on boards (auxiliary-boards). The standard specifies

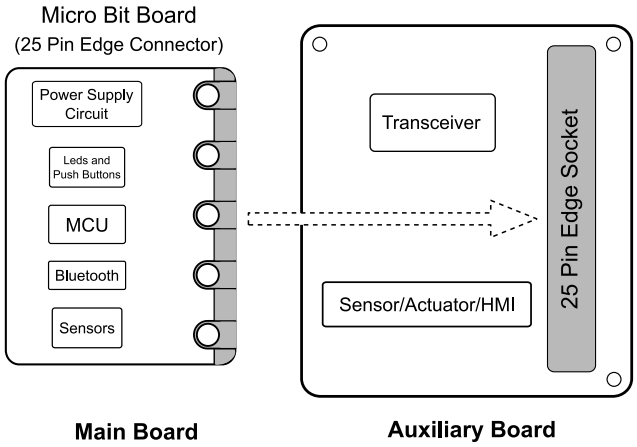


FIGURE 2.3: Micro Bit : Main-Board and Auxiliary-Board

FIGURE 2.4: Micro Bit - Main Board

the physical layout of the mikroBUS^{TM} pinout, the size & shape of auxiliary-boards and the positioning of the mikroBUS^{TM} socket on the main-board and finally the silk screen marking conventions for both sockets and auxiliary-boards. Figure 2.5 depicts mikroBUS^{TM} system and along with example board in Figure 2.6. The mikroBUS^{TM} socket comprises a pair of 1x8 female header on the main-board with proprietary pin configuration that offers various peripherals listed in Table 2.5. The auxiliary-boards are known as mikroBUS^{TM} add-on boards and each such boards provide additional functionalities - wireless connectivity, sensing, actuation, HMI, etc. - to the system. The size of the auxiliary-boards are limited to three sizes S (25.4×28.6 mm), M (25.4×42.9 mm) and L (25.4×57.15 mm) and because of this it is difficult to provide multiple IoT resources on a single auxiliary-board and as a result the main-board may require multiple mikroBUS^{TM} auxiliary boards. This creates variability in main-board size (form factor) from one IoT application to another.

Pmod Standard : Pmod (Peripheral Module) standard [144] is an open standard defined by Digilent Inc. [145]. Interface boards that adhere to Pmod

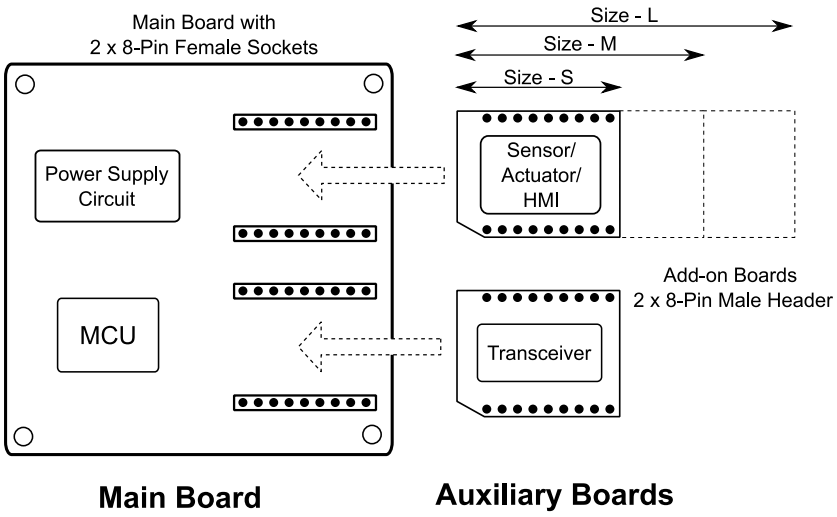


FIGURE 2.5: mikroBUS^{TM} : Main and auxiliary-boards

FIGURE 2.6: MikroBUS - Auxiliary Board

standard are known as pmod modules (auxiliary-boards). These modules communicate with the main-board using 1x6, 2x4, or 2x6-pin right angle *female socket* that carry standard peripherals and digital control signals. Pmod standard is a guideline for - form factor, peripheral pin mapping, reference manual, example code, user guides and technical support.

Figure 2.7 depicts a pmod based system. The connectors on the pmod modules are standard *male pin-header* style connectors, these connectors are generally right angle at the edge of the boards for direct connection to the main-boards. Similarly the main-board carries an equivalent *female socket* connector on the edge of the board. Although the width of the pmod module is not prescribed in the standard and due to this some modules might interfere mechanically with the adjacent boards, but when multiple *female sockets* are used on the main-board then the width of the module is restricted to 20.32 *mm* in order to avoid mechanical interference.

Like mikroBUS^{TM} the limitation of Pmod is that it only supports SPI, I2C, and UART protocols and standard digital control signals. Most pmod modules provide single peripheral due to a limited number of pins, for example a 1 x 6-pin *female socket* provides either SPI or UART or I2C, but a pair of 2 x

6-pin can provide SPI or UART or I2C on each 1 x 6-pin separately. Table 2.5 lists various peripherals supported by pmod along with size specification. Figure 2.8 shows an example of Pmod board.
The drawback of both mikroBUS^{TM} and Pmod system is that multiple auxiliary-boards are needed if an IoT application requires multiple IoT resources, which can effect the design of main-boards from one application to another.

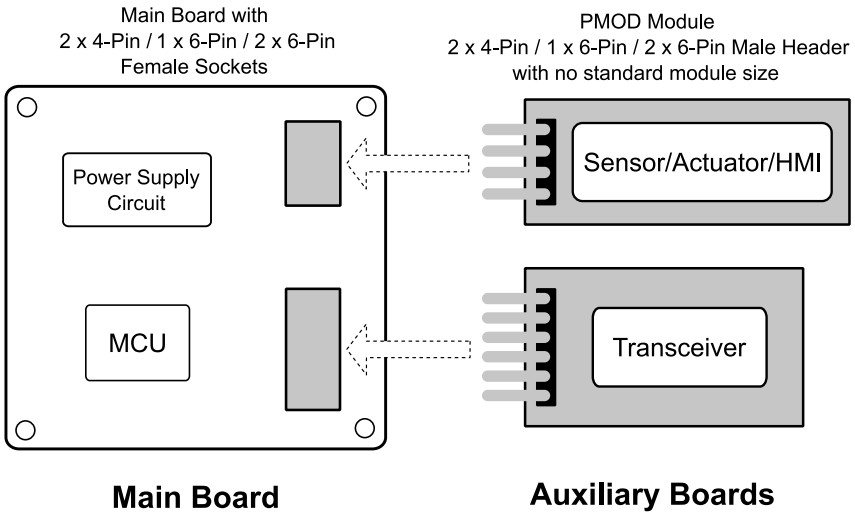


FIGURE 2.7: PMOD : Main-Board and Auxiliary-Boards

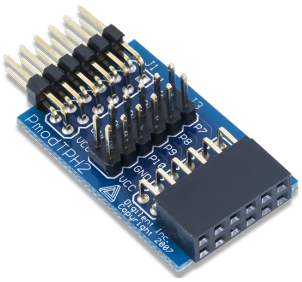


FIGURE 2.8: PMOD - Auxiliary Board

Grove System : Grove system [146] is another building block approach to assemble electronics using standardized grove connectors. Both main and auxiliary-boards (grove modules) carries similar grove connectors and are connected together using a grove cable. The grove connectors are 4-pin standardized size connectors and are keyed to prevent plugging them backwards. There are four different kinds of connectors based on the type of embedded peripherals, *Grove Digital* - provides two general purpose input output, *Grove Analog* - with two analog input, *Grove UART* and *Grove I2C* for UART and I2C respectively. The drawback of grove system is that it only

supports four types of embedded peripherals and not all peripherals are simultaneously available on a single board. Figure 2.9 shows an example of Grove auxiliary board.

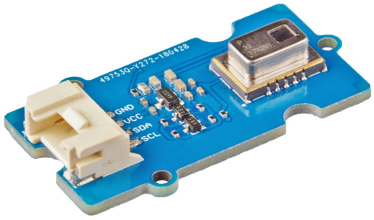


FIGURE 2.9: Grove System - Auxiliary Board

Arduino Shields : *Shields* are one of the older and widely used pluggable board architecture for Arduino based systems. *Shields* (auxiliary-board) mounted on the Arduino board (main-board) gives extra capabilities in terms of IoT resources. The nominal size of *shield* is 68.58×53.34 *mm*, which provides standard *male pin-header* style connectors (known as "shield connectors") for mounting directly onto the main board that has an equivalent *female socket*. The 32 pin connector exposes various standard peripherals like - SPI, UART, I2C, upto 6 PWM & Analog each, 2 interrupts and upto 20 GPIOs. Due to its popularity with Arduino based system these boards has been adopted for other non-Arduino based boards, like STM Nucleo [147], NXP LPCXpresso [148]. Figure 2.10 shows an example of Arduino *Shield*.

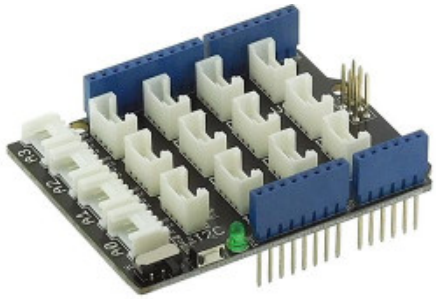


FIGURE 2.10: Arduino Shield - Auxiliary Board

Raspberry Pi HATs : It is important to mention modular systems based on single board computers that allows prototyping of IoT gateways, access points, routers, proxies and edge devices due to the support of full operating systems like Linux.
In this category, HAT (Hardware Attached on Top) [149] is an add-on board (auxiliary-board) for Raspberry Pi (RPi) [150] family of low-cost single-board

computers (main-board). Figure 2.11, shows an example of HAT on top of RPi board. A HAT that confirms to HAT-specifications allows RPi to identify the connected HAT and automatically configures the kernel (Linux) device tree [151] using an external EEPROM (Electrically Erasable Programmable Read-Only Memory) that holds various device meta-data. This helps kernel to setup the physical interface and load the specific HAT drivers at boot time. In order to facilitate this all official HAT has an EEPROM that hold apart from meta-data, information about interface setup and a fragment of device tree known as device tree overlay. During boot the overlay is merged with the device tree and directs the kernel to load the relevant drivers. In order to access the HAT correctly, this assumes that the drivers are pre-installed in the kernel or provided by the manufacturer of the HAT. The 40 pin HAT interface exposes various peripherals listed in Table 2.5. Finally, there are several disadvantages of using HATs, first the official HAT-specification is not designed to allow the use of multiple HATs on a single main-board because the EEPROM specification does not have meta-data that can distinguish between various attached HATs and therefore the last read EEPROM will override the previous device tree overlay hence only one HAT will be recognized, which might introduce shared pin conflicts with other HATs. Second, the HAT is specifically designed to be compatible with RPi based main-board that hinders its use with other main-boards.

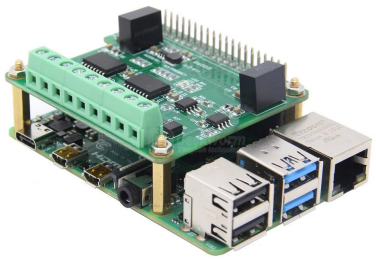


FIGURE 2.11: Raspberry Pi HAT - Auxiliary Board

BeagleBoard Capes Similar to RPi HATs, BeagleBoard (BB) capes are daughter boards for single board computers based on BeagleBone and PocketBeagle family of boards. Capes also has an EEPROM which fulfills a similar function to that of HAT-EEPROM (interface & driver setup), but the data format is not compatible, unlike RPi the BB has the ability to accept up to four capes that can be stacked on top of the BB. The information about which pins and features used by capes are stored in the EEPROM on each cape. The

processor used in BB supports multiple features (peripherals) via pin multiplexing and therefore each pin can be configured for various features allowed by each pin. Figure 2.12, shows an example of BeagleBoard Cape auxiliary board.

FIGURE 2.12: BeagleBoard Cape - Auxiliary Board

Table 2.5 summarizes various peripherals supported by individual modular systems along with form factor (size).
In order to complement our proposed software oriented approach in Chapter 4, we proposed in Chapter 5 a hardware oriented approach where we design a new modular system for designing interoperable embedded systems that better caters the IoT device peripheral heterogeneity in comparison with existing modular systems. We also showed using quantitative analysis that our system is better suited for implementing embedded systems for IoT scenarios.

Name	Size ($w \times l$ mm)	Embedded Peripherals								
		SPI	UART	I2C	I2S	SDIO	PWM	Analog	Interrupt	GPIO
M2.COM$^{\circ}$	22×30	$2^{\star}$	1	2	1	1	upto 2	upto 6	$*$	upto 16
Micro Bit$^{\circ}$	43×52	1	1	1	0	0	upto 3	upto 6	$*$	upto 3
mikroBUS$^{TM\bullet}$	25.4×57.15	1	1	1	0	0	1	1	1	0
pmod$^{\dagger\bullet}$	$20.32 \times l^{\ddagger}$	1	1	1	1	0	upto 2	0	upto 1	upto 8
Grove System$^{\dagger\bullet}$	No Standard	0	1	1	0	0	0	2	0	2
Arduino Shield$^{\bullet}$	68.58×53.34	1	1	1	0	0	upto 6	upto 6	2	upto 20
RPi HAT$^{\bullet}$	65×56.5	2	1	1	1	1	upto 4	0	upto 28	upto 28

†Not all supported embedded peripherals are available on a single board, $^{\circ}$main-board standard
$^{\bullet}$auxiliary-board standard, ‡no prescribed standard length, *no explicit mention of dedicated interrupt lines

TABLE 2.5: Comparison Between Various Modular Systems

2.3.1.2 Power Requirements in Embedded Systems for IoT

A lot of IoT applications are relying on IoT devices with limited resources of processing, storage, communication but also energy. Certain IoT applications are using other IoT devices with enough resources such as connected vehicles, connected sensors in home automation with sensors energised from the

mains electricity. In this work we are interested in the power constrained IoT devices as it is for the moment widely used for different sensing and monitoring IoT applications. Moreover, the true power of IoT lies in the fact that in the real world scenario the network (wired and wireless) of IoT devices are expected to be operated in energy constrained environment with multi year of lifetime. This requires IoT devices to better cope with various external power and energy sources. Garg et al. [152] provide a comprehensive survey on various energy sources available that are used in energy harvesting applications. In order to understand the energy limitations of IoT devices, Bormann et al. [25] introduced the terminology for energy constrained devices and partition the devices into various classes according to energy limitations as shown in Table 2.1.

Arshad et al. [13] discusses and evaluates the best strategies for minimizing the energy consumption for their vision of Green IoT 2020. To maximize the wireless node's lifetime the node has to exploit energy aware techniques in circuits, architecture, algorithm and protocols [153]. One method to understand energy consumption of protocols is to model the wireless power consumption of IoT devices [154], although it is not an easy task as it requires many technology dependent parameters, nevertheless the power models are good for comparing various wireless technologies at an early stage of technology selection. For example, Casals et al. [155] presents analytical models of LoRaWAN nodes's current consumption that is derived from the measurements performed on existing prevalent LoRaWAN hardware platform, these models are useful to study the impact of various LoRaWAN physical and Medium Access Control (MAC) parameters (data rates, acknowledge transmission, payload size and bit error rate) on power consumption and ultimately to have a rough estimate of battery life.

There are also other techniques to reduce power at the protocol level, for example Adame et al. [156] proposes an energy efficient protocol stack for multi-hop communication for LPWANs technology.

Sinha et al. [153] has developed an energy aware embedded operating system (OS) that uses dynamic power management techniques such as shutting down sensor nodes if no event occurs and wake them up when necessary. Raghunathan et al. [157] explains the various design considerations at circuit level for designing energy harvesting sensor nodes. From a hardware design perspective, an energy efficient ultra low power wake-up receivers (WURx) [158, 159, 160] are gaining a lot of attentions for reducing power consumption by relieving the main radio from continuously monitoring the channel

for incoming messages. The WURx acts as an auxiliary receiver and wake-up the wireless node from sleep using interrupt, only after detecting a potential incoming message.

Finally, we have studied that in order to implement the above power management techniques ([153, 157, 158, 159, 160, 156]) a wireless node should be capable of dynamically monitoring the available energy source (light intensity, wind speed, vibration, temperature, etc.) & consumption (voltage, current, charge, etc.) and also taking appropriate action to manage power consumption both in software and hardware. In this context, Georgiou et al. [161] discusses the role of software in controlling the energy consumption of IoT devices that requires constant feedback from the hardware on the state of energy consumption. This requires an intelligent modular power supply unit that is usable across diverse classes of energy limitations and can provide the necessary features to the device software to better optimize power utilization during runtime. The existing modular systems as discussed in Subsection 2.3.1.1 does not take into account the energy requirements of IoT application and hence does not provide any power management features.

In summary, our contribution work presented in Chapter 5 is built on the previous research that are presented in this section to identify the various power requirements & management techniques of IoT devices. As a result of which we propose a modular system named *Power-Bus* (Chapter 5 - Section 5.3) that provides the necessary features required for better power optimization and exposes an intelligent homogeneous interface that is usable across various power requirements of IoT applications.

Chapter 3

IoT Application Characteristics and Its Common Invariant Functionalities

3.1 Introduction

This chapter introduces our first contribution that describes the underlying design philosophy of our proposed framework (Chapter 4). We propose 4-layer IoT architecture and most importantly we identified the common invariant functionalities (IFs) and the corresponding programming patterns (PPs) present in most IoT scenarios. These IFs and PPs are needed to reduce the complexity of managing IoT scenarios on heterogeneous IoT devices. These high level abstractions are needed to systematically understand the high level description of IoT scenarios and also for designing software abstraction layer (SAL) that is agnostic to the underlying IoT device and protocol heterogeneity. For validating the IFs and PPs, we incorporated these abstractions in our PrIoT framework (Chapter 4) to manage IoT life cycle on heterogeneous IoT devices.

The chapter is organized as follows, we begin (Section 3.2) with our proposed 4-layer IoT Architecture that will allow us to systematically structure an IoT application scenario, its characteristics and identify the underlying problems associated with each layer. Next, in Section 3.4 we introduce a high level abstraction in the form of programming concepts that captures the most common IoT applications invariant functionalities (IFs) and programming patterns (PPs) that we identified and gathered in a limited and simple list of commands. We identified these abstraction from our study (Chapter 2)

of various IoT architectures used by commercial IoT service providers, industry alliances, industrial standardized bodies and in research community and also in various IoT applications mentioned in the literature as well as commercially deployed successful IoT applications. In Section 3.5, we evaluate our proposed high level abstraction with existing IoT development platforms along with example use cases before concluding the chapter in Section 3.6.

3.2 High Level-Description of IoT Application Scenarios

3.2.1 A proposed 4-Layer IoT Architecture

In this section, we use an IoT architecture approach to understand the structure of IoT application and the problems encountered at various stages of IoT application development. The main idea behind IoT architecture is to help IoT application developers to map application problem statements onto more structured and well defined building blocks, which makes it easier for an application developer to go from a concept to a real world realization. An ideal IoT architecture that has the ability to capture all the characteristics of IoT scenarios is far from reality, mostly due to the multi and interdisciplinary nature of IoT. Although the concept of architecture in IoT is not new [98, 58, 86, 46, 45], the architecture presented here focuses mainly to define and capture the IoT characteristics which forms the basis for IoT application invariant functionalities and programming patterns proposed in (Section 3.4). The architecture illustrated in Figure 3.1 consist of 4 layers, namely Layer-1 (L1) : *Device-Layer*, Layer-2 (L2) : *Edge-Layer*, Layer-3 (L3) : *Cloud-Layer* and finally a Layer-4 (L4) : *Cross-Layer*. A brief description of each layer is given as follows:

3.2.1.1 Layer-1 : Device-Layer

This layer is responsible to *measure physical quantity*, *detect an event* or *control* in the environment of interest. One of the characteristic of this layer is that it consist of (large) network of *heterogeneous* IoT end-devices and to go further in the heterogeneity components considered in this layer compared to the architectures described in the related work, we also consider fine grained hardware components of the IoT devices such as like sensor, actuator, transceiver and processing unit - which together perform the desired application task.

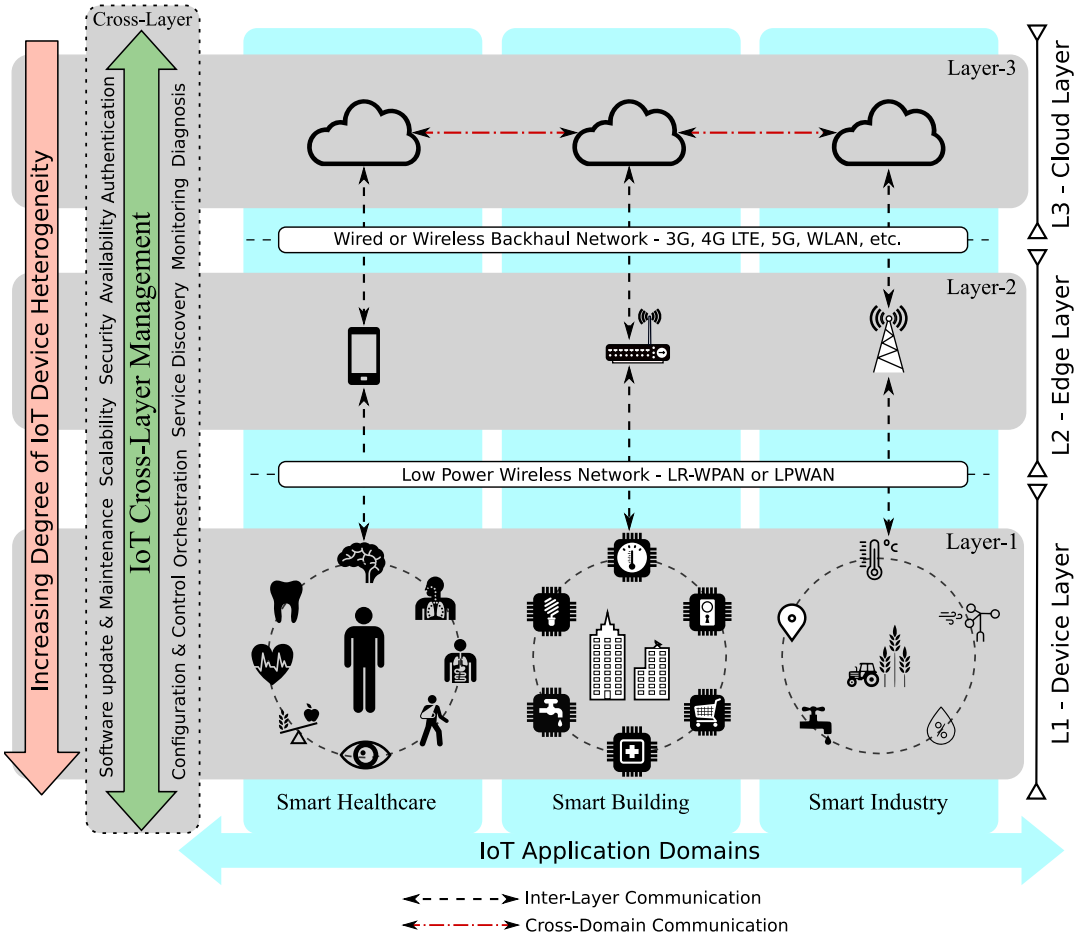


FIGURE 3.1: Proposed IoT System Architecture

These devices are available with varying degrees of capabilities and are provided by various OEMs (Original Equipment Manufacturers). Selecting a particular device for a given application is not trivial, as the application requirements are generally domain dependent [162] and application developers do not have much freedom in defining them. For example a monitoring application in the smart-city domain has a firm requirement of high reliability, low cost and scalability apart from others.
From a given application requirements selecting the right device/hardware is a daunting task because of the following reasons listed below and is a trade-off between interlinked "*selection criteria*" mentioned in Table 3.1 :

1. There are numerous devices available in the market from various hardware vendors therefore selecting a particular hardware is a trade off between many "*hardware requirements*" listed in Table 3.1.

2. The hardware vendors do not have a shared common standard to program these devices, one has to look for the available software development tools both proprietary and open source, software libraries, available operating systems (OSes), technical and open community support before selecting a suitable hardware. This further complicates the device selection process because a suitable hardware for a particular application does not guarantee the availability of suitable software support and vice versa.

3. During maintenance phase the legacy devices are replaced due to, for instance a change in application requirements may impose new hardware features to be added or system upgrade. In this situation the deployed hardware is difficult to upgrade in case the devices are replaced from different hardware vendors, as this will enforce to hire experienced embedded system developers to implement and maintain embedded software codebase from various hardware vendors.

Selection Criteria	**Requirements**
Application	data rate, reliability, battery powered, coverage(range), cost, security, scalability, etc.
Hardware	power, memory, peripherals, MIPS, customer support, open community support, development, evaluation kits, etc.
Software	development tools, networking and communication driver library, sensor-actuator-HMI library, open community support, etc.

TABLE 3.1: IoT Architecture Layer-1 Device Selection Criteria

Another important characteristic of this layer is the availability of a large number of communication and networking protocols (LR-WPANs vs LP-WAN) to choose from. The reason for such a large selection is that these protocols are specifically designed to satisfy application requirements listed in Table 3.1 and there is no one fit protocol that satisfies all applications requirements from various IoT domains. Although this might be reconsidered in the future 5G specifications [163]. There are a number of possible ways a network of IoT end-systems in layer-1 can communicate with the rest of the IoT system as shown in Table 3.2 along with example application and technologies.

Inter/Intra-Layer Communication	**Example Application and Technology**
L1 ↔ L1	M2M, Mesh network, Infrastructure network, Autonomous deployment
L1 → L2 → L3	Monitoring, Data acquisition, Geolocation, etc.
L1 ← L2 ← L3	Command & Control, Diagnostic, Firmware upgrade, etc.
L1 ↔ L3	GSM/GPRS/3G/4G - Monitoring, Data acquisition, Geolocation, etc.
L1 ↔ L2	M2M, Edge Computing, Intranet, Access Network, etc.
L3 ↔ L3	Cross-domain

TABLE 3.2: Inter/Intra-Layer Communication

In conclusion this layer has the highest degree of heterogeneity in terms of both hardware devices, software support, communication and network protocols. It is very difficult for the IoT application developers to escape this heterogeneity and has to rely on intermediate software abstraction layer [19] for device independent IoT hardware programming. In our research we focused precisely on how to handle this complex device heterogeneity to complete the state of the art with our proposed heterogeneity control up to the hardware circuits component of the IoT devices and hide it from the upper layers to ease IoT applications development and maintenance regardless of the IoT devices used and allow fast prototyping.

3.2.1.2 Layer-2 : Edge-Layer

At the very least this layer acts as a bridge to exchange data between bottom-layer and upper-layer. The hardware system used in this layer are known as gateway and are made up of high performance processing unit with abundant resources to support state of the art operating systems, for example the famous *Raspberry Pi* with *Raspbian OS* (a linux based OS). The presence of such operating systems at this layer hides the hardware heterogeneity for gateway-like devices, thereby relieving application developers from knowing the underlying hardware details. The gateway exchanges data to and from the top-layer using well know standard protocols (also known as Internet backhaul) from the application layer of the Internet protocol suite - like

HTTP, MQTT, CoAP, etc. - over wireless/wired Internet technologies like 4G/5G, LTE, IETF IEEE 802.11 (WiFi, Ethernet), etc. In the simplest case a gateway can be a relay that transfers data received from layer-1 directly to layer-3 without any data manipulation (for example - Sigfox, LoRaWAN, etc), but in other cases a gateway might implement some complex cloud services which also referred to as "Fog Computing" or "Edge Computing" this includes functionalities like - sensor data filtering, security, firmware update, network diagnosis, facilitate local data storage & analysis, sensor fusion, start and stop service, protocol interoperability algorithm, Machine learning algorithm, M2M interface, etc. The facility to port complex cloud services closer to layer-1 allows reduce bandwidth utilization and latency for critical applications for example, the well known cloud service providers AWS and Azure extends their cloud services using solution like *AWS Greengrass* and *Azure IoT Edge* respectively for the Edge devices. In conclusion all these compute intensive functionalities facilitate the use of high performance devices and operating systems (Linux, Windows, Unix).

3.2.1.3 Layer-3 : Cloud-Layer

Bulk data received from the previous layer are stored in the cloud for further processing. This layer exposes the required tools to help in the development of various cloud applications, such as visualization, analytics, cross-domain data exchange, cognitive computing, big data, machine learning algorithm, artificial intelligence (AI) and so on. Also this layer exposes close to no hardware heterogeneity as IoT application developers take advantage of available cloud services like SaaS (Software as a Service), PaaS (Platform as a Service) and IaaS (Infrastructure as a Service) from various cloud based service providers (Amazon, Google, Microsoft, IBM, Oracle, etc.). In general the services at this layer are accessed through secure publish/subscribe (pub/sub) protocols such as MQTT, AMQP and RabbitMQ and request/response protocols like HTTP.

3.2.1.4 Layer-4 : Cross-layer

The IoT device management (DM) plays a very important role in the scalability and sustainability of IoT systems by ensuring maximum uptime, preemptive security vulnerabilities & greater customer experience and can be

considered as a backbone of an IoT system. Provisioning for system management helps in the *deployment* and *maintenance* of IoT devices and is also considered an important requirement from business point of view. As a result, there are number of DM solution that exist such as TR-069 [119], LWM2M [117, 121], Field Device Integration (FDI) [122].
Irrespective of application domain an IoT device management possesses the following four functions across all three layers (L1, L2 & L3) :

- *Authentication & Provisioning* - It consists of securely establishing the *identity* of IoT devices (both at L1 and L2) and the process of adding (registration) it into the system.
- *Configuration & Control* - It provides the capability to remotely *configure* and *control* IoT devices (both at L1 and L2) by end users. This capability also spans to L3 for example, configuring the cloud architecture in terms of number of virtual machines, types of database, the selection of machine learning algorithms, etc and configuring the building blocks of an Edge (L2) container or the type of network access to forward data, etc.
- *Monitoring & Diagnosis* - It provides the capability to remotely *monitor* the various parameters of IoT devices at layer L1 and L2 without affecting the normal operation and also the capability to remotely diagnose if required.
- *Software maintenance* - It provides the ability to remotely *update* and *upgrade* IoT end-system application software, firmware, patches, etc. For such cases, devices should embed a bootloader/firmware that has the ability to receive and run updates over the air such as Mender [164].

In general, due to resource (compute & memory) limitations at L1 & L2, IoT system management functions listed above are implemented and executed as cloud services by various "IoT platform" solution providers [165], notable examples are Amazon AWS, Google IoT, Microsoft Azure, IBM Watson IoT, Oracle IoT Cloud Enterprise, etc. In certain scenarios where the latency is of prime importance the optimized version of cloud services are also exported to edge devices (L2), also known as fog computing or edge computing.

3.3 Terminology of IoT Scenarios

In this section, we try to label various disparate elements at each layer - L1, L2, L3 and L4, that makes up an IoT system. This allows us to better organize the IoT scenario understanding and construction, which also forms the basis for Chapter 4 and Chapter 5. As Figure 3.2 shows, an IoT system as a whole is made up of four systems corresponding to each layer namely, ***End-System*** (L1), ***Edge-System*** (L2) and ***Cloud-System*** (L3) and ***CrossLayer-System*** (L4). The system is further composed of various ***Resources*** and ***Connectors*** and are explained hereafter.

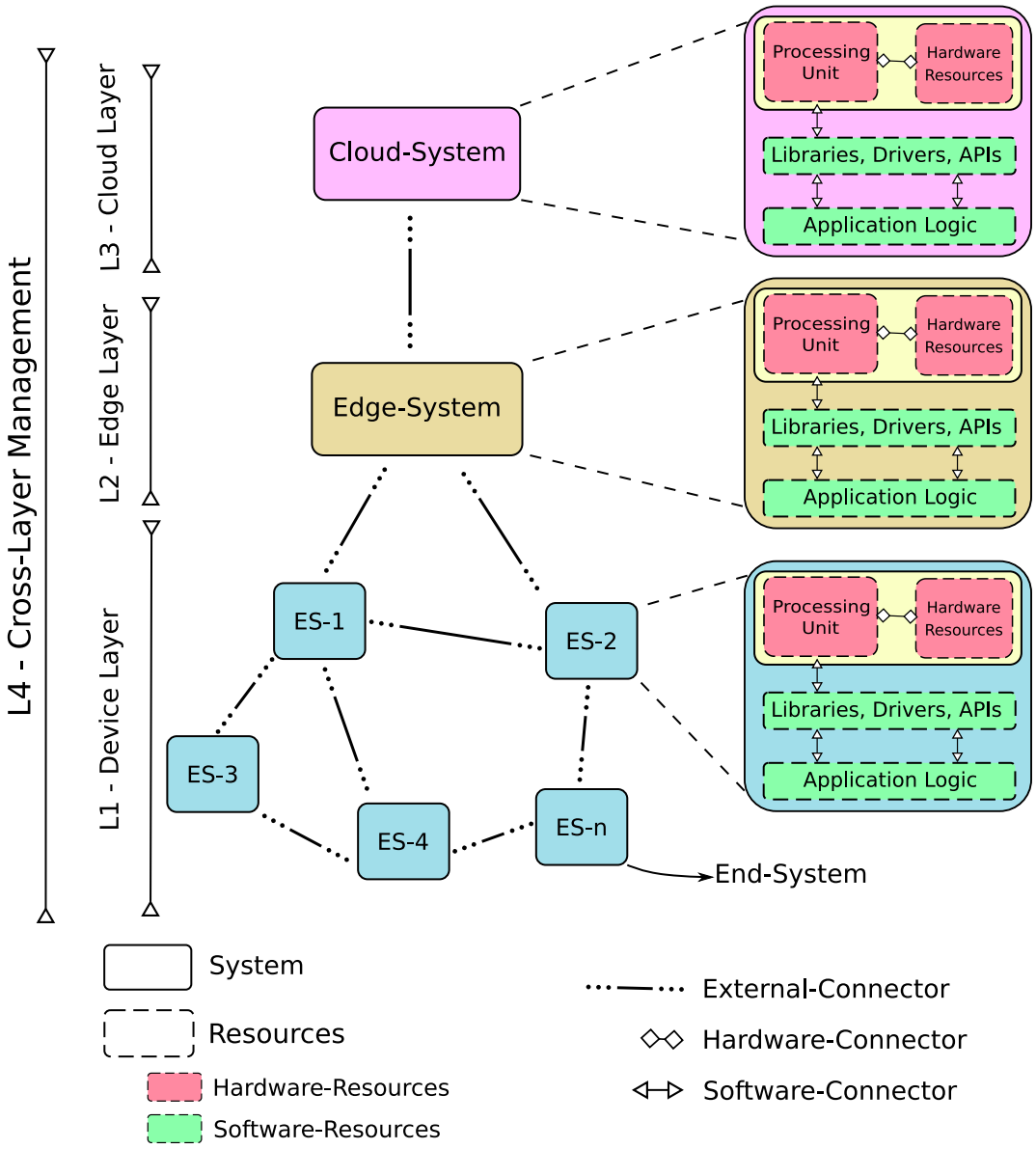


FIGURE 3.2: IoT Scenario - Vocabulary and Concepts

Resources : As shown in Figure 3.3, *Resources* are of two types *Hardware* and *Software*. For example, *Hardware-Resources* includes various hardware devices that makes up a working *End-System* or *Edge-System* such as processing unit, sensors, actuators, transceivers, hmi, etc and on the other hand *Software-Resources* includes hardware independent libraries, APIs, drivers, application software, etc. More details pertaining to *Software Resources* and *Hardware Resources* will be dealt in Chapter 4 and 5 respectively.

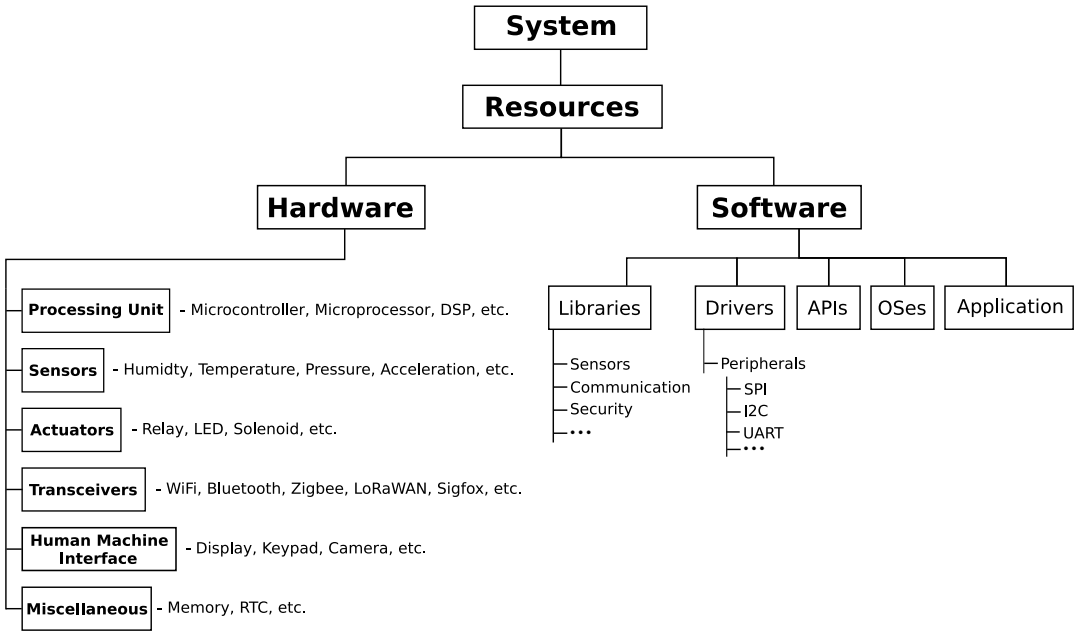


FIGURE 3.3: Resource Hierarchy

Connector : The interaction between and within various systems - (*End-System*, *Edge-System*, *Cloud-System* and *CrossLayer-System*) - is made possible using an abstract interface known as *Connectors*, for this we have defined three types of *Connectors* :

1. *External-Connector* - It defines a communication interface between *End-System*, *Edge-System*, *Cloud-System* and *CrossLayer-System*. A pictorial representation of an external-connector is shown in Figure 3.4.

2. *Hardware-Connector* - It defines a communication interface between various *Hardware-Resources*, i.e. between processing unit and various other resources such as sensor, actuator, actuators, transceivers, hmi, etc. A pictorial representation of a hardware-connector is shown in Figure 3.5. This interaction is made possible by using various embedded peripheral inside the processing unit of the device by exposing its features

to the API. More details about embedded peripherals are discussed in Chapter 5.

3. *Software-Connector* - It defines an interface between various *Software Resources*.

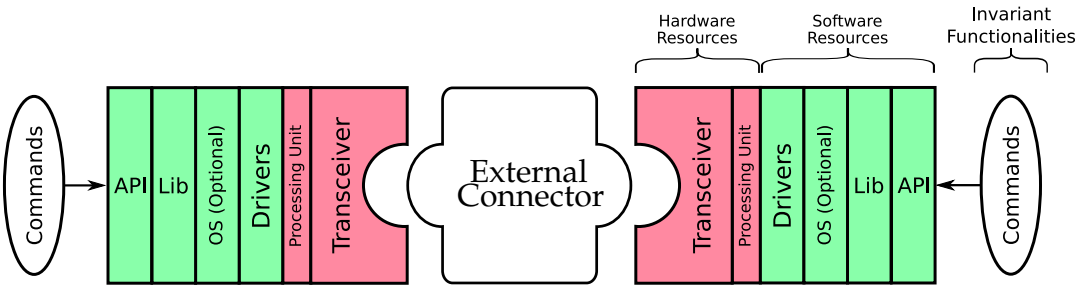


FIGURE 3.4: External Connector

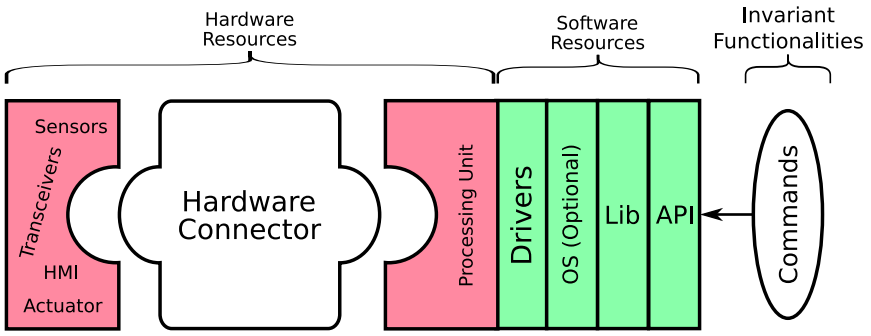


FIGURE 3.5: Hardware Connector

From an application developers point of view, an IoT scenario is structured into three entities - *Cross-Layer Management*, *Configuration* and *Application Logic*.

- *Cross-Layer Management* - The cross-layer management is the front end of an IoT application scenario that allows application developers to describe IoT scenarios using the vocabulary defined above. This acts as a starting point to describe a high level description/view of a scenario without going into technical or implementation details of the scenario. It also handles the IoT device management functions as described in Section 3.2.1.4.
- *Configuration* - The configuration as the name implies defines the configuration parameters of each system. It allows application developers to select various *resources* and configure their associated *connectors*.
- *Application Logic* - It describes the application *behavior* independent of the underlying *resources* and *connector* used.

In summary, for an IoT application scenario the *cross-layer management* allows application developers to manage various systems at L1, L2 and L3. Whereas, the *configuration* allows application developers to select and configure various technologies in terms of *resources* and *connectors*. On the other hand, *application logic* allows developers to write application behavior independent of *configuration*. These three entities are discussed in detail in Chapter 4.

3.4 IoT Applications Invariant Functions (IFs) and Programming Patterns (PPs)

In this section we examine that within each layer of IoT architecture an IoT application performs various invariant functions (IFs) and follows certain programming patterns (PPs). In our work, we consider *invariant functionalities* as well defined functions associated with IoT applications that do not change from one application to another. It provides a useful concept in our work to implement high level abstraction for hiding the underlying heterogeneous device technology and communication protocols. Whereas, programming patterns are execution flow of an IoT application that are reusable across applications, thereby reducing the development time of IoT applications.
The similar IFs have been studied in the past - [166], [167], [168] - for designing programming and domain models for IoT applications, but they didn't examine the programming patterns observed in IoT.
In order to understand, if there exist any functions and programming patterns that are common across IoT scenarios. We analysed different IoT scenarios, and observed variety of common functions and programming patterns found in various IoT applications published in the literature as well as some of the successful FP7-ICT experimental research projects deployed in the real world like SmartSantander [6], TEFIS [8] and ELLIOT [7]. We describe in the following the identified IFs and PPs in each layer of Figure 3.1.

3.4.1 Layer-1 : IFs and PPs

As defined in Section 3.3, *L1* is composed of one or more heterogeneous wired or wireless networks of IoT *End-System* and each IoT *End-System* is further composed of *Hardware-Resources* (processing unit, sensor, actuator, transceiver, HMI, etc.) as shown in Figure 3.6. Irrespective of IoT application

domain and its requirements an IoT *End-System* performs some basic high-level invariant functions as explained hereafter and is also summarized in Table 3.3.

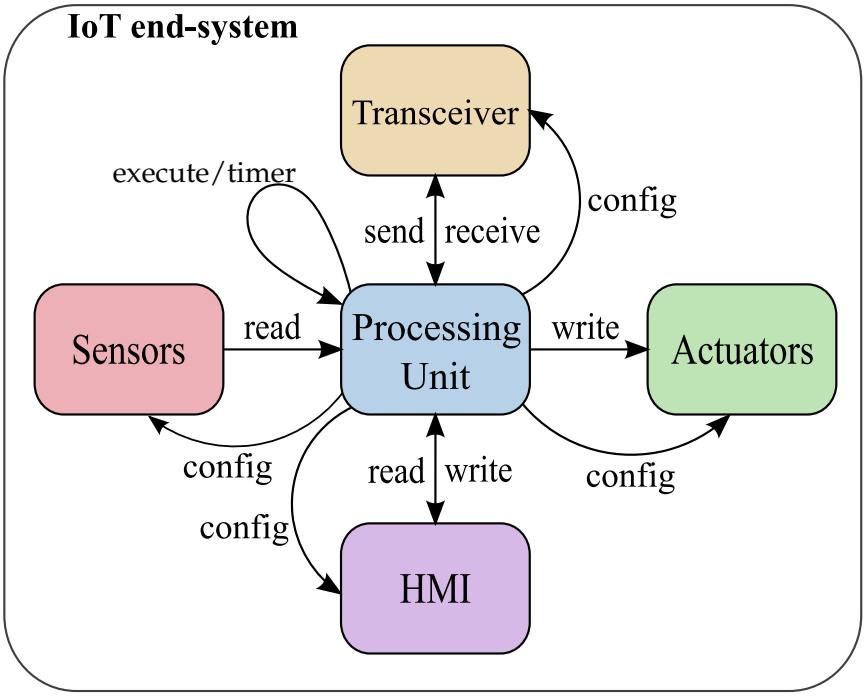


FIGURE 3.6: IoT end-system

Function Name	Description
configure	configure IoT end-devices
read	read data from sensor, memory or HMI
write	write data to actuator, memory, or HMI
send	send data using transceiver
receive	receive data using transceiver
execute	execute user defined function
timer	delay, wait and sleep

TABLE 3.3: Layer-1 - Invariant Functionalities

These functions are programmed in the memory of the processing unit (PU) and are responsible for controlling the behavior of other IoT *Hardware-Resources*. One of the control functions issued from PU is to configure (**configure**) the end-system into desired functional state depending on the type of *Hardware-Resources*. For example the sampling rate of the sensor needs to be configured before using it and also after transmission a transceiver can be configured in low power mode.

Other control functions perform operations that are based on the type of *Hardware-Resources* they are associated with. For example, a **read** function associated with a sensor is used for reading sensor data, while the similar read function for memory is used for reading stored data. Similarly a **write** function is used for controlling the action of actuators and also it can be used to display information on HMI devices like display.
Also based on the underlying network protocol and configuration a PU can issue **send** and **receive** control function for transmitting and receiving data to and from layer above (layer-2 and layer-3) or directly to another IoT *End-System* in layer-1.
A PU can also perform some auxiliary functions for example, an **execute** command can call library functions or user defined custom functions (for eg. an algorithm for local sensor data processing before transmitting).
Finally the **timer** command can execute three low level functionalities based on the application requirements - first is the standard delay operation where a PU waits for a defined time interval, second allows PU to wait for an external event to occur and third is a sleep operation that allows PU to initiate a safe system level power down sequence to save power in a battery operated scenario.
In our work the collection of these invariant constructs forms the basis for an PU to interact with other *Hardware-Resources* in a technology agnostic way. We also observed [4, 169] that an application code for IoT *End-System* (*device element*) follow some basic programming patterns listed in Table 3.4 along with example applications. Although these patterns are limited in scope but can be used to program highly practical IoT scenarios.

Layer-1 Programming Pattern	**Example Application**
read → send → timer	Monitoring without post-processing, etc.
read → execute → send → timer	Monitoring with post-processing, etc.
receive → execute → read → send → timer	Monitoring with post-processing, Actuation, Diagnosis, etc.
receive → execute → write → timer	Actuation, Post-processing, Configuration, Control, Diagnosis, etc.
receive → write → timer	

TABLE 3.4: Layer-1 : Programming Patterns

As Figure 3.7 shows, these programming patterns can be easily visualized as a finite-state machine, where each state is one of the basic functions (Table 3.3) performed by the processing unit. Figure 3.7 also shows the various inter-layer communication that are possible using these programming patterns. Similar types of PPs for layer-1 devices have been recognized in [4] for

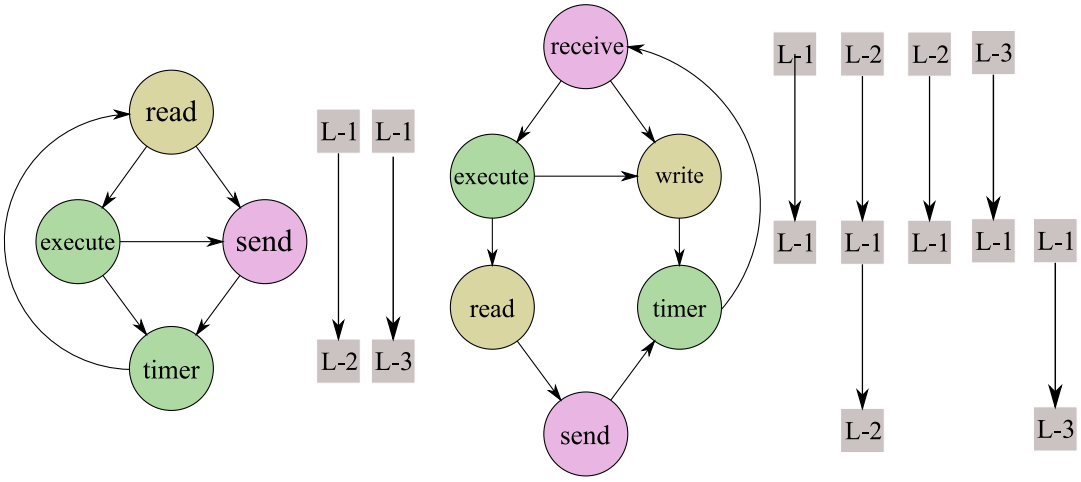


FIGURE 3.7: Layer-1 PP State Machine

general stages of IoT applications that includes - data acquisition, data processing, data storage and data transmission. They are designed specifically according to real-time requirements of an application and whether the data is processed locally or not.
Here we also argue that the device management functions are also covered by these programming patterns which requires systematic exchange of invariant commands between IoT *end-system* and cross-layer management controller, which will be discussed in Section 3.4.4.

3.4.2 Layer-2 : IFs and PPs

The gateways at this layer consist of two sets of **send** & **receive** commands, the first set of commands is used to exchange data from layer-1 using the available low power wireless communication (LR-WPAN or LPWAN, for example) and the second set of commands is used to interconnect the IoT devices to the cloud by exchanging data from layer-3 via Internet backhaul using standard wired or wireless communication technologies such as cellular (LTE, 5G, NB-IoT, etc.), WLAN, etc. One of the important feature of this layer is the capability to execute relevant cloud services closer to layer-1 as discussed in Section 3.2.1.4, therefore a gateway can expose a high level **execute** command that a user can bind to one or more cloud services or micro-services [170, 66].

We have observed that a gateway follows a very simple programming pattern to communicate with layer-1 and layer-3 as shown in Table 3.6 and Figure 3.8.
There are number of solutions exist at this layer and Table 3.5, lists various

open source and proprietary home automation (but not restricted to) solutions, for IoT *edge-system* to integrate and connect IoT *end-systems* to IoT *cloud-system* or it can also be used as decentralized standalone controller for applications that require low latency and privacy. All these solutions provide almost equal capabilities but differ in terms of supported device, protocol and visualization features.

The presence of so many solutions indicates the fact that one needs to adapt and learn new solutions every time there is a decision to switch between various solutions. Here we argue that the high level invariant functions - **send** & **receive** that are visible to application developers can be attached to any device and communication protocols using a configuration file that hides the underlying device details. More details about binding communication protocols with invariant functions using configuration files are discussed in Chapter 4.

Home Automation Controllers	
Ago Control [171]	Calaos [172]
Domoticz [173]	FHEM [174]
Homebridge [175]	Home Assistant [176]
HomeGenie [177]	HomeSeer [178]
Homey [179]	HoMIDoM [180]
Indigo Domotics [181]	ioBroker [182]
Jeedom [183]	nodeRed [184]
MyController.org [185]	Misterhouse [186]
MyNodes.NET [187]	MajorDoMo [188]
OpenHAB 2.x [189]	PiDome [190]
pimatic [191]	XTension [192]
smarthomatic [193]	EventGhost [194]

TABLE 3.5: Layer-2 : Platforms for Implementing IoT *edge-system* Solutions.

Layer-2 Programming Pattern	**Example Application**
receive → execute → send	Edge Computing, Control, Configuration, Diagnosis, etc.
receive → send	Gateway, Packet-forwarding, etc.

TABLE 3.6: Layer-2 : Programming Patterns

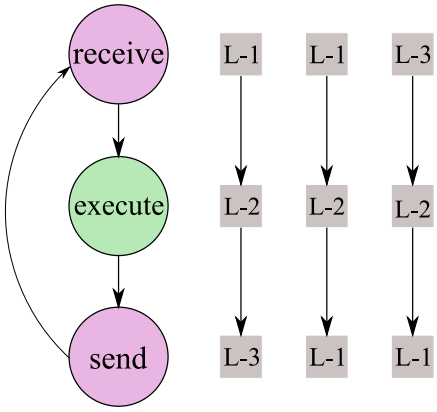


FIGURE 3.8: Layer-2 PP State Machine

3.4.3 Layer-3 : IFs and PPs

Similar to layer-1 and layer-2 this layer also consists of **send** & **receive** commands to exchange data with layer-2 or directly with layer-1 bypassing IoT *edge-systems*. The most important feature is to **execute** various cloud services. There are a number of cloud solution providers that compete [165] at this layer, Table 3.7 lists various cloud based IoT platforms, each of them differs in the number and type of micro-services, tools, supported devices, pricing model, etc.
Figure 3.9 shows PP for layer-3 and the state machine along with inter and intra layer communication. The Table 3.8 summarizes the PP along with example applications.

Cloud based IoT Platforms	
Amazon AWS IoT [195]	Google IoT [196]
Microsoft Azure IoT [197]	IBM Watson IoT [198]
Oracle IoT Cloud Enterprise [199]	ThingSpeak [200]
IFTTT [96]	Eclipse Kapua [110]
Bosch IoT Suite [201]	nodeRed [184]
OpenRemote [202]	CISCO IoT Control Center [203]
Kaa [204]	Predix [205]

TABLE 3.7: Layer-3 : Cloud based IoT Platforms

Layer-3 Programming Pattern	**Example Application**
receive → execute	Visualization, Status, Storage, Monitoring, etc.
execute → send	Control, Configure, Actuation, etc.

TABLE 3.8: Layer-3 : Programming Patterns

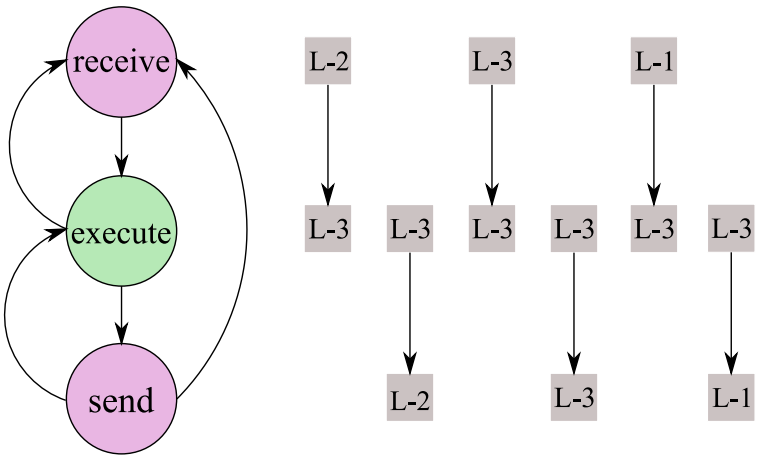


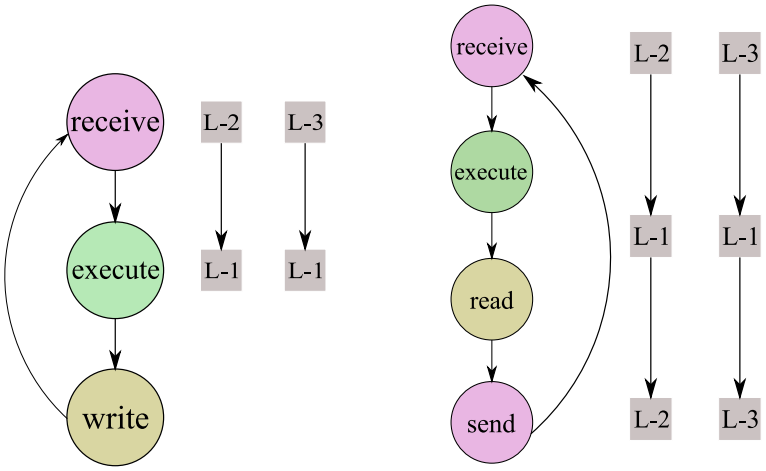
FIGURE 3.9: Layer-3 PP State Machine

3.4.4 Layer-4 : IFs and PPs

As shown in Figure 3.1 the Cross-layer will be in charge of management and control of all the functionalities of the previously described three layers. We recognized four fundamental (atomic) invariant functions (commands) listed in Table 3.9 to accommodate the cross-layer functionalities mentioned in Sub-Section 3.2.1.4. The partition of these cross-layer functionalities into fundamental functions depends on the similar programming pattern it uses, for example **configure**, **control** and **diagnosis** all follows a similar execution sequence : receive → execute → write, where as **upgrade** uses the same execution sequence multiple times until the upgrade is completed as shown in Figure 3.10a. On the other hand **status** follows : receive → execute → read → send as shown in Figure 3.10b, finally **register** which follows the programming pattern : send → timer → receive → execute as shown in Figure 3.11 where an IoT object *sends* an authentication request followed by *waiting* for a response to be *received* and finally decoding the response using *execute*.

Function Name	Description
update	*configure*, *control* and *diagnosis* an IoT end-system.
upgrade	Over-The-Air (OTA) firmware & application software upgrade and patch fixing.
register	IoT *end-system* or *edge-system provisioning* after successful *authentication*.
status	Pull current status and device information.

TABLE 3.9: Layer-4 - Invariant Functionalities



(A) Update and Upgrade PP (B) Status PP

FIGURE 3.10: Programming Patterns for update, upgrade and status IFs

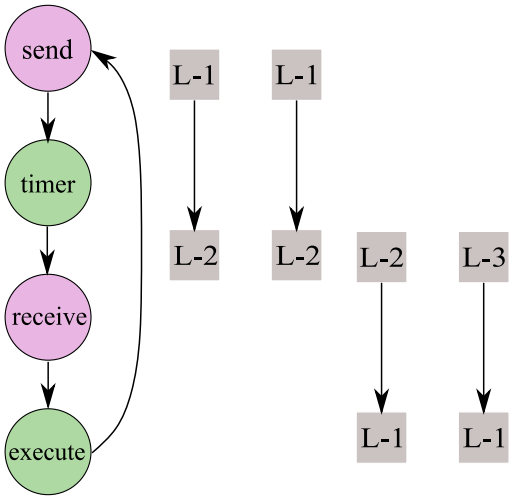


FIGURE 3.11: Programming Pattern for register

In summary, the IFs and PPs presented in this section provides a high level abstraction for IoT application development that are usable across a variety of IoT application domains. These high level abstractions are designed to hide the underlying IoT device heterogeneity in terms of device architecture and communication protocol.

3.5 Evaluation and Analysis

One can realize the effectiveness of any IoT development platform, OSes and framework for implementing end-to-end IoT scenarios by analyzing how

rapid and easy is to implement any IoT scenario (vertical scaling) in various application domains (horizontal scaling) [168]. One factor that is important to consider while comparing is the extent of high level abstraction exposed at various layers of IoT Architecture. In our work we proposed two high level abstractions in the form of IFs that are technology and IoT device agnostic and PPs. For example some of the well known development platforms (Arduino, Energia and mbed and embedded OSes) for embedded applications does not provide application invariant functionalities to aid application developers, whereas PPs can be considered as a subset of any programming language in this case C/C++.

Implementing these IFs using the aforementioned development platforms can be time consuming as the IFs needs to be independent of underlying technology and IoT devices to hide extreme heterogeneity at layer-1 and layer-2, whereas embedded OSes solves this issue by exposing a HAL (Hardware Abstraction Layer) for implementing IFs, but as far as we know till date there are no OSes that exposes these functionalities. On the other hand commercial IoT service providers take the advantage of minimum heterogeneity at layer-3 and provide IFs in the form of cloud services.

In Chapter 4 we explain the implementation of IFs and PPs by separating application logic from IoT scenario configuration. The application logic in an application file that contains IFs and PPs is independent of underlying IoT device hardware and technology, whereas the configuration contains details pertaining to communication protocol, network devices, sensor, actuators, MCU, etc. This separation is helpful in two cases, first when the same application scenario requires for example the use of different communication protocols then only the configuration file is changed and therefore maintaining the integrity of the application logic and second is the reusability of the application logic (IF and PP) for different IoT scenarios. Both these cases help in rapid development of IoT application scenarios. Table 3.10 shows various types of high level abstractions exposed by development platforms, OSes and commercial IoT service providers.

<table>
<tr><td></td><td colspan="4">Extent of High Level Abstraction</td></tr>
<tr><td></td><td>Layer-1</td><td>Layer-2</td><td>Layer-3</td><td>Cross-Layer</td></tr>
<tr><td>Arduino</td><td colspan="2" rowspan="3">Hardware Dependent Standard Libraries</td><td>-</td><td>-</td></tr>
<tr><td>Energia</td><td>-</td><td>-</td></tr>
<tr><td>mbed</td><td>-</td><td>-</td></tr>
<tr><td>RIOT</td><td colspan="2" rowspan="3">OS HAL</td><td>-</td><td>-</td></tr>
<tr><td>Zephyr</td><td>-</td><td>-</td></tr>
<tr><td>Contiki</td><td>-</td><td>-</td></tr>
<tr><td>IoT cloud service providers</td><td>-</td><td>Edge Computing Services</td><td colspan="2">Cloud Services</td></tr>
</table>

TABLE 3.10: Extent of High Level Abstraction Exposed by Various Development Platforms

We validate the usefulness of IFs and PPs by mapping IoT scenarios submitted in *IEEE IoT - IoT Scenarios* [169] onto IFs and PPs. Without loss of generality we selected two IoT scenarios to showcase the mapping from dissimilar application categories. This allows us to show the similarity in IFs and PPs across dissimilar IoT applications. We also extended the original scenarios requirements to accommodate the device management features. The IoT scenarios are listed in Table 3.11, the detailed description of these scenarios can be found in [169]. The first scenario is sense only IoT application describing the one way IoT connectivity and second scenario is sense and actuate IoT application describing two way IoT connectivity.

Scenario Name	**Description**
Pollution Monitoring 2 [206]	Distributed sensor on public transport to monitor pollution
GreenIQ [207]	Smart HUB for garden irrigation

TABLE 3.11: IoT Scenarios

3.5.1 Scenario 1 : Pollution Monitoring

This scenario uses distributed sensors on public transport to generate spatial and temporal pollution levels in the city. Let us assume the sensor node on each public transport has two sensors one for pollution level and other for location (GNSS module), also the sensor node communicates with a gateway using LPWAN like LoRaWAN, Sigfox, etc. The LPWAN gateways are

distributed across the city in such a way to have a city wide coverage. The gateway relays data back to the cloud where an application server generates the map of pollution levels. Figure 3.12 depicts the scenario diagram.

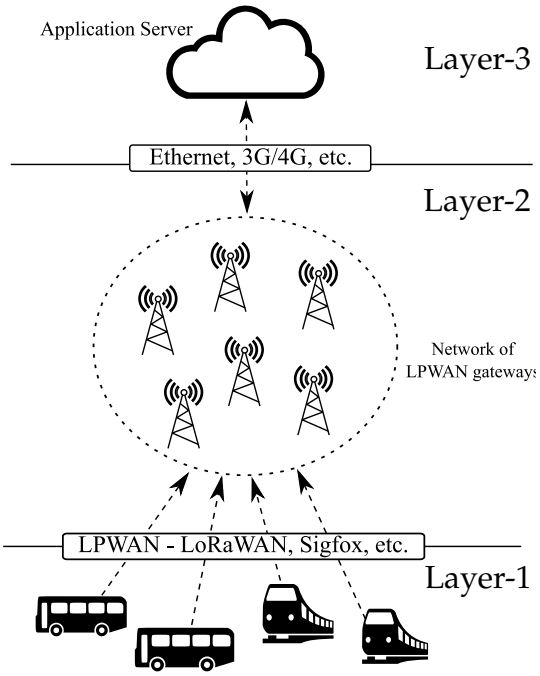
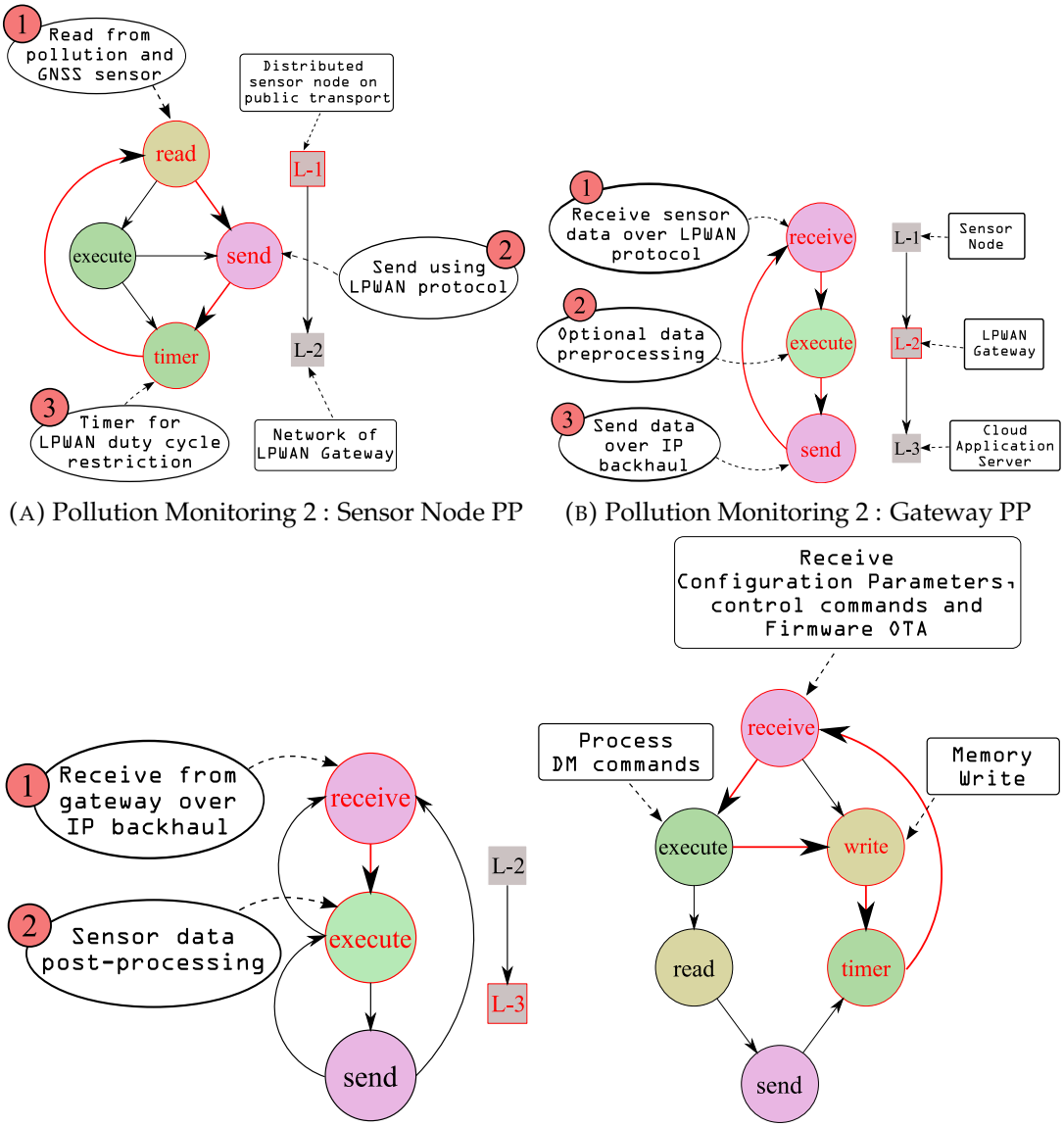


FIGURE 3.12: Pollution Monitoring 2

The sensor node is programmed to transmit sensor data at a time interval governed by regional ISM radio regulation. Figure 3.13a shows the mapping of sensor node functionalities and program execution onto the IF and PP of layer-1. The sensor node *reads* data (pollution level and location) from the sensors followed by data transmission using *send* IF and finally the sensor node goes into sleep and respecting the duty cycle limitation of LPWAN network using *timer* IF.

On the other hand, the LPWAN gateway follows the programming pattern of layer-2 (Section 3.4.2). As shown in Figure 3.13b the LPWAN gateway *receives* data from sensor node followed by optional data processing using *execute* IF and finally sending data to application server in the cloud at layer-3 using *send* IF. The application server at cloud follows the programming pattern of layer-3 (Section 3.4.3). As shown in Figure 3.13c the application server in the cloud *receives* data for further post-processing of sensor data using *execute* IF.



(A) Pollution Monitoring 2 : Sensor Node PP (B) Pollution Monitoring 2 : Gateway PP

(C) Pollution Monitoring 2 : Cloud PP (D) Pollution Monitoring 2 : DM PP

FIGURE 3.13: Pollution Monitoring 2 : Programming Patterns

The sensor node also accepts device management (DM) commands from the application server that includes - *configuration* parameters such as unique ID of the vehicle (license plate number), driver's name, sensor sampling time, sending interval, etc. Moreover the sensor node accepts certain *control* commands such as - remote reset, switch to factory default configuration, etc. The application server can also issue over-the-air software updates during *maintenance*. On the other hand, the process of *authentication* and *provisioning* of sensor nodes are governed by LoRaWAN protocol specifications.

Figure 3.13d shows the mapping of DM functionalities (Layer-4) onto IFs and PPs of layer-4. The *configuration* and *control* follows the PP of **update** (Section 3.4.4), whereas the software *maintenance* follows the PP of **upgrade** (Section

3.4.4). The sensor node receives the *configuration* parameters, *control* commands, software *maintenance* using *receive* IF, followed by processing of DM commands using *execute* IF and then finally wiring the configuration parameters, control command and software updates in the memory using *write* IF.

3.5.2 Scenario 2 : GreenIQ - Smart Irrigation

The GreenIQ is a smart HUB (gateway) for garden irrigation. The HUB connects to the Internet via Wi-Fi or 3G/4G. The irrigation scheduling algorithm is based on the data received from the sensors and also on current & forecasted weather data received from the Internet. The HUB can be *configured* and *controlled* using mobile application (restart HUB, change address, revert back to factory settings etc.) and is connected to various off-the-shelf irrigation sensors and actuators. For simplicity let's assume the sensors & actuator communicates with HUB using MQTT over 6LoWPAN (LR-WPANs) and mobile devices communicate with HUB via cloud. The user can also monitor the *status* of irrigation sensors and actuators, for example - battery level, etc. It is also possible for users to initiate device (HUB, sensor and actuator) firmware upgrade from a mobile phone. The scenarios description is shown in Figure 3.14.

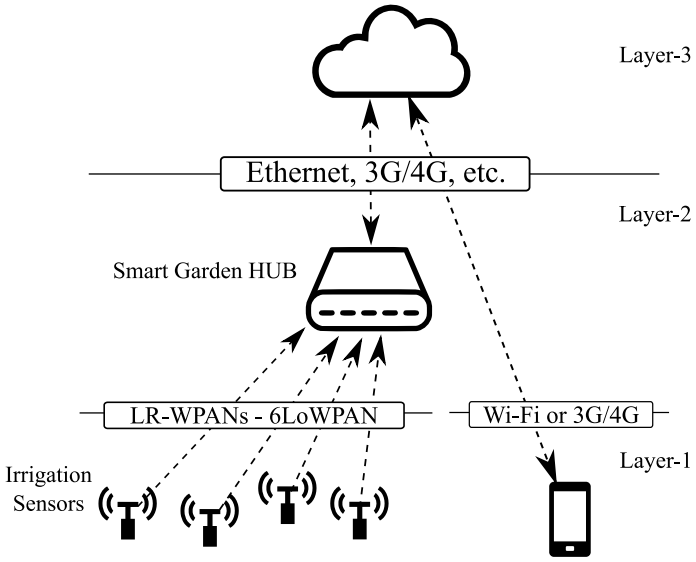
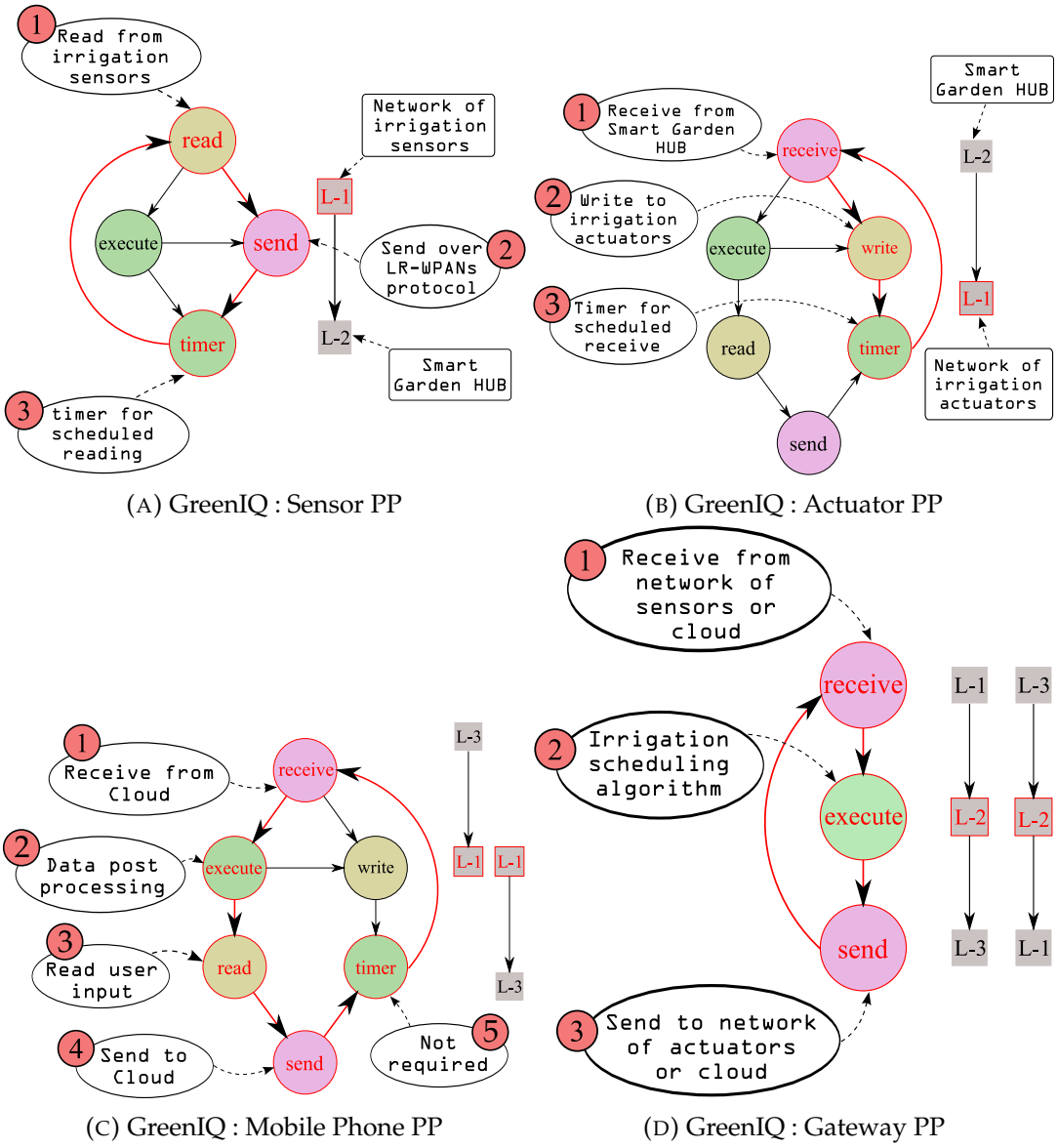


FIGURE 3.14: GreenIQ

There are three distinct devices - sensors, actuators and a mobile phone - at layer-1. As shown in Figure 3.15a, the sensor executes the same IF and follows the same PP of that of the previous example scenario in Section 3.5.1.

The actuator follows the programming pattern of layer-1 (Section 3.4.1). As shown in Figure 3.15b, it receives the command from Smart Garden HUB at layer-2 followed by controlling the actuator using *write* IF and finally the actuator goes into sleep using *timer* IF.
On the other hand the mobile phone also follows the PP of layer-1 (Section 3.4.1). As shown in Figure 3.15c, it can receive data from cloud using *receive* IF, followed by data post processing using *execute* and then finally reading the user input using *read* IF.
The gateway follows the programming pattern of layer-2 (Section 3.4.2). As shown in Figure 3.15d, it can *receives* data from the network of sensors at layer-1 as well as from cloud application at layer-3. It then processes (irrigation scheduling algorithm) the data locally using *execute* IF and if required it can send data back to the network sensor and actuator or to cloud application.
The cloud on the other hand follows the programming pattern of layer-3 (Section 3.4.3). As shown in Figure 3.16a it receives the data either from gateway or directly from mobile phone using *receive* IF, then the received data is processed using *execute* IF and if required it can send data back to gateway or mobile phone using *send* IF.
Finally, for device management (DM) the sensor & actuator follows the programming pattern of layer-4 (Section 3.4.4). In this scenario use case, the configuration parameters and control commands (restart HUB, change address, revert back to factory settings, etc.) follows the programming pattern of **update** (Section 3.4.4), whereas the firmware update follows the programming pattern of **upgrade** (Section 3.4.4) and the status request (battery status) follows the programming pattern of **status** (Section 3.4.4). As shown in Figure 3.16b, for configuration parameters, control commands and firmware update, the sensor and actuator receives the device management commands using *receive* IF followed by processing of DM commands using *execute*, then finally writing the data in the memory using *write* IF before doing into sleep using *timer* IF.
On the other hand, for status request the sensor and actuator receive request for status command via *receive* IF followed by processing of DM commands using *execute* IF, then reading battery status using *read* IF and finally sending the battery status using *send* IF before going into sleep using *timer* IF.



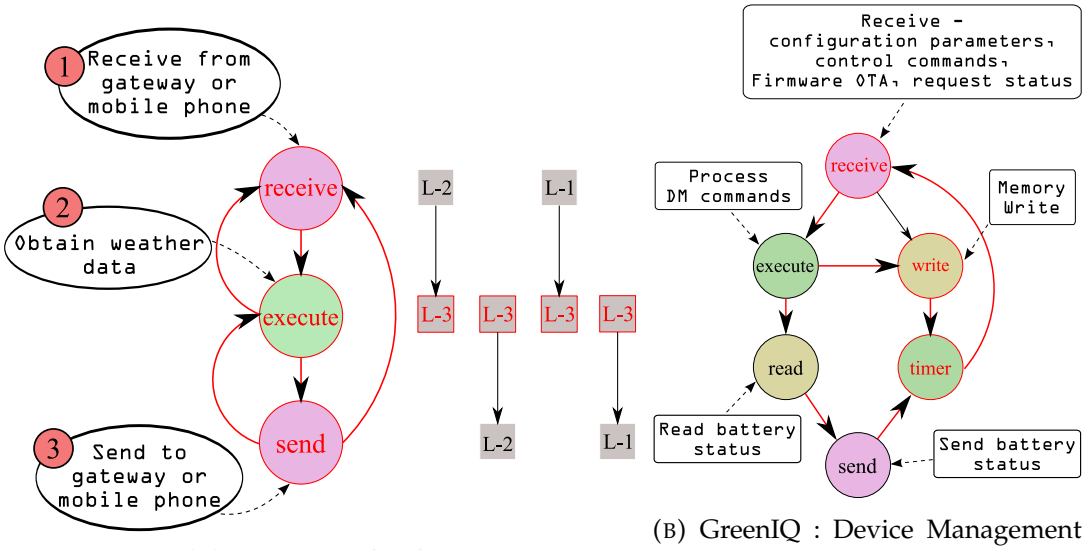
(A) GreenIQ : Sensor PP

(B) GreenIQ : Actuator PP

(C) GreenIQ : Mobile Phone PP

(D) GreenIQ : Gateway PP

FIGURE 3.15: GreenIQ : Programming Patterns



(A) GreenIQ : Cloud PP

(B) GreenIQ : Device Management PP

FIGURE 3.16: GreenIQ : Programming Patterns

3.5.3 Discussion and Analysis

It is interesting to see that both scenarios described above share some similar programming patterns at their respective layers even though both scenarios belong to different application categories. For example, the sensor devices in both the scenarios follow the same programming pattern of layer-1 and they execute the same invariant functions - *read* $\rightarrow$ *send* $\rightarrow$ *timer*, similarly the gateways also follow the same programming pattern of layer-2 and execute the same invariant functions - *receive* $\rightarrow$ *execute* $\rightarrow$ *send*.

The cloud also follows the same programming pattern of layer-3, except that in the first scenario (Section 3.5.1) which is sense only application (one way communication) the cloud application does not send data back to layer-1 or layer-2.

Moreover, the programming pattern related to device management also shows similar program execution flow of layer-4 in both the scenarios - *receive* $\rightarrow$ *execute* $\rightarrow$ *write* $\rightarrow$ *timer* for **update** and **upgrade** DM functionalities.

In conclusion, the invariant functionalities and programming patterns provide a useful means to ease programming of IoT devices that allows reusability of application logic and functionalities across IoT applications irrespective of application domain.

3.6 Concluding Remarks

In this chapter, we described the requirement of a software abstraction layer (SAL) for easy and rapid IoT application lifecycle management that hides the underlying IoT device heterogeneity. For this, we systematically presented using an IoT architecture the common IoT characteristics - starting from the network of low power devices to gateways then to cloud infrastructure and finally the device orchestration. Following this we presented two types of abstractions in the form of IoT application's invariant functionalities (IFs) and programming patterns (PPs) at each layer of IoT architecture. We also showcased the usefulness of IFs and PPs in implementing IoT application use cases.

In summary, this chapter lays the groundwork for implementing our framework in Chapter 4 that implements these high level abstractions (IFs and PPs).

Chapter 4

PrIoT - A Framework for Prototyping IoT Applications on Heterogeneous Hardware Devices

4.1 Introduction

In this chapter, we introduce PrIoT [19], an open source[1] framework for rapid and easy IoT prototyping that helps to hide the IoT device manufacturers heterogeneity from the application developers, thus solving the identified research problem of complexity related to the IoT device heterogeneity explained in Chapter 3. This is our software oriented approach that will be followed in Chapter 5 with the hardware oriented approach. Also PrIoT will be useful in different phases of the IoT service definition and development. The main design philosophy behind PrIoT framework is - "code once port everywhere" allowing IoT application developers to develop hardware independent embedded code which can be ported to any hardware supported by PrIoT. After the design of PrIoT framework, we also validated its proposed concept with an implementation using an open source approach. As shown in Figure 4.1, PrIoT proposes to stitch together the different components that makes an IoT system namely the **hardware** consisting of sensors, actuators, transceiver and Human-Machine Interface, the **embedded systems** that are used to program the IoT devices, the **components** that serves at providing the higher level libraries for Network and Cloud functionalities. Finally, PrIoT leverages IoT device programming by exposing a **service** element with a high level language for heterogeneous device programming and an orchestrator

[1] The PrIoT framework is under GPL-v3 license. For more information please visit - http://www.priot.org/

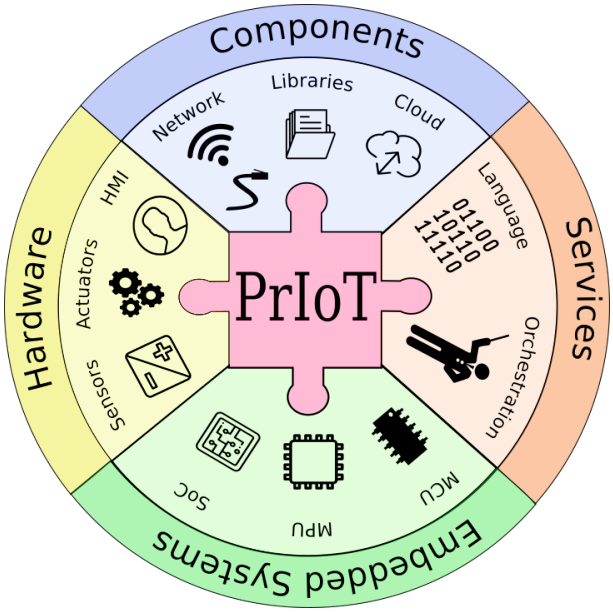


FIGURE 4.1: PrIoT Components

to synchronize all devices together and easily upgrading the entire system. We propose in PrIoT to achieve this design goal by maintaining a separation between application logic and scenario configuration. PrIoT exposes developers with a device independent application programming interface (API) specially designed to cover most IoT scenarios and yet concise enough for easy application development. In fact, PrIoT is tailored to better manage IoT device heterogeneity and fasten the IoT prototyping phase by introducing an intelligent intermediate software layer over the IoT physical device that allows service developers to easily and quickly program IoT devices independently of the hardware manufacturer. Also this PrIoT intelligence is able to understand the commands of the IoT device firmware and uses a hardware abstraction layer (HAL) that can translate the high level IoT service functionalities into hardware specific commands. In addition, PrIoT aggregates different acknowledged and accepted tools under a common umbrella to foster and ease IoT prototyping. We also gather in the same framework the best tools together to enhance and ease user action. As stated earlier, PrIoT will mainly stitch together tools, frameworks and libraries to ease prototyping, our intelligence and own programs is more as a scheduler and translator to other tools that are not functioning together.

The chapter comprises the following sections. Section 4.2 introduces PrIoT design objectives, followed by overview of PrIoT framework building blocks in Section 4.3, then in Section 4.4 we describe the conventional method for implementing IoT application on embedded hardware. Next, in Section 4.5,

we validate the PrIoT design philosophy "code once port everywhere" by implementing PrIoT framework using various open source tools and programming language. Then in Section 4.6, we describe the steps required in order to implement IoT scenarios using the PrIoT framework along with an example use case that will illustrate the different PrIoT concepts and provide an example of its usage. Finally, Section 4.7 contains conclusion and provides the perspective and orientation of PrIoT framework.

4.2 Framework Design Objectives

The main design goal of PrIoT is to allow not only inexperienced personnel (domain experts like - doctor, farmer, engineer, etc.) but also professionals to develop, deploy and maintain IoT application on a distributed network of IoT end-systems. It also allows well experienced personnel to develop applications on new IoT devices different from their previous IoT devices knowledge. PrIoT achieves this design goal by satisfying three important requirements of prototyping applications for IoT end-systems as described below.

4.2.1 Rapid Prototyping

PrIoT hides the low level details of programming IoT hardware resources (sensors, actuators, processing unit, Human Machine Interfaces, communication devices, etc.) and allows developers to build IoT application logic independent of any IoT devices. PrIoT also hides the implementation details of standard IoT communication protocols (HTTP, CoAP, MQTT, IETF 6LoWPAN, etc) and provides users with protocol independent abstract functions to communicate with gateways and other IoT devices. This high level of abstraction is achieved by exposing developers with device independent PrIoT-API (Subsection 4.3.1). Once the application logic is framed it can be deployed largely on different kinds of IoT hardware resources that can be selected through PrIoT database - PrIoT-DB (Subsection 4.3.7).

4.2.2 Hardware Configuration

PrIoT provides an interface for the community to add and maintain a repository of IoT hardware resources that includes resource metadata (cost, power consumption, memory, peripherals, operating voltage, technology, etc.) and

recommends an optimum bill of materials (BOM) based on user defined application requirements for example, cost, power budget, connectivity, security, cloud interface, data acquisition, physical design, etc. PrIoT also delivers a hardware configuration template in accordance to user requirements and IoT application.

4.2.3 Scenario Deployment

PrIoT helps in the deployment of IoT scenarios on a distributed network of IoT end-systems, whether they are linked to the same logic or if they are distributed and connected to different gateways. Thus it is possible to manage, monitor, update and upgrade complex IoT scenarios during deployment and maintenance phases. This is achieved by using a specific PrIoT tool that interprets the high level IoT application required functionalities into specific hardware commands. Table 4.1 qualitatively compares the shortcomings of existing work (Section 2.2) with respect to PrIoT design goals.

PrIoT Design Goals	**Platform**	**OS**	**Framework**	**PrIoT**
Rapid Prototyping	✷	✓	✷	✓
Scenario Deployment	✗	✗	✗	✓
Hardware Configuration	✗	✗	✗	✓
✓ - Full Support, ✗ - No Support, ✷ - Partial Support				

TABLE 4.1: PrIoT Design Goal Comparison

4.3 PrIoT Framework Overview

This section explains the main building blocks of PrIoT Framework. The overall framework workflow is shown in Figure 4.2. In PrIoT the application logic is independent of any IoT hardware resources, this implies that the same application logic can be ported to any hardware supported by PrIoT, thus solving the IoT device heterogeneity problem introduced in Chapter 3.

4.3.1 PrIoT Language and Application Programming Interface : PrIoT-Lang & PrIoT-API

PrIoT is equipped with a high level programming language - **PrIoT-Lang**, which exposes users with device independent high level programming interface - **PrIoT-API**. The set of PrIoT-APIs is kept limited but captures most

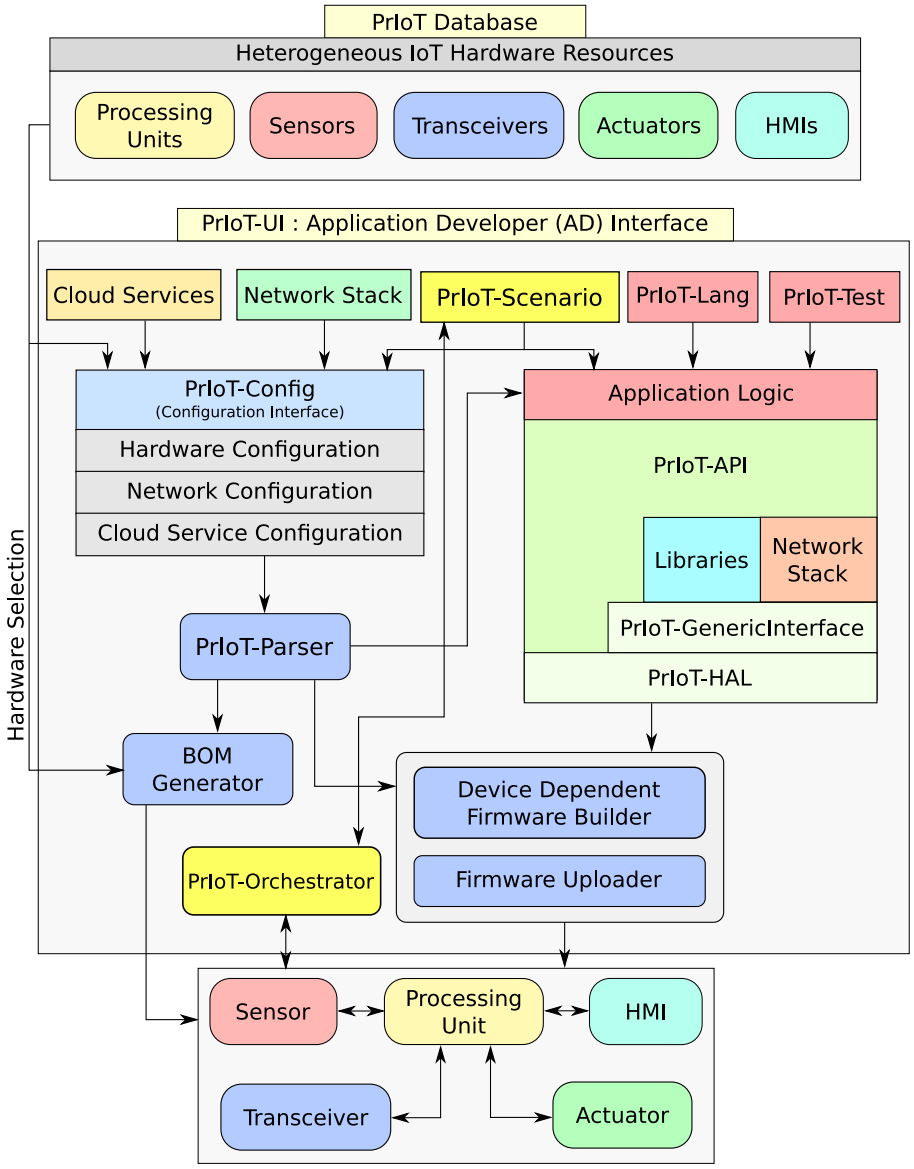


FIGURE 4.2: PrIoT Framework Block Diagram

practical invariant functionalities (Section 3.4) of IoT application scenarios, the set includes functions like - *read*, *write*, *execute*, *send*, *receive* and *wait* as shown in Table 4.2.

There are a number of existing solutions (Arduino [68], Energia [70] and Mbed [69]) which uses the similar API approach combined with C-like programming language to expose device specific features and to program device side application logic. Another approach for the implementation of APIs & languages is Domain Specific Language (DSL) and is heavily used and described in academic research - [208], [209], [210], [112], [211], [212], [213], [214]. In PrIoT, we also used the DSL approach to implement our PrIoT-Config as will be discussed in Section 4.5.

List of PrIoT-APIs	
configure	Configure hardware resources.
read	Read data from sensor, memory, etc.
write	Write data to actuator, memory, etc.
send	Send data using transceiver.
receive	Receive data from transceiver.
execute	Execute user defined operations.
timer	Delay, wait or sleep.

TABLE 4.2: PrIoT-API - Device Independent Programming Interface

4.3.2 PrIoT Application Debugging : PrIoT-Test

The **PrIoT-Test** allows specifying metrics to be tested as output results. Such metrics can be described in terms of power consumption, communication latency, memory usage, processing unit performance, etc.

4.3.3 PrIoT Generic Interface : PrIoT-GI

PrIoT-GenericInterface (**PrIoT-GI**) is an abstract interface layer that provides an uniform interface for the development of **PrIoT-Lang** and **PrIoT-API**, hardware & software resource libraries, etc. The generic interface is a concept which is designed in such a way that the application written with PrIoT-GI can be mapped easily on existing embedded OSes like - RIOT [111], Zephyr [72], Contiki [73], etc. This abstract interface gives users the freedom to compare and experiment with different OSes without rewriting the application code. The concept of generic interface is not new, for example CMSIS-RTOS [215] that provides a standard programming interface that is portable across various RTOS.
Note that, the generic interface in general allows standard interface over existing software library, operating system APIs, etc. thereby allowing uniform interface across various software solutions.

4.3.4 PrIoT Hardware Abstraction Layer : PrIoT-HAL

PrIoT-HAL is a hardware abstraction layer (HAL) which consist of high level functions for interfacing peripherals using standard peripheral communication such as SPI, I2C, UART, 1-wire, etc. This layer also contains a device independent interface for input-output operations such as - sleep, delay, memory access, etc. In PrIoT, the HAL is inherited from the Wiring framework, as discussed in subsection 4.5.6. The concept of HAL is not new, for example

CMSIS (Cortex Microcontroller Software Interface Standard) [216] that provides an abstraction layer for ARM based architectures.
Note that, the HAL in general allows common standard interface over various hardware peripherals that exist inside heterogeneous IoT devices. This makes it possible to maintain a uniform interface for accessing hardware peripherals irrespective of IoT device architecture. Various device manufacturers define their own HAL for their respective device architecture. In PrIoT, we propose to integrate these HALs under PrIoT-HAL to allow common interface for device peripherals across heterogeneous device architecture.

4.3.5 PrIoT Configuration : PrIoT-Config

PrIoT-Config provides users with the interface to add information about the type & configuration of hardware resources, cloud service and communication protocols, etc. and is parsed using **PrIoT-Parser**. The configuration information is independent of application logic and therefore any change in configuration parameters does not affect the application logic. Further details regarding the use case of **PrIoT-Config** are discussed in Section 4.5.

4.3.6 PrIoT Parser

The **PrIoT-Parser** parses the configuration file generated by **PrIoT-Config** to determine the type of hardware resources, communication protocol, network stack used by application logic. The output of PrIoT-Parser together with IoT application logic is passed to **PrIoT-HAL** which generates the necessary device dependent code.

4.3.7 PrIoT Database : PrIoT-DB

PrIoT maintains a database of IoT hardware resources ranging from - sensors, actuators, transceivers, Human Machine Interface (HMI), etc. to processing units such as microcontroller, microprocessor, SoC, etc. The primary goal of this database is to provide users with a large set of hardware components that can be programmed using **PrIoT-Lang** with a high level abstract set of functionalities and parameters that can be accessed through the **PrIoT-APIs**. Second, this database can also target specific device vendor items while keeping the device programming transparent. For example, we will see in Section 4.6.1 an example of using **PrIoT-DB** where a specific family of component, namely an RFID reader can be used as a default hardware component or it

can be defined as a vendor specific component when needed. The implementation of **PrIoT-DB** is discussed in more detail in subsection 4.5.2.
Also, the BOM (Bill of Material) generator uses PrIoT-DB to provide application developers with a list of hardware resources based on application requirements and scenario configuration.

4.3.8 PrIoT User Interface : PrIoT-UI

To increase the productivity of application development, PrIoT encapsulates the building blocks (**PrIoT-GI**, **PrIoT-HAL**, **PrIoT-Config**, **PrIoT-Parser**) with graphical interface (**PrIoT-UI**) which helps developers to build application without dealing directly with the underlying building blocks. There are a number of tools available such as Node-Red [184] and Blockly [217], that uses graphical interface for implementing application logic. In PrIoT, the **PrIoT-CLI** provides a user friendly command line interface (CLI) to access the underlying building blocks and is discussed in more detail in subsection 4.5.7. Moreover, the advantage of CLI is that it is possible to create a graphical interface on top of PrIoT-CLI without making significant changes in the existing implementation.

4.3.9 PrIoT Firmware Builder and Uploader

The final PrIoT application code can be *built* using available open-source or proprietary vendor specific tools. Regarding the device firmware *upload*, there are many existing solutions and depending on the type of target hardware one can use proprietary vendor specific tools as listed in Table 2.4. Also there are a number of open-source tools to *upload* device firmware and one such tool is PlatformIO [104], which is a collection of vendor specific open-source tools for building and uploading device firmware.
Due to the scale of IoT, firmware over-the-air (FOTA) is getting a lot of attention [218]. In FOTA a wireless communication interface, such as WiFi, Bluetooth, ZigBee, etc. is used to receive firmware from the host.
At this stage of our work we have selected PlatformIO as our underlying block for firmware *upload* and *build* tool. The reason for choosing PlatformIO is that it covers most vendor specific tools that allows for testing various heterogeneous hardware platforms.

4.3.10 PrIoT Scenario

The PrIoT Scenario acts as a starting point to describe a high level view of a scenario without going into technical or implementation details of the scenario. The scenario is defined in a file to expose the various elements used in an IoT scenario such as those discussed in Section 3.3. This file is further processed by PrIoT Orchestrator (Section 4.3.11) to generate template for each *end-system*, *edge-system*, *controler-system* and *cloud-system* which makes it easier for application developers to work with. In this work we limit our implementation that does not require PrIoT scenario description and is left for future work.

4.3.11 PrIoT Orchestrator

PrIoT Orchestrator implements the cross-layer management proposed in our 4-layer architecture in Chapter 3. It is in charge of different functionalities such as authentication & provisioning, configuration & control, monitoring & diagnosis and software maintenance of all the elements needed by the IoT application and services in the 3 layers - Device layer, Edge layer and Cloud layer.
In our work, in addition to above functionalities we also propose new functionality of the PrIoT Orchestrator that allows it to use machine readable description defined by PrIoT Scenario and generate automatically the corresponding configuration of the heterogeneous devices. It helps in the deployment of IoT scenarios onto the network of IoT end-systems & edge-systems and also provides provision for executing IoT cross-layer (L4) management functionalities (Section 3.4.4). More details about PrIoT orchestrator is described in Section 4.5.8.
Finally, Table 4.3 summarizes the available solutions against various components comprising PrIoT. As we can see PrIoT takes the best of IoT development techniques proposed by available solutions and integrates it into one platform for end-to-end IoT application development.

In summary, apart from integrating best of the available IoT development techniques, PrIoT also introduces a new way to handle device heterogeneity by exposing IoT application invariant functionalities (IFs) and Programming Patterns (PPs) in the form of PrIoT-API and PrIoT-Lang. Moreover it can also suggest from its database (PrIoT-DB) optimum bill of material (BoM) based on application scenario requirements. At last, thanks to PrIoT-Orchestrator,

PrIoT also helps developers in the deployment and maintenance of IoT devices.

IoT development Techniques	PrIoT Block	Existing Solution
High level language and API	PrIoT-Lang PrIoT-API	Arduino, Energia, mbed and Embedded OSes
Hardware abstraction	PrIoT-GI PrIoT-HAL	Embedded OSes
Device Configuration	PrIoT-Config PrIoT-Parser	PlatformIO
User Interface	PrIoT-UI	Node-Red, Blockly
IoT service and device management	PrIoT-Orchestrator	Kubernetes

TABLE 4.3: IoT Development Techniques Integration in PrIoT

4.4 Conventional Method for Implementing IoT Applications on Embedded Hardware

The conventional methods for designing embedded systems follows a top down approach as shown in Figure 4.3. As per application scenario requirements there are two options (Step-1A and Step-1B) for deciding target hardware. In Step-1A, the hardware resources (sensor, actuator, transceiver and processing, etc) are selected to design and build (Step-2) the required application specific embedded board, but this option is only possible if the necessary engineering resources are available. Instead, it is possible to select (Step-1B) off-the-shelf embedded boards such as Arduino, Raspberry Pi, STM32 Nucleo, etc. In Step-3 IoT application is designed and implemented, that is specific to the hardware resources selected in Step-1A or Step-1B. Finally in Step-4, the application is tested and validated on the target hardware, any discrepancy in test results or change in IoT scenario requirements might force embedded designers to either select a new embedded board (Step-1B) or re-iterate the hardware design (Step-1A & Step-2). Conventional methods for implementing IoT applications suffer from the problem of hardware heterogeneity, which hinders its use for rapid prototyping because of the following reasons:

1. *Very limited freedom to experiment with multiple hardware architectures* - There are a number of embedded hardware available from various silicon manufacturers and not all hardware are similar with respect to technology, architecture, cost, power consumption, performance, peripherals, etc. Therefore the initial selection (Step-1A & Step-1B) of

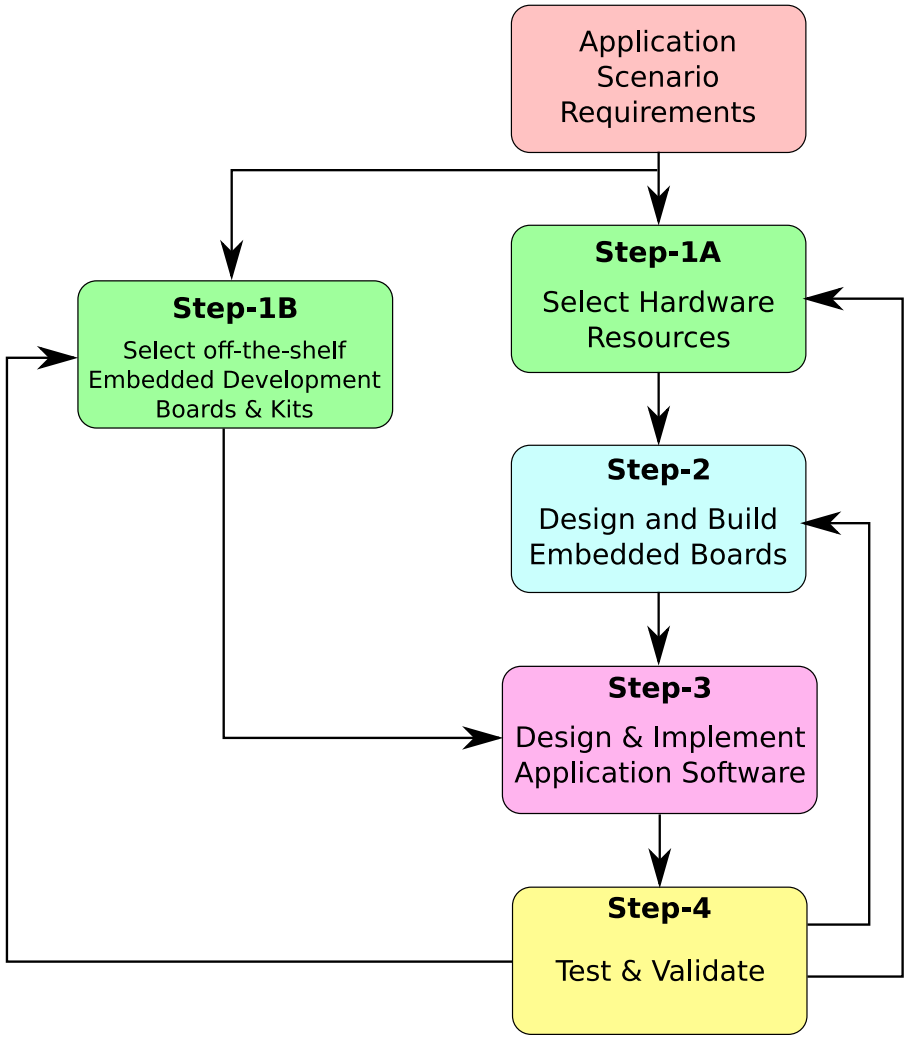


FIGURE 4.3: Conventional Steps to Design Embedded Systems

hardware determines the functionality and performance of IoT application software. Since the software design is heavily tied with the choice of hardware, so any change in hardware requirements might force application developers to rewrite or port the existing software on a different hardware. This increases the software development, testing and validation time thereby inhibiting application developers to experiment with multiple hardware from various vendors.

2. *Vendor lock-in* - Depending on IoT application requirements, there are multitude of embedded hardware to choose from various silicon vendors. Moreover, each silicon vendor provides its own set of tools, Integrated Development Environment (IDE), compilers, reference designs & documentations, etc. which requires a learning phase in order to rewrite or port the existing software thus making IoT prototyping difficult and slow process.

4.5 PrIoT Framework Implementation and Evaluation

In this section, we describe the high level implementation details of the PrIoT framework that is used for IoT device application development. In order to implement the framework, we have used various open source tools and programming languages that allows us to achieve the design philosophy of "code once port everywhere".
The implementation described in this section is not the only way to implement the PrIoT framework, there might exist other solutions from a software engineering perspective. Here our objective is to validate the PrIoT design philosophy via implementation without going into details about our choices with respect to tools, programming language, implementation methodology, etc, but for interested readers these details are mentioned in Appendix **??**. Note that PrIoT Scenario and PrIoT-Orchestrator are still under development, but in Section 4.5.8 we introduce the building blocks of PrIoT-Orchestrator that are part of the future development.
Figure 4.4 shows the overall block diagram of PrIoT framework implementation used for device application development and we briefly explain each individual block hereafter.

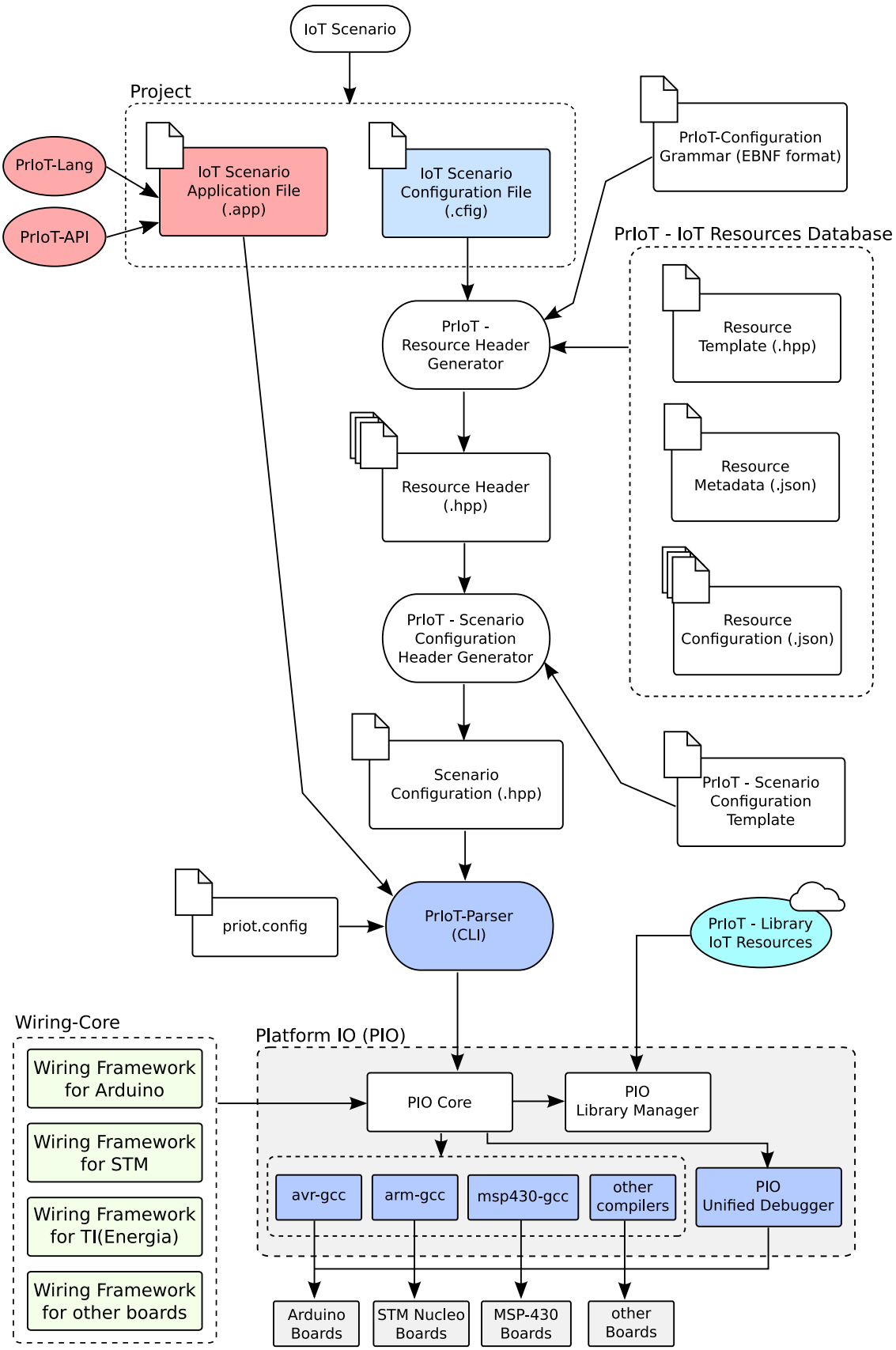


FIGURE 4.4: PrIoT Framework Workflow for IoT Device Application Development

4.5.1 PrIoT Application and Configuration Entities

In order to implement IoT applications on heterogeneous hardware devices, PrIoT requires only two entities first is in the form of *application file* (*.app) that contains the application logic using PrIoT-Lang (Section 4.3.1) and second is the *configuration file* (*.cfig) that contains scenario configuration written using PrIoT-Configuration DSL (Section 4.5.3). Further details about the content and the structure of application and configuration files are discussed in Section 4.6.

4.5.2 IoT Resources Database

The *"IoT Resource database"* is a database maintained by PrIoT for all hardware and software resources. As listed below, each resource is made up of three types of database files that allows *IoT Resource Header Generator* (Section 4.5.3) to generate the necessary resource headers (*.hpp). These database files are transparent to the application developers and are only maintained by developers of PrIoT framework.

1. *Resource Template* (*.hpp_temp) - A template for a resource header file written using Jinja [219] templating programming language. It provides various placeholders that are filled by *IoT Resource Header Generator* based scenario configuration along with resource metadata and resource configuration.

2. *Resource Metadata* (*.json) - The resource metadata contains information like peripheral (I2C, UART, SPI, etc.), interface (number and location of physical pins), library, etc. This is required by the processing unit to correctly interface with resources.

3. *Resource Configuration* (*.json) - In general, a resource can be configured in various operating modes which requires different configuration settings. Therefore, for every operating mode there exist a resource configuration file. For example, a hardware resource of type WiFi (Transceiver-IEEE 802.11) in HTTP client mode requires various configuration settings such as - port, request page, method, host name, etc.

4.5.3 Resource Header Generator

As shown in Figure 4.5, the *resource header generator* for both hardware and software resources generates "C Standard" header files (*resource header*) from scenario configuration and *IoT resource database* (Section 4.5.2) . The *resource header* contains configuration settings and options that are required to program resources in a knows initial state as indicated in scenario configuration. The user inputs the scenario configuration in a configuration file (**.cfig*) using a domain specific language of PrIoT configuration named PrIoT-Config-DSL and without going into details, listing 4.1 shows an example of how a typical configuration file looks like. The PrIoT-Config-DSL grammar is written using EBNF (Extended Backus–Naur form) format and is described in Appendix B.1.
The configuration file is parsed using lark-parsee [220] that checks the integrity of configuration file and also provides tools for efficient processing of configuration files.

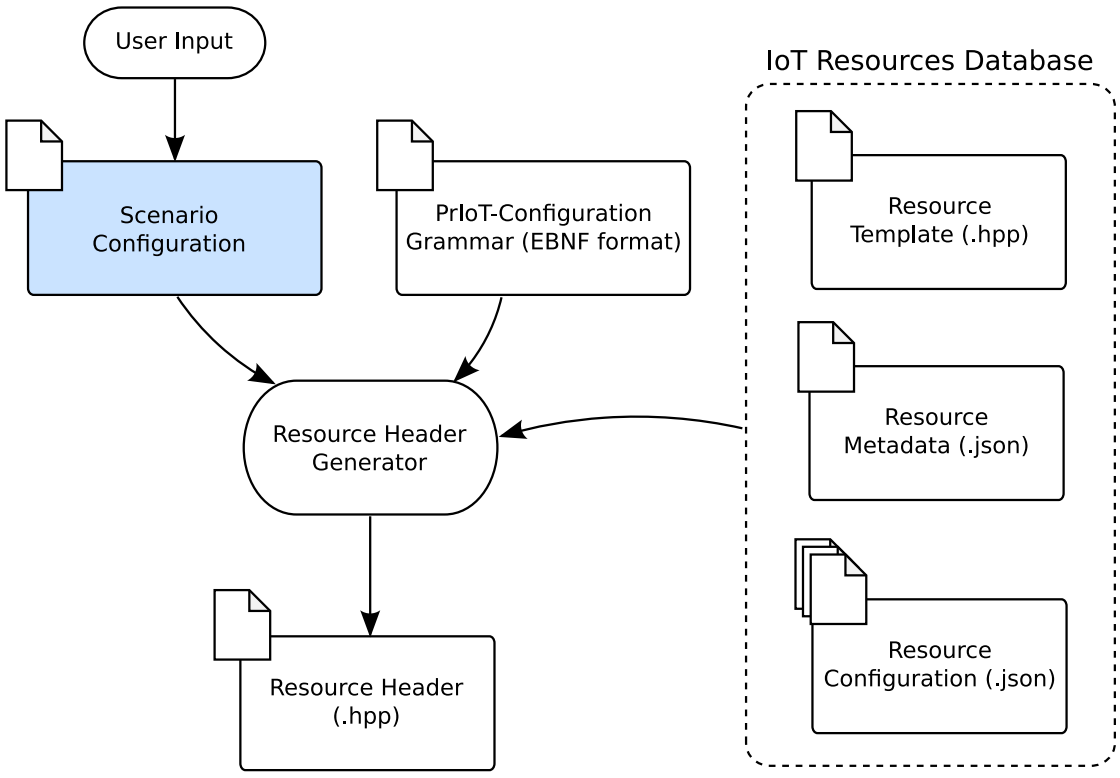


FIGURE 4.5: IoT Resource Header Files Generator

```
# File name - example.cfig

# select hardware resources
Select Sensor.Temp as s_temp
```

```
5  Select Transceiver.IEEE80211 as t_esp8266
6
7  # select embedded system
8  Select Hardware.MCU.Family as AVR-8bit
9
10 # import communication protocol
11 Import Communication.80211 as wc
12 Import Gateway.HTTP.client as http_c
```

LISTING 4.1: Example Scenario Configuration File

4.5.4 Scenario Header Generator

The *"scenario header generator"* generates "C Standard" header file for scenario configuration (*.hpp). As shown in Figure 4.6 it requires both *"resource headers"* and *"scenario configuration template"* to generate the overall scenario configuration that contains the initialization code for all the resources used in the scenario. The *"scenario configuration template"* is similar to *"resource template"* (Section 4.5.2) in the sense that it uses the same templating programming language - Jinja and provides the necessary placeholder to include the required resource libraries, initialization code, etc.

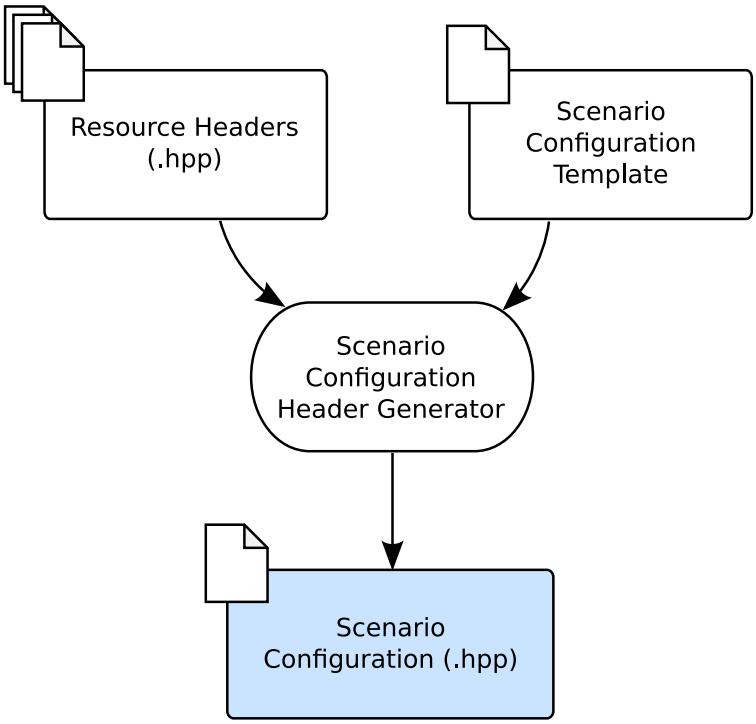


FIGURE 4.6: Scenario Header Generator

4.5.5 PrIoT IoT Resource Library

The *IoT Resource Library* is the database of software libraries to use & access all the *Connectors* (External, Hardware & Software) (Section 3.3). It is equivalent to communication libraries for *external-connectors*, device drivers for *hardware-connectors* and standard libraries (such as - algorithms, math, string processing, etc.) for *software-connectors*. These libraries are implemented in C++ taking advantage of object oriented features and exposes various invariant functions (IFs) as described in Section 3.4 in the form of PrIoT-APIs. The developers of PrIoT can implement new libraries to extend the database and thus allow continuous evolution of PrIoT with the growing number of hardware resources.

4.5.6 PrIoT Embedded Toolchain and Hardware Abstraction Layer

An embedded toolchain is a set of compiler, linker, debugger, standard library and other tools and is required to compile source code (application software or firmware) into an executable that can run on target hardware devices. In general, there exist many open source and proprietary toolchains for various processing unit architecture and for our work we used PlatformIO [104] to build, upload and debug PrIoT projects. As shown in Figure 4.4, PlatformIO provides in one place a collection of open source cross-compiler for various architectures such as ARM, Atmel-AVR, TI-MSP, etc. along with debugger and library manager. It is possible to interact with PlatformIO independently from PrIoT but to make things easier we create a wrapper around PlatformIO that allows developer to interface with PlatformIO using PrIoT command line interface (PrIoT-CLI) 4.5.7.
Hardware Abstraction Layer (HAL) provides an abstraction layer between hardware devices and high level application software it includes apart from device drivers, any other software that directly interacts with hardware. For our work we used an open source framework named Wiring [94] as our HAL (PrIoT-HAL). As shown in Figure 4.4, we used one implementation of the Wiring framework for each target hardware architecture and is transparent to application developers.

4.5.7 PrIoT Command Line Interface (PrIoT-CLI)

The PrIoT-CLI is the command line interface for the PrIoT framework and is written in python programming language. Through its command line options it allows application developers to execute various behaviors listed below.

- *create project* - Generate project directories in the workspace along with scenario application file (*.app) and configuration file (*.cfig) templates for application developer.
- *create configuration* - Triggers *"resource header generator"* (Section 4.5.3) and *"scenario header generator"* (Section 4.5.4) to generate various header files required for successful compilation of scenario project.
- *build* - It builds and compiles the scenario project for the target board specified in the configuration file using PlatformIO.
- *upload* - It uploads the project binary on the target board using PlatformIO.
- *create database* - Create PrIoT *IoT resource database* (Section 4.5.2).
- *add resources* - Add metadata, configuration and header template (Section 4.5.2) for various IoT resources such as processing units, sensors, actuators, transceivers, etc. in the PrIoT database. It is used by PrIoT framework developers to add new resources or update the existing ones.

4.5.8 PrIoT Orchestrator

The PrIoT Orchestrator is responsible for handling the execution of cross layer invariant functionalities as described in Chapter 3 (Section 3.4.4). We define the implementation of PrIoT Orchestrator as our cross-layer management (CLM). The CLM is further divided into four main subsystems - 1) Device meta-data. 2) Controller-Application. 3) Rule-Engine. 4) Controller-Core. Figure 4.7 shows the architecture of our CLM. The internal functioning of the CLM and detailed description of the four main parts are described hereafter.

The four main subsystems perform a dedicated function that are required for the proper functioning of the controller and interact with each other via common *message bus*. As Figure 4.7 shows, the *edge interface* which is responsible for communication with downstream devices (*end-systems* & *edge-systems*)

and can support multiple protocol via *protocol-translators/adapters*. The message broker & router (for example, MQTT for message broker) combines the data in a unified manner and may induce more contextual information which may be useful to other modules. Whereas, *device meta-data storage* maintains the most recent *end-system* and *edge-system* meta-data. For each system the meta-data consist of, for example - unique ID, firmware version, last status, last updated, authentication policy, etc. The *rule engine* monitors the message bus and performs action based on the rules and the content of the message. The cross-layer controller can resides inside the edge-system or cloud-system and is responsible to execute cross-layer functionalities in response to commands received from the cloud-system.

1. *Device meta-data* - For each IoT scenario, this contains the database of all the physical devices present at L-1 (device layer) and L-2 (edge layer). For each device the meta-data consist of, for example - unique ID, firmware version, last status, last updated, authentication policy, etc. The meta-data can be created automatically during device provisioning or manually during the life cycle of IoT scenario. The repository is dynamic and maintains the current status of all the devices at L1 and L2. Any queries to the database are initiated via controller-core. The control-application can use this information to monitor the status of devices.

2. *Controller-Application* - It consists of a sequence of control commands exposed by various cross-layer management functionalities. It can combine commands from distinct cross-layer functionalities, for example batch update of device firmware, send device configuration parameters, periodically check the availability of device, execute network reorganization algorithm, etc. It is also possible to define rules within a control application, for example rules that execute weekly network monitoring, a rule to execute network wide device availability after modifying the network configuration.
   The control-applications are stored in a buffer and are executed automatically by controller-core based on the defined rules or sequentially.

3. *Rule-Engine* - The rule-engine monitors the message bus and can instruct controller-core to execute predefined actions based on the message content. For example, monitoring sensor data for abnormality, initiating provisioning and authentication sequence after detecting request message from new device. The rule-engine can also instruct controller-core to execute control applications.

4. *Controller-Core* - It processes the control-application and based on the control commands it issues the standard communication command based on the underlying network protocol using a protocol adapter. The inherent task of the controller-core is to periodically maintain the repository of device meta-data and also to execute control applications issued by the user or triggered by the rule-engine.

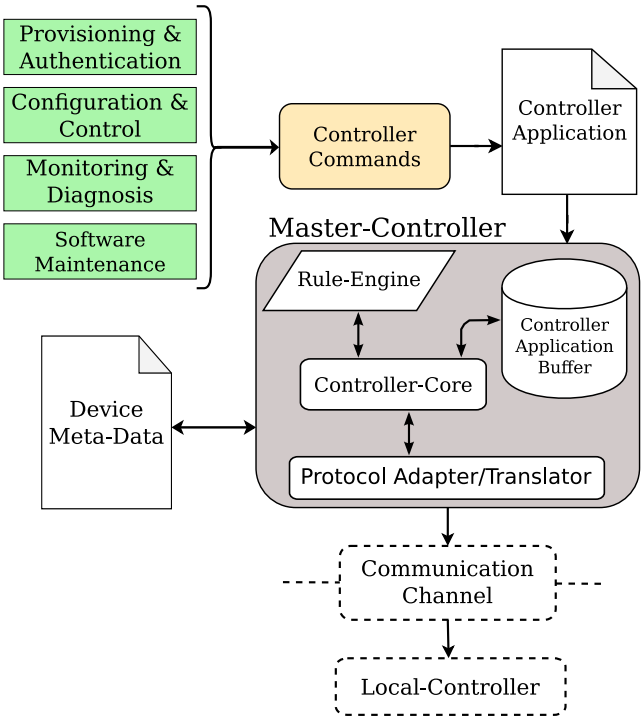


FIGURE 4.7: Cross-Layer Management

4.6 IoT scenario implementation with PrIoT Framework

In this section we describe the steps required to implement an IoT scenario with the PrIoT framework along with an example use case in Section 4.6.1.

As mentioned in Section 4.2, PrIoT main design goal is to allow inexperienced IoT service provider to quickly develop, deploy and maintain end-to-end IoT system from different hardware manufacturers that normally requires specific programming skills per manufacturer.
PrIoT follows a bottom-up approach for implementing IoT applications. As shown in Figure 4.8, based on the application requirements - application scenario orchestration (for simple scenarios with small number of IoT devices, scenario orchestration can be omitted), application logic (PrIoT-Lang &

PrIoT-API), scenario configuration (PrIoT-Config-DSL) and scenario orchestration is designed and implemented first (Step-1). Since PrIoT provides a uniform access interface (**PrIoT-APIs**) across all hardware components, so at this point the components selected from **PrIoT-DB** represent high level hardware elements with default outcomes which is independent of hardware manufacturer. Once the application logic is created in Step-1, then in Step-2A, the hardware resources are selected to design and build (Step-3) the required application specific embedded board. Instead, it is also possible to select (Step-2B) off-the-shelf embedded boards such R-Bus (Section 5), Arduino, Raspberry Pi, STM32 Nucleo, etc. In Step-4 application logic is tested and validated on the target board and hardware selected in Step-2. At this point, it is possible to gather test information (power consumption, CPU & memory usage, etc) using **PrIoT-Test** and can easily replace hardware resources (update scenario configuration) from any vendor without rewriting the application logic.
The IoT application developer defines the application logic in application files **(*.app)** and hardware configuration as per application requirements in configuration files **(*.cfig)** as it will be shown in Section 4.6.1 using an example use case. The configuration file includes - selecting hardware resources, standard communication protocol (*external-connectors* Section 3.3), interface between hardware resources (*hardware-connectors* Section 3.3) and software libraries (*software-connectors* Section 3.3). The configuration file(s) have all the necessary information required by PrIoT to automatically configure and compile the whole IoT application onto the selected hardware resources.

4.6.1 Use Case : Security Access System in Smart Building

In order to show the added value of our PrIoT framework, in this section we explain with an IoT service use case what are the components to be developed and the design steps used with our PrIoT framework.

Scenario Description : Consider an IoT scenario where we have to design a smart security access system in a building entrance where only authorized personnel are allowed to enter. For simplicity, we assume that the scenarios consist of a single object where all the requests for access are made.
The object consists of RFID reader for unique identification and human machine interface (display and keyboard) to enter personal code. For enhanced security the personal code is matched against the database which resides in

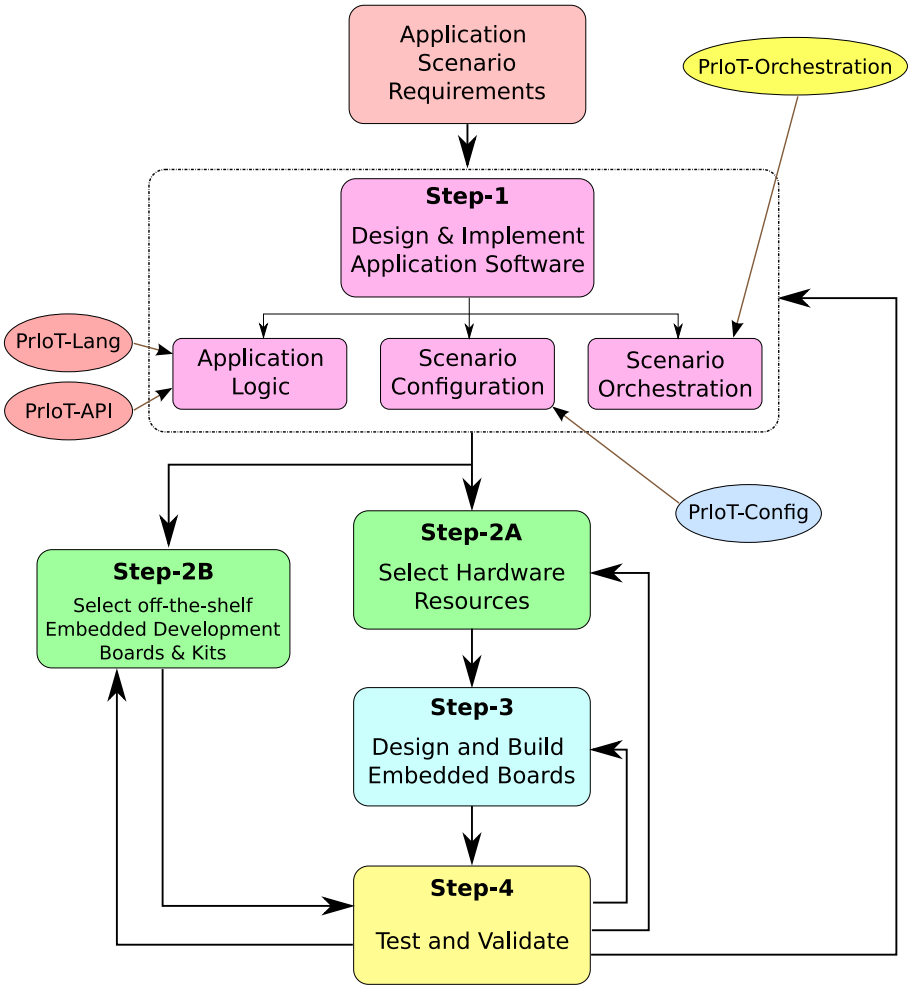


FIGURE 4.8: PrIoT Steps to Implement IoT Application

the cloud instead of the object's local storage. The object also has a relay (actuator) to actuate Motor which controls the entrance door. To access cloud services the object goes through a gateway. The object uses IEEE 802.11 and MQTT (Message Queuing Telemetry Transport) protocol to communicate with the gateway. Such scenarios can be enhanced after deployment for example in the context of smart building where spaces are scheduled at a daily granularity by analyzing the personnel agenda (meetings, development zone, team work, etc.) stored in the cloud and assisting them finding a workstation that suits their daily needs.

Implementation using PrIoT : As Figure 4.8 shows, the first step before designing the IoT application logic is to select the high level description of IoT resources used in the target deployment using PrIoT-DB. This step doesn't require to know the specific hardware vendor components but rather define

what are the high level hardware components that are selected to be manipulated by the application and configured using configuration file (*.cfig).
In our example scenario, the configuration file for the object and gateway is shown in Listing 4.2 and 4.3 respectively. In configuration file, the **Select** keyword allows the user to select hardware from the repository of IoT hardware resources maintained by **PrIoT-DB**, this also includes any relevant device libraries used by the application. The **Import** keyword is used to inform PrIoT which library to include in the project, in this example we included standard communication library for IEEE 802.11 and PubSub.
At this stage, the selected elements are defined as default elements from the PrIoT-DB and are not linked to a specific hardware vendor. Thus for the object, a generic RFID reader, a relay, a WiFi transceiver, an LCD display and a keyboard have been selected as electronic components and communication protocols such as a Wifi Station mode and PubSub Client have been selected as communication libraries.
For the gateway, we have exposed in the configuration file the usage of a WiFi transceiver and a PubSub library used as a server. However, as we will see in 4.7 and 4.6 we also have the ability to specify dedicated hardware and configuration at this stage by using the **Define** keyword allowing users to define precise hardware vendors for IoT devices. In case where no specific hardware vendor is specify, a default component is provided and used in the application logic.
After having defined the IoT components to be used, we can program the targeted scenario through the application file that consist of application logic written using device independent **PrIoT-APIs** and **PrIoT-Lang**. The application logic file (*.app) for the object and gateway is shown in Listing 4.4 and 4.5 respectively. The high level language uses a procedural c-like structure with conditional operators. To be coherent with the hardware components selected, we have to use the **Import** keyword and specify the related configuration file to be imported. In Listing 4.4 we see that the object senses the ID collected by the RFID reader and sends to the PubSub server channel the information that is executed back by the object to allow the user to access the premises if it has been identified. The application programmed for the gateway is shown in Listing 4.5, where information sent from the objects are collected through the PubSub channel and compared to the access database located in the cloud service. If the user identity is detected then the access is granted for the user.

```
# File name - config_sa_object.cfig

Select Sensor.RFID as s_rfid
Select Actuator.RELAY as a_relay
Select Transceiver.IEEE80211 as t_wifi
Select HMI.DISPLAY.LCD as h_display
Select HMI.KEYBOARD as h_keyboard

# Embeded System
Select Hardware.MCU.Family as AVR-8bit

# Communication Protocol
Import Communication.80211 as wc
Import Gateway.PubSub.client as pbs
```

LISTING 4.2: Configuration file : Security Access - Object

```
# File name - config_sa_gateway.cfig

Select Transceiver.IEEE80211 as t_wifi
Select Hardware.MCU.Family as ARM

# Communication Protocol
Import Communication.80211.Station as wc
Import Gateway.PubSub.server as pbs
Import Cloud.Registry.AccessID as aid
Define Cloud.Address.Name as dns1
```

LISTING 4.3: Configuration file : Security Access - Gateway

```
Import config_sa_object.cfig

void loop()
{
 payload = s_rfid.sense("ID")
 t_wifi.send(wc, pbs, tcp, "channel", payload)
 t_wifi.receive(wc,pbs,tcp,"channel",payload)
 if payload == OK
 {
  a_relay.execute(actuate,"HIGH")
  h_display.execute(display,"Access approved")
 }
 else
```

```
14   h_display.execute(display,"Access denied")
15 }
```

LISTING 4.4: Application Logic : Security Access - Object

```
1  Import config_sa_gateway.cfig
2
3  void loop()
4  {
5   t_wifi.receive(wc,pbs,tcp,"channel",payload)
6   NewPayload = aid.execute(dns1,compare,payload)
7   t_wifi.send(wc,tcp,"cloud_address", \\
8               "HTTP_REQUEST",payload)
9   t_wifi.send(wc,pbs,tcp,"channel",NewPayload)
10 }
```

LISTING 4.5: Application Logic : Security Access - Gateway

Following this approach, we have the ability to program a distributed scenario with default hardware specification. As not all hardware are similar, there are many factors - like technology, cost, power consumption, MIPS (Million Instruction Per Seconds) and number of peripherals - which influence the initial selection of hardware and ultimately determines the functionality and performance of embedded software. Thus, when the scenario are defined and validated, it is possible to target a more precise hardware components as shown in Listing 4.7 and 4.6 where first the former configuration file is imported with the **Import** keyword and the specific hardware components are selected and configured with the **Define** keyword.

```
1 import config_sa_object.cfig
2 Define Transceiver.IEEE80211 as ESP8266
3 Define Hardware.MCU.Type as rpi2
4
5 Define rpi2.GPIO.RX2 as t_wifi.TX
6 Define rpi2.GPIO.TX3 as t_wifi.RX
7 Define Cloud.Address.Name as mycloud.com
```

LISTING 4.6: Configuration file : Detailed Gateway Hardware

```
1 import config_sa_object.cfig
2
3 Define Sensor.RFID as NXP-Explore
4 Define Actuator.RELAY as Grove-Motor
5 Define HMI.DISPLAY.LCD as Adafruit
```

```
Define HMI.KEYBOARD as Logitec_K780

Define Transceiver.IEEE80211 as ESP8266

Define Hardware.MCU.Family AVR
Define Hardware.MCU.Type ATmega328p

Define ATmega328p.GPIO.RX4 as t_wifi.TX
Define ATmega328p.GPIO.TX5 as t_wifi.RX

Define ATmega328p.GPIO.OUT9 as a_relay.IN
```

LISTING 4.7: Configuration file : Detailed Object Hardware

In conclusion, prototyping IoT scenarios requires various heterogeneous hardware resources for implementing & testing application logic, this requires domain knowledge to understand heterogeneous device architecture and communication protocols. As a result the proof-of-concept takes considerable time thereby hindering the early adoption of IoT technology and services. In PrIoT, we rectified this problem by separating the dependency of application logic from scenario configuration. It also exposes high level abstraction in the form of PrIoT-API and PrIoT-Lang that hides the underlying heterogeneity of IoT devices and communication protocol.

4.7 Summary

In this chapter we introduced a new approach to tackle IoT device heterogeneity issues associated with device architecture, device peripheral diversity and protocol heterogeneity. and proposed a framework called PrIoT, for IoT application lifecycle management (development, deployment and maintenance). PrIoT introduces an intermediate intelligence between the IoT device hardware and the IoT application that usually resides in a cloud infrastructure. We provided the three design objectives of the PrIoT framework i.e. Rapid Prototyping, Hardware Configuration and Scenario Deployment. We described the various PrIoT framework's building blocks (PrIoT-Lang, PrIoT-API, PrIoT-Test, PrIoT-GI, PrIoT-HAL, PrIoT-Config, PrIoT-Parser, PrIoT-DB, PrIoT-UI, PrIoT firmware builder & uploader and PrIoT Orchestrator) along with implementation details. In addition, we described the steps needed to implement an IoT application scenario using PrIoT along with an example use case.

Finally, in PrIoT we integrated the best of available techniques (high level programming, hardware abstraction, hardware configuration, hardware database, user interface) proposed by existing solutions (IoT development platforms, embedded OSes, etc.) under one framework. More importantly, we implemented the IoT application invariant functionalities and programming patterns (Chapter 3) in PrIoT-API and PrIoT-Lang that provides minimalist programming APIs and language for IoT life cycle management. .

Chapter 5

R-Bus and P-Bus : Modular Systems for Designing Interoperable and Energy Aware Embedded Systems

5.1 Introduction

In this chapter, we address the challenge of easy and rapid prototyping of IoT hardware systems (IoT objects) by introducing a two new open hardware[1] specification named R-Bus (Resource Bus) and P-Bus (Power Bus) to design modular systems that better meet IoT hardware requirements and are usable across diverse IoT applications & also takes into account their disparate power requirements. Note that this approach is complementary to the software oriented approach (PrIoT) that we proposed in Chapter 4.
From an embedded system perspective, the difficulty in prototyping IoT object arises from the fact that an IoT object incorporates various disparate enabling technologies (processing unit, sensors, actuators, transceivers, power supply, human machine interfaces, etc.) [4, 2, 9], which requires systems designers to constantly test and evaluate new technologies and upgrade the final system appropriately. For our work, we adopted a simplified view of an IoT Object architecture centered around processing unit (PU) and peripherals [2]. The PU represents the "brain" in the form of ultra-low power microcontroller, microprocessor, system-on-chip (SoC), etc. and integrates a wide variety of heterogeneous *peripherals* (communication interfaces). This wide

[1] R-Bus and P-Bus are under creative commons share alike license, for more information on the specification please visit http://www.rbus.io

heterogeneous peripheral integration is intended to facilitate communication with systems outside of the processing unit. The number and type of peripherals supported by PU and their pin mapping vary from one PU to another, therefore this peripheral interface heterogeneity can lead to different and incompatible hardware design that we showed in previous chapters as being challenging for IoT developers.
Furthermore, from the power consumption perspective of an IoT object [25], little has been done to consider the power management system (PMS) in existing hardware platforms including modular systems as discussed in Section 2.3.1. As an integral part of the R-Bus system we also proposed and analysed a power reduction technique for incorporating in the future R-Bus revisions that will allow battery powered operation for testing the final system in the field.
Note that the power technique proposed can also be used with other IoT system designs and is not restricted to R-bus based modular systems.

This chapter comprises the following sections. In Section 5.2 we introduce a new modular system named R-Bus for designing embedded systems based on the shortcoming of existing solutions discussed in Section 2.3.1. In Section 5.2.4, we systematically evaluate R-Bus along with existing solutions and thereby compare the advantages of R-Bus in designing modular embedded systems. In Section 5.2.5, we validate R-Bus by implementing the requirements of a real IoT application scenario and its assessment against existing solutions. In Section 5.3, we introduce R-Bus power-board (PB) that caters the need for disparate power requirements of various IoT applications and exposes various features for runtime power optimization. Moreover in order to better understand the application use case of R-Bus power module, in Section 5.3.5, we proposed and implemented two power reduction technique named *Power Gating* and *Wake-Up Radio* using R-Bus power module along with detailed analysis. Finally, we conclude (Section 5.4) and highlight key improvements offered by the R-Bus system compared to the other standard approaches and provide important future directions to meet further IoT requirements.

5.2 R-Bus - A Resource Bus for Modular System Design

The goal of R-Bus is to allow easy integration of IoT resources by using a homogeneous interface and facilitate configuration of R-Bus interface based on available IoT resources. In order to accomplish this goal, we utilize a *modular architecture* [125, 126] approach to partition the system based on the concept of *bus modularity* [124], which is common in computer based system where all modules (IoT resources) are connected to a single common module (Processing Unit). In the R-Bus system, a collection of various peripherals (buses and channels) (see Table 5.1) forms a single bus that we named as *R-Bus*. The ad-

IoT Resources	Example Applications	Peripherals								
		Buses					Channels			
		UART	I2C	SPI	I2S	SDIO	Analog	Interrupt	PWM	GPIO
Sensors	Temperature, Humidity, Pressure, etc.		✓	✓			✓	✓	✓	✓
Actuators	Relay, Led, Motors, etc.		✓	✓				✓	✓	✓
Transcievers	WiFi, Ethernet, LoRa, SigFox, etc.	✓		✓				✓		✓
HMI	Display, Keyboard, Joystick, etc.		✓	✓				✓		✓
Memories	Flash, EEPROM, etc.		✓	✓		✓		✓		✓
Audio	Audio Codec, Speaker, Mic				✓		✓	✓		✓
Data Converters	ADCs and DACs		✓	✓				✓		✓
Miscellaneous	Real-time clocks, debugging, etc.	✓	✓					✓		✓

UART : Universal Asynchronous Receiver-Transmitter, I2C : Inter-Integrated Circuit, SPI : Serial Peripheral Interface, I2S : Inter-IC Sound
SDIO : Secure Digital Input Output, PWM : Pulse-Width Modulation, GPIO : General Purpose Input Output

TABLE 5.1: Main peripherals used in IoT applications

vantage of modular systems using bus modularity is that a product variant can be generated using the same bus, for example in computer systems using the same data bus and combining different types of CPUs & memory units it is possible to create systems with different processing power and memory capacity. Likewise in IoT due to diversity in IoT applications, the IoT resource requirements varies from one application to another therefore a modular system like R-Bus allows to easily generate an application specific product variant by interchanging R-Bus modules named *R-Bus auxiliary-board* and *R-Bus main-board* without redesigning the whole system.

By introducing an auxiliary-board that will contain the most changing components (such as sensors, actuators, transceiver, human machine interface, etc.) of an IoT device, and maintaining the most stable components (such as processing unit, debug circuit, etc.) of IoT devices on the main-board, a running IoT device will then be used longer and only some of its components that are in the auxiliary-board can easily be changed. This will allow from IoT application point of view, easy prototyping with heterogeneous IoT devices. On the other hand it will reduce the IoT devices related e-waste.

In order to accomplish this goal we propose to use for R-Bus the existing PCI-e x1 connector & socket specifications [221] for homogeneous interface and an on board memory to access the information related to IoT resource peripheral usage and that can be used for implementing plug & play architecture.

5.2.1 R-Bus Components

The system based on R-Bus is divided into two parts - R-Bus main-board and R-Bus auxiliary-board - as illustrated in Figure 5.1 & Figure 5.2 and are explained in detail hereafter.

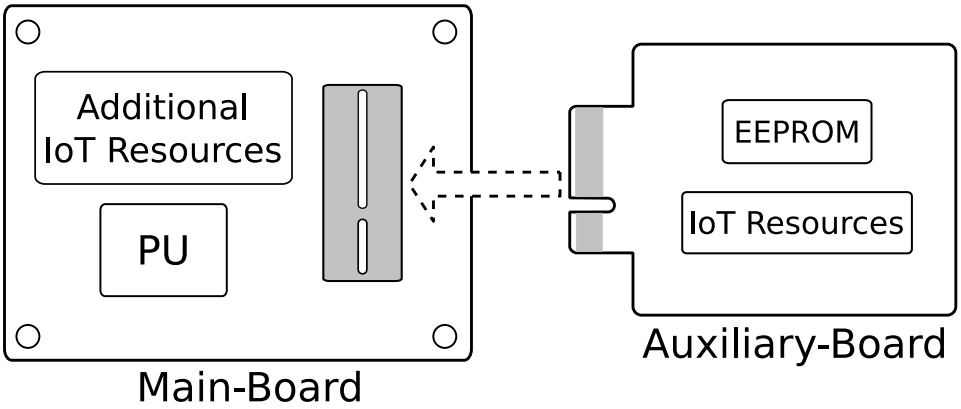


FIGURE 5.1: R-Bus : Main-Board and Auxiliary-Board

5.2.1.1 R-Bus Main-Board

The main-board contains at most one PU that communicates with IoT resources via R-Bus interface and is categorized into three classes based on the type of PU used. The different class borrows from the IETF RFC7228[25] that classify constrained devices based on PU capabilities in terms of memory, such as code size (ROM/Flash) and data size (RAM). The three different R-bus main-board classes and their characteristics are given below.

1. **Class 0 - Bare Metal** : The PUs with small memory capacity (RAM $\ll$ 10 KB & Flash $\ll$ 100 KB), limited functionalities through low level programming languages and reduced communication features.
2. **Class 1 - RTOS** : PUs with medium size memory capacity (RAM $\sim$ 10 KB & Flash $\sim$ 100 KB), enhanced functionalities through Real Time OSes [222] (RioT, contiki) including communication protocol stacks (CoAP, 6LowPAN, etc.) for constraint networks.

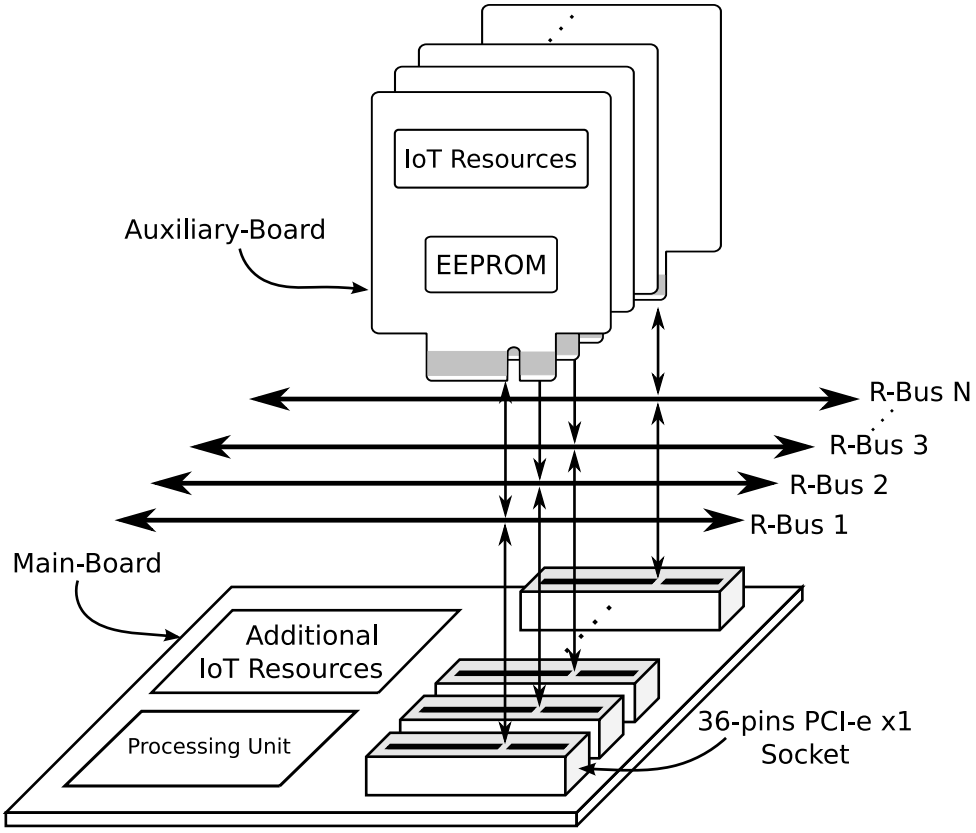


FIGURE 5.2: R-Bus : Main-Board and Auxiliary-Board

3. **Class 2 - OSes** : PUs with sufficient memory capacity (RAM $<$ 1 MB & Flash $<$ 10 MB) to host a light embedded Operating System (LINUX, BSD, etc) with access to advanced service packages (IP stack, Programs) and high level programming language.

This class designation allows users to easily identify the board capabilities and possible application use cases. Table 5.2 shows a non exhaustive list of commercially available processing units and their corresponding R-Bus class along with supported peripherals. Interestingly the number of PU's peripherals and cost increases with higher classes providing boundaries on available resources and relative cost per classes. The main-board can also contain additional peripherals that are not supported by R-Bus such as CAN (Controller Area Network), Ethernet, USB, etc. It can also contain additional one or more secondary processing unit for housekeeping purpose and complementary circuitries like - power, debug, etc. R-Bus system does not enforce any special design specification and specific component requirements on the main-board thus allowing various manufacturers to create a variety of application dependent main-board along with R-Bus interface to add IoT resources.

Processing Unit		Device Class	Peripherals							
			Buses					Channels		
Name	**Architecture**	*	**UART**	**I2C**	**SPI**	**I2S**	**SDIO**	**ADC**	**Interrupt**	**PWM**
ATtiny25	AVR	0	0	0	0	0	0	4	1	2
MSP430FR2000	MSP430	0	1	0	1	0	0	0	8	2
Atmega328P	AVR	0	1	1	1	0	0	8	2	6
STM32L011x3	ARM Cortex-M0+	0	2	1	1	0	0	10	16	7
STM32F042K6	ARM Cortex-M0	0	2	1	2	1	0	10	16	17
STM32L010RB	ARM Cortex-M0+	1	2	1	1	0	0	16	16	7
Atmega1284P	AVR	1	2	1	1	0	0	8	3	6
STM32F722ZE	ARM Cortex-M7	2	8	3	5	5	2	24	16	37
MSP430F6659	MSP430	2	3	3	6	0	1	12	32	18
AM3351	ARM Cortex-A8	2	6	3	2	2	3	8	8	3

*Class of constraint devices - RFC7228 [25]

TABLE 5.2: Commercially available processing units with corresponding device class

5.2.1.2 R-Bus Auxiliary-Board

The second part of the R-Bus system consists of one or more R-Bus auxiliary-boards that carries various IoT resources such as transceivers, sensors, actuators, HMI, various connectors for remote IoT resources, etc. The auxiliary-board is designed separately from the main-board and has a PCI-e x1 connector that plugs into the main-board which carries an equivalent PCI-e x1 socket.
R-Bus defines two types of auxiliary-boards based on the presence or absence of R-Bus memory on the board. This R-Bus memory holds additional board information stored in an EEPROM and reachable from the main-board through I2C with known addresses and attributes.

1. **Type 0** : When no R-Bus memory is detected on the board, a fixed pin mapping is considered as the default pin connection between the main and auxiliary-boards.

2. **Type 1** : When an R-Bus memory is detected on the board, it provides additional information for flexible interface configuration either as alternate pin-mapping or with full plug and play features for Class 2 devices using device tree.

5.2.2 R-Bus Pin-Mapping and Interface Configuration

The R-Bus connector has 36 pins (18 on each side), enough to *simultaneously* accommodate 5 (SPI, UART, I2C, I2S and SDIO) widely used peripherals, upto 2 PWM (output), upto 3 Analog (output) and upto 3 Interrupt signals. We propose in Table 5.3 the pin mapping of peripherals on R-Bus connector.

The pin mapping is carefully done in such a way that most widely used peripherals like UART, I2C, SPI, Interrupt and Analog, are on one side (Side-B) which makes PCB layout an easy task. As Table 5.3 shows, there are 20 pins

PCI-e Side-B Pin Name	Peripheral Pin Mapping	Peripheral Name	Alternate Function		PCI-e Side-A Pin Name	Peripheral Pin Mapping	Peripheral Name	Alternate Function
B1	$+3.3V$	Power	-		A1	$+3.3V$	Power	-
B2	GND	Ground	-		A2	GND	Ground	-
B3	I2C_SDA	I2C	None		A3	PWM0	PWM	GPIO6
B4	I2C_SCL				A4	PWM1		GPIO7
B5	SPI_MOSI	SPI			A5	UART1_TX	UART1	GPIO8
B6	SPI_MISO				A6	UART1_RX		GPIO9
B7	SPI_CLK				A7	I2S_SCK	I2S	GPIO10
B8	SPI_CS0				A8	I2S_WS		GPIO11
B9	SPI_CS1				A9	I2S_SD_OUT		GPIO12
B10	UART0_TX	UART0			A10	I2S_SD_IN		GPIO13
B11	UART0_RX				A11	GND	Ground	-
B12	GND	Ground	-		A12	SDIO_CLK	SDIO	GPIO14
B13	INT0	Interrupt	GPIO0		A13	SDIO_CMD		GPIO15
B14	INT1		GPIO1		A14	SDIO_DATA0		GPIO16
B15	INT2		GPIO2		A15	SDIO_DATA1		GPIO17
B16	AN0	Analog	GPIO3		A16	SDIO_DATA2		GPIO18
B17	AN1		GPIO4		A17	SDIO_DATA3		GPIO19
B18	AN2		GPIO5		A18	GND	Ground	-

TABLE 5.3: R-Bus: Pin Mapping

(B13-B18, A3-A10 and A12-A17) with alternate functions (marked as GPIO). This allows users to add those IoT resources that require GPIOs and can be reprogrammed for custom applications, for example - control signals, leds, push buttons, etc. To facilitate automatic configuration of R-Bus interface and to properly configure alternate function, each type-1 R-Bus auxiliary-board must include its own I2C compatible R-Bus memory in the form of EEPROM that holds a machine readable description of the board. In this way the descriptor becomes part of the board rather than available as separate file and provides a crucial building block in realizing plug & play architecture and resource discovery protocols. The R-bus memory specification inherits partly from Raspberry PI HAT [150] specifications by assigning known I2C addresses to reach R-Bus memory and define a specific memory structure in the form of memory blocks of meta-data.

Following is the list of memory blocks meta-data stored in R-Bus memory where each block represents a certain type of data information. Although the detailed discussion of EEPROM requirements and memory blocks structure

and data are beyond the scope of this work, interested readers are invited to follow the project website for detailed information [223].

- **Header**: Provides memory access validation for the master board with a signature, a version number and information about the total number of blocks and their size in memory.
- **Board Information**: Presents specific board information such as a *Universally Unique Identifier* (UUID), *Board Version*, *Board Name*, *Vendor Name* and *Serial Number*.
- **GPIO Map**: It defines an alternate pin-mapping layout for R-bus master board (Class 0-1) that can only assign a default GPIO peripheral as alternate pins. Generally, a software based peripheral implementation can be set on the master PU to transform standard GPIO into virtual peripherals (UART, I2C, SPI, etc.) but with less performance compared to hardware defined peripherals.
- **Device Tree Blob**: The Device Tree provides a way to describe non-discoverable hardware to the Linux kernel as a textual representation of the tree data structure of the hardware. This feature allows flexible assignments of different peripherals per pin but is only available to Class 2 devices that have an embedded Linux kernel and bootloader.
- **Custom Data**: These information presents the *board configuration* (IoT resources name, vendors, data format, etc.), *options* (Additional board configuration options) and *board function API* (When a MCU is active on the auxiliary-board and expose extra functionalities through a serial communication).
- **Bootcode**: This block allows to define a driver implementation written in a meta Language that the master board can include and execute. Domain-specific language (DSL) are becoming a suitable option for meta languages and proposal such as IoT-Link [115], ThingML [224] or PrIoT-LANG [19] are example of existing approaches.
- **Board Version** : Hardware version code to identify board revision and can also be used to locate compatible test software as recommended by the board developer.
- **Future Use**: This block is a reserved memory space for future usage such as new specification for additional hardware requirements, specification for plug & play capabilities such as the IEEE 1451.4 standard

for analog transducers, etc.

5.2.3 R-Bus Form Factor

R-Bus does not follow the standard PCI-e x1 form factor specification, nonetheless it provides some restrictions based on the type of (90° or 180°) of PCI-e x1 socket used on the main-board. A main-board with 90° socket provides a vertical approach to mount auxiliary-boards (see Figure 5.2), in this approach, there are no restrictions in the form factor as long as it does not interfere with components and connectors on the main-board. This allows board designers to customize auxiliary-boards based on application requirements or electronic enclosures, similar to "expansion cards" on desktop computers. On the other hand a main-board with 180° socket provides a horizontal approach, similar to mikroBUS^{TM} , Pmod, Arduino Shield and M2.COM. But this approach requires carefully defining the form factor for auxiliary-board as it might affect the form factor of the main-board, which requires thorough understanding and investigation into existing - modular systems, various components (RF modules, sensor, actuators, connectors etc.) packaging & form factor, feedback from industries, etc.

5.2.4 Evaluation

In this section we evaluate the R-Bus approach for modular system design, first by analyzing various qualitative design features of R-Bus with existing standards and finally by quantitative analysis using two metrics - *coverage* and *suitability* (defined later) - to measure the extent of compatibility of various modular systems for a given processing unit.

5.2.4.1 Qualitative Analysis

In contrast with existing standards (Section 2.3.1.1), the R-Bus offers following advantages to system designers some of them are summarized in Table 5.4 :

- **Peripheral Diversity**: The total number of peripherals offered by R-Bus is more than any other competing auxiliary-board standards (see Table 5.4), it exposes 5 peripheral (SPI, UART, I2C, I2S, SDIO) and up to 10 pins for general purpose interfaces (Interrupt, Analog, PWM, GPIO, AF).

- **Peripheral Density :** It provides the possibility to realize a complete IoT solution on a single board, which is otherwise impossible in auxiliary-board standards like Pmod and Grove because not all peripherals are simultaneously accessible on a single Pmod & Grove connector and is restricted in mikroBUS^{TM} because of the form factor.
- **Peripheral Access :** It offers simultaneous use of all 5 peripherals and 8 pins for general purpose interface, which is not possible with Pmod, Grove and restricted in mikroBUS^{TM} because of the form factor.
- **Cost :** Since R-Bus borrows its connector and socket from PCI-e x1 standard, which is maintained by PCI-SIG (PCI Special Interest Group) [225] and has a proven commercial success history in personal computers, therefore there are many vendors with existing manufacturing capabilities to provide inexpensive sockets and the abundance of available CAD (Computer Aided Design) references for both connector and socket. This makes R-Bus an easy and inexpensive solution for modular design and is comparable with other auxiliary-board standards.

In summary as shown in Table 5.4, R-Bus provides an extensive set of peripherals on a single module in comparison with other modular systems. In the next Section 5.2.4.2, we mathematically evaluate this benefit using two metrics - *coverage* and *suitability*.

Name	Size ($w \times l$ mm)	Embedded Peripherals								
		SPI	UART	I2C	I2S	SDIO	PWM	Analog	Interrupt	GPIO
R-Bus$^{\bullet}$	Flexible$^{\otimes}$	2*	2	1	1	1	upto 2	upto 3	upto 3	upto 10
mikroBUS$^{TM\bullet}$	25.4 × 57.15	1	1	1	0	0	1	1	1	0
pmod$^{\dagger\bullet}$	20.32 × $l^{\ddagger}$	1	1	1	1	0	upto 2	0	upto 1	upto 8
Grove System$^{\dagger\bullet}$	No Standard	0	1	1	0	0	0	2	0	2
Arduino Shield$^{\bullet}$	68.58 × 53.34	1	1	1	0	0	upto 6	upto 6	2	upto 20
RPi HAT$^{\bullet}$	65 × 56.5	2	1	1	1	1	upto 4	0	upto 28	upto 28

†Not all supported embedded peripherals are available on a single board, *One SPI with two chip select signals

$^{\bullet}$auxiliary-board standard, ‡no prescribed standard length, *no explicit mention of dedicated interrupt lines

TABLE 5.4: Comparison between various auxiliary-boards

5.2.4.2 Quantitative Analysis

In order to systematically evaluate R-Bus with other similar modular systems we define two metrics that allow us to measure the extent of compatibility of auxiliary-boards for a given processing unit on the main-board. The first

metric is called *coverage* denoted by C_n and is defined in Equation 5.1 as percentage ratio.

$$C_n = \frac{\alpha_n}{\beta} \times 100, n \in \mathbb{N} \tag{5.1}$$

where n is the "total number of auxiliary-boards", α_n is the "total number of peripherals occupied on n auxiliary-boards" and β is the "total number of peripherals supported by processing unit". The second metric is called *suitability* denoted by S_n and is defined in Equation 5.2 as percentage ratio.

$$S_n = \frac{\alpha_n}{n \times \gamma} \times 100, n \in \mathbb{N} \tag{5.2}$$

where n and α_n is same as in Equation 5.1 and γ is the "total number of peripherals supported by auxiliary-board".
The *coverage* is used to indicate how many peripherals can be covered if one or more auxiliary-boards are used, whereas *suitability* is used to indicate if the combined set of one or more auxiliary-boards are under utilized or not. Both C_n & S_n are dependent on each other through α_n and are used together to assess various auxiliary-boards. An auxiliary-board has high compatibility with PU when it has high C_n and S_n relative to other auxiliary-boards. In general, since β is constant for a given processing unit, one can increase the *coverage* by adding more auxiliary-boards on the main-board but this also decreases *suitability* by factor n.
In order to showcase the usefulness of *coverage* and *suitability*, we assume a worst case scenario where an application requires the use of all the peripherals available on PU, this condition allows us to evaluate various auxiliary-boards for worst case compatibility.
For this we selected one processing unit from each class of constraint devices listed in Table 5.2 and plot C_n and S_n as we increase n (number of auxiliary-boards). Without loss of generality of Equation 5.1 & 5.2 we decided to account for all the peripherals listed in Table 5.2 except interrupt. Figures 5.3, 5.4 and 5.5 shows C_n and S_n for AM3351 (class 2), STM32F042K6 (class 1) and Atmega328P (class 0) devices respectively.

As Figures 5.3, 5.4 and 5.5 shows, R-Bus provides a suitable balance between *coverage* and *suitability* as compared to other auxiliary-boards when used across three distinct class of processing units. C_n in R-Bus tends to reaches 100% coverage more sharply for each subsequent addition of auxiliary-board, for example it requires only 3 R-Bus auxiliary-boards to reach 100% coverage, i.e. $C_3 = 100$ in Figure 5.3 and 5.5. This sharp increase is due to the fact that R-Bus supports more peripherals than any other auxiliary-boards.

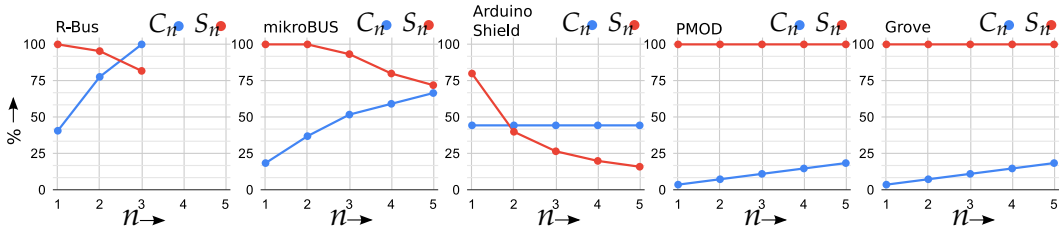


FIGURE 5.3: C_n and S_n vs n for AM3351 class 2 device for various auxiliary board based modular systems

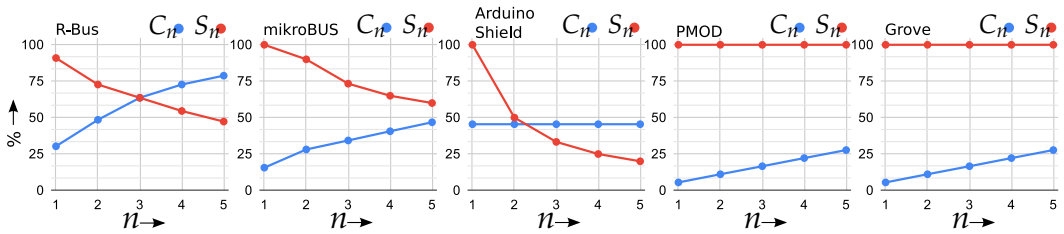


FIGURE 5.4: C_n and S_n vs n for STM32F042K6 class 1 device for various auxiliary board based modular systems

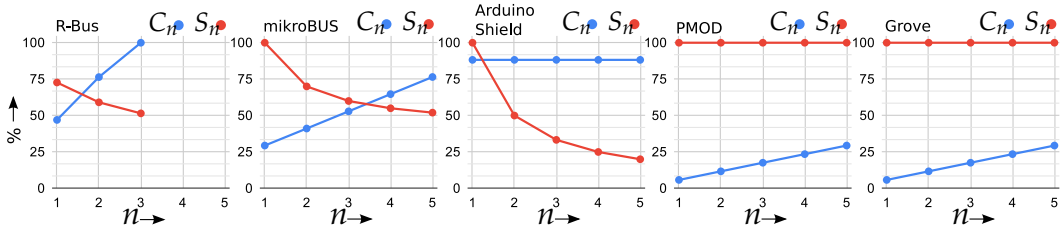


FIGURE 5.5: C_n and S_n vs n for Atmega328P class 0 device for various auxiliary board based modular systems

In Figure 5.5, R-Bus has relatively low *suitability* of $S_1 = 72.7\%$ because it supports some advance peripherals like I2S and SDIO which are not commonly found in class 0 PUs and hence are never used.

In mikroBUS system, as shown in Figure 5.3, 5.4 and 5.5, the initial *suitability* S_1 & S_2 is relatively very high but at the expense of low *coverage* C_1 & C_2 and therefore requires more auxiliary-board to reach comparable coverage to that of R-Bus. For example, in Figure 5.4 mikroBUS require 5 auxiliary-board to reach $C_5 = 46.8\%$ with $S_5 = 60\%$ whereas R-Bus requires only 2 auxiliary-boards to reach comparable $C_2 = 48.48\%$ with better *suitability* of $S_2 = 72.7\%$ even though mikroBUS show high initial *suitability*.

On the other hand Arduino shield uses a stackable approach (on top of each other) to add auxiliary-boards which hinders its use to add more than one auxiliary-board without pin conflict once all the peripheral pins are exhausted on the shield. Due to this, shields are not useful when a processing unit exposes mores peripherals than those supported by shields. This situation is clearly visible in Figure 5.3, 5.4 and 5.5 where the first shield exhausted all its

peripherals (high S_1), indicating that the addition of another shield does not change C_n, and therefore S_n tends to decreases sharply towards 0. Similar behaviour in C_n and S_n is observed in RPi HAT which uses similar stacking approach for adding auxiliary boards.
Finally, for auxiliary-boards like pmod and grove that supports one peripheral per board, in order to reach 100% coverage it requires as many auxiliary-boards as there are peripherals on the processing unit and hence the linear rise of C_n for pmod and grove in Figure 5.3, 5.4 and 5.5. Moreover since pmod does not support SDIO and grove does not support SPI, I2S and SDIO, therefore C_n cannot reach 100% while S_n remains at 100% till all the peripherals supported by pmod and grove are exhausted from PU. In general, this trend in C_n and S_n is expected from system that supports one peripheral per board.
The extent to which a main-board can support multiple auxiliary-boards are not only driven by the number of peripherals supported by the processing unit used, but more importantly an application requirement can also limit the requirement of multiple auxiliary-boards even though a processing unit can accommodate more. Although having multiple auxiliary-boards is not a requirement, but it can increase the usability of main-board across diverse IoT applications that requires more peripherals than those provided by the single auxiliary-board.
Both *coverage* and *suitability* provide the necessary tools to do a preliminary analysis of modular systems for designing application specific IoT based embedded systems.

5.2.5 Validation : Implementation & Assessment

In this section, we showcase the practical advantage and proof-of-concept of R-Bus (Type-0) by implementing a practical wireless sensor node. We also compared the sensor node implementation with other auxiliary-board standards for implementing the similar node.

5.2.5.1 A LoRaWAN enabled Environmental Sensor Node

For this validation, we decided to implement an environmental sensor node that records barometric pressure and temperature. In order to actualize this sensor node, the IoT resource requirements is as follows : we decided to use LoRaWAN [226] technology for communication using SPI based RFM95 transceiver module. There is also an external memory (SPI based W25X40CL

4-MB flash) for data logging and sending measurements in batches to save transmission power. An external I2C based battery backed RTC (Real Time Clock) is used to timestamp measurements and to wake-up MCU from sleep. Moreover an external I2C based secure element (ATECC608A) is used to run AES encryption algorithm in hardware to save on chip MCU memory (ROM + RAM) and processing time associated with encryption algorithms. Finally, for the sensor we decided to use an I2C based barometric pressure sensor BMP280, that also has an inbuilt temperature sensor. Table 5.5 summarizes the list of various components used along with respective standard interface.

IoT Resource	**Description**	**Standard Interface**
Transceiver	RFM95W LoRaWAN module	SPI
Sensor	BMP280 Barometric Pressure	I2C
RTC	PCF8523 with backup battery	I2C
Security	ATECC608A Crypto-Authentication	I2C
Memory	W25X40CL 4-MB Flash	SPI

TABLE 5.5: IoT Scenario : List of IoT Resources & Standard Interface

Figure 5.6 shows our auxiliary-board prototype based on R-Bus standard. For the main-board prototype we decided to use ATmega328P MCU, the bootloader inside the MCU is same as *Arduino Pro Mini 3.3V-8MHz* therefore it is possible to use Arduino IDE to create and debug applications. Figure 5.7 shows auxiliary-board plugged into the main-board using 90° PCI-e x1 socket.

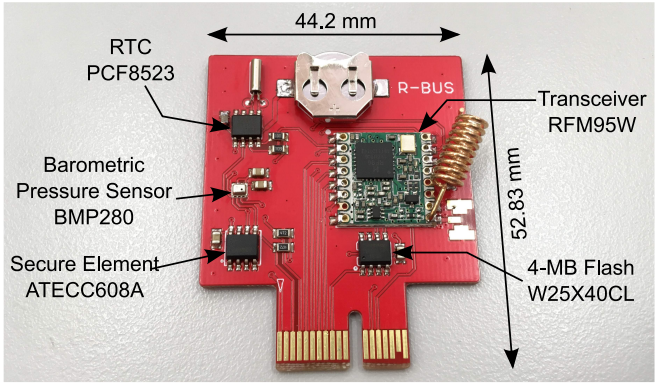


FIGURE 5.6: R-Bus : Auxiliary board Prototype

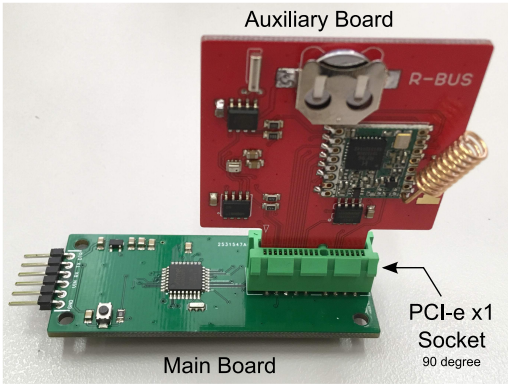


FIGURE 5.7: R-Bus : Main and Auxiliary board Prototype

In comparison with other auxiliary-board standards to implement the similar sensor node using the same IoT resources. For example - PMOD will require at least 3 auxiliary-boards - two SPI boards one for transceiver & one for memory and one I2C board for sensor, RTC and secure element, it may require an additional I2C board if it is difficult to accommodate the three IoT resources on a single board due to form factor restriction. On the other hand it is impossible to realize the sensor node using Grove System as it does not support SPI. Although MikroBUS^{TM} supports simultaneous use of SPI and I2C, considering the maximum size of 25.4×57.15 mm it may still require 2 auxiliary-boards to accommodate all the IoT resources. Finally the Arduino Shield can easily accommodate all the IoT resources to implement the sensor node.
Although in this scenario we only used two embedded peripherals - SPI and I2C, but because of the number of IoT resources required (5 in this case) we need more than one auxiliary-boards for PMOD and MikroBUS^{TM} based solutions. One can easily interpolate, when the requirement of distinct peripherals increases the number of auxiliary-boards for PMOD, MikroBUS^{TM} and Grove System also increases. Also the size of the main-board increases with the number of auxiliary-boards.

5.3 P-Bus - A Power Bus for Modular System Design

The benefits of IoT can only be realized when the devices (for example, Wireless Sensor and Actuator Networks) are capable of battery-operated or working under extreme energy limitations. This requires careful consideration (Section 2.3.1.2) into designing low-power systems that are energy aware.

In this section we introduce a P-Bus module that is designed to satisfy the IoT power requirements (energy harvesting, battery operated, wall powered, etc.) and more importantly provides an intelligent homogeneous interface to capture or inquire the necessary features to better optimize power utilization during runtime. We also show in Section 5.3.5 using two power optimization application use cases (Power Gating and Wake-Up Radio) that it is possible to implement different power optimization techniques thanks to its intelligent homogeneous interface.

5.3.1 P-Bus : Overview

The P-bus modular system consists of three parts, 1) *P-Bus Module*. 2) *P-Bus Interface*. 3) *P-Bus Connector*. as illustrated in Figure 5.8. The *P-Bus module* is an electronic board that provides the necessary power required for proper functioning of an IoT object (main-board). More importantly, it also includes a smart power management block that can expose various features that are accessible to via *P-Bus interface* (Section 5.3.2). The P-Bus module is connected to the main-board via *P-Bus connector*.

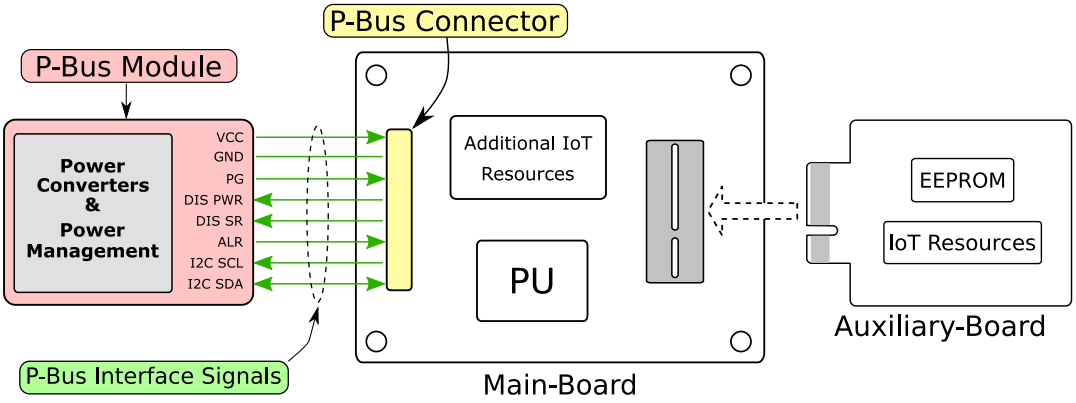


FIGURE 5.8: P-Bus : P-Bus Module, P-Bus Connector and P-Bus Interface

The list of non-exhaustive features that are considered important for better power optimization are listed in Table 5.6. The features mentioned in the table are implemented directly on *P-Bus module* using existing integrated circuit solutions in order to consume less power or it can be implemented as a firmware on a low power microcontroller. These design options will be considered in more detail in Section 5.3.3.

Feature Name	Description
Battery Voltage	Monitoring voltage level.
Battery Current	Monitoring current consumption.
Battery State of Charge (SoC)	Level of charge remaining relative to battery capacity.
Battery Temperature	Monitoring battery temperature.
Status and Alerts	Alerts for under and over voltage & current protection
Power Good	Indicates when the system voltage is at an acceptable level.
Disable Power	Allows host to disable power for energy conservation.

TABLE 5.6: P-Bus Module Feature List

5.3.2 P-Bus : Interface

In order to interface with external boards (for example, R-Bus main-board), the P-Bus module exposes various signals that allows the main-board to send and receive features necessary for runtime power optimization. Table 5.7 lists various signals supported by the P-Bus module along with a short description. The selection decision of signals are based on the usefulness of these signals in various energy aware IoT applications as mentioned in Section 2.3.1.2. The importance of each signal is explained hereafter.

Signal Name	Description	Signal Direction
VCC	Regulated Output for R-Bus system	PB $\rightarrow$ MB
GND	Common Ground	-
PG	Power Good signal indicating the VCC has reached a predefined upper threshold	PB $\rightarrow$ MB
DIS PWR	Disable Power	PB $\leftarrow$ MB
DIS SR	Disable Switching Regulator	PB $\leftarrow$ MB
ALR	Alert - Inform host MCU for abnormal behavior or status information	PB $\rightarrow$ MB
I2C SCL	I2C clock signal	PB $\leftarrow$ MB
I2C SDA	I2C data signal	PB $\leftrightarrow$ MB

PB : R-Bus Power-Board, MB : R-Bus Main-Board

TABLE 5.7: P-Bus Module Interface signals

1. **VCC** - VCC is the regulated output voltage ($3.3V$ or $5.5V$) generated by the P-Bus module for the R-Bus system. The regulated supply voltage is needed for proper functioning of various sub-systems within R-Bus.
2. **GND** - Common ground between P-Bus module and R-Bus main-board.
3. **PG** - In a battery powered application, the battery voltage is not always at an acceptable level. If not taken into account, the host can experience abrupt power failure. The Power Good (PG) signal can indicate to the host module that the supply voltage has reached an acceptable level.

Normally, the main-board can stay in sleep to conserve power and use PG signal as a wake-up source and perform the necessary power hungry tasks like transmission, etc.

4. **Disable Power** - The main-board can use this signal to disable supply voltage (VCC) to conserve leakage power in energy sensitive applications. This operation is also known as power gating and is one of the power optimization techniques discussed in detail in Section 5.3.5.1 as an application of the P-Bus module.

5. **Disable Switching Regulator** - The switching regulators are widely used in many energy harvesting circuits for charging batteries or super capacitors. But in noise or EMI (Electromagnetic Interference) sensitive applications, the switching regular can interfere with proper functioning of RF circuits. Therefore using this signal, the host can temporarily disable switching regulators during transmission.

6. **Alert & Status** - This signal can indicate abnormal behavior such as, under & over voltage and current protection for battery powered devices, charging status, high temperature for battery protection, etc.

7. **I2C** - The I2C [15] can be used to access a programmable power management integrated circuits in the form of battery gauge, battery management unit, etc. The processing unit on the main-board can use this interface to access power management features directly from the integrated circuit available on the P-Bus module. The I2C can also be used to access a low power slave microcontroller if available on the P-Bus module to implement, access & configure customized power management features that are not available directly from existing integrated solutions. The importance of I2C interface is discussed in Section 5.3.5.2 using an example application.

5.3.3 P-Bus Module : Class Distinction

The R-Bus power modules are divided into three classes based on the availability and usage of various interface signals.

1. **Class** *P*0 - The *P*0 is the most basic power-board with only VCC and GND signals. The is useful in those applications that are wall powered and the application does not utilize or need any power optimization features. Example use case - Wall powered edge devices or gateways,

it can also be battery powered devices that does not expose any power optimization features back to the host MCU.

2. **Class** *P*1 - The *P*1 is more advanced than the *P*0. It contains a low power EEPROM that holds a feature descriptor. The feature descriptor contains the detailed information about each signal and list of features that are available from the power-board. The main-module can use this feature descriptor to configure its peripheral interface to access the available features and use the appropriate libraries to access power management ICs over I2C to collect features. The behavior of other signals are the same as described in previous Section 5.3.2.

3. **Class** *P*2 - It contains a dedicated low-power power management slave MCU that allows the user to implement customized power optimization algorithms directly inside the MCU. The host MCU can access the algorithm features directly from slave MCU via I2C. In the *P*2 module, the *Alert* signal is used by the slave MCU to inform any abnormal behavior, the host MCU can use this signal to wake up from sleep and ask for relevant information from the slave via I2C that caused the abnormal behavior. It also contains an EEPROM similar to Class *P*1 that holds the feature descriptor. The behavior of other signals are the same as described in previous Section 5.3.2.

5.3.4 P-Bus Module : Application Class

In this section we will describe and map various application use cases that are possible using various classes of P-Bus Modules. For this we have partitioned applications into various classes, designated by *PACx* (**P**-Bus **A**pplication **C**lass), where *x* is class number. The partition is based on the source of power, type of energy storage and the requirements of power optimization features. The various application classes are explained below along with Table 5.8 that shows the mapping of application class with P-Bus module class.

Application Class	Energy Source		Energy Storage		P-Bus Class
	Wall Powered	Renewable	Non-Rechargeable	Rechargeable	
*PAC*0	✓	✗	✗	✗	P0
*PAC*1	✓	✗	✗	✓	P0
*PAC*2	✓	✗	✗	✓	P1
*PAC*3	✗	✗	✓	✗	P1
*PAC*4	✗	✓	✗	✓	P1
*PAC*5	✓	✓	✗	✓	P2

Non-Rechargeable - AA, AAA, etc.

Rechargeable - Lithium-ion, Lithium-polymer, Supercapacitor, etc.

Renewable - Solar, Wind, Vibration, etc.

TABLE 5.8: P-Bus Module Application Class

- *PAC*0 **:** This is the most basic application class, the system is wall powered with no energy storage element and does not utilize any *external* features for optimizing power consumption. Typical examples for this application class are - gateways or edge devices that are based on powerful hardware such as Raspberry Pi, BeagleBoard, etc. The devices used in this application class have energy limitation class of type *E*9 as shown in Table 2.1.
- *PAC*1 **:** In this class, the system is wall powered backed with rechargeable storage in case of power outage and does not utilize any *external* features for optimizing power consumption similar to *PAC*0.
- *PAC*2 **:** This application class is similar to *PAC*0 and *PAC*1, but can also utilize external features for better power optimization.
- *PAC*3 **:** In this class, the application uses non-rechargeable batteries as input power source and uses various P-Bus signals to optimize power consumption to extend the end of life period. The devices used in this application class have energy limitation class of type *E*2 as shown in Table 2.1.
- *PAC*4 **:** The application uses renewable energy as a power source with energy storage element in terms of rechargeable battery or super capacitor. It uses various P-Bus signals to optimize energy consumption and manage charging and recharging of storage elements. The devices

used in this application class have energy limitation class of type *E*1 as shown in Table 2.1.

- *PAC*5 **:** This is the most advanced application class, in this class the application uses a low power microcontroller for implementing intelligent power optimization algorithms. The features of the algorithm are available to the main processing unit via I2C interface. The system can have any type of power input source such as wall powered, wall powered with battery backup, non-rechargeable battery, energy harvesting with rechargeable battery or super capacitor storage. The application can also use other P-Bus signals for power optimization.

5.3.5 P-Bus : Validation

In order to validate the P-Bus module we first propose and evaluate in detail two power optimization techniques namely - *Power Gating* [227] and *Wake-Up Radio* (WUR) [158], that are used in a variety of energy aware wireless sensor nodes. The reason for this detailed discussion is not only to showcase the benefits of energy awareness in IoT objects but also to state the requirements of extra interface to implement such techniques that are not available in existing 2.3.1.1 development boards. Also, more importantly we showcase how these techniques can be implemented using the P-Bus module in order to take advantage of the homogeneous interface that allows easy integration of these power optimization features across a wide variety of wireless sensor nodes, etc.

5.3.5.1 Application-1 : Power Gating

In this application use case we implement and evaluate a power optimization strategy known as *Power Gating* [227] to reduce leakage power between two consecutive transmits - when the microcontroller (MCU) is in inactive state. We showcase using this technique and the measurement results presented in [155] the significant increase in battery life, when the inactive period is large and inactive current starts to dominate. The power gating technique presented here is independent of any wireless technology and other hardware components. On a side note, our solution can also be used to complement the technique presented in [156], where [156] reduces active state current and our solution reduces inactive state current. Finally, we showcase how power gating can be implemented using class *P1* of P-Bus module.

Power gating brief description : Power gating [227, 228] is a technique to reduce power consumption by temporarily and selectively shutting down current from the circuits/subcircuits that are not in use. This allows reducing overall system standby current during inactive (sleep) state. The goal of power gating is to minimize leakage current [228] (inactive/sleep current). Figure 5.9 illustrates a generalized block diagram of a typical system utilizing power gating. The power gating controller generates the required power gating signals to selectively and temporarily disable power from various subcircuits/sub-systems using electronic switches.

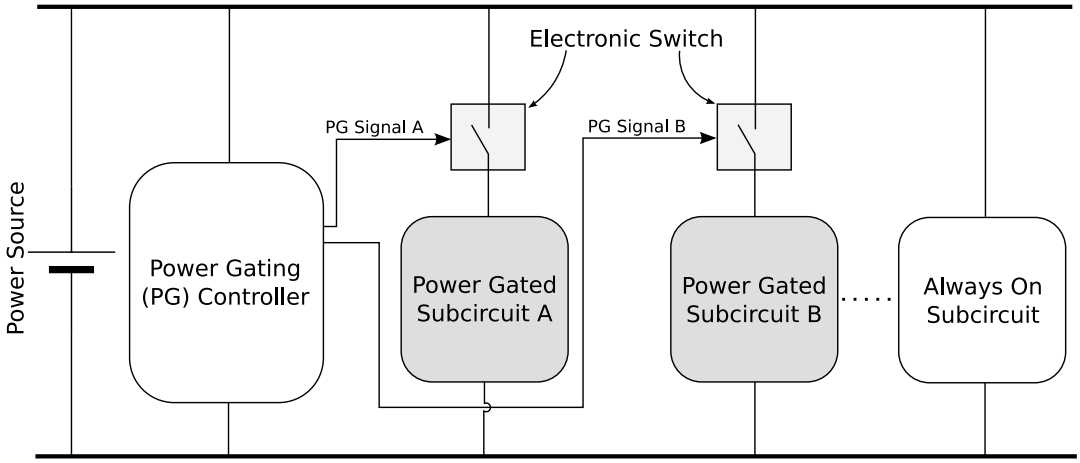


FIGURE 5.9: Illustrative block diagram of power gating

Evaluation : It is possible to generalize the average current (I_{avg}) consumption of any hardware platform, let us consider that the platform is programmed for periodic message transmission (IoT monitoring application) with period T_{Notif} (notification period), than I_{avg} can be calculated as :

$$I_{avg} = \frac{1}{T_{Notif}} \int_0^{T_{Notif}} i(t)dt = \frac{1}{T_{Notif}} \left(\int_0^{T_{active}} i_{active}(t) + \int_{T_{active}}^{T_{Notif}} i_{inactive}(t) \right) dt \tag{5.3}$$

T_{active} is the total time spent in active state (wake-up, read sensor, transmit, etc.), where $i_{active}(t)$ and $i_{inactive}(t)$ are current consumption profile in active and inactive state respectively. Also $T_{inactive}$ can be obtained as :

$$T_{inactive} = T_{Notif} - T_{active} \tag{5.4}$$

It is important to note that, $i_{active}(t)$ depends on numerous factors such as processing unit, operating voltage & frequency, transceiver, application algorithm, wireless network & technology, temperature, etc. and therefore it is difficult to model the behavior accurately which requires careful measurement setup & methods [229], nevertheless [154] presents analytical models of energy consumption for various operations in wireless sensor device like - data acquisition (regular and event driven), data processing (hardware dependent), data communication (point-to-point - SIGFOX & LoRa and time synchronized network - TSCH) and also [155] presents analytical models for both acknowledged and unacknowledged LoRaWAN transmission for various spreading factors (SF) based on the measured current consumption on an actual hardware platform.
On the other hand $i_{inactive}(t)$ can be considered constant throughout $T_{inactive}$ and is equivalent to the summation of all the sleep currents of various components available on the platform and active currents of always-On components (for example, Equation (5.10)). Therefore Equation (5.3) can be simplified as shown in Equation (5.5), where $I_{inactive}$ is total sleep current of the platform and $\frac{T_{active}}{T_{Notif}}$ is duty cycle.

$$\begin{aligned} I_{avg} &= \frac{1}{T_{Notif}} \left(\int_0^{T_{active}} i_{active}(t)dt + I_{inactive}.\, T_{inactive} \right) \\ &= \frac{1}{T_{Notif}} \int_0^{T_{active}} i_{active}(t)dt + I_{inactive} \left(1 - \frac{T_{active}}{T_{Notif}} \right) \end{aligned} \tag{5.5}$$

Further using Equation 5.5, we can calculate the theoretical lifetime, denoted by $T_{lifetime}$, of a battery-operated end-device as shown in Equation (5.6), where C_{bat} is battery capacity expressed in mAh (milliampere hour)

$$T_{lifetime} = \frac{C_{bat}}{I_{avg}} \tag{5.6}$$

One of the main goal of IoT embedded designers and application developers is to increase $T_{lifetime}$ by systematically modifying the factors that influence I_{avg}. It is easy to visualize from Equation (5.5) the factors - T_{Notif}, T_{active}, i_{active} and $I_{inactive}$ - that determines $T_{lifetime}$ for a given C_{bat}. We consider the impact of T_{Notif} and I_{sleep} on $T_{lifetime}$.
We can calculate the asymptotic theoretical upper bound of $T_{lifetime}$ denoted by $\hat{T}_{lifetime}$, using Equation (5.5) & (5.6) and taking the limit $T_{Notif} \rightarrow \infty$,

$\hat{T}_{lifetime}$ is given by :

$$\begin{aligned} \hat{T}_{lifetime} &= \frac{C_{bat}}{I_{inactive}} \\ I_{inactive} &= \lim_{T_{Notif} \to \infty} I_{avg} \end{aligned} \tag{5.7}$$

The implication of Equation (5.5) & (5.7) is that, the contribution of $I_{inactive}$ in I_{avg} starts to dominate as T_{Notif} becomes large as compared to T_{active} i.e $\frac{T_{active}}{T_{Notif}} \ll 1$. Furthermore, while designing an IoT end-device one can easily gather sleep currents ($I_{inactive}$) of various components from manufacturer's datasheet and can easily predict $\hat{T}_{lifetime}$ early in design phase. Also it is relatively easy to measure $I_{inactive}$ accurately using inexpensive equipment.
Now we illustrate the effect of $i_{active}(t)$, T_{active} and T_{Notif} on $T_{lifetime}$ for a given $I_{inactive}$. Without loss of generality and for realistic assumption of $i_{active}(t)$ and T_{active}, we decided to reuse the measurement data of LoRaWAN transmission from [155] and is summarized in Table 5.9, where $C_{active} = \int_0^{T_{active}} i_{active}(t)dt$. We also assumed a battery capacity of 2400 mAh, which is equivalent to AA battery capacity.

Parameters		C_{active}	T_{active}	$I_{inactive}$
SF	Payload			
7	242 bytes	95.93 mA.s	2.934 sec	$40\mu A$
12	51 bytes	304.15 mA.s	5.577 sec	$40\mu A$
Common Parameters - Class A protocol, Bandwidth BW = 125 KHz, 8-symbol preamble length				

TABLE 5.9: LoRaWAN Transmission Measurements

As illustrated in Figure 5.10, for relatively lower values of $i_{active}(t)$ and T_{active} (i.e SF7) 50 percent of $\hat{T}_{lifetime}$ (1000 days) is achieved earlier as we increase T_{Notif} ($T_{Notif} \approx 27$ min for SF7 and $T_{Notif} \approx 85$ min for SF12).

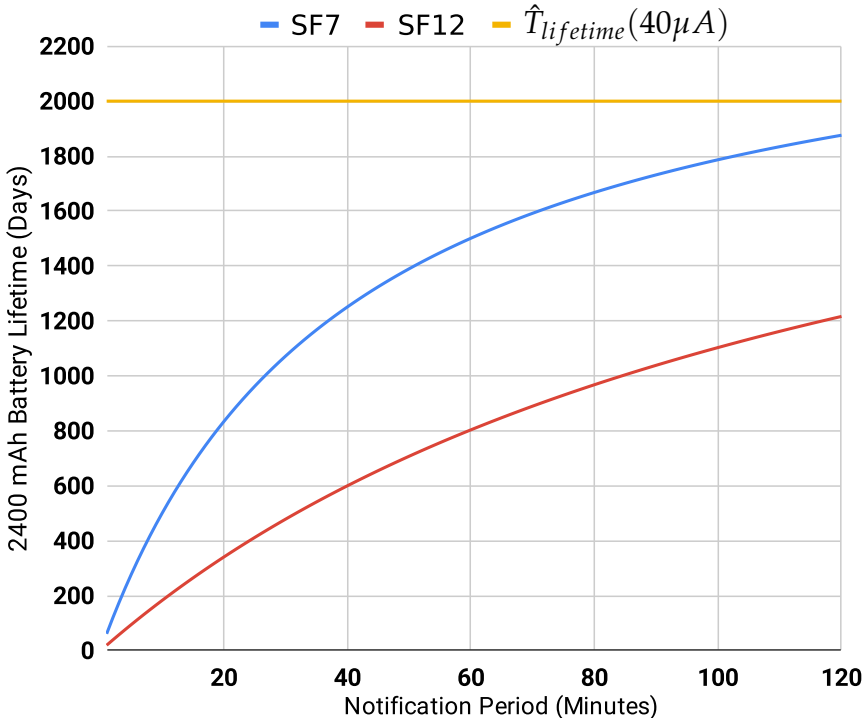


FIGURE 5.10: $T_{lifetime}$ for LoRaWAN SF7 & SF12 transmission as a function of T_{Notif}

Next, we illustrate the effect of $I_{inactive}$ on $T_{lifetime}$ as a function of T_{Notif} for a given $i_{active}(t)$ and T_{active}, again for realistic assumption and without loss of generality we use the data of Table 5.9. Figure 5.11 illustrates $T_{lifetime}$ for various $I_{inactive}$ as a function of T_{Notif} for LoRaWAN SF7 & SF12 transmission. Note that $I_{inactive} = 50nA$ (Equation 5.9) corresponds to the situation, where power gating is used to reduce inactive current and $I_{inactive} = 40\mu A$ as used in [155].

It is interesting to note that for lower values of T_{Notif}, $\leq$ 9 min for SF7 and $\leq$ 20 min for SF12 there is not much difference in $T_{lifetime}$ because C_{active} dominates I_{avg} and hence $T_{lifetime}$. The implication of this result is that in an application where there is a requirement of frequent transmission which corresponds to short T_{Notif}, one should try to reduce $i_{active}(t)$ & T_{active} rather than inactive current in order to obtain the desired $T_{lifetime}$.
Furthermore, the effect of $I_{inactive}$ starts to dominate early for lower values of $i_{active}(t)$ & T_{active} as we increase T_{Notif} ($\geq$ 9 min for SF7 and $\geq$ 20 min for SF12). Interestingly in an application with infrequent transmission which corresponds to large T_{Notif}, where $I_{inactive}$ dominates, one can try to incorporate techniques (for example - power gating) to reduce inactive current rather than $i_{active}(t)$ & T_{active} to obtain better end-device lifetime.

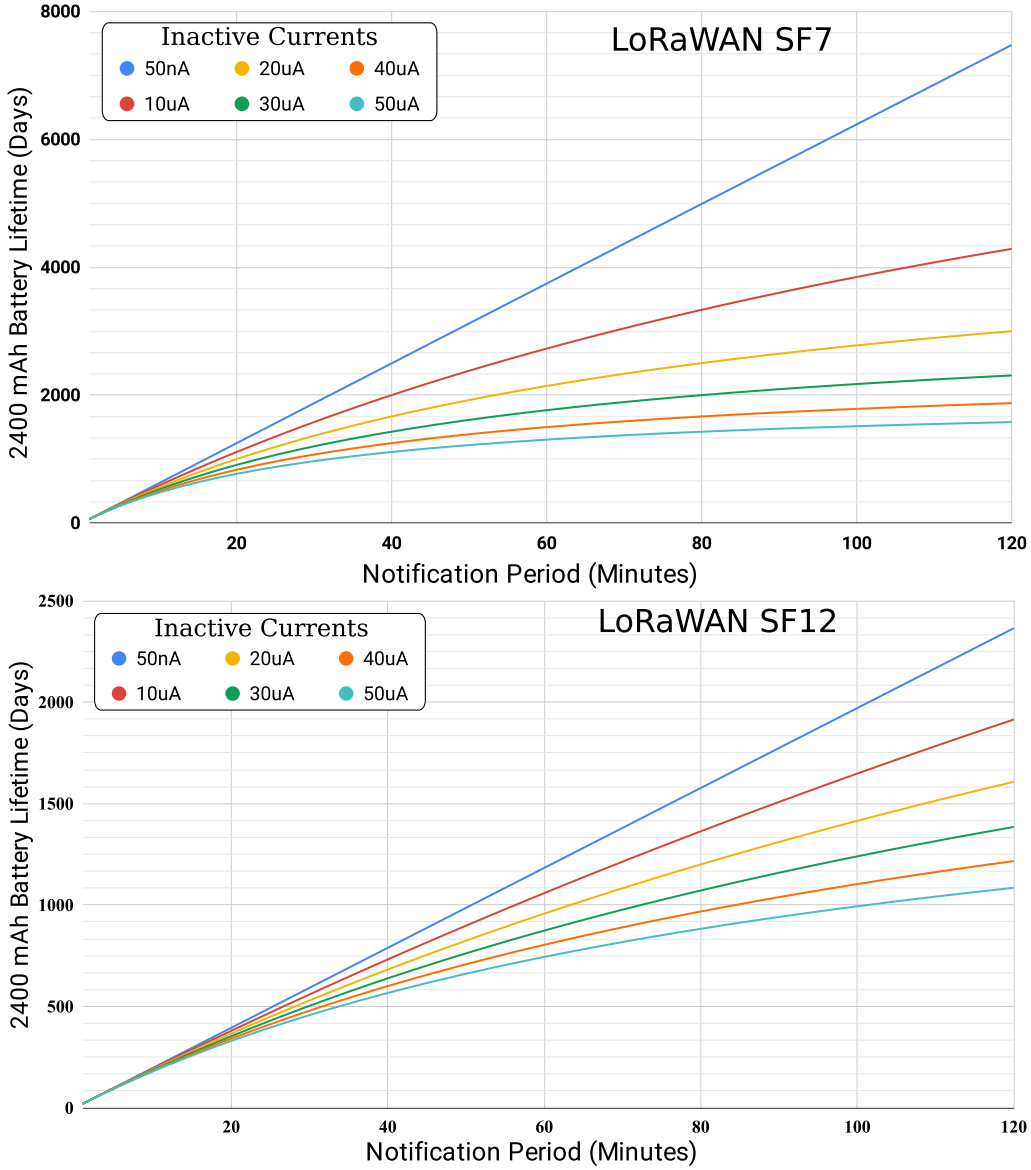


FIGURE 5.11: Battery life vs notification period for various inactive currents - LoRaWAN : SF7 (Top) and SF12 (Bottom)

Validation : We validated the importance of power gating technique by implementing a battery operated Arduino compatible LoRaWAN enabled wireless sensor node. As a byproduct of our work, the proposed platform enables easy, cost effective, battery operated and low power solution for experimental LoRaWAN [226] field studies and can also be used to validate the power models [154] of LoRaWAN technology and better estimate battery life. Although our platform uses LoRaWAN radio, the power gating technique is independent of any wireless technology and other hardware components. Moreover, our solution can also be used to complement the technique presented in [156], where [156] reduces active state current and our solution reduces inactive state current.

Hardware Platform Details : The platform consists of two boards that together form a battery operated wireless node. As shown in Figure 5.12, the *first board* (referred to as B1) combines on a single board an Arduino compatible MCU (Atmega328p) with LoRaWAN radio (RFM95W module). The bootloader inside the MCU is the same as *Arduino Pro Mini 3.3V-8MHz* therefore it is possible to use the Arduino IDE to create applications and debug via external FTDI serial adapter, similar to Arduino Pro Mini. The *second board* (referred to as B2) is a power-supply and power-management board for B1. It uses nanoPower boost converter (MAX17223 [230]) to convert voltage from two 1.5V AAA batteries (3.0V in series) into regulated 3.3V output voltage for B1. The value of the boost converter inductor is carefully selected to reliably convert input voltage ranging from 2.0V to 3.0V into regulated 3.3V at 200 mA output current. For the power gating controller, a nano-power system timer (TPL5111 [231]) is used to temporarily disable power from B1, the timer's time interval is programmable using an external resistor.

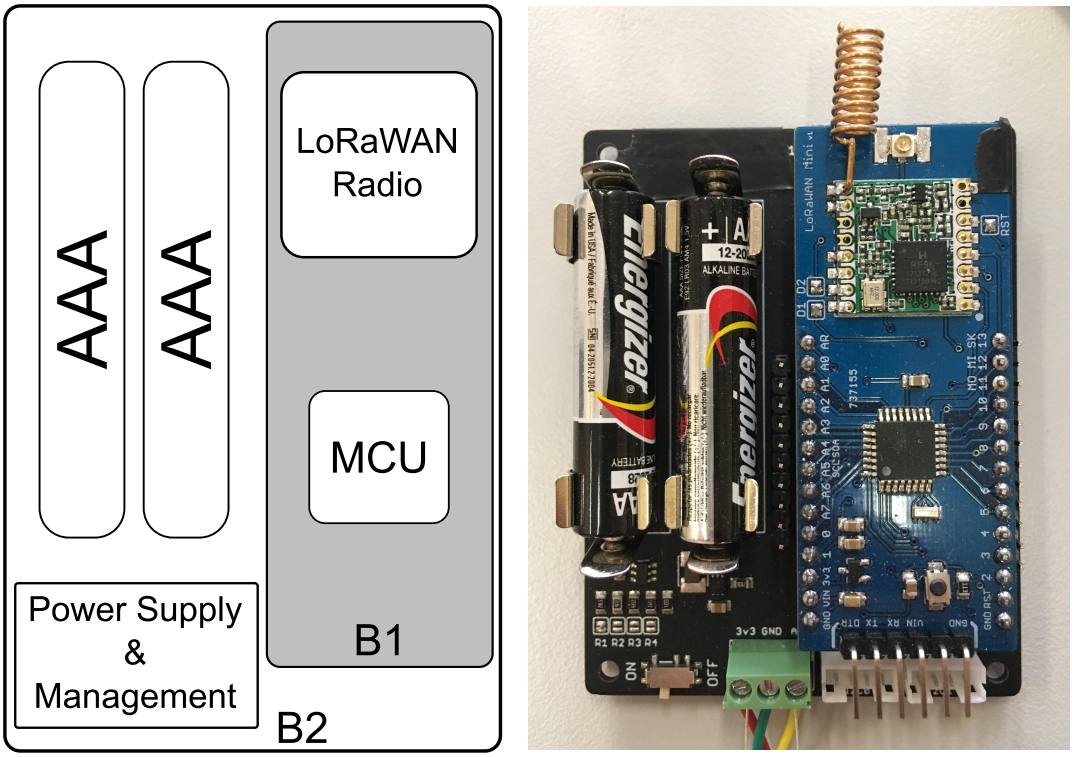


FIGURE 5.12: Platform block diagram

Power Management & Operation : The power management is purely in hardware and is independent of wireless technology. As illustrated in Figure 5.13, the B2 has two important components - boost converter (electronic

switch) and system timer (power gating controller). The boost converter generates the regulated 3.3V for the proper functioning of B1 and the system timer. The system timer waits for the MCU (B1) to generate power disable signal and after receiving the signal, the timer disables the boost converter, thereby removing power from B1, this process is called *"self destruction"*. Since the system timer is continuously powered, therefore it is still ticking and after a predefined time interval it enables the boost converter and the power is back again. Before disabling the power the MCU performs the necessary IoT application task as programmed by the user.

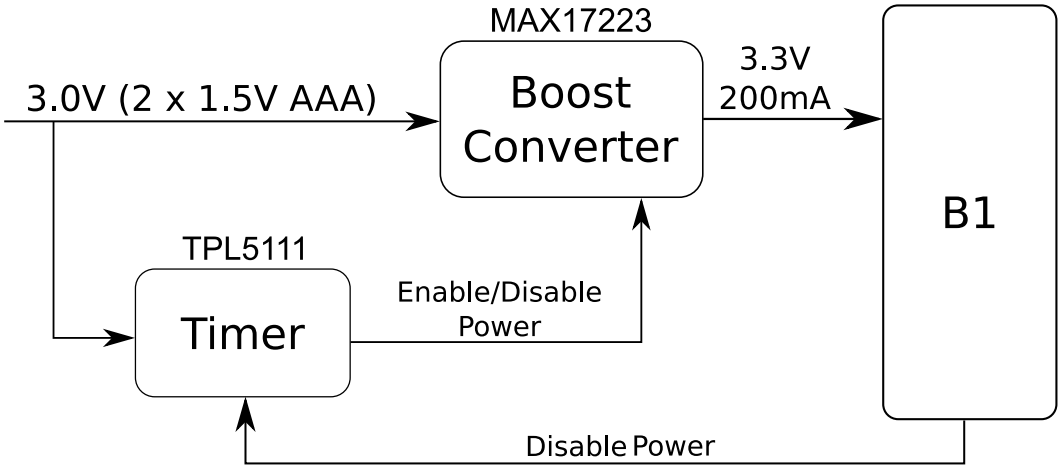


FIGURE 5.13: Block diagram of B2

Figure 5.14 shows the timing diagram of platform operation. This technique is useful for star networks like LPWAN (LoRaWAN, Sigfox, etc.) and for applications with infrequent transmission, for example - Smart Cities, Smart Agriculture, etc.

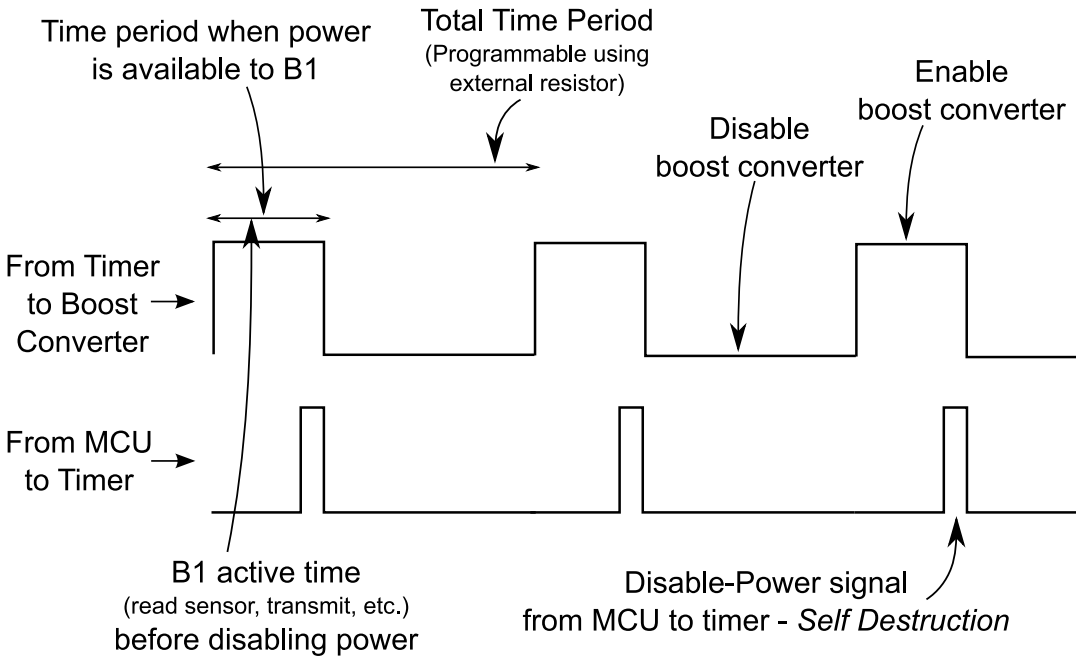


FIGURE 5.14: Platform Operation - Timing Diagram

Platform Power Consumption : The advantage of technique mentioned in Section 5.3.5.1 is that between two consecutive transmit/receive the total shutdown current drawn $I_{T_{SD}}$ from the batteries is effectively reduced to the current consumed by the system timer I_{ST}, shutdown current of boost converter $I_{BC_{SD}}$ and an unknown platform dependent leakage current $\delta_{leakage}$ - see Equation (5.8). The $\delta_{leakage}$ is calculated as the difference between analytical and measured current consumption of the platform, this allows to take into account the quality of overall platform circuit design and is usually very small. After obtaining the values of I_{ST} and $I_{BC_{SD}}$ from their respective device datasheet, the $I_{T_{SD}}$ is approximately equal to Equation (5.9).

$$I_{T_{SD}} = I_{ST} + I_{BC_{SD}} + \delta_{leakage} \tag{5.8}$$

$$I_{T_{SD}} = 35nA + 0.5nA + \delta_{leakage} \tag{5.9}$$

In contrast with other techniques, where we utilize various power down modes of the components to reduce the effective power during sleep, the total power down (sleep) current ($I_{T_{PD}}$) consumed is given by Equation (5.10). Where I_{BC_A} is boost converter active current (always on to supply sleep current) and $I_{MCU_{PD}}$ & $I_{R_{PD}}$ are MCU & Radio power down currents respectively. In this case the system timer (I_{ST}) is used to wake up MCU from sleep. After obtaining the values of I_{ST}, I_{BC_A}, $I_{MCU_{PD}}$ & $I_{R_{PD}}$ from their respective device

datasheet, the $I_{T_{PD}}$ is approx. equal to Equation (5.11).

$$I_{T_{PD}} = I_{ST} + I_{BC_A} + I_{MCU_{PD}} + I_{R_{PD}} + \delta_{leakage} \tag{5.10}$$

$$I_{T_{PD}} = 35nA + 500nA + 100nA + 200nA + \delta_{leakage} \tag{5.11}$$

It is important to note that Equation (5.10) does not include current consumed by external components attached to B2 such as sensors, actuators, etc. as they are application dependent. One can imagine the addition of more sleep currents if these external components are attached to the platform, where as Equation (5.8) is independent of various sub-system sleep and leakage currents. Also in general the total platform's sleep current depends on the type of components used and the overall circuit design, therefore Equation (5.10) varies from one platform to another. Table 5.10 lists sleep currents for few popular hardware platforms. Note from Table 5.10 that sleep current of [232] and [233] differs considerably even though they have same MCU and Transceiver, this is because [232] is a module and requires additional hardware components (power supply, debug circuit, etc.) before it can be used.

Device Name	**MCU**	**Transceiver**	**Sleep Current**
MultiTech mDot [234] ⋆	STM32F411 ⋆	SX1272 [235]	$40\mu A$
iM880B-L [232] ⋆	STM32L151 •	SX1272	$1.85\mu A$
NetBlocks XRange [233] †	STM32L151	SX1272	$70\mu A$
iM222A [236] ⋆	CC2530 °		$1\mu A$
⋆ Module, † Platform, ⋆ [237], • [238], ° [239]			

TABLE 5.10: Sleep Current for various IoT Devices

P-Bus Module - Power Gating : Figure 5.15 shows the proposed implementation of wireless node architecture using Class *P*1 module of the P-Bus. The wireless node takes advantage of the power optimization technique (power gating) using the available P-Bus interface signal (DIS_PWR). The same P-Bus module can be easily used with other wireless nodes that have the similar P-Bus connector and can take advantage of power gating technique provided by this module.

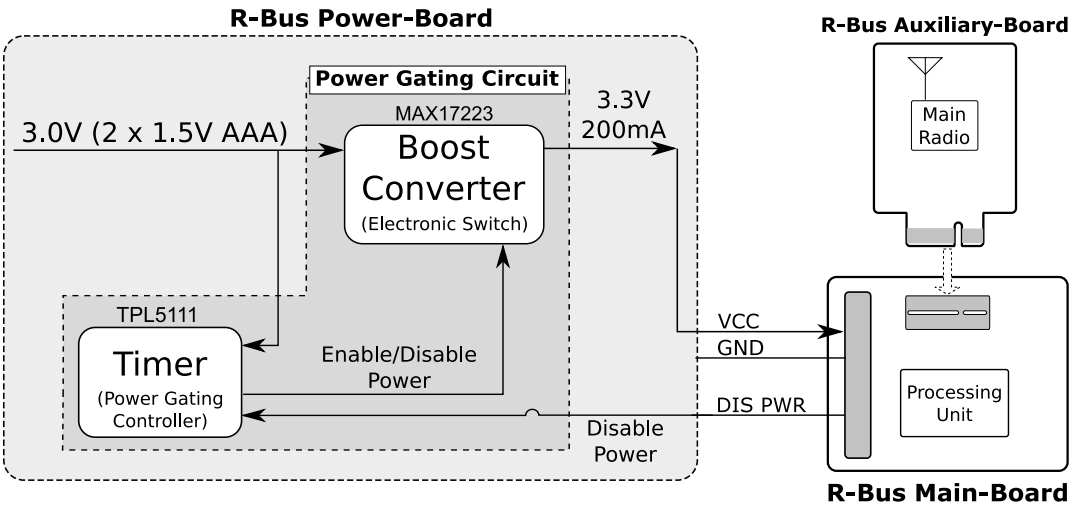


FIGURE 5.15: P-Bus : Power Gating

5.3.5.2 Application-2 : Power Gating & Wake-up Radio

In this subsection, we demonstrate the use of P-Bus (Class *P*2) by implementing *Power Gating* along with *Wake-up Radio* on the same board.

The wake-up radio (WUR) is a technique to reduce communication power of wireless nodes whereby an auxiliary receiver circuit known as WUR receiver relieves main radio from continuous listening of transmission medium for an incoming messages [158, 160]. Wake-up radios employ an asynchronous wake-up mechanism to notify the main processing unit for a potential incoming message. In contrast with wireless node without wake-up radio the idle listening of main radio receiver consumes more energy, therefore one of the main goal in wake-up radio is to design an ultra low power (also, low latency and high sensitivity) receiver circuit that consumes significantly low power in comparison with idle listening of main radio.

Figure 5.16 illustrates the block diagram of wireless node that takes advantage of both WUR and power gating as a power optimization technique. For simplicity Figure 5.16 shows only relevant signals and system blocks. The power gating circuit comprises an ultra low power system *timer* that acts as power gating controller and a *boost converter* that acts as an electronic switch to enable or disable power in response to the signal generated by the timer.

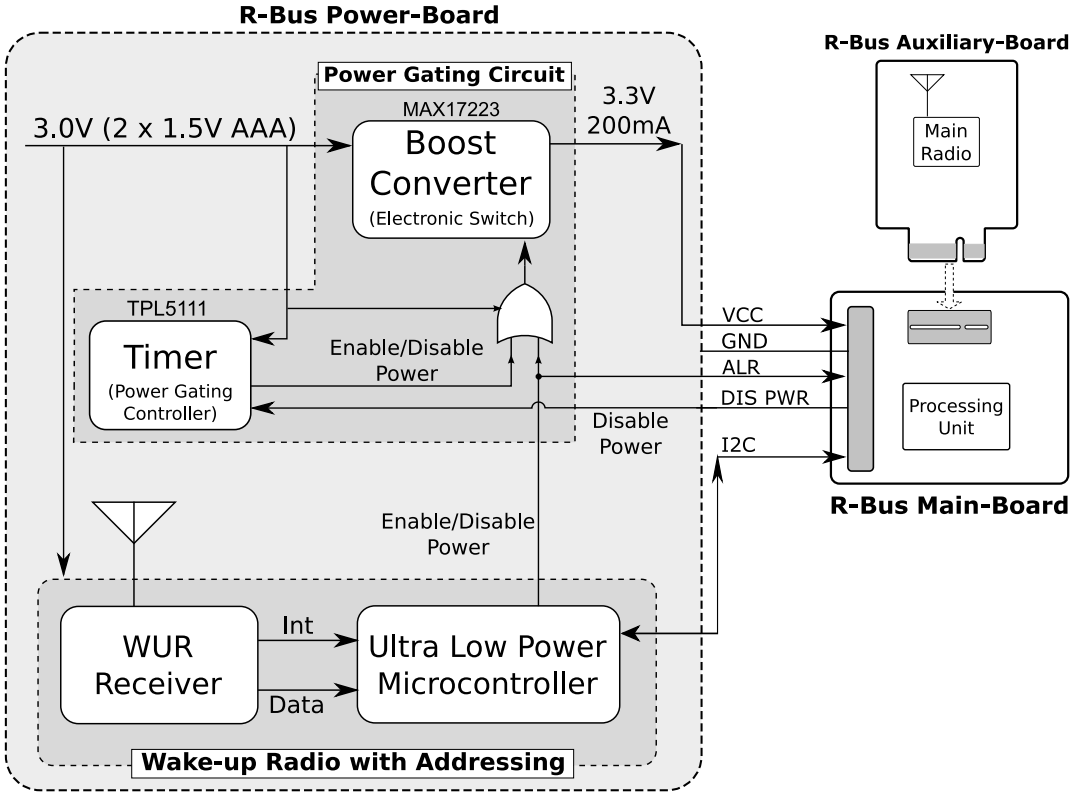


FIGURE 5.16: P-Bus : Power Gating & Wake-up Radio (WUR)

The timer waits for the processing unit on the R-Bus main board to generate the power disable signal (*DIS_PWR*) and in response to this signal the timer turns off the electronic switch (boost converter), thereby removing power from the main radio (R-Bus Auxiliary-Board) and processing unit. After a predefined time interval the timer enables the boost converter, thereby restoring the power.

On the other hand, the WUR consists of in addition to wake-up receiver an ultra low power microcontroller (ULPμC) for address matching, similar to the one presented in [160]. On successful address matching the ULPμC issues a power enable signal which also acts as level-interrupt (ALR) to the main processing unit (PU) to distinguish WUR wake-up from timer wake-up, this allows processing of incoming messages. The main PU can instruct ULPμC to disable power through I2C interface. The OR gate allows electronic switch to receive power enable/disable signal from both timer and WUR.

As shown in Figure 5.16, the proposed wireless node architecture can be implemented using P-Bus system. The wireless node can take advantage of the power optimization techniques (*Power Gating* and *WUR*) offered by the P-Bus module using a homogeneous interface that is usable across various wireless

node that requires similar power optimization techniques.
In summary, we showed using two IoT scenarios the benefits of P-Bus modular systems. The P-Bus exposes an intelligent interface that better caters the power requirements of an IoT object. The P-Bus module can be used with any IoT object that has a P-Bus connector for interfacing with a P-Bus module and is not just limited to R-Bus modular systems. Finally it is also possible to complement P-Bus with our software oriented approach (PrIoT - Chapter 4) by using PrIoT-API for accessing features (Table 5.6) supported by the P-Bus module.

5.4 Concluding Remarks

In this chapter, we addressed the IoT device heterogeneity with a hardware oriented approach, where we have proposed two new modular systems named R-Bus (Resource Bus) and P-Bus (Power Bus) to cater the requirements of IoT application while designing an embedded system (wireless sensor actuator node, edge node, etc.). We provided a detailed description of both modular systems and how they satisfy the peripheral heterogeneity control problem by exposing a homogeneous intelligent interface. We provided a detailed evaluation of R-Bus against existing modular systems and showed that R-bus poses many advantages over existing modular approaches in terms of the number of supported peripherals, simultaneous access to peripherals, compatibility across a diverse class of constraint devices, form factor, interface configuration, etc. We also implemented a R-Bus based hardware prototype of an environmental wireless sensor node and compared its advantages if the similar node is realized using existing modular approaches.
For P-Bus, we provided a description of its usability across various energy requirements of IoT applications. We also provided a detailed validation by implementing two distinct energy aware IoT applications.
Finally, more work is needed in order to facilitate the widespread use and evolution of both modular systems and also to ensure compatibility of boards across other modular ecosystems.

Chapter 6

Conclusions and Future Work

In the context of generalised Internet of Things for enabling and accelerating digital transformation in different domains, we identified that IoT end device heterogeneity is slowing down this process. We argue that IoT interoperability implemented in the industry today is actually handling only the heterogeneity from the IoT gateways up to the IoT applications and services in the IoT cloud platforms, thus the end to end interoperability up to the heterogeneous IoT end devices is still a big research and development challenge. In this thesis we identified different aspects of heterogeneous IoT ecosystem and its corresponding efforts to handle it in order to build this end to end interoperability and ease IoT applications and services development, deployment, and maintenance regardless of the IoT end devices heterogeneity.
This chapter summarises the contributions of this thesis, in Section 6.1 we address the problem of IoT device heterogeneity from both hardware and software perspective. Then potential future research directions are presented in Section 6.2.

6.1 Summary of Contributions

The research proposed to solve the heterogeneity control problem of the IoT devices, this will bring solutions that will reduce the IoT prototyping and development complexity, it will thus enable inexperienced personnel to *develop*, *deploy* and *maintain* IoT applications on the network of heterogeneous IoT devices. The development and the design of our solutions are guided by the challenges and problems presented in Chapter 1. Having identified the shortcomings of various solutions in the form of framework, development platforms & tools, embedded operating systems, etc. in Chapter 2, the

requirements to implement the proposed solutions were established in Chapter 3. The contributions associated with this research work are as follows:

- We realized the importance of the software abstraction layer (SAL) for hiding various heterogeneous technologies underlying IoT devices for allowing easy and rapid IoT adoption. For implementing this SAL, in Chapter 3 we studied the IoT application characteristics using an IoT architecture approach and extracted most common functionalities and programming patterns that are IoT applications invariant [17, 18]. We proposed the requirements of high level abstraction based on these invariant characteristics that are necessary for implementing software abstraction layers. Our objective is to lay the foundation for our proposed framework (Chapter 4) for IoT application lifecycle management that utilizes this abstraction layer.

- We proposed a new Open Source framework named PrIoT [19] in Chapter 4 that hides the IoT device manufacturer heterogeneity from the application developer. The main design philosophy behind the framework is "code once port everywhere". We defined the three objectives of the framework i.e. *rapid prototyping* - by utilizing high level abstraction as described in Chapter 3 in the form of PrIoT-API and PrIoT-Lang, *hardware configuration* - that provides configuration template and optimum bill of material in accordance with IoT application requirements and finally *scenario deployment* - that is achieved using an orchestrator that interprets high level IoT application functionalities into hardware specific commands. We also explained the main building blocks that make up the entire PrIoT framework.

- We described in detail the reference implementation of the PrIoT framework as a proof of concept. In order to implement our framework we used various open source languages and tools. We showed the benefit of the framework by implementing an example IoT use case.

- From a hardware perspective, we defined the problems associated with IoT device peripheral heterogeneity while designing hardware solutions for prototyping IoT applications. In order to systematically tackle this problem we studied various requirements of IoT applications in terms of - peripherals usage, types of IoT hardware and their associated peripherals, memory, processing and energy constraints of IoT devices. Based on this study, we proposed two new Open Hardware modular systems named R-Bus (Resource Bus) [20, 21, 223] and P-Bus (Power

Bus) for designing interoperable embedded systems. The full details of the proposed solutions are provided in Chapter 5.
We systematically evaluated the R-Bus with existing modular systems using two metrics - *coverage* & *suitability* ratios and showed the advantage of our system in implementing IoT scenarios along with an example implementation of practical wireless sensor node. For P-Bus, we validated our solution by systematically analysing and implementing two practical power optimization techniques - power gating and wake-up radio - that are widely used in energy aware IoT applications [22, 23].

6.2 Future Directions

In this section we discuss the potential extension to our work and future directions. This research demonstrated the importance of intelligently designed software and hardware solutions in implementing IoT use cases by reducing the complexity due to IoT devices heterogeneity. To extend our work, we have identified the future related work to follow up, some of which we already started to explore. Based on that, we have identified the following potential future research directions.

Software Perspective : From a software perspective, we proposed a new framework for rapid prototyping of IoT applications along with its reference implementation. However, we have planned to extend our framework along the following lines.

- A key challenge for its wide adoption will be to design HAL (Hardware Abstraction Layer) with a generalized API to accommodate a larger number of heterogeneous IoT devices and also to support a developer friendly environment for device driver development.
- In order to facilitate its widespread adoption and also its continuous evolution in the IoT community, we plan to continue the development and foster it's community through PrIoT website (`http://www.priot.org`), repository of example scenario, extensive documentation, software design guidelines for implementing device drivers that adhere to the specification of PrIoT-API, etc.

- It is interesting to analyze the best methodology to connect our framework with the IoT interoperability layer that allows communication protocol homogenization and service deployment.
- It is also interesting to analyze the performance of the firmware generated by the reference implementation of our framework on various heterogeneous hardware devices and also a qualitative and quantitative comparison with different frameworks available.
- In order to tackle the problem of massive firmware update due to the scale of IoT, it is interesting to investigate firmware over-the-air (FOTA) in our framework, to remotely update the firmware. For this we plan to analyze the limitations and merits of various existing FOTA projects such as Mender [164].
- We plan to study and make use of semantic interoperability technologies such as SSN (Semantic Sensor Network) by W3C (World Wide Web Consortium), OWL (Web Ontology Language), WSDL (Web Services Description Language), etc. in our framework to allow interoperability among other frameworks and platforms.
- We also plan to continue the development of the device management feature (PrIoT Orchestrator) of our framework with the ability to design and control a complete IoT scenario. As orchestration in the cloud has been extensively covered by solutions such as kubernetes, we are interested to follow the ongoing efforts of the community to adapt it to the edge with approaches such as Cloud4IoT and K3S [240, 241]. Furthermore, to bridge the gap of orchestrating the end-systems at Layer-1, we will extend our PrIoT proposal to separate configuration and application logic to be harmonized with the orchestrator solutions stated before.

Hardware Perspective : From a hardware perspective, we proposed two new modular systems named R-Bus and P-Bus. It constitutes a step forward in building an interoperable modular embedded system that hides underlying peripheral heterogeneity of IoT hardware resources. More work is needed to adapt R-Bus and P-Bus based systems to meet further requirements of IoT and also for the widespread adoption in the IoT community. For this we see the following important directions :

- In order to facilitate the widespread use and evolution of R-Bus & P-bus in the educational, research & industrial community and also to ensure compatibility of boards across our modular ecosystem we will create a hardware design guidelines for implementing R-Bus & P-Bus based systems. This will include but is not limited to - Form factor, connector selection, reference PCB design files, supporting documents, example boards, learning guide, etc. To strengthen our community around R-Bus & P-bus proposals and promote our open hardware specification, we will also maintain a website and repository (`http://www.rbus.io`).
- The success of any IoT hardware is based on the availability of software development tools and example applications, although R-Bus & P-Bus systems does not have any restriction on the use of particular development tools but the requirement of sample application accompanying both systems is important therefore a repository of example code and demonstration is required for rapid prototyping and experimentation.
- Machine readable description of IoT hardware resources is another important aspect that is not available by default in existing systems. It plays a crucial role in designing plug & play architecture, resource discovery, universal resource interface, etc. To accomplish this, one of the options is to extend/reuse the existing similar standards such as *"IEEE 1451.4 Transducer Electronic Data Sheets (TEDS)"* [102] and SensorML [101] for IoT. Both R-Bus & P-Bus support on-board non-volatile memory to add machine readable description. It is then interesting to investigate the merits of both TEDS and SensorML along with other standards and adopt the one that is best suitable for our modular system. This will also facilitate easy sharing of both systems among industries, researchers and open communities.
- The ability to uniquely identify an IoT object plays an important role in authentication, provisioning and security. One solution is to attach a globally unique node address using IEEE EUI-48 or IEEE EUI-64 (Extended Unique Identifier) global identifier [242], interestingly these identifiers are available pre-programmed in EEPROMs from various vendors. Therefore a single EEPROM on the system will serve two important functions - machine readable board description and unique node address. Apart from this, it is interesting to investigate other standards comparing their merits for unique identification of hardware devices.

The future work presented in this section requires considerable support from the open community to support its continuous development and foster its usage in the education, research and industry. Therefore, for our PrIoT framework we decided to use an open license based on GPL-v3 and our two modular systems R-Bus and P-Bus are under Creative Commons ShareAlike license.

Bibliography

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A survey". In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[2] Ala Al-Fuqaha et al. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys & Tutorials* 17.4 (2015), pp. 2347–2376.

[3] Jayavardhana Gubbi et al. "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future generation computer systems* 29.7 (2013), pp. 1645–1660.

[4] Farzad Samie, Lars Bauer, and Jörg Henkel. "IoT Technologies for Embedded Computing: A Survey". In: *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM. 2016, p. 8.

[5] *PrIoT: Prototyping the Internet of Things*. http://www.priot.org.

[6] Luis Sanchez et al. "SmartSantander: IoT experimentation over a smart city testbed". In: *Computer Networks* 61 (2014), pp. 217–238.

[7] *Experiential Living Lab for the Internet Of Things*. https://cordis.europa.eu/project/rcn/95205_en.html. [Online; last accessed 25-Oct-2018].

[8] *TEstbed for Future Internet Services*. https://cordis.europa.eu/project/rcn/96812_en.html. [Online; last accessed 25-Oct-2018].

[9] In Lee and Kyoochun Lee. "The Internet of Things (IoT): Applications, investments, and challenges for enterprises". In: *Business Horizons* 58.4 (2015), pp. 431–440.

[10] Thiago Teixeira et al. "Service oriented middleware for the internet of things: a perspective". In: *Towards a Service-Based Internet* (2011), pp. 220–229.

[11] Soma Bandyopadhyay et al. "Role of middleware for internet of things: A study". In: *International Journal of Computer Science and Engineering Survey* 2.3 (2011), pp. 94–105.

[12] Soma Bandyopadhyay et al. "A survey of middleware for internet of things". In: *Recent trends in wireless and mobile networks*. Springer, 2011, pp. 288–296.

[13] Rushan Arshad et al. "Green IoT: An investigation on energy saving practices for 2020 and beyond". In: *IEEE Access* 5 (2017), pp. 15667–15681.

[14] André Glória, Francisco Cercas, and Nuno Souto. "Comparison of Communication Protocols for Low Cost Internet of Things Devices". In: *2017 South Eastern European Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*. IEEE. 2017, pp. 1–6.

[15] Frédéric Leens. "An Introduction to I2C and SPI Protocols". In: *IEEE Instrumentation & Measurement Magazine* 12.1 (2009), pp. 8–13.

[16] Jerad Lewis. "Common Inter-IC Digital Interfaces for Audio Data Transfer". In: *EDN-Electronic Design News* 57.16 (2012), p. 46.

[17] Nahit Pawar, Thomas Bourgeau, and Hakima Chaouchi. "Study of IoT Architecture and Application Invariant Functionalities". In: *IFIP/IEEE International Symposium on Integrated Network Management (IM 2021)*. 2021.

[18] Nahit Pawar, Thomas Bourgeau, and Hakima Chaouchi. "PrIoT Demo: Example of Invariant Functionalities". In: *IFIP/IEEE International Symposium on Integrated Network Management (IM 2021)*. 2021.

[19] Nahit Pawar, Thomas Bourgeau, and Hakima Chaouchi. "PrIoT: Prototyping the Internet of Things". In: *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE. 2018, pp. 216–223.

[20] Nahit Pawar, Thomas Bourgeau, and Hakima Chaouchi. "R-Bus: a resource bus for modular system design". In: *Proceedings of the 10th International Conference on the Internet of Things*. 2020, pp. 1–7.

[21] Nahit Pawar, Thomas Bourgeau, and Hakima Chaouchi. "Poster: R-Bus-A Resource Bus for Modular System Design". In: *Proceedings of the 2020 International Conference on Embedded Wireless Systems and Networks on Proceedings of the 2020 International Conference on Embedded Wireless Systems and Networks*. 2020, pp. 168–169.

[22] Nahit Pawar, Thomas Bourgeau, and Hakima Chaouchi. "Power Gating and Its Application in Wake-Up Radio." In: *EWSN*. 2020, pp. 218–223.

[23] Nahit Pawar, Thomas Bourgeau, and Hakima Chaouchi. "Low-cost, Low-power Testbed for Establishing Network of LoRaWAN Nodes." In: *EWSN*. 2020, pp. 192–194.

[24] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. "Interoperability in Internet of Things: Taxonomies and Open Challenges". In: *Mobile Networks and Applications* 24.3 (2019), pp. 796–809.

[25] Carsten Bormann, Mehmet Ersue, and Ari Keranen. "Terminology for constrained-node networks". In: *Internet Engineering Task Force (IETF): Fremont, CA, USA* (2014), pp. 2070–1721.

[26] Anuj Sehgal et al. "Management of resource constrained devices in the internet of things". In: *IEEE Communications Magazine* 50.12 (2012), pp. 144–149.

[27] Behrouz A Forouzan and Sophia Chung Fegan. *TCP/IP Protocol Suite*. McGraw-Hill Higher Education, 2010.

[28] Ed. S. Farrell. "Low-Power Wide Area Network (LPWAN) Overview". In: *Internet Engineering Task Force (IETF)* (2018).

[29] Nandakishore Kushalnagar, Gabriel Montenegro, Christian Schumacher, et al. "IPv6 over low-power wireless personal area networks (6LoWPANs): overview, assumptions, problem statement, and goals". In: (2007).

[30] Andrew S Tanenbaum, David Wetherall, et al. *Computer networks*. 1996.

[31] Shree Krishna Sharma et al. "Physical Layer Aspects of Wireless IoT". In: *2016 international symposium on wireless communication systems (ISWCS)*. IEEE. 2016, pp. 304–308.

[32] Harsh Tataria et al. "6G wireless systems: Vision, requirements, challenges, insights, and opportunities". In: *Proceedings of the IEEE* (2021).

[33] Godfrey Anuga Akpakwu et al. "A survey on 5G networks for the Internet of Things: Communication technologies and challenges". In: *IEEE access* 6 (2017), pp. 3619–3647.

[34] Kushalnagar N, Montenegro G, and Schumacher C. *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals*. https://tools.ietf.org/html/rfc4919. 2007.

[35] Josiah Balota, Colin Pattinson, and ah-lian Kor. "Wireless Personal Area Networks: A Survey of Low- Rate and Low-Power Network Technologies". In: Dec. 2017. DOI: 10.6084/m9.figshare.10315448.v2.

[36] Vasileios Karagiannis et al. "A survey on application layer protocols for the internet of things". In: *Transaction on IoT and Cloud computing* 3.1 (2015), pp. 11–17.

[37] Roy Fielding et al. *RFC2616: Hypertext Transfer Protocol–HTTP/1.1*. https://dl.acm.org/doi/pdf/10.17487/RFC2616. 1999.

[38] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The constrained application protocol (CoAP), IETF Internet-Draft 08*. http://tools.ietf.org/html/draft-ietf-core-coap-08. (last accessed : 01-01-2021). 2014.

[39] *MQTT - Message Queuing Telemetry Transport*. https://mqtt.org/. (last accessed : 01-01-2021).

[40] *XMPP - Extensible Messaging and Presence Protocol*. https://xmpp.org/. (last accessed : 01-01-2021).

[41] *AMQP - Advanced Message Queuing Protocol*. https://www.amqp.org/. (last accessed : 01-01-2021).

[42] Lavinia Nastase. "Security in the internet of things: A survey on application layer protocols". In: *2017 21st international conference on control systems and computer science (CSCS)*. IEEE. 2017, pp. 659–666.

[43] Bardia Safaei et al. "Reliability side-effects in Internet of Things application layer protocols". In: *2017 2nd International Conference on System Reliability and Safety (ICSRS)*. IEEE. 2017, pp. 207–212.

[44] Matthias Pohl et al. "Performance evaluation of application layer protocols for the internet-of-things". In: *2018 Sixth International Conference on Enterprise Systems (ES)*. IEEE. 2018, pp. 180–187.

[45] Srdjan Krco, Boris Pokric, and Francois Carrez. "Designing IoT Architecture(s): A European Perspective". In: *Internet of Things (WF-IoT), 2014 IEEE World Forum on*. IEEE. 2014, pp. 79–84.

[46] *IoT-A - Internet of Things Architecture*. https://cordis.europa.eu/project/rcn/95713_en.html.

[47] *IoT6 - Universal Integration of the Internet of Things through an IPv6-based Service Oriented Architecture Enabling Heterogeneous Components Interoperability*. https://www.iot6.eu/. (last accessed : 01-01-2021).

[48] Jorg Swetina et al. "Toward a standardized common M2M service layer platform: Introduction to oneM2M". In: *IEEE Wireless Communications* 21.3 (2014), pp. 20–26.

[49] Soohong Park. "OCF: A New Open IoT Consortium". In: *2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. IEEE. 2017, pp. 356–359.

[50] *IoTivity*. www.iotivity.org.

[51] *Thread Group*. https://www.threadgroup.org.

[52] *LoRa Alliance*. https://lora-alliance.org/.

[53] *Zigbee Alliance*. https://zigbeealliance.org/. (last accessed : 01-01-2021).

[54] *Z-Wave Alliance*. https://z-wavealliance.org/.

[55] *Wi-SUN Alliance : Wireless Smart Ubiquitous Network*. https://wi-sun.org/.

[56] *NFC Forum*. https://nfc-forum.org/.

[57] Zhengguo Sheng et al. "A survey on the ietf protocol suite for the internet of things: Standards, challenges, and opportunities". In: *IEEE Wireless Communications* 20.6 (2013), pp. 91–98.

[58] Miao Wu et al. "Research on the architecture of Internet of Things". In: *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*. Vol. 5. IEEE. 2010, pp. V5–484.

[59] Alexander Gluhak et al. "A survey on facilities for experimental internet of things research". In: *IEEE Communications Magazine* 49.11 (2011).

[60] Miguel Angel López Peña and Isabel Muñoz Fernández. "SAT-IoT: An Architectural Model for a High-Performance Fog/Edge/Cloud IoT Platform". In: *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*. IEEE. 2019, pp. 633–638.

[61] Prasant Misra, Yogesh Simmhan, and Jay Warrior. "Towards a practical architecture for the next generation internet of things". In: *arXiv preprint arXiv:1502.00797* (2015).

[62] Kumar Yelamarthi, Md Sayedul Aman, and Ahmed Abdelgawad. "An Application-Driven Modular IoT Architecture". In: *Wireless Communications and Mobile Computing* 2017 (2017).

[63] Takeshi Yashiro et al. "An Internet of Things (IoT) Architecture for Embedded Appliances". In: *2013 IEEE Region 10 Humanitarian Technology Conference*. IEEE. 2013, pp. 314–319.

[64] Julien Mineraud et al. "A gap analysis of Internet-of-Things platforms". In: *Computer Communications* 89 (2016), pp. 5–16.

[65] Antonios Pliatsios, Christos Goumopoulos, and Konstantinos Kotis. "Interoperability in IoT: A Vital Key Factor to Create the "Social Network" of Things." In: *The Thirteenth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies UBICOMM*. 2019, pp. 63–69.

[66] Johannes Thönes. "Microservices". In: *IEEE software* 32.1 (2015), pp. 116–116.

[67] Partha Pratim Ray. "A survey of IoT cloud platforms". In: *Future Computing and Informatics Journal* 1.1-2 (2016), pp. 35–46.

[68] *Arduino*. https://www.arduino.cc/. (last accessed : 01-01-2021).

[69] *ARM Mbed*. http://energia.nu. (last accessed : 01-01-2021).

[70] *TI Launchpad Energia*. http://energia.nu. (last accessed : 01-01-2021).

[71] Emmanuel Baccelli et al. "Survey of Operating Systems for the IoT Environment". In: *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on* (2013), pp. 79–80.

[72] *Zephyr Project*. www.zephyrproject.org.

[73] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. "Contiki-a lightweight and flexible operating system for tiny networked sensors". In: *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE. 2004, pp. 455–462.

[74] Philip Levis et al. "TinyOS: An operating system for sensor networks". In: *Ambient intelligence* 35 (2005), pp. 115–148.

[75] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[76] ISO/IEC. *ISO/IEC 2382:2015 Information Technology — Vocabulary*. https://standards.iso.org/ittf/PubliclyAvailableStandards/index.html. (last accessed : 01-01-2021). 2015.

[77] Mahmoud Elkhodr, Seyed Shahrestani, and Hon Cheung. "The Internet of Things: New Interoperability, Management and Security Challenges". In: *arXiv preprint arXiv:1604.04824* (2016).

[78] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. "RDF and Property Graphs Interoperability: Status and Issues." In: *AMW*. 2019.

[79] W3C - World Wide Web Consortium. *Semantic Integration and Interoperability Using RDF and OWL*. https://www.w3.org/2001/sw/BestPractices/OEP/SemInt/. (last accessed : 01-01-2021). 2005.

[80] Paul Murdock et al. *Semantic interoperability for the Web of Things*. Research Report. Dépt. Réseaux et Service Multimédia Mobiles (Institut Mines-Télécom-Télécom SudParis) ; Services répartis, Architectures, MOdélisation, Validation, Administration des Réseaux (Institut Mines-Télécom-Télécom SudParis-CNRS) ; TNO [Netherlands] (.) ; National Institute of Standards and Technology [Gaithersburg] (Agency of the U.S. Department of Commerce) ; British Telecom Research &

Technology (British Telecom) ; Intel corporation [USA] (Intel corporation) ; Krypton Brothers (.) ; EURECOM [Sophia Antipolis] (Institut Mines-Télécom) ; Fraunhofer-Institut für Offene Kommunikationssysteme (FOKUS Fraunhofer) ; LG Group (.) ; Rockwell automation (.) ; Huawei Technologies [Nanjing] (Huawei Technologies Co. Ltd.) ; Ericsson Research (Ericsson) ; Insight Centre for Data Analytics [Galway] (National University of Ireland Galway (NUIG)) ; Network Research Division, NEC Laboratories Europe (NEC Europe Ltd.) ; Sensinov (.) ; Huawei Technologies [San Francisco] (Huawei Technologies Co. Ltd.) ; Department of Computer Science [Binghamton] (Binghamton University) ; Honeywell Process Solutions (Honeywell International Inc.) ; Senslytics (Senslytics Corporation) ; Telecom Orange (.) ; Laboratoire d'analyse et d'architecture des systèmes (CNRS (UPR8001)-Université Toulouse 3-INPT-Institut National des Sciences Appliquées de Toulouse) ; OpenDOF (Panasonic) ; Deutsche Telekom Laboratories (Deutsche Telekom) ; Schneider Electric (.) ; Nokia Bell Labs [Paris Saclay] (Nokia) ; Institut de Recherche en Informatique de Toulouse (CNRS : UMR5505; INPT de Toulouse; Universités de Toulouse I, II et III) ; FESTO (.) ; Landis+Gyr (Toshiba) ; IoTecha (.) ; Comcast (.) ; Orange Labs [Issy les Moulineaux] (France Télécom) ; InterDigital Communications (.) ; World Wide Web Consortium (.), Aug. 2016, p. 18. URL: https://hal.archives-ouvertes.fr/hal-01362033.

[81] George Hatzivasilis et al. "The Interoperability of Things: Interoperable solutions as an enabler for IoT and Web 3.0". In: *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. IEEE. 2018, pp. 1–7.

[82] *Web of Things Working Group*. https://www.w3.org/WoT/WG/. Accessed on 01 January 2021.

[83] W3C. *Semantic Sensor Network (SSN) Ontology*. https://www.w3.org/TR/vocab-ssn/. (last accessed : 01-01-2021). 2017.

[84] Thinagaran Perumal et al. "Interoperability for smart home environment using web services". In: *International Journal of Smart Home* 2.4 (2008), pp. 1–16.

[85] Oladayo Bello, Sherali Zeadally, and Mohamad Badra. "Network layer inter-operation of Device-to-Device communication technologies in Internet of Things (IoT)". In: *Ad Hoc Networks* 57 (2017), pp. 52–62.

[86] Arne Bröring et al. "Enabling IoT ecosystems through platform interoperability". In: *IEEE software* 34.1 (2017), pp. 54–61.

[87] *oneM2M*. www.onem2m.org.

[88] *Open Connectivity Foundation (OCF)*. https://openconnectivity.org.

[89] OpenMTC. *A reference implementation of oneM2M*. https://www.openmtc.org/. (last accessed : 01-01-2021). 2004.

[90] OS-IoT. *Open Source Internet of Things - IoT device side library of oneM2M standard*. https://os-iot.org/. (last accessed : 01-01-2021). 2004.

[91] IoTivity. *IoTivity-Lite - A light-weight implementation of the OCF 1.3 spec*. https://iotivity.org/about-iotivity-lite. (last accessed : 01-01-2021). 2004.

[92] ETSI. *ETSI-M2M: Internet of Things and Machine to Machine Solutions*. https://www.etsi.org/technologies/internet-of-things. (last accessed : 01-01-2021). 2004.

[93] Mohammad Abdur Razzaque et al. "Middleware for Internet of Things: A survey." In: *IEEE Internet of Things Journal* 3.1 (2016), pp. 70–95.

[94] *Wiring*. www.arduino.cc.

[95] Per Persson and Ola Angelsmark. "Calvin–merging cloud and iot". In: *Procedia Computer Science* 52 (2015), pp. 210–217.

[96] *IFTTT - A Web-based Service*. https://ifttt.com/. Accessed on 01 January 2021.

[97] *NoFlo - JavaScript implementation of Flow-Based Programming (FBP)*. https://noflojs.org/. Accessed on 01 January 2021.

[98] Sergios Soursos et al. "Towards the cross-domain interoperability of IoT platforms". In: *2016 European conference on networks and communications (EuCNC)*. IEEE. 2016, pp. 398–402.

[99] Michael Blackstock and Rodger Lea. "IoT interoperability: A hub-based approach". In: *2014 international conference on the internet of things (IOT)*. IEEE. 2014, pp. 79–84.

[100] Ioannis Chatzigiannakis et al. "True self-configuration for the IoT". In: *2012 3rd IEEE International Conference on the Internet of Things*. IEEE. 2012, pp. 9–15.

[101] OGC. *SensorML - Sensor Model Language*. https://www.ogc.org/standards/sensorml. (last accessed : 01-01-2021).

[102] *IEEE 1451.4 Transducer Electronic Data Sheets (TEDS)*. https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/tutorials/teds.pdf (last accessed on 17/09/2019).

[103] W3C. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. https://www.w3.org/Submission/SWRL/. (last accessed : 01-01-2021). 2004.

[104] Ivan Kravets. *PlatformIO*. http://platformio.org.

[105] Eclipse Foundation. *Eclipse IoT - Open Source for IoT*. https://iot.eclipse.org/. (last accessed : 01-01-2021).

[106] Eclipse Foundation. *Eclipse Edje - Hardware Abstraction Layer (HAL) for accessing hardware features*. https://projects.eclipse.org/projects/iot.edje. (last accessed : 01-01-2021).

[107] Eclipse Foundation. *Eclipse Paho - A light weight implementation of MQTT*. https://www.eclipse.org/paho/. (last accessed : 01-01-2021).

[108] Eclipse Foundation. *Eclipse Wakaama - OMA Lightweight M2M C implementation*. https://www.eclipse.org/wakaama/. (last accessed : 01-01-2021).

[109] Eclipse Foundation. *Eclipse Kura - Open source IoT Edge Framework based on Java/OSGi*. https://www.eclipse.org/kura/. (last accessed : 01-01-2021).

[110] Eclipse Foundation. *Eclipse Kapua - Modular IoT cloud platform to manage and integrate devices and their data*. https://www.eclipse.org/kapua/. (last accessed : 01-01-2021).

[111] *RIOT OS*. https://riot-os.org.

[112] Maurizio Gabbrielli et al. "A language-based approach for interoperability of iot platforms". In: *51st Hawaii International Conference on System Sciences (HICSS-51)* (2018).

[113] *Jolie Website*. https://www.jolie-lang.org, last accessed on 30/07/2019. Accessed: 2019-07-30.

[114] *Eclipse Mita - Programming Language of Internet of Things*. https://www.eclipse.org/mita/. Accessed on 01 January 2021.

[115] Ferry Pramudianto et al. "IoTLink: An Internet of Things Prototyping Toolkit". In: *Ubiquitous Intelligence and Computing, 2014 IEEE 11th Intl Conf on and IEEE 11th Intl Conf on and Autonomic and Trusted Computing, and IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UTC-ATC-ScalCom)*. IEEE. 2014, pp. 1–9.

[116] Dimitris Soukaras et al. "IoTSuite: a ToolSuite for prototyping internet of things applications". In: *The 4th International Workshop on Computing and Networking for Internet of Things (ComNet-IoT), co-located with 16th International Conference on Distributed Computing and Networking (ICDCN)*. 2015, p. 6.

[117] Zhengguo Sheng et al. "Lightweight management of resource-constrained sensor devices in internet of things". In: *IEEE internet of things journal* 2.5 (2015), pp. 402–411.

[118] William Stallings. "SNMP and SNMPv2: the infrastructure for network management". In: *IEEE Communications Magazine* 36.3 (1998), pp. 37–43.

[119] Houda Rachidi and Ahmed Karmouch. "A framework for self-configuring devices using TR-069". In: *2011 International Conference on Multimedia Computing and Systems*. IEEE. 2011, pp. 1–6.

[120] OMA. *OMA LWM2M - A lightweight device management protocol for IoT and M2M*. https://omaspecworks.org/what-is-oma-specworks/iot/lightweight-m2m-lwm2m/. (last accessed : 01-01-2021).

[121] Suhas Rao et al. "Implementing LWM2M in constrained IoT devices". In: *2015 IEEE Conference on Wireless Sensors (ICWiSe)*. IEEE. 2015, pp. 52–57.

[122] Peter Neumann et al. "Field device integration". In: *ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No. 01TH8597)*. Vol. 2. IEEE. 2001, pp. 63–68.

[123] Ed. R. Enns. *RFC4741 - NETCONF Configuration Protocol*. https://tools.ietf.org/html/rfc4919. 2006.

[124] Chun-Che Huang and Andrew Kusiak. "Modularity in Design of Products and Systems". In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 28.1 (1998), pp. 66–77.

[125] Karl Ulrich. "Fundamentals of Product Modularity". In: *Management of Design*. Springer, 1994, pp. 219–231.

[126] Tarek AlGeddawy and Hoda ElMaraghy. "Optimum granularity level of modular product design architecture". In: *CIRP Annals* 62.1 (2013), pp. 151–154.

[127] Karsten Schischke et al. "Modular products: Smartphone design from a circular economy perspective". In: *2016 Electronics Goes Green 2016+(EGG)*. IEEE. 2016, pp. 1–8.

[128] *PuzzlePhone : A Modular Smart Phone*. http://www.puzzlephone.com/ (last accessed on 27/09/2019).

[129] CP Kruger, AM Abu-Mahfouz, and SJ Isaac. "Modulo: A modular sensor network node optimised for research and product development". In: *2013 IST-Africa Conference & Exhibition*. IEEE. 2013, pp. 1–9.

[130] Nicholas Edmonds, Douglas Stark, and Jesse Davis. "Mass: modular architecture for sensor systems". In: *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.* IEEE. 2005, pp. 393–397.

[131] Jorge Portilla et al. "A modular architecture for nodes in wireless sensor networks." In: *J. UCS* 12.3 (2006), pp. 328–339.

[132] Carsten Buschmann and Dennis Pfisterer. "iSense: A modular hardware and software platform for wireless sensor networks". In: *6. Fachgespräch Sensornetzwerke* (2007), p. 15.

[133] EG Straser et al. "A modular, wireless network platform for monitoring structures". In: *Proceedings-SPIE The International Society for Optical Engineering*. Vol. 1. Citeseer. 1998, pp. 450–456.

[134] Mohieddine Benammar et al. "A modular IoT platform for real-time indoor air quality monitoring". In: *Sensors* 18.2 (2018), p. 581.

[135] Pablo Merino et al. "A Modular IoT Hardware Platform for Distributed and Secured Extreme Edge Computing". In: *Electronics* 9.3 (2020), p. 538.

[136] Yao Li. "Teaching embedded systems using a modular-approach microcontroller training kit". In: *World Transactions on Engineering and Technology Education* 6.1 (2007), p. 135.

[137] Konstantin Mikhaylov et al. "Extensible modular wireless sensor and actuator network and IoT platform with Plug&Play module connection". In: *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*. 2015, pp. 386–387.

[138] Wei-Ying Yi, Kwong-Sak Leung, and Yee Leung. "A modular plug-and-play sensor system for urban air pollution monitoring: design, implementation and evaluation". In: *Sensors* 18.1 (2018), p. 7.

[139] Zhaohua Wang, Bin Zhang, and Dabo Guan. "Take responsibility for electronic-waste disposal". In: *Nature News* 536.7614 (2016), p. 23.

[140] *M2.COM Platform*. http://www.m2com-standard.org/en-us (last accessed on 17/09/2019).

[141] *micro:bit*. https://microbit.org (last accessed on 17/09/2019).

[142] *Mikroe mikroBUS*. https://www.mikroe.com/mikrobus (last accessed on 17/09/2019).

[143] *Mikroe*. https://www.mikroe.com (last accessed on 17/09/2019).

[144] *Pmod Standard*. https://reference.digilentinc.com/reference/pmod/specification?redirect=1 (last accessed on 17/09/2019).

[145] *Digilent Inc.* https://store.digilentinc.com (last accessed on 17/09/2019).

[146] *Grove System*. http://wiki.seeedstudio.com/Grove_System (last accessed on 17/09/2019).

[147] *STM32 Nucleo Boards*. https://www.st.com/en/evaluation-tools/stm32-nucleo-boards.html (last accessed on 17/09/2019).

[148] *NXP LCPXpresso Boards*. https://www.nxp.com/design/development-boards/lpcxpresso-boards:LPCXPRESSO-BOARDS (last accessed on 17/09/2019).

[149] Raspberry Pi (RPi) Foundation. *RPi HATs*. https://github.com/raspberrypi/hats. 2020.

[150] Jayavardhana Gubbi et al. "Examples of Raspberry Pi usage in Internet of Things". In: *International Conference on Applied Internet and Information Technologies* (2016), pp. 112–119.

[151] David Gibson and Benjamin Herrenschmidt. "Device trees everywhere". In: *OzLabs, IBM Linux Technology Center* (2006).

[152] Neha Garg and Ritu Garg. "Energy harvesting in IoT devices: A survey". In: *2017 International Conference on Intelligent Sustainable Systems (ICISS)*. IEEE. 2017, pp. 127–131.

[153] Amit Sinha and Anantha Chandrakasan. "Dynamic power management in wireless sensor networks". In: *IEEE Design & Test of Computers* 18.2 (2001), pp. 62–74.

[154] Borja Martinez et al. "The Power of Models: Modeling Power Consumption for IoT Devices". In: *IEEE Sensors Journal* 15.10 (2015), pp. 5777–5789.

[155] Llués Casals et al. "Modeling the Energy Performance of LoRaWAN". In: *Sensors* 17.10 (2017), p. 2364.

[156] Toni Adame Vázquez et al. "HARE: Supporting Efficient Uplink Multihop Communications in Self-Organizing LPWANs". In: *Sensors* 18.1 (2018), p. 115.

[157] Vijay Raghunathan et al. "Design considerations for solar energy harvesting wireless embedded systems". In: *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.* IEEE. 2005, pp. 457–462.

[158] Ilker Demirkol, Cem Ersoy, and Ertan Onur. "Wake-up Receivers for Wireless Sensor Networks: Benefits and Challenges". In: *IEEE Wireless Communications* 16.4 (2009), pp. 88–96.

[159] Michele Magno et al. "WULoRa: An Energy Efficient IoT End-Node for Energy Harvesting and Heterogeneous Communication". In: *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association. 2017, pp. 1532–1537.

[160] Michele Magno et al. "Design, Implementation, and Performance Evaluation of a Flexible Low-Latency Nanowatt Wake-Up Radio Receiver". In: *IEEE Transactions on Industrial Informatics* 12.2 (2016), pp. 633–644.

[161] Kyriakos Georgiou, Samuel Xavier-de-Souza, and Kerstin Eder. "The IoT energy challenge: A software perspective". In: *IEEE Embedded Systems Letters* 10.3 (2017), pp. 53–56.

[162] Ovidiu Vermesan, Peter Friess, et al. *Internet of things-from research and innovation to market deployment*. Vol. 29. River publishers Aalborg, 2014.

[163] Juho Lee and Yongjun Kwak. "5G standard development: technology and roadmap". In: *Signal Processing for G* 5 (2016).

[164] *Mender: Over-the-air software updates for connected Linux devices*. https://mender.io. [Online; last accessed 25-Oct-2018].

[165] Paola Pierleoni et al. "Amazon, google and microsoft solutions for iot: Architectures and a performance comparison". In: *IEEE Access* 8 (2019), pp. 5455–5470.

[166] Pankesh Patel et al. "Towards application development for the internet of things". In: *Proceedings of the 8th Middleware Doctoral Symposium*. ACM. 2011, p. 5.

[167] Ryo Sugihara and Rajesh K Gupta. "Programming models for sensor networks: A survey". In: *ACM Transactions on Sensor Networks (TOSN)* 4.2 (2008), p. 8.

[168] Pankesh Patel et al. "Evaluating the Ease of Application Development for the Internet of Things". In: (2013).

[169] *IEEE IoT - IoT Scenarios*. iot.ieee.org/iot-scenarios.html.

[170] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.

[171] *Ago Control – Open Source Home Automation System*. https://www.agocontrol.com/. Accessed on 01 January 2021.

[172] *Calaos - Open Source Home Automation*. https://calaos.fr/en/. Accessed on 01 January 2021.

[173] *Domoticz - Home Automation System*. https://www.domoticz.com/. Accessed on 01 January 2021.

[174] *FHEM - Server for House Automation*. https://fhem.de/. Accessed on 01 January 2021.

[175] *Homebridge - Modern and Lightweight NodeJS Server*. https://homebridge.io/. Accessed on 01 January 2021.

[176] *Home Assistant - Awaken Your Home*. https://www.home-assistant.io/. Accessed on 01 January 2021.

[177] *Home Genie - Home Automation Server*. http://www.homegenie.it/. Accessed on 01 January 2021.

[178] *HomeSeer - Smart Home Systems For Every Need & Budget*. https://homeseer.com/. Accessed on 01 January 2021.

[179] *Homey - Make It Your Home*. https://homey.app/. Accessed on 01 January 2021.

[180] *HoMIDoM - Open Source Home Automation Software*. https://github.com/Homidom/HoMIDoM. Accessed on 01 January 2021.

[181] *Indigo Domotics - Advanced Mac-based Smart Home Hub*. https://www.indigodomo.com/. Accessed on 01 January 2021.

[182] *ioBroker - Open Source Automation Platform*. https://www.iobroker.net/. Accessed on 01 January 2021.

[183] *Jeedom - Innovative Home Automation*. https://www.jeedom.com/. Accessed on 01 January 2021.

[184] Michael Blackstock and Rodger Lea. "Toward a distributed data flow platform for the web of things (distributed node-red)". In: *Proceedings of the 5th International Workshop on Web of Things*. ACM. 2014, pp. 34–39.

[185] *MyController.org - The Open Source Controller*. https://www.mycontroller.org. Accessed on 01 January 2021.

[186] *MisterHouse - Open Source Home Automation Program*. http://misterhouse.sourceforge.net/. Accessed on 01 January 2021.

[187] *MyNodes.NET - Open Source .NET Controller*. https://github.com/derwish-pro/MyNodes.NET. Accessed on 01 January 2021.

[188] *MajorDoMo - Open Source for Smart Home DIY*. https://majordomohome.com/. Accessed on 01 January 2021.

[189] *OpenHAB - Open Home Automation Bus*. https://www.openhab.org/. Accessed on 01 January 2021.

[190] *PiDome - Open Source Home Automation Platform*. https://pidome.org/. Accessed on 01 January 2021.

[191] *pimatic - Home Automation Framework*. https://pimatic.org/. Accessed on 01 January 2021.

[192] *XTension - Mac Home Automation*. https://machomeautomation.com/doku.php. Accessed on 01 January 2021.

[193] *smarthomatic - A Secure and Extendable Open Source Home Automation System*. https://www.smarthomatic.org/. Accessed on 01 January 2021.

[194] *EventGhost - Home Automation Program*. http://www.eventghost.net/. Accessed on 01 January 2021.

[195] *Amazon AWS IoT Platform*. https://aws.amazon.com/iot/. Accessed on 01 January 2021.

[196] *Google Cloud IoT Platform*. https://cloud.google.com/solutions/iot. Accessed on 01 January 2021.

[197] *Microsoft Azure - IoT Cloud Platform*. https://azure.microsoft.com. Accessed on 01 January 2021.

[198] *IBM Watson IoT Platform*. https://www.ibm.com/cloud/watson-iot-platform. Accessed on 01 January 2021.

[199] *Oracle IoT Cloud Services*. https://docs.oracle.com/en/cloud/paas/iot-cloud/index.html. Accessed on 01 January 2021.

[200] *ThingSpeak - IoT Analytics Platform*. https://thingspeak.com/. Accessed on 01 January 2021.

[201] *A toolbox in the cloud for IoT developers*. https://developer.bosch-iot-suite.com/. Accessed on 22 October 2019.

[202] *OpenRemote - Open Source IoT Platform*. https://openremote.io/. Accessed on 01 January 2021.

[203] *CISCO IoT Control Center*. https://www.cisco.com/c/en/us/solutions/internet-of-things/iot-control-center.html. Accessed on 01 January 2021.

[204] *Kaa - Cloud to Edge IoT Solution*. https://www.kaaproject.org/. Accessed on 01 January 2021.

[205] *Predix - IIoT Platform*. https://www.ge.com/digital/iiot-platform. Accessed on 01 January 2021.

[206] *IEEE IoT - IoT Scenarios, Pollution Monitoring 2*. iot.ieee.org/iot-scenarios.html?prp=oc-6f436520-882f-4c4f-9224-54d9f25413bb.

[207] *IEEE IoT - IoT Scenarios, GreenIQ*. iot.ieee.org/iot-scenarios.html?prp=24.

[208] Cristian González Garcıéa et al. "Midgar: Domain-specific language to generate smart objects for an internet of things platform". In: *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2014 Eighth International Conference on*. IEEE. 2014, pp. 352–357.

[209] Cristian González Garcıéa et al. "Midgar: Generation of heterogeneous objects interconnecting applications. A Domain Specific Language proposal for Internet of Things scenarios". In: *Computer Networks* 64 (2014), pp. 143–158.

[210] Adnan Salihbegovic et al. "Design of a Domain Specific Language and IDE for Internet of Things Applications". In: *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*. IEEE. 2015, pp. 996–1001.

[211] Manfred Sneps-Sneppe and Dmitry Namiot. "On web-based domain-specific language for Internet of Things". In: *2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. IEEE. 2015, pp. 287–292.

[212] Behailu Negash et al. "DoS-IL: A Domain Specific Internet of Things Language for Resource Constrained Devices." In: *ANT/SEIT*. 2017, pp. 416–423.

[213] Edel Sherratt et al. "SDL - The IoT Language". In: *International SDL Forum*. Springer. 2015, pp. 27–41.

[214] Nicolas Harrand et al. "ThingML: A Language and Code Generation Framework for Heterogeneous Targets". In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 2016, pp. 125–135.

[215] *CMSIS-RTOS - Generic API Layer for OS*. https://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html. (last accessed : 01-01-2021).

[216] *CMSIS - Vendor Independent Hardware Abstraction Layer*. https://developer.arm.com/tools-and-software/embedded/cmsis. (last accessed : 01-01-2021).

[217] *Blockly*. https://developers.google.com/blockly.

[218] Jan Bauwens et al. "Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles". In: *IEEE Communications Magazine* 58.2 (2020), pp. 35–41.

[219] *Jinja - Designer-friendly Templating Language*. https://jinja.palletsprojects.com/en/2.11.x/. (last accessed : 01-01-2021).

[220] *Lark - A parsing toolkit for Python*. https://github.com/lark-parser/lark. (last accessed : 01-01-2021).

[221] *PCI Express Electromechanical Specification Rev 1.1*. https://pcisig.com/specifications (last accessed on 06/08/2019).

[222] Oliver Hahm et al. "Operating systems for low-end devices in the internet of things: a survey". In: *IEEE Internet of Things Journal* 3.5 (2015), pp. 720–734.

[223] Thomas Bourgeau, Nahit Pawar, and Hakima Chaouchi. *R-Bus : A Resource Bus for Modular System Design*. www.rbus.io. 2020.

[224] Brice Morin, Nicolas Harrand, and Franck Fleurey. "Model-based Software Engineering to Tame the IoT Jungle". In: *IEEE Software* 34.1 (2017), pp. 30–36.

[225] *PCI Special Interest Group*. https://pcisig.com (last accessed on 06/08/2019).

[226] Ferran Adelantado et al. "Understanding the Limits of LoRaWAN". In: *IEEE Communications magazine* 55.9 (2017), pp. 34–40.

[227] Preeti Ranjan Panda et al. *Power-efficient System Design*. Springer Science & Business Media, 2010.

[228] Farzan Fallah and Massoud Pedram. "Standby and Active Leakage Current Control and Minimization in CMOS VLSI Circuits". In: *IEICE transactions on electronics* 88.4 (2005), pp. 509–519.

[229] David Charles Harrison et al. "Busting myths of energy models for wireless sensor networks". In: *Electronics Letters* 52.16 (2016), pp. 1412–1414.

[230] *MAX17223 Nano-Power Boost Converter, Datasheet*. https://datasheets.maximintegrated.com/en/ds/MAX17220-MAX17225.pdf. Accessed on 22 October 2019.

[231] *TPL5111 Nano-Power System Timer, Datasheet*. http://www.ti.com/lit/ds/symlink/tpl5111.pdf. Accessed on 22 October 2019.

[232] *WiMOD iM880B Long Range Radio Module*. https://www.wireless-solutions.de/downloads/Radio-Modules/iM880B/General_Information/iM880B_Datasheet_V1_6.pdf. Accessed on 22 October 2019.

[233] *NetBlocks XRange SX1272 LoRaNode*. https://www.netblocks.eu/xrange-sx1272-lora-datasheet/. Accessed on 22 October 2019.

[234] *MultiTech mDot Long Range LoRa Module*. https://www.multitech.com/documents/publications/data-sheets/86002171.pdf. Accessed on 22 October 2019.

[235] *Semtech SX1272 Transceiver*. https://www.semtech.com/products/wireless-rf/lora-transceivers/sx1272. Accessed on 22 October 2019.

[236] *WiMOD iM222A IEEE 802.15.4 Radio Module*. https://www.wireless-solutions.de/downloads/Radio-Modules/iM880B/General_Information/iM880B_Datasheet_V1_6.pdf. Accessed on 22 October 2019.

[237] *STMicroelectronics : STM32F411 MCU*. https://www.st.com/resource/en/datasheet/stm32f411ce.pdf. Accessed on 22 October 2019.

[238] *STMicroelectronics : STM32L151 MCU*. https://www.st.com/resource/en/datasheet/stm32l151qc.pdf. Accessed on 22 October 2019.

[239] *Texas Instruments : CC2530 System-on-Chip*. http://www.ti.com/lit/ds/symlink/cc2530.pdf. Accessed on 22 October 2019.

[240] Daniele Pizzolli et al. "Cloud4iot: A heterogeneous, distributed and autonomic cloud platform for the iot". In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2016, pp. 476–479.

[241] Corentin Dupont, Raffaele Giaffreda, and Luca Capra. "Edge computing in IoT context: Horizontal and vertical Linux container migration". In: *2017 Global Internet of Things Summit (GIoTS)*. IEEE. 2017, pp. 1–4.

[242] *IEEE Global Identifier EUI-48 & EUI-64*. https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/tutorials/eui.pdf (last accessed on 17/09/2019).

Appendix A

List of Publications

A.1 International Conferences

[1] Nahit Pawar, Thomas Bourgeau and Hakima Chaouchi. *R-Bus: A Resource Bus for Modular System Design.* The 10^{th} International Conference on Internet of Things (IoT2020), 6-9 October 2020, Malmö, Sweden.

[2] Nahit Pawar, Thomas Bourgeau and Hakima Chaouchi. *Power Gating and Its Application in Wake-Up Radio.* International Conference on Embedded Wireless Systems and Networks (EWSN), 17-19 February 2020, Lyon, France.

[3] Nahit Pawar, Thomas Bourgeau and Hakima Chaouchi. *PrIoT: Prototyping the Internet of Things.* In 2018 IEEE 6^{th} International Conference on Future Internet of Things and Cloud (FiCloud), 6-8 August 2018, Barcelona, Spain.

A.2 Posters

[1] Nahit Pawar, Thomas Bourgeau and Hakima Chaouchi. *Study of IoT Architecture and Application Invariant Functionalities* IFIP/IEEE International Symposium on Integrated Network Management (IM 2021), 17-21 May 2021, Bordeaux, France.

[2] Nahit Pawar, Thomas Bourgeau and Hakima Chaouchi. *R-Bus - A Resource Bus for Modular System Design* International Conference on Embedded Wireless Systems and Networks (EWSN), 17-19 February 2020, Lyon, France.

A.3 Demos

[1] Nahit Pawar, Thomas Bourgeau and Hakima Chaouchi. *PrIoT Demo: Example of Invariant Functionalities.* The IEEE/IFIP International Symposium on Integrated Network Management 2021 (IM 2021), 17-21 May 2021, Bordeaux, France.

[2] Nahit Pawar, Thomas Bourgeau and Hakima Chaouchi. *Low-cost, Low-power Testbed for Establishing Network of LoRaWAN Nodes.* International Conference on Embedded Wireless Systems and Networks (EWSN), 17-19 February 2020, Lyon, France.

Appendix B

Code Listings

B.1 PrIoT Configuration DSL Grammar

```
start: cfig+

cfig: select_instruction | config_instruction

?select_instruction: "select" cname"."cname"."cname "as"
   cname                    -> select_resource
?config_instruction: "config" cname"."cname parameter
                               -> config_resource

?parameter: dict
     | list
     | string
     | INT                  -> integer
     | NUMBER               -> number
     | "true"               -> true
     | "false"              -> false
     | "null"               -> null

list : "[" [parameter ("," parameter)*] "]"
dict : "{" [pair ("," pair)*] "}"
pair : string ":" parameter

string : ESCAPED_STRING
cname : CNAME

COMMENT: "//" /[^\n]/*

```

```
%import common.LETTER
%import common.DIGIT
%import common.CNAME
%import common.WORD
%import common.ESCAPED_STRING
%import common.NUMBER
%import common.INT
%import common.WS
%import common.WS_INLINE
%ignore WS
%ignore WS_INLINE
%ignore COMMENT
```

LISTING B.1: PrIoT Configuration DSL Grammar

ECOLE DOCTORALE

Titre : Conception d'une architecture complete pour l'interopérabilité des objets connectes heterogenes et des services de l'Internet des Objets

Mots clés : Objets connectés, Hétérogénéité, Interopérabilité

Résumé : L'Internet des objets (IoT) combine de nombreuses technologies et s'est étendu à des domaines d'application divers et multidisciplinaires. Chaque domaine a son propre ensemble d'exigences d'application en termes de matériel, de communication, de logiciel, de source d'énergie, etc. Cela empêche l'utilisation de modèles de programmation conventionnels de l'informatique distribuée qui suppose que les systèmes sont toujours connectés, disposant de ressources de calcul abondantes et d'accès à l'énergie électrique infinie. De plus, l'IoT englobe une large gamme d'appareils IoT embarqués hétérogènes (unités de traitement, capteurs, actionneurs, émetteurs-récepteurs, etc.) fournis par divers fabricants, chacun avec une architecture d'appareil différente, par conséquent, les logiciels d'application développés pour ces appareils ne sont pas compatibles avec chacun. autre. Cette hétérogénéité des appareils pose de sérieux problèmes pour l'interopérabilité des appareils et également pour les outils de développement IoT harmonisés sur une large gamme d'appareils IoT hétérogènes. Un défi important non seulement pour les experts du domaine, mais aussi pour les professionnels est de réaliser une preuve de concept (PoC) lors de l'industrialisation des services IoT, ce qui implique - le développement, le déploiement et la maintenance de services d'application IoT de bout en bout nécessitant différents types et niveaux d'expertise.

La principale contribution de cette thèse est d'introduire un nouveau framework nommé PrIoT (Prototyping Internet of Things) qui permet une programmation simple et rapide des appareils IoT, de la conception au déploiement, qui gère mieux l'hétérogénéité de l'architecture des appareils IoT. Plus spécifiquement, le framework PrIoT est basé sur le concept que les applications IoT possèdent diverses caractéristiques invariantes que nous avons étudiées et rassemblées à partir de diverses architectures et applications IoT présentées dans la littérature. Nous avons ensuite développé un langage de programmation minimaliste de haut niveau et des API pour montrer la composabilité facile de nos fonctionnalités invariantes dans le développement d'applications IoT. Nous validons notre framework PrIoT à travers l'implémentation de référence et également le développement d'implémentation prototype de différents scénarios IoT en utilisant notre framework et en le comparant à diverses solutions existantes.

D'un point de vue matériel, afin de mieux contrôler l'hétérogénéité des appareils, nous proposons deux nouveaux systèmes modulaires nommés R-Bus et P-Bus pour concevoir des systèmes embarqués sous la forme d'un ensemble de modules matériels pouvant être montés et démontés en fonction des besoins des applications IoT. Cela résout l'hétérogénéité de l'interface des appareils et prend en charge diverses classes d'appareils de contrainte, ainsi qu'une configuration système avancée et des fonctionnalités plug-and-play pour faciliter le prototypage matériel IoT. Nous validons notre système modulaire à l'aide de deux mesures - l'adéquation et le taux de couverture qui mesurent la compatibilité des systèmes modulaires embarqués par rapport aux unités de traitement. Nous avons utilisé ces métriques pour comparer notre solution avec les systèmes modulaires existants.

Cette approche complète notre proposition de cadre PrIoT car elle offre une nouvelle façon de créer des prototypes d'applications IoT de bout en bout avec une flexibilité à la fois matérielle et logicielle des appareils IoT.

En fait, notre objectif est de permettre un prototypage rapide de l'IoT de bout en bout en mettant en œuvre une couche d'abstraction de haut niveau qui cache les détails de diverses technologies sous-jacentes à l'IoT et en mettant en œuvre des systèmes modulaires pour une intégration flexible des appareils ciblés pour la conception de systèmes IoT. Ce travail a permis de faire un pas en avant dans le contrôle de l'hétérogénéité des appareils du point de vue matériel et logiciel, mais il manque toujours la normalisation au sein de la communauté IoT pour favoriser son développement continu.

Title : On Interoperability and Network Architecture Bottom-Up Heterogeneity Control in Internet of Things

Keywords : Internet of Things, Heterogeneity, Interoperability

Abstract : Internet of Things (IoT) combines many technologies and it has spanned across diverse and multidisciplinary application domains. Each domain has its own set of application requirements in terms of hardware, communication, software, source of energy, etc. This inhibits the use of conventional *programming models* of distributed computing which assumes that the systems are always connected, having abundant computational resources and access to infinite electric energy. Additionally, IoT encompasses a wide range of heterogeneous embedded IoT devices (processing units, sensors, actuators, transceivers, etc.) provided by various manufacturers each with different device architecture, as a result the application software developed for these devices are not compatible with each other. This device heterogeneity poses serious problems for device interoperability and also for harmonized IoT development tools over a wide range of heterogeneous IoT devices. An important challenge not only for domain experts but also for professionals is to realize proof-of-concept (PoC) during industrialization of IoT services, that involves - development, deployment and maintenance of end-to-end IoT application services requiring different types and levels of expertise.

The main contribution of this thesis is to introduce a new framework named PrIoT (Prototyping Internet of Things) that allows easy and rapid IoT device programming from design to deployment that better handles the heterogeneity of IoT device architecture. More specifically, the PrIoT framework is based on the concept that IoT applications possess various invariant characteristics that we studied and gathered from various IoT architectures and applications presented in the literature. We then developed a minimalist high level programming language and APIs to show the easy composability of our invariant functionalities in the development of IoT applications. We validate our PrIoT framework through reference implementation and also the development of prototype implementation of various IoT scenarios using our framework and comparing it against various existing solutions.

From a hardware perspective, in order to control better the device heterogeneity, we propose two novel modular systems named R-Bus and P-Bus for designing embedded systems as a set of hardware modules that can be mounted and dismounted based on the IoT applications needs. This resolves device interface heterogeneity and accommodates various classes of constraint devices along with advance system configuration and plug-&-play functionalities to ease IoT hardware prototyping. We validate our modular system using two metrics - *suitability* and *coverage ratio* that measure the compatibility of embedded modular systems with respect to processing units. We used these metrics to compare our solution with existing modular systems.

This approach complements our proposed PrIoT framework as it offers a new way to build end-to-end IoT application prototypes with flexibility in both hardware and software of the IoT devices.

In fact, our objective is to enable rapid end-to-end IoT prototyping by implementing a high level abstraction layer that hides the details of various technologies underlying IoT and implementing modular systems for flexible device integration targeted for IoT system design. This work has provided a step forward in controlling device heterogeneity from both hardware and software perspective, but it still lacks the standardization among the IoT community to foster its continuous development.

Institut Polytechnique de Paris
91120 Palaiseau, France