A Lightweight and Chip-Level Reconfigurable Architecture for Next-Generation IoT End Devices

Chong Zhang, Songfan Li, *Member, IEEE*, Yihang Song, Qianhe Meng, Li Lu, *Member, IEEE*, Hongzi Zhu, *Senior Member, IEEE*, and Xin Wang

***Abstract*—The rapid development of IoT applications calls for re-configurable IoT devices that can easily extend new functionality on demand. However, in the current architecture, updating chip functions on the end device is highly coupled with the local microprocessor in both hardware and software aspects, leading to inadequate flexibility. In this paper, we propose LEGO, a lightweight architecture with chip-level plug-and-play capabilities for IoT end devices. To achieve this, we first decoupling the control over heterogeneous chips from end devices to the gateway, and design a novel Unified Chip Description Language (UCDL) to access various types of functional chips uniformly. To supporting chips plug-and-play, we design a novel signal converting circuit on end devices to generate all required underlying signals for chip control. We also design a layered instruction orchestrator and hierarchical scheduler to minimize transmission overhead. The results show that our LEGO system can respond to chips plug-and-play within 0.13 seconds, and the lightweight architecture could reduce 49%∼61% of power consumption in practical scenarios compared with traditional IoT end devices that are controlled by a microprocessor. The lightweight and easy-to-deploy features of LEGO makes it helpful to reduce deployment cost, thus conducive to accelerating large-scale applications.**

***Index Terms*—Reconfigurable architecture, chip level plug-and-play, description language.**

I. Introduction

INTERNET of Things (IoT) end devices, usually equipped with a set of functional chips (*e.g.*, sensor chips, memory chips), sensing the environment and upload data to a gateway to provide data support for various IoT applications, such as intelligent transportation, modern agriculture and smart healthcare [1], [2], [3]. To meet the ever-increasing application requirements, instead of deploying new end devices, it would be fabulous if new functional chips could start to work after being plugged into existing end devices, minimizing the customization difficulty and cost.

In the current IoT application architecture, however, deploying a new chip on an end device is tightly coupled with both hardware and software modifications. Specifically, different functional chips are highly heterogeneous in their logic and signal aspects, which have no unified standards. To fit the heterogeneity of deployed chips, developers design the current device architecture as an independent microcomputer, and make hardware-software integrated developments around the device's microprocessor (MCU) to couple with the on-board chip in both control logic and signal feature aspects. Once a new chip needs to be deployed for new functions, developers often need to re-program the device's MCU, re-build hardware connections, and even create a new printed circuit board (PCB), which is tedious for developers and technically infeasible for most ordinary end-users.

An ideal re-configurable end device design should satisfy three requirements as follows: 1) Low difficulty: such devices should be easily configured by simply plug-and-play chips, making IoT system customization even available for unskilled end users; 2) good versatility: such a device should support most Commercial Off-The-Shelf (COTS) chips or modules, without concern with their heterogeneity in use; 3) low deployment cost: such a device should have a simple hardware architecture with low cost and power consumption, suitable for large-scale deployment.

To reduce the difficulty of customizing functional chips on end devices, a rich set of schemes has been proposed. One branch focuses on the rapid development of end devices by either simplifying hardware design [4], [5] or software development [6], [7], *e.g.*, employing over-the-air (OTA) technology [8], [9], [10] to facilitate device programming. However, in these approaches, developers still need to complete professional system developing in advance to fit the heterogeneity of deployed chips. Another branch concentrates on the design of plug-and-play modules, aiming to shield the heterogeneity of underlying chips by unified hardware packaging, *e.g.*, designing modules [11], [12] compatible with the IEEE 1451.4 plug-and-play standard [13]. However, packaging heterogeneous chips often involves mass hardware-software integrated

Manuscript received 22 June 2023; revised 19 October 2023; accepted 3 December 2023. Date of publication 14 December 2023; date of current version 12 February 2024. This work was supported in part by the National Natural Science Foundation of China under Grant U21A20462 and in part by the Natural Science Starting Project of SWPU under Grant 2023QHZ002. An earlier version of this paper was presented at the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2023 [DOI: 10.1145/3582016.3582050]. Recommended for acceptance by T. Chantem. *(Corresponding authors: Li Lu; Hongzi Zhu.)*

Chong Zhang is with the Computer Science, Southwest Petroleum University (SWPU), Chengdu 610500, China, and also with the University of Electronic Science and Technology of China (UESTC), Chengdu 611731, China (e-mail: zhangchong92@swpu.edu.cn).

Songfan Li, Yihang Song, Qianhe Meng, and Li Lu are with the School of Computer Science and Engineering, University of Electronic Science and Technology of China (UESTC), Chengdu 611731, China (e-mail: songfanli@std.uestc.edu.cn; songyihang@std.uestc.edu.cn; qianhe@std.uestc.edu.cn; luli2009@uestc.edu.cn).

Hongzi Zhu is with the Shanghai Jiao Tong University, Shanghai 200241, China (e-mail: hongzi@sjtu.edu.cn).

Xin Wang is with the Computer Science, Southwest Petroleum University (SWPU), Chengdu 610500, China (e-mail: xinwang@swpu.edu.cn).

Digital Object Identifier 10.1109/TC.2023.3343094

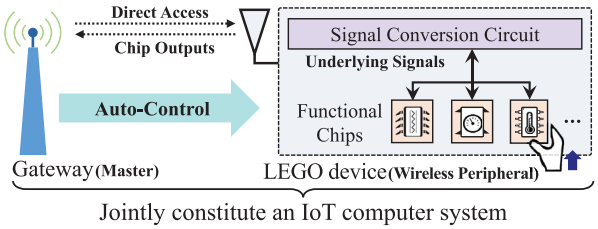


Fig. 1. LEGO empowers chip-level functionality plug-and-play on end devices by decoupling chip control logic to the gateway for unified access.

designs with adding extra components, leading to higher deployment costs. In summary, to the best of our knowledge, there is no successful solution that meets all aforementioned requirements for the ideal re-configurable end device.

In this paper, we propose LEGO, a novel computing architecture for chip-level re-configurable end devices with lightweight structure. As illustrated in Fig. 1, we decouple the control over heterogeneous chips from individual LEGO (IoT) devices to a centralized gateway for unified access. In this architecture, the gateway and end devices are no longer independent but work as a unified computer system, where the gateway handles all chips access and the end device serves as a "wireless peripheral" of the gateway, which supports unified plug-and-play for various heterogeneous chips via a simple circuit, which not need a local MCU. Hence, conventional designs like embedded developing are no longer needed, and the "processor-free" lightweight architecture is also helpful to reduce cost and power consumption, which envisioning cost-efficient and joyful IoT applications where building an end device resembles building LEGO blocks.

Challenges and contributions: There are three main challenges in the design of LEGO. First, as different functional chips are highly heterogeneous in neither control logic, signal timing and data format, it is hard for the system to dynamically generate the required control logic for various deployed chips uniformly. To tackle this challenge, we summarize chip control into six basic types of meta-operations and design a novel *Unified Chip Description Language* (UCDL) to construct the required control logic for various heterogeneous chips by the orchestration of meta-operations. To support chips plug-and-play, we build up a description library database with chip specifications for 100+ types of chips and design a novel coding logic to generate instructions on the gateway. When a newly plugged chip is detected, the gateway automatically reads its description file and issues the generated instructions to the target end device for chip control.

Second, it is challenging for LEGO devices to provide all the required underlying signals for the control of various heterogeneous onboard chips uniformly via a simple circuit without manual modifications. To deal with this challenge, we design a universal signal converting circuit as an interpreter between a LEGO gateway and various types of plug-in chips on LEGO devices. In general, the circuit carries out three main functions: 1) dynamically fit electrical connections for plugged chips, 2) automatically convert gateway instructions into corresponding pin signals with the required timing and formats for chip control, 3) receive chips outputs ($e.g.$, data, control signal), and respond key information to the gateway for control response. By this, closed-loop control for the plugged chips can be achieved.

Finally, decoupling the chip control logic all back to the gateway could effectively avoid manual modifications on LEGO devices when plug-and-play chips, but bring additional communication overheads. To address this issue, we have developed a novel three-layer orchestration logic for gateway instructions as well as a hierarchical scheduler on LEGO devices to handle the plugged chips more efficiently. By this, in downlink aspects, the gateway only needs to issue complete instructions once when the chip is plugged in, and mass subsequent tasks can be scheduled by a simple instruction; in uplink aspects, the scheduler handles chip output, and only uploads the key data required by IoT applications, minimizing communication overhead.

We have implemented a LEGO prototype and conducted comprehensive evaluations. The results show that LEGO can respond to chips plug-and-play within 0.13 seconds. The "processor-free" lightweight architecture could reduce 49%∼61% of overall power consumption compared with traditional IoT end devices that are controlled by a MCU. Besides, our UCDL is easy to use, allowing developers to create a new chip in less than 8 minutes on average. In summary, benefiting from the lightweight and chip-level plug-and-play features, LEGO is helpful in accelerating the large-scale deployments of IoT end devices for various applications. The main contributions are summarized as follows:

1) We propose a "processor-free" and chip-level re-configurable architecture for IoT end devices, called LEGO, which could facilitate the large-scale deployment of IoT end devices through easy deployment and lightweight features.
2) We design a unified chip description language with simple syntax to describe the feature of heterogeneous types of chips uniformly and share a description library database for public use;
3) We build up a novel signal interaction circuit on end device to dynamically convert gateway instructions into required underlying pin signals for chip control, which is lightweight and supports plug-and-play by avoiding manual efforts on end devices;
4) We design a novel three-layer instruction orchestration logic and corresponding hierarchical scheduler on the gateway and LEGO devices, respectively, which minimizes the communication overhead under remote chip control.

II. Related Works

To the best of our knowledge, a rich set of schemes have proposed to reduce the difficulties in IoT end device design, as follows:

1) *Rapid development of end devices*. Representatively, Gaoyang et al. proposed a solution that can generate a recommended list of hardware configurations based on the user code [4] as guidance for design. They further improve application logic expression through machine

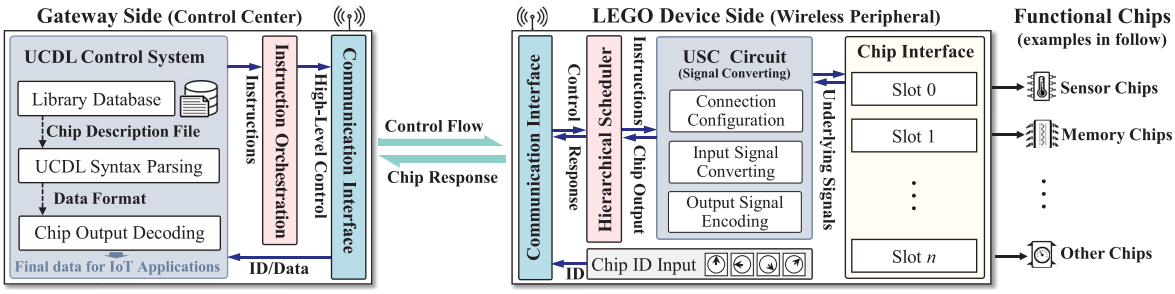


Fig. 2. The architecture of a LEGO system, consisting of a gateway and distributed LEGO devices connected via wireless network.

learning [5]. However, the design still relies on traditional "processor-centric" architecture of IoT end devices, where the system design has to be compatible with the MCU on end devices and can only support standardized hardware platforms like Arduino and Raspberry Pi. There are also other works that focus on this direction, aiming to simplify end device design either in software or hardware aspects [14], [15], [16], [17]. However, as those works not change the hardware architecture of end devices, developers still need to make professional development on the local MCU to fit the heterogeneity of deployed chips.

2) *Wireless programming*. Technologies such as Over The Air (OTA) [18] could help developers to conveniently update firmware or program on their devices via a wireless way [19], [20], [21], [22], [23]. However, these works only change the way of program write-in and cannot reduce the workload in system design. Developer need to complete all development and debugging processes in advance to ensure that the program could work properly with both the device's MCU and deployed chips before uploads to the device. We made corresponding evaluations in Section VIII-G.

3) *Modularized device design*. This direction focuses on the design of standard plug-and-play modules with unified packages to shield the heterogeneity of underlying chips [7], [24]. A rich set of works focus on end device design with several plug-and-play models [25], [26], and some of designs [11], [12], [13] are base on IEEE 1451.4 plug-and-play standard[1]. However, building a plug-and-play module often involves both hardware and software designs. Developers often need to add specific peripheral circuits with a master chips to build modules with standard interaction rules, which is much more complicated than deploying a simple chip and usually results in higher costs due to the additional components, making it not conducive for large-scale applications.

In summary, no previous work has supported chip-level functionality plug-and-play on IoT end devices. The root reason is that they still rely on traditional architecture, where the device works as an independent microcomputer and the deployed functional chips are tightly coupled with its MCU, in both hardware and software aspects. To deploy a new chip for function updates, it is inevitable to make proprietary adjustments around the local MCU to fit the heterogeneity of newly plugged chips. In contrast, LEGO proposes a novel architecture that decouples all chip control to the gateway and simplifies IoT end devices to a simple circuit that requires no changes when deploying new chips, which empowers chips plug-and-play and also helpful to further reduce device cost.

[1]The standards for IEEE 1451.4 "Plug and Play" sensors: https://standards.ieee.org/wp-content/uploads/import/documents/tutorials/1451d4.pdf

III. System Overview

We design a novel architecture for LEGO. As illustrated in Fig. 2, we decouple chip control to the gateway and simplify LEGO devices as a "wireless peripheral" of the gateway, which supports chip-level function plug-and-play through a simple circuit.

A. LEGO Gateway

To minimize application cost, LEGO utilizes the existing gateway to send and receive data, which can be controlled by a simple computing device like Raspberry Pi. We set the following three functional components on the gateway:

1) UCDL Control System: To access various heterogeneous chips uniformly, we design a novel Unified Chip Description Language (UCDL) and its control system, which carries out three main function units, $i.e.$, a library database, a UCDL syntax parsing unit and a chip output decoding unit. In general, the library contains description files with specifications ($e.g.$, connection configuration, control logic) for various COTS chips, written in the UCDL. When receiving a chip ID uploaded from a LEGO device (a new chip is plugged-in), the gateway looks up the library for the corresponding description file and uses the UCDL syntax parsing unit to analyze the found file, and then generate required orchestration logic on meta-operations for chip accessing. Further, the system also works for analysis output data from deployed chips, and can convert the raw data into final application data based on the data format from its description file, $e.g.$, converting raw data $'11100010'$ into 36°C for temperature sensing. The details are presented in Sections IV and V-A.

2) Instruction Orchestration: We design a novel three-layer architecture to orchestrate gateway instructions on three different levels ($i.e.$, chip, sub-function, meta-operation). Based on this design, instruction transmissions between gateway and

LEGO devices can be greatly reduced, minimizing communication overheads of the system. The details are presented in Section V-B.

3) Communication Interface: As LEGO uses the existing gateway to reduce application cost, the communications could be traditional wireless networks such as WiFi, LoRa, and NBIoT.

B. LEGO (IoT) Devices

By decoupling chip control logic to the gateway, we simplify the LEGO device into a circuit that connects the gateway and its onboard chips. It consists of the following five components:

1) Hierarchical Scheduler: To minimize communication overhead, we designed a hierarchical scheduler on LEGO devices associated with the instruction orchestration architecture on the LEGO gateway. In input aspects, the scheduler buffers gateway control flow according to independent sub-functions (*e.g.*, sensing range setting, mode selecting, data reading) of the chip. In output aspects, the scheduler buffers chip output, making data aggregation and handling control signal for chip cooperation. Based on it, the complete control flow only need to be issued only once from the gateway, and subsequent chip operations can be achieved by simple instructions to schedule the buffer. Moreover, the device can only upload the required application data to the gateway, thus minimizing communication loads (details in Section VI-A).

2) Universal Signal Converting (USC) circuit: To support chips plug-and-play, the USC circuit plays an essential role, handling all input and output Transistor-Transistor Logic gate (TTL) signals of onboard chips. In general, the circuit integrates three functional units: a) *Pin Configuration*. Under gateway instructions, it dynamically sets pin connections with required electrical feature for the plugged chip; b) *Input Signal Converting*. For chip control, this unit directly converts gateway instructions into specific physical signals (*e.g.*, address and bus data signals) with the required timing and formats for each pin of a chip; 3) *Output Signal Encoding*. For chip response, this unit dynamically converts the output signal from a chip into uplink instructions for the hierarchical scheduler which may further uploads to the gateway. By this, closed-loop control between the gateway and deployed chips can be achieved. The details are presented in Section VI-B.

3) Chip ID Dialer: To identify different functional chips for plug-and-play, we assign a unique ID to each type of chip and associate it with its description file. To load the ID when plugging in chips on the device, we provide a chip ID dialer as an example for users to enter the chip ID by turning hexadecimal dial switches.

4) Chip Slots: A LEGO device provides a set of n chip slots to accommodate functional chips. When a new chip is plugged in a vacant slot, the power supply connection in that slot closes and triggers the LEGO device to upload the chip ID recorded by the dialer to the gateway. Similarly, when a chip is removed from the LEGO device, the gateway would also be notified.

5) Communication Interface: LEGO devices connect to the same network as the gateway.

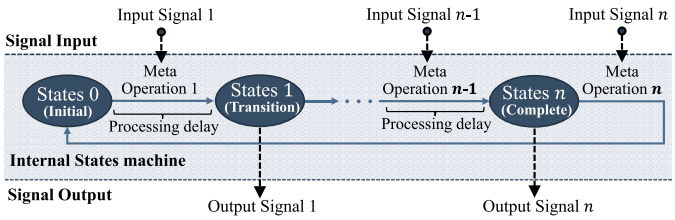


Fig. 3. The abstract model of chip operation.

TABLE I
SIX KEYWORDS FOR UCDL

	Keywords	Description
Configuration	PIN	Set pin functions for a chip.
	DF	Define data format for a chip.
Chip Control (Meta-Operation)	DW	Write data to a chip.
	DR	Read data from a chip.
	CW	Send a control signal to a chip.
	SR	Read a status signal from a chip.

IV. UNIFIED CHIP DESCRIPTION LANGUAGE

One major barrier to plug-and-play chips on end devices is the requirement of software modifications to fit the heterogeneous feature of newly plugged chips. Such modification is tedious for application developers and technically infeasible for end-users. To combat this difficulty, we analyze the principle of chip control in-depth and design a novel Unified Chip Description Language (UCDL) with simple syntax to describe specifications for various types of heterogeneous chips uniformly.

A. Overview

Through in-depth research, we have found that most chips are operated in the state machine, so we build up a abstract model of chip operation. As shown in Fig. 3, specific signal inputs drives the chip jump to the next state and a series of state transitions form up the required chip functions. For instance, the data reading operation on a chip involves controlling its internal state machine to jump to the data ready state from which we can read data.

Next, considering one-step of internal states transition of a chip is the minimum operation, which is atomic and indivisible, we define it as a meta-operation. Through massive chip analysis, we discovered that despite different chips are highly heterogeneous in their control logic and signal timing, the required meta-operations can be summarized into four basic types: 1) write data into a chip; 2) read data out from a chip; 3) write control signals into a chip, *e.g.*, write logic high to a chip's chip select to pull it up; 4) read state signals from a chip, *e.g.*, check the interrupt signal. Furthermore, to access a chip, in addition to the four basic meta-operations, the configurations for electrical connections on pins and data format definitions for chip outputs are also required. Hence, we design six keywords to describe the configuration and control operations for various heterogeneous chips uniformly, as summarized in Table I.

B. Basic Grammar of UCDL

We introduce the basic grammar[2] of UCDL for pin connection configuration, meta-operations, and data format definition.

1) Pin Connection Configuration: We design the keywords *PIN* to dynamically set electrical connections for each pin of a plugged chip. We define basic syntax *PIN(j, type, function, connection)* to configure the j-th pin of a chip with specific *type* ($e.g.$, power, ground, I2C data bus), *function* ($e.g.$, data read, clock input), and *connection* ($e.g.$, open-drain, push-pull). For example, *PIN(3, SPI, DR, push−pull)* means that the third pin of the chip is set to be the data-read line of the SPI bus and its connection type is push-pull. In practice, the PIN statement drives the LEGO device to set the required electronic connections for a specified pin of the target chip. The detailed working flow is presented in Section VI.B.1.

2) Meta-Operations for Chip Control: We define the following four keywords to carry the summarized meta-operations with specific timing requirements:

1) *Data Write (DW)*: We define the keyword *DW* for the meta-operation of data-write, and the basic syntax is *DW(j,0xYY)*, which means to write the hex data *YY* to a particular register of the chip via Pin *j*.
2) *Data Read (DR)*: We define the keyword *DR* to read data from a chip, and the syntax is *DR(j, x)*, which means to read *x*-byte data out from the chip via Pin *j*. For the convenience of subsequent data conversion, we divide the raw read data into individual bytes. For instance, $DR(3, 4, X_1, X_2, X_3, X_4)$ means to read 4 bytes from Pin 3, and stored as X_1, X_2, X_3 and X_4, respectively.
3) *Control Write (CW)*: We define the keyword *CW* to set or clear a control pin of a chip, and the syntax is *CW(j, H/L)*, which means to send a logic high (*H*) or logic low (*L*) signal to a control pin j (*e.g.*, chip select, interrupt) of the chip.
4) *State Read (SR)*: We define the keyword *SR* to read the state information from a chip, and the syntax is *SR(j, H/L)*, means to check the state on the j-th Pin (*e.g.*, interrupt request) of a chip.

With above meta-operations, it is easy to define a chip control sequence according to its datasheet. For instance, to read the acceleration data from an ADXL362 accelerometer chip, the following steps should be conducted: 1) enable the chip by pulling down the chip select pin to logic-low (*e.g.*, the chip select pin is referred to as CS) $\rightarrow$ 2) send a data reading command (*i.e.*, 0x0B) to the chip $\rightarrow$ 3) specify from which address (*e.g.*, 0x00) to read data $\rightarrow$ 4) read the 3-axis acceleration data of three bytes from the chip $\rightarrow$ 5) turn the chip into sleep mode by pulling up the chip select pin to logic-high. The corresponding control sequence using meta-operations is *'CW(CS Pin, Pulldown to logic-low)* $\rightarrow$ *DW(0x0B)* $\rightarrow$ *DW*$(0x00)$ $\rightarrow$ *DR(3 bit acceleration data)* $\rightarrow$ *CW(CS Pin, Pullup to logic-high)'*.

[2] We construct a comprehensive Backus-Naur Form (BNF) [27], [28] grammar for the UCDL, in the anonymous link: https://anonymous.4open.science/r/UCDL-grammar-and-Chip-specifications-86D5/README.md

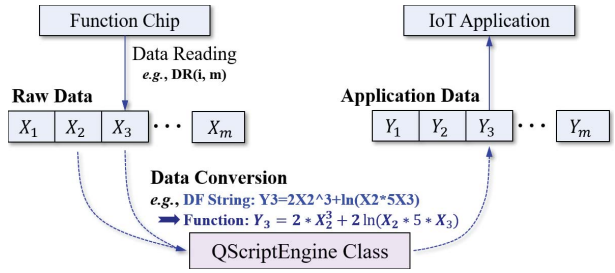


Fig. 4. Example of data conversion.

Finally, as functional chips have a certain processing time for meta-operations, $i.e.$, it takes time for state transitions in its internal states machine (as discussed in Fig. 3), hence subsequent operations can only be initiated when the state transitions is complete, otherwise it will be ignored. To settle this issue, we attach an additional time delay to each meta-operation to define the required time for chip processing (state transition), defined as *'operation + processing delay'*. For instance, $DW(j, 0x0b) + 6\mu$S means that the chip requires 6 microseconds to process the data $(0x0b)$ after it is written in, only after which it transition to the next states and the subsequent meta-operation can be executed. The exact operation duration of each meta-operation can also be obtained from the chip datasheet.

3) Data Format Definition: A chip may have multiple types of raw data output ($e.g.$, a BME280 sensor can output data on temperature, humidity, and air pressure). We set the keyword DF to define the output data format for a chip with syntax of *DF(j, type, func, unit)*, which the type of the j-th output is *type* and the data conversion functions and unit are *func* and *unit*, respectively. For example, $DF(1, air\ pressure, x/322.5, Kpa)$ means that the chip first returns the air pressure data. The data conversion function is $x/322.5$ (x is the raw data), and the unit of the result is Kpa (*i.e.*, 1000N/m^2).

To support complicated data conversion, we integrates the QScriptEngine [29] on the LEGO gateway for data conversion with functions in strings. By this, the user can directly write the conversion formula as a string. As illustrated in Fig. 4, the gateway first uses DR instruction to get m bytes of raw data, $e.g.$, X_1, X_2, $\cdots$, X_m from a deployed chip on a LEGO device, and converts the raw data into the desired information, $e.g.$, Y_1,Y_2, $\cdots$, Y_m, according to the given DF string. This data conversion can be one-to-one or many-to-one across multiple readings.

C. Chip Description File and Library

By writing UCDL statements, we can create description files for different chips, with which the gateway can dynamically generate the required control logic for various deployed chips. The structure of a UCDL-based chip description file contains two parts, $i.e.$, one part for chip configuration and another part for chip control. For instance, as shown in Fig. 5, the description file of an ADXL362 accelerometer chip only needs 27 lines of statements to describe its pin connection, data definition, and three sub-functions.

```
----------Connection configuration-(1)----------
1 Pin(1, VDD,3.3V,Open-Drain(type2)); // Set Pin 1 to 3.3V Voltage Supply
2 Pin(2, SPI, Clock, Push-Pull(Input));// Set Pin 2 to SPI Clock input
3 Pin(3, SPI, DW, Push-Pull(Input));   // Set Pin 3 to SPI data-write
  …Code…
----------Data format configuration-(2)----------
8 DF(Y1, X-Acc, (y=X1/64, X1<128; y=(X1-256)/64, X1>=128), g);
  //Data converting for X-axis Acceleration(convert raw data to app data)
9 DF(Y2, y-Acc, (y=X2/64, X2<128; y=(X2-256)/64, X2>=128), g);
  //Data converting for Y-axis Acceleration(convert raw data to app data)
  …Code…
----------Data read operation-(3)----------
11 CW(5, L), +2us;          // Set Pin 5 to logic low to enable the chip
12 DW(3,0x0b),+2us;         // Write in 0x0b(data-read command) to Pin 3
13 DW(3,0x08),+2us;         // Write in 0x08(data register address) to Pin 3
14 SR(Pin 7,H),+0.4us;      // Wait Pin 7 turn high
15 DR(4,3,X1,X2,X3),+6us;   // Read out 3 byte data from Pin 4
16 CW(5, H)+0.2us; }        // Set Pin 5 to high to disable the chip
----------Set Standby Mode-(4)----------
17 CW(5, L),+2us;           // Set Pin 5 to logic low to enable the chip
18 DW(3,0x0A),+2us;         // Write in 0x0A(write-register command) to Pin 3
19 DW(3,0x2D),+2us;         // Write in 0x2D(data register address) to Pin 3
20 DW(3,0x0),+2us;          // Write in 0x00(data byte) to Pin 3
21 CW(5, H),+2us;           // Set Pin 5 to high to disable the chip
----------Set Wakeup Mode-(5)----------
22 CW(5, L),+2us;           // Set Pin 5 to logic low to enable the chip
23 DW(3,0x0A),+2us;         // Write in 0x0A(write-register command) to Pin 3
24 DW(3,0x2D),+2us;         // Write in 0x2D(data register address) to Pin 3
25 DW(3,0x08),+2us;         // Write in 0x08(data byte) to Pin 3
26 CW(5, H),+2us;           // Set Pin 5 to high to disable the chip
--------**26 lines code for 5 function blocks in total**-------------
```

Configuration

Meta-operation (Chip control)

Fig. 5. An example description file for a COTS ADXL362 accelerometer chip, where the configuration and control keywords are marked in blue and red, respectively.

The LEGO system has an efficient operating mechanism, when a new chip is plugged into an LEGO device, the corresponding description file only needs to be parsed once. For chip configuration, the UCDL codes for PIN and DF keywords are coded into instructions to set electronic connections and define data structure for the newly plugged chip, respectively. The chip control part ($i.e.$, UCDL codes for DW, DR, CW, and SR) would be buffered in the hierarchical scheduler (Section VI-A) on the LEGO device once it is received, which further drives the universal signal conversion (USC) circuit (Section VI-B) to achieve efficient chip access. Based on such a chunked design, the complete control flow with all instructions for a plugged chip only needs to be transmitted once as it plug in, and specific chip functions can be scheduled ($e.g.$ read an sensor chip at 200 Hz) on the LEGO device by a simple gateway command, minimizing the communication overhead.

Finally, to boost the application of our LEGO architecture, we build an online library database for public use. To assign unique IDs to various chips, we establish a hash table using MD5 as the hash function to map each chip model name to a 16-bit digital ID space. Hash collisions are handled by increasing the collided number until it is not occupied. Then, the assigned ID of a chip is associated with its corresponding description file. In the initial stage, the library contains description files for 100 types of COTS functional chips, the open source of 100 supported COTS chips can be accessed via the anonymous link in footnote 2 (on page 4).

D. Error Detection

As functional chips on LEGO devices operate based on a state machine and are controlled by specific underlying signals on the target pin, as defined in UCDL (as described in Section IV-A), any errors in UCDL statements or chip connections will cause the corresponding internal state transitions

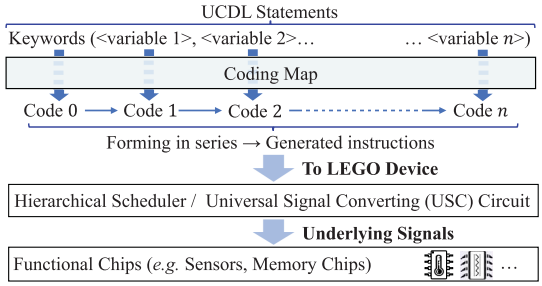


Fig. 6. The process for UCDL statement encoded into instruction and go through LEGO device for chip control.

(meta-operations on the target chip) to fail, and the error can be localized. Consequently, LEGO has a simple error detection mechanism; once errors occur in the UCDL codes and chip connections, the user will be notified.

Base on it, LEGO could also find potential errors in chip connections, as each UCDL code points to a specific pin on the target functional chip. For instance, the UCDL code ′CW(5,L)′ indicates pulling the 5th pin of the chip to logic-low. When an error occurs, it might be incorrect connections on pin 5 or bugs in the code, and the user will be notified.

The limitation of the initial error checking mechanism is that it can only detect errors but unable to identify the exact types of errors ($e.g.$, errors in meta-operation sequences, signal timing, or chip connection definitions). In future work, we will further optimize the error checking mechanism, exploring possible solutions discussed in Section 9 (error handling).

V. LEGO Gateway

The LEGO gateway works as the control center, handling the unified access of various functional chips. Its key functions are UCDL syntax parsing and instruction orchestration as follows.

A. UCDL Syntax Parsing

We build an efficient coding logic to parsing UCDL statements into downlink instructions for unified chip control. As shown in Fig. 6, for a UCDL statement, the gateway automatically maps its keywords and variables into a predefined code and connects them in series to form an independent instruction, which further drives the USC circuit on LEGO devices to generate the required signal for chip control.

For a chip description file, the UCDL statements for pin configurations (keywords: PIN) and logic control (keywords: *DW, DR, CW, SR*) are converted into gateway instructions. But the statements for data formats (keywords: DF) only work on the gateway, which guides the gateway to convert chip output into final data for IoT applications. The basic encoding scheme of the downlink 5 gateway instructions is as follows (the detailed coding rules can also be accessed via the anonymous link in footnote 2):

1) *PIN instruction*: The basic code format for pin configuration instruction consists of five fields, $i.e.$, 4-bit keyword code, 4-bit pin number code, 4-bit pin type code

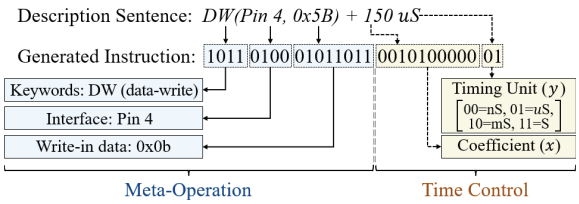


Fig. 7. The example of UCDL syntax parsing.

($e.g.$, power supply, SPI data bus), 4-bit pin function code ($e.g.$, data-write, interrupt), and 5-bit connection type code ($e.g.$, push-pull).

2) *DW instruction*: Its code format consists of three fields, $i.e.$, 4-bit keyword code, 4-bit pin number code, and n-bit write-in data code (the data to write in the target chip).
3) *DR instruction*: Its code format consists of three fields, $i.e.$, 4-bit keyword code, 4-bit pin number code, and 4-bit read length definition code (number of Bytes read out from the target chip).
4) *CW instruction*: Its code format consists of three fields, $i.e.$, 4-bit keyword code, 4-bit pin number code, and 1-bit control flag code to set the target pin of a chip to logic high or logic low.
5) *SR instruction*: Its code format consists of two fields, $i.e.$, 4-bit keyword code and 4-bit pin number code.

Based on this, the gateway can automatically generate the binary commands needed to control the target chip based on the UCDL statements. As illustrated in Fig. 7, for instance, the meta-operation statement $'DW(Pin\ 4,\ 0x0b) + 150\ \mu S'$ is converted to three-byte code '1011-0100-01011011-0010100000-01' (not include device code), where the former part (1011-0100-00001011) represents the meta-operation to writes the hex data $'0x0b'$ into pin 4 of the chip, and the later part (0010100000-01) represents time control $i.e.$, the chip needs 150 microseconds to process the current meta-operation, and the next operation can only be executed after that. In the design, the exact value of time control is the product of the coefficient (x) and the timing base (y). By this, we can provide a wide range of processing delay for a single meta-operation with only 12 bit of instruction.

B. Three-Layer Instruction Orchestration

To reduce the system communication overhead, we designed a three-layer orchestration architecture for gateway commands, which could minimize the interaction frequency between the gateway and end devices. As illustrated in Fig. 8, we orchestrate gateway instructions in three levels, $i.e.$, deployed chips, sub-functions, and meta-operations.

Specifically, a sub-function of the chip ($e.g.$, working mode setting, data reading) could be achieved by an orchestration sequence of UCDL instructions, and each instruction contains a meta-operation and its timing control (as described in Section V-A). When a new chip is plugged into a LEGO device, the gateway first writes all the relevant instructions into the hierarchical scheduler of the device in independent sub-functions. After that, the gateway can directly invoke specific

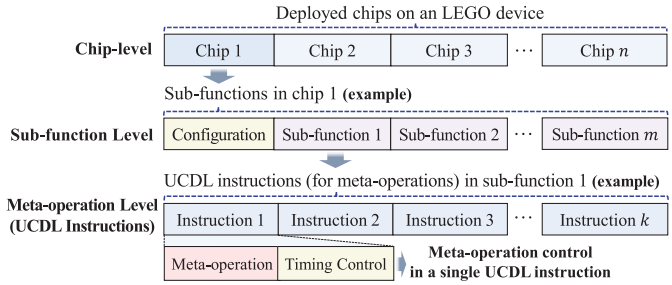


Fig. 8. Three-layer orchestration logic for gateway instructions in LEGO.

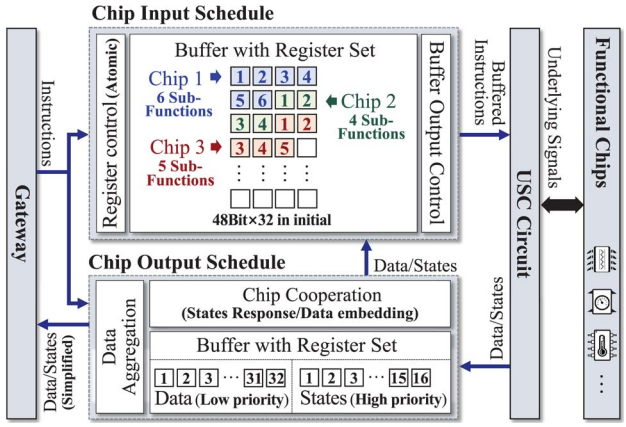


Fig. 9. The structure of the hierarchical scheduler on LEGO devices.

sub-functions for the chip by a simple command, and the LEGO device executes all meta-operations according to the orchestration sequences and timing control in the selected sub-function ($e.g.$, read temperature from the sensor chip on slot 5 at 20 Hz). By this, the complete control flow for a functional chip only needs to be issued once as it first plugs into a LEGO device, thereby minimizing communication overhead.

VI. LEGO Device

The essential component of a LEGO device is the hierarchical scheduler and USC Circuit, where the former handles communication overheads in instruction transmission, and the latter deals with all TTL signal interactions for the control of various types of heterogeneous on-board chips.

A. Hierarchical Scheduler

To reduce communication overhead, we design a hierarchical scheduler to work with the three-layer instruction orchestration architecture (Section V-B) that handles chip control efficiently without the need of a MCU. As illustrated in Fig. 9, the scheduler handles chip interaction in both input and output aspects.

1) Chip Input Schedule: In input aspects, the scheduler integrates four units, as follows:

1) *Register Set*: The register set consists of multiple registers, and each register can buffer instructions for an independent sub-function of the deployed chip ($e.g.$, data

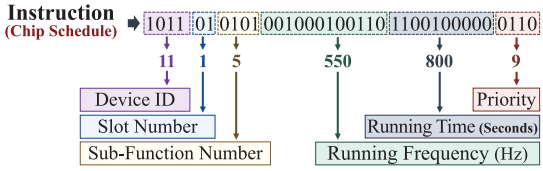
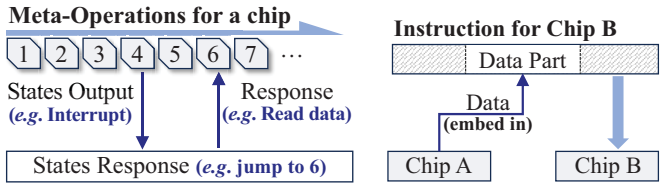


Fig. 10. Format of the chip schedule (register control) instruction.



(a) States response (b) Data embedding

Fig. 11. Example for local chip cooperation in LEGO.

reading, work-mode setting). We set the initial size of each register by exploring the control of many COTS chips. The space can also be further improved by adding a external memory in specific applications.

2) *Register Control:* This unit controls the schedule of a specific register to send out buffered instructions, which further drives the Universal Signal Converting (USC) circuit to generate the required underlying signals for chip control. Since the buffered instructions are pre-orchestrated according to control logic and chip specifications, the generated underlying signals can meet the timing and electrical characteristic requirements of the target chip. Additionally, this unit also controls the writing of registers, which often occurs after a new chip is plugged in.

3) *Buffer output control:* This unit ensures the atomicity of chip operations. As the sub-function of a chip can be realized by orchestrating a set of meta-operations, and it takes effect only when all meta-operations are complete. We design this unit to drive out all meta-operation instructions from the selected register when scheduled, and it also counts instruction outputs during the progress. Once a sub-function is interrupted in special cases ($e.g$, system pause for debugging), the unit can resume the progress by outputting the current instruction according to the count record.

By this, mass tasks of the selected sub-function for a chip can be excited by scheduling the corresponding register. As illustrated in Fig. 10, the instruction '1011-01-0101-001000100110-1100100000-0110' means to schedule the selected chip (on slot 1) with sub-function 5 at 550 Hz for 800 seconds on the target LEGO device ($ID = 11$), and the task priority is 9 (a smaller number means a higher priority). Hence, mass tasks with the selected sub-function can be scheduled by issuing a simple instruction from the gateway, minimizing the communication overhead.

In practical applications, the scheduler can schedule multiple chip sub-functions in a sequential way according to their priorities. For instance, if a LEGO device receives two instructions, $i.e.$, one is to schedule sub-function A at 600 Hz with a priority of 3, and the other is to schedule sub-function B at 450 Hz with a priority of 5. In this case, sub-functions A and B will be executed interleaved at 600 Hz and 450 Hz, respectively. Besides, as sub-function A has a higher priority, it will be executed first when the two sub-functions coincide in the same time.

Finally, base on the end-to-end control mechanism, LEGO could handle errors in chip scheduling without introducing extra components. Specifically, the operation of LEGO end devices is predictable as it is predefined by the gateway. Once errors occur in the scheduler ($e.g.$, changes in scheduling frequency) or USC circuit (causing chip control to be terminated), the LEGO device cannot return the target data or control signals within the anticipated time-frame to the gateway. In such cases, the gateway can identify the error and restart the corresponding unit in the LEGO devices, and users could also be notified.

2) Chip Output Schedule: To further improve task efficiency, we set three functional units with a simple structure for chip outputs, as follows:

1) *Chip Cooperation.* In many cases, updating all chip outputs to the gateway is required by IoT applications, but inefficient in practical. Hence, we design this unit to handle chip cooperation locally without need for a MCU. With this unit, mass chip output could be processed locally instead of upload to the gateway, which could further reduce communication overhead and improve task performance under a given resource.

The chip cooperation unit have main two functions: a) *States Response*. For instance, as illustrated in Fig. 11(a), the chip outputs a state signal ($e.g.$, interrupt) that indicates a jump to meta-operation 6 to read data out. This process can be executed directly in local without the need for uploading the state to the gateway. Additionally, it is also applicable to handling status signals if a user presses a button. b) *Data Embedding*. The unit also supports direct data transfer between different chips, as shown in Fig. 11(b). For example, if a LEGO device plugs a sensor chip to sense environmental temperature and wishes to record the data in a EEPROM, the unit can directly the embedded the obtained temperature data in the control flow of the EEPROM, without the need of a round trip to the gateway.

2) *Register Set (for chip output)*. We set independent registers on LEGO devices for data and state outputs, which can buffer chip outputs and handling collisions when multiple chips generate output simultaneously. The register adopts a first-in first-out mode for real-time insurance, and the state register has a higher priority than the data register.

3) *Data aggregation*. We set this unit to further improve transmission efficiency, if needed, it can aggregate multiple chip outputs into one data packet to reduce communication protocol overhead when upload to the gateway.

Base on above design, the hierarchical scheduler could help LEGO devices to schedule the plugged chips efficiently via a simple architecture, which does not need the assist of a MCU, also embedded design are no longer required. The scheduler further drives the USC circuit to generate underlying signals for chips plug-and-play, which we discuss in Section VI-B.

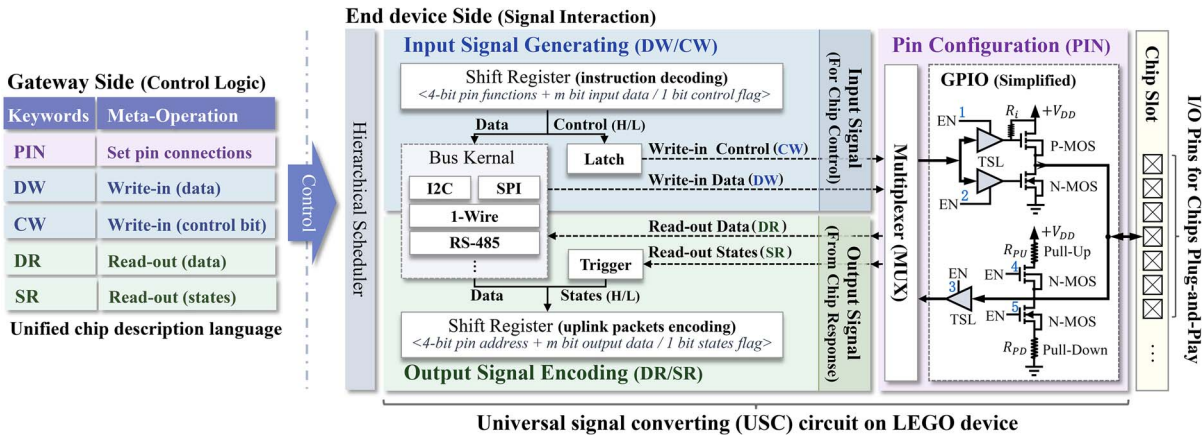


Fig. 12. The USC circuit is composed of three parts $i.e.$, pin configuration, input signal generation and output signal encoding. It works passively under the control of UCDL statements and can provide required underlying signals for accessing various heterogeneous chips uniformly.

B. Universal Signal Converting (USC) Circuit

To achieve unified plug-and-play for various heterogeneous chips, we have design a universal signal conversion circuit on LEGO devices that serves as "middle-ware" connecting the gateway and various deployed chips. As illustrated in Fig. 12, the circuit integrates three effective units passively derived by the description language: 1) *Pin Configuration*. This unit works by the drive of PIN statements, which dynamically configures electrical connections for deployed chips; 2) *Input Signal Generating*. It works by the control of DW and CW statements and dynamically generate bus data signals ($e.g.$, I2C bus signal) and specific control signals ($e.g.$, interrupt) for chip access, respectively. 3) *Output Signal Encoding*. It drives by DR and *SR* statements, controlling the LEGO device to read out the data and states ($e.g.$, interrupt) signals from the chip for closed-loop control, respectively. The working flow of each unit are as follows.

1) Pin Configuration: This unit (right part of Fig. 12) is passively driven by PIN statements, which dynamically set electronic connections for each pin of a plugged chip. Specifically, although different chips are highly heterogeneous in their interfaces, their connection features on pins can be achieved with basic functions ($e.g.$ interrupt output) and electrical characteristics ($e.g.$, push-pull, open-drain). To support chips plug-and-play, we need dynamically set pin functions as well as electrical characteristics for the plugged chips.

To achieve this, in pin function aspects, we integrate a multiplexer (MUX) in front of the I/O interface, which could connect internal functional modules ($e.g.$, specific bus kernel, trigger, etc.) in the circuit to the target pin of a chip under the control of PIN statements. For instance, we can connect the clock and data lines of the I2C bus core to pins 3 and 4 of the selected chip to build up data connection with I2C bus. Alternatively, we can connect a trigger to pin 4 of a chip to detect interrupt signals.

Next, to flexibly set pin connections for various plugged chips, we integrate a simplified GPIO control circuit. As listed in Table II, different pin connections can be set by controlling a set of five enable (EN) signals:

TABLE II
CONNECTION CONFIGURATIONS

Configuration	Command	EN signal States[1]				
		1	2	3	4	5
Push-Pull (output)	11000	H	H	L	L	L
Push-Pull (input)	00100	L	L	H	L	L
Open-Drain (type 1)	01010	L	H	L[2]	H	L
Open-Drain (type 2)	10001	H	L	L[3]	L	H
High Impedance	00000	L	L	L	L	L

[1] The states of the five enable (EN) signals in the GPIO circuit, where H (logic high) represents enable, and L (logic low) disable.
[2,3] This signal is required to be set to logic high when receiving (reading) data from chips, and the instructions are 01110 and 10101 for type 1 ($e.g.$, I^2C) and type 2 ($e.g.$, RS485), respectively.

1) *Push-pull connection*: In the push-pull connection, the input and output signals are physically separated. To configure an output pin, EN 1 and EN 2 are set to high with other EN signals set to low to form a push-pull structure. Similarly, to configure an input pin, EN 3 is set high, and all other EN signals are set low.
2) *Open-drain connection*: The open-drain connection supports bidirectional signal interaction on the same pin, which need one MOSFET and the matched pull resistors for signal output ($i.e.$, an N-MOS and pull-up resistor for Type 1 and a P-MOS and pull-down resistor for Type 2), and a TSL gate (3) for signal input.

2) Input Signal Converting: To dynamically generate required bus data ($e.g.$ I2C, SPI) and control signals ($e.g.$, chip select or interrupt signals) for chip access, we design the *Input Signal Converting* unit, which passively controlled by DW (data write) and CW (control write) statements. As shown in Fig. 12 (upper part), we employ a simple Shift Register (with serial input to parallel output) for instruction decoding, which could splits instructions into independent control bits, and selects the target bus kernel or latch for bus data or control signal transmission, respectively.

Specifically, the instruction for DW and CW statements contains 4-bit code for *pin functions* and m-bit code for input

data or 1-bit code for *control flag*, respectively. For bus data input (DW), it controls the selected bus kernel to generate underlying signals with required bus protocol ($e.g.$, I2C, SPI), and send to the target pin of the deployed chip to write-in data; for control signal input (CW), it controls a latch to generate target voltage level that pull the selected pin of a chip to logic high (H, *control flag*=1) or logic low (L, *control flag*=0). As long as instructions are outputted from the hierarchical scheduler (Section VI-A) with predefined timing requirements, the circuit can provide all required signals to access all the plugged chips dynamically.

3) Chip Output Encoding: To dynamically convert the chip outputted data and state signals ($e.g.$, interrupt) into uplink package for control response, we design the *Chip Output Encoding* unit, which is controlled by the DR (data read) and SR (state read) statements. As depicted in 12 (bottom part), the read-out chip data is received by the corresponding bus kernel via the pre-configured MUX (as described in Section VI.B.1), and the state signals are detected by a Trigger. For data signal readout (DR), the selected bus kernel derives the chip data and assembles a packet with the data and pin address. For states signal readout (DR), the voltage change on that pin changes the on-off state of the trigger, which drives the circuit to construct a new packet with one-bit state information (1 for logic high and 0 for logic low) and the corresponding pin addresses. Such packets are transmitted to the hierarchical scheduler for data aggregation or local chip cooperation, and can also upload to the gateway for further processing.

VII. Implementation

To verify the feasibility of LEGO, we implement a LEGO demo and build up three proof-of-concept applications.

A. Implementation Details

1) *LEGO Device*. As shown in Fig. 13, the demo of our LEGO device consists of a mainboard, multiple COTS chips, and different communication modules. Besides, the block diagram of the hardware implementation is illustrated in Fig. 15.

In the prototype, the core USC circuit (Section VI-B) and hierarchical scheduler (Section VI-A) are implemented using a low-power FPGA (AGLN060V2) with less than 45,000 logic gates. The USC circuit provides standard TTL I/O interfaces, which could be compatible with the majority of digital chips available in the market. To fascinate users, the board contains standard pin slots with 2.54 mm intervals, allowing connections to commercial off-the-shelf (COTS) chips via DuPont wires. We also provide chips with a Type-C interface and Type-C slots on the prototype, enabling users to plug-and-play chips more conveniently.

Further, to extend the support for analog chips, we incorporated an 8-bit ADC (LTC1096L) and an 8-bit DAC (MCP4901) for analog signal input and output, respectively. For chips with higher interface voltage, we add a DC-DC converter (TPS61378) and two level shifters (MC14504B) on the input and output side of the FPGA to compact the I/O voltage from 4

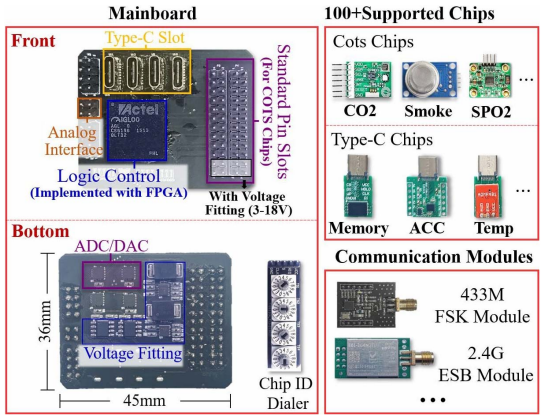


Fig. 13. In the initial stage, we implemented our prototype LEGO device by an FPGA with different communication modules to support chips plug-and-play.

TABLE III
Chip Types Supported in LEGO Demo

Sensing Functions	Temperature	Humidity	GPS	PM2.5
	Acceleration	CO2	Distance	PM10
	Air Pressure	Illuminance	Angular	Pulse
	Flow Velocity	Vibration	Alcohol	Smoke
	Displacement	Pulse oxygen saturation (SpO2)		
Other Functions	LED screen \ Electronic ink screen refreshing			
	Electronic ink screen refreshing			
	Writing and reading memory chips			
	ADC(analog-digital converter) Sampling			

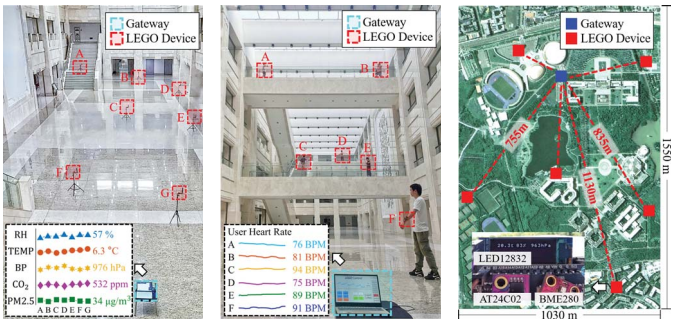
V to 18 V. All components are selected in low power consumption at the micro-amp level. When using only standard TTL I/O, the DC-DC converter, ADC, DAC, and level shifter could also be turned off to further reduce power consumption. We may also implement LEGO devices through SOC design to further reduce power consumption and cut-down hardware costs to : $3 [30], which is much less than Arduino devices (often costing from $22 to $46).

2) *LEGO Gateway*. To reduce the cost of deployment, we utilize the existing gateway for transmitting and receiving data. The gateway demo is implemented with a Raspberry Pi 4B, which features a 1.5 GHz quad-core 64-bit ARM Cortex-A72 CPU, 4 GB of memory, and 16 GB of TF card space. It can control different communication modules through USB or GPIO interfaces.

3) *COTS Chips*. For the convenience of use, we directly utilize Commercial Off-The-Shelf (COTS) chips that available in the market, and only led out the chip layout to standard pins with 2.54 mm intervals or to a Type-C interface to fit the mainboard. We have verified a total of 105 COTS chip models, containing 19 sensor types and four non-sensor types, as shown in Table. III.

B. Proof-of-Concept Applications

We deploy our LEGO device in three demonstrating applications, as follows. With chip-level plug-and-play capabilities, all the applications are user-friendly and can be set up in a few minutes.



(a) Air monitoring (b) Wireless pulse oximeter (c) Remote display

Fig. 14. Three example LEGO applications for proof-of-concept.

1) *Indoor Air Monitoring*. As depicted in Fig. 14(a), we have set up the demo in a lobby using seven LEGO devices and a gateway. Each LEGO device is equipped with a BME280 chip for measuring temperature (TEMP), relative humidity (RH), and barometric pressure (BP), a CCS811 chip for measuring CO_2 concentration, and a GP2Y1014AU chip for measuring PM2.5 concentration. The data link between LEGO devices and the gateway is established using a 433 MHz FSK (Frequency-shift keying) communication module (SI4460).
2) *Wireless Pulse Oximeter*. As shown in Fig. 14(b), we deploy our LEGO devices on the wrists of six volunteers to collect their heart rate information. To build the system, we remove all sensors in the air monitoring application and plug a MAX30102 chip on each LEGO device to collect the heart pulse of the volunteer. We also deploy a 315-MHz RXM-315 ASK (Amplitude Shift Keying) module on the device for data transmission.
3) *Remote Display*. As shown in Fig. 14, we have developed an application to monitor the temperature and air quality in a 1030×1550 m^2 area. In this application, we use BME280 sensor chips on LEGO devices to collect temperature (TEMP), relative humidity (RH), and barometric pressure (BP) information, along with AT24C02 EEPROM memory chips to store the display formats. We have deployed 2.4 GHz Enhanced Shock Burst (ESB) modules (E01-24GM27D) on both the gateway and LEGO devices to provide kilometer-level connectivity. The gateway collects sensory data from six LEGO devices, calculates the average value for the entire area, and then sends the results back to all the LEGO devices, which further displays the average value of TEMP, RH, and BP for the six points on its LCD screen through SPI interface.

VIII. Performance Evaluation

A. Response Time for Chips Plug-and-Play

The chip-level plug-and-play capability for IoT applications with LEGO is revolutionary. It indicates that end devices could play a new role once the desired chips are plugged-in. Therefore, we first evaluated the response performance for chips plug-and-play. We use an oscilloscope to record the time duration

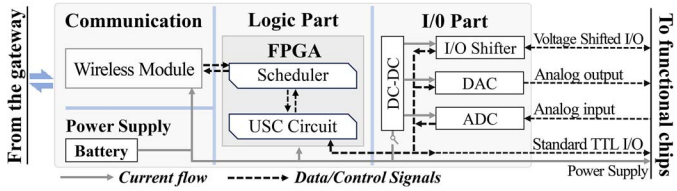


Fig. 15. The hardware diagram of the LEGO demo implementation mainly consists of three parts, $i.e.$, communication, logic control, and I/O part.

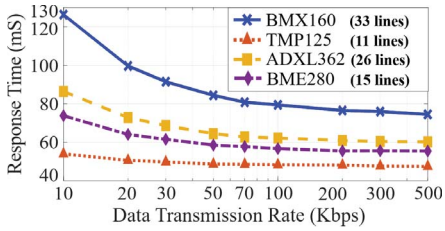


Fig. 16. Response time for four different functional chips under various data rates.

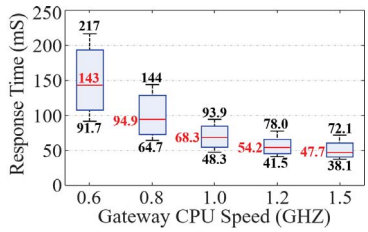


Fig. 17. Response time distribution under different gateway CPU speeds for 20 type of chips.

from when the chip was plugged in until it was available for use ($i.e.$, a response instruction was uploaded by the LEGO device).

We first selected four commonly used sensor chips and test the response time under data rates from 10 kbps to 500 kbps with the gateway CPU speed set at 1.0 GHz. The results of the response times are presented in Fig. 16. It can be seen that all chips can be accessed within 0.13 seconds after they are plugged into a LEGO device. We also find the results are highly distinct in different chips due to the complexity of description files, but the impact of transmission rate only being obvious at low speeds ($\leq 50\ Kbps$). The reason is that to respond to a newly plugged chip, only a few hundred bits of ID and instructions need to be transmitted, but the complexity of chip description logic may vary significantly. For instance, the BMX160 chip, a 9-axis gyroscope, has 33 lines of code in its description file, which is much more complicated than that of the TMP125 temperature sensor chip (11 lines).

Further, we expanded the test range to 20 chips and recorded the response time for plug-and-play under different gateway CPU speeds. We fixed the communication data rate at 500 kbps to control the variables. The box plot of the results is summarized in Fig. 17. It can be seen that as the gateway CPU running speed decreases, the response time increases, and the value change not obvious when it is above 1.0 GHz. This is

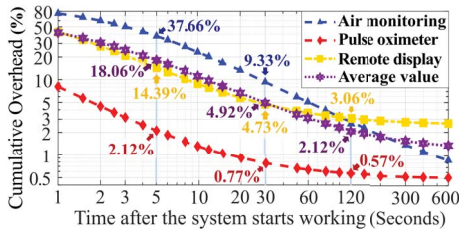


Fig. 18. The proportion of cumulative communication overhead in remote chip control.

because data transmission occupies a portion of the time, and syntax parsing operations only account for a small fraction of the total time when the clock frequency is high ($\geq$ 1.0 GHz).Further, when the gateway CPU speed is higher than 0.8 GHz, the response time for most chips is less than 0.1 seconds. In summary, LEGO supports chip-level plug-and-play and is capable of responding promptly.

B. Communication Overhead

By decoupling chip control uniformly to the gateway, LEGO avoids manual development when deploying new chips and also simplifies the architecture for end devices with lower cost and power consumption, but it also introduces extra communication loads. Therefore, we conducted a comparison experiment between LEGO devices and traditional end devices (controlled by a local processor) on communication overhead in three scenarios in Section VII-B. To set standardized tests, we configure the chip reading frequency at 3 Hz, 15 Hz, and 120 Hz for the scenarios of air monitoring, remote display, and wireless pulse meter, respectively.

In the experiment, traditional end devices are pre-configured and only need to upload the obtained sensory data to the gateway. However, LEGO devices are customized in the field by plugging in the required chips. We recorded the total communication overhead of a single device from deployment to subsequent usage and take traditional devices as a benchmark to calculate the increased overhead in LEGO. The results are summarized in Fig. 18.

We can see that the remote control has a significant impact at the beginning (within 5 seconds of system working), which introducing 37.66%, 2.12%, and 14.39% of the communication overheads in the scenarios of air monitoring, wireless pulse oximeter, and remote display. This is because the LEGO device is customized on-site through chips plug-and-play, where the gateway needs to write the control flow for each plugged chips into the hierarchical scheduler of the device. We found that the control flow often takes 300$\sim$700 bits for transmission, hence it has a high proportion in communications at the beginning as the application data (chip output) is relatively small. Besides, the air monitoring scenario is most affected as it reads sensor chips at a low frequency (3Hz) with the fewest data for applications.

However, with the system working, the impact decreases dramatically. After 30 seconds, the increased overhead in LEGO drops to 0.77%$\sim$9.33% in those three scenarios, and it further reduces to 0.57%$\sim$3.06% after 2 minutes of system operation. This is benefited by the design of three-layer instruction orchestration and hierarchical scheduler, which handle chips efficiently and minimize the additional overhead in remote chip control. In practical applications, the running time for IoT end devices often much longer than two minutes, and the additional overhead in LEGO will be less than 3% compared with traditional processor-based end devices. Hence, we believe the overhead of shifting chip control to the gateway is acceptable for most applications.

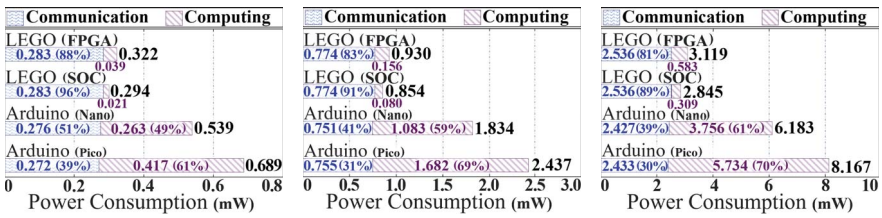
C. Power and Cost Benefit

The "processor-free" architecture of LEGO is helpful in reducing power consumption and costs. For power consumption, we made comparison evaluations with LEGO and traditional end devices. To be fair, all devices using the same Bluetooth (BLE) chip CYBT-423028-02 for data transmission. For traditional end devices, we select Arduino nano ABX00028 with an ATmega4809 MCU and Arduino pico with an ATmega32U4 MCU. We also give all devices the same task in the test $i.e.$, to sample a 12-bit ADC MCP33143-05, and use a micro power analyzer EMK8503 to measure power consumption of each device.

Besides, we also utilize Libero SoC v12.0 [31] to simulate the possible power consumption for LEGO devices if we implement it by System-on-Chip (SoC) with 45nm process. We record the average power consumption on computing ($i.e.$, the overhead for the MCU on Arduino device, or the FPGA or SOC on LEGO device) and communication ($i.e.$, the overhead for data transmission) for each device under different task rates (ADC sample rates). The results are summarized in Fig. 19. For the communication overhead, as the SOC simulation of LEGO only changes the implementation of the USC circuit (Section VI-B) and hierarchical scheduler (Section VI-A), which not change the communication mechanism from the FPGA demo, so we directly take the same value of from the FPGA demo for the SOC simulation.

The results show that LEGO devices consume much less power under the same task. For the FPGA prototype, LEGO reduces 49% and 61% of overall power consumption compared with Arduino Nano and Arduino Pico, respectively. For the SOC version, it is possible to have 57$\sim$72% lower power overhead than Arduino devices. Although LEGO devices have slightly higher power consumption for communication as they need to receive gateway instructions, it have much less power overhead in total.

The root reason is that the LEGO device has a simpler architecture, which directly accesses on-board chips by a simple hardware circuit. In contrast, the Arduino device is actually a microcomputer, which emulates the chip access function in software aspects by running a customized program on its MCU. Hence, LEGO has lower active and quiescent current (especially for the SOC simulation) in running tasks and standby, respectively. Moreover, as LEGO directly accesses chips in hardware aspects, it takes fewer clock cycles to handle tasks on chips than traditional processor-based devices that work by running programs.



(a) 1 KHz (b) 5 KHz (c) 20 KHz

Fig. 19. Power consumption under different task rates (ADC sample rates).

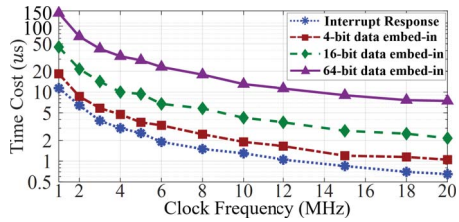


Fig. 20. Time cost for chip cooperation operations on a LEGO device under different clock speeds.

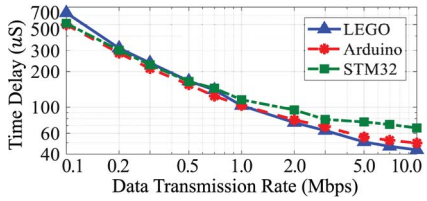


Fig. 21. Time cost to respond to a single task under different communication data rates.

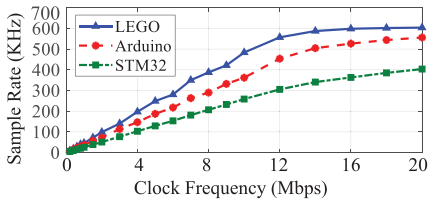


Fig. 22. Task throughput (ADC sample rate) under different clock speeds of the end device.

D. Performance of Chips Cooperation

In Section VI-A, we proposed a novel hierarchical scheduler that handles chips efficiently on the "processor-free" LEGO devices, which also support efficient cooperation for deployed chips without need a round trip to the gateway. Hence, we evaluated the cooperation performance for chips and used an oscilloscope to record the time cost of operation under different clock speeds. We considered three types of cooperation operation, $i.e.$, responding to an interrupt signal from an ADXL 372 accelerometer chip; embedding 4-bit, 16-bit and 64-bit data from a SST25VF010A flash memory chip into the instruction of another flash memory chip. To reduce errors, we repeated the test 20 times and recorded the average value. The results are shown in Fig. 20.

We can see that LEGO can respond to interrupt operations in a few microseconds (μs). Additionally, for short data embedding (4bit, 16bit), it can also be completed within 10 μs when the clock frequency is higher than 4 MHz. For long data (64bit) embedding, it takes relatively more time because the data take two registers in the hierarchical scheduler (Section VI-A). However, the operation can also be completed within a few microseconds as we improve the clock frequency ($\geq$12 MHz). Hence, despite LEGO achieving a lightweight and chip-level re-configurable architecture by decoupling chip control logic uniformly to the gateway, it can also efficiently respond to simple collaborative operations for deployed chips in local and not need a MCU.

E. Task Response and Throughput

Further, we conducted comparative experiments on task response delay and throughput. To mitigate performance limitations in functional chips, we deployed a high-speed 14-bit ADC (ADS7052) on a LEGO device, as well as on two traditional end devices: a STM32-based device with a STM32L431 MCU and an Arduino Nano device with an ATmega4809 MCU for the evaluation.

We first evaluate the time cost to respond to a single task, $i.e.$, the time taken from when the gateway sends out a request to when the target end device completes ADC reading and returns all the data. To control variables, we fix the clock frequency of each end device at 4 MHz and change the communication data rate from 100 kbps to 10 Mbps. The results are recorded in Fig. 21.

We find that when the communication data rate is low, the response time on LEGO devices is slightly higher than that on the other two traditional end devices. This is because LEGO requires 48-bit commands to point to the end device (with 12-bit device code) and activate the scheduler (with 36-bit code, as illustrated in Fig. 10), which is longer than the required code (16 bits) for the Arduino and STM-32 devices. However, when the transmission data rate is high ($\geq$ 1 Mbps), LEGO has a lower time cost for task response. This is because, at high data rates, the impact of wireless transmission is significantly reduced, and LEGO accesses the ADC directly through hardware, which is more efficient than traditional platforms that work by running embedded programs.

Next, we examine the task throughput under different local clock speeds of the end device. To be fair, we set the communication data rate at 10 Mbps, all the end devices continuously

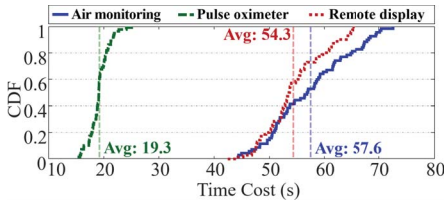


Fig. 23. Time required for end-users to build up a LEGO device for three example applications.

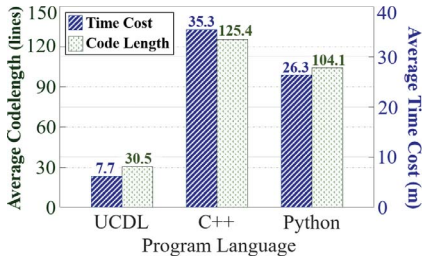


Fig. 24. The UCDL greatly reduces the cost of developing description files for various chips.

read ADC and return data. We change the clock frequency of each end device via an external clock source and record the task throughput (ADC sample rate) in Fig. 21.

We can see that under the same clock frequency, LEGO achieves a higher throughput (sample rates) than traditional end devices. This is because LEGO can schedule the selected functional chips continuously via a simple gateway command, besides, controlling functional chips directly via hardware on the LEGO device is more efficient than running an embedded program on the MCU. Limited by the communication data rate (10 Mbps), LEGO reached a maximum sampling rate when the clock frequency is high ($\geq$ 16Mhz). In summary, LEGO demonstrates good task performance when handling tasks on functional chips.

F. User Experience

1) End-User Experience: To assess the user experience of LEGO for end-users, we recruited 27 unskilled student volunteers as end-users to rebuild the three applications in Section VII-B. We recorded the time required for each volunteer to build a LEGO device by plugging and playing corresponding functional chips. The cumulative distribution function (CDF) of the results is summarized in Fig. 23.

It can be observed that for the LEGO device with one chip deployed (wireless pulse oximeter), end-users can complete the construction in 20 seconds on average. For the LEGO device with three chips deployed (indoor air monitoring, remote display), the average setup time is also less than 1 minute. The results indicate that it is easy for end-users to customize LEGO devices for real-world applications. Hence, with the extremely low technical threshold, we believe LEGO could open the door for non-professional end-users to customize IoT end devices,

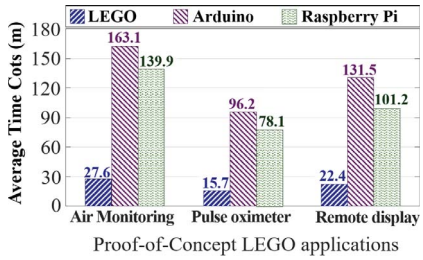


Fig. 25. Time cost of developing the three proof-of-concept real-world applications.

which allowing more users to participate in and thus helpful to accelerate the further development of IoT applications.

2) Developer Experience: In the initial stage, we have developed chip description files for 105 chips with diverse functions. To investigate the convenience of programming in UCDL, we recruit 16 skilled CS (Computer Science) student volunteers as developers to create these files and record the time cost. Each developer undergoes a 40-minute training session in advance to become familiar with UCDL. Fig. 24 plots the average time required to develop a typical chip using different programming languages.

We can find that, compared to C++ and Python, the UCDL can save up to 78% and 71% of development time, respectively. Moreover, the code length is also reduced by 76% and 71%, respectively. The root reason is that traditional programming languages like C++ and Python lack the keywords and logical statements for chip description. In contrast, UCDL is designed directly based on chip characteristics. To add a new chip to the LEGO system, developers only need to write a chip description file and upload it to a LEGO gateway. After that, users can plug the corresponding chip into a LEGO device and the gateway will automatically generate the required instructions based on the uploaded description file to activate chip functions immediately.

Next, we ask our developers to rebuild the three proof-of-concept LEGO applications, $i.e.$, as described in Section VII-B, which contains multiple functional chips on the end device. All developing progress is started equally from the device only deployed with communication modules and compares the average development time cost on LEGO system with two traditional IoT development platforms, $i.e.$, an Arduino R3 platform using C++ and a Raspberry Pi 4B platform using python, as illustrated in Fig. 25.

The results show more significant effects that LEGO reduces 83.2% and 79.5% development time cost compared with Arduino and Raspberry Pi platforms, respectively. The reason stems from the fact that in the LEGO system, developers only need to write the chip description file and upload the file to a LEGO gateway. Then, simply plug in the corresponding chip to a LEGO device, and the gateway can automatically generate the required orchestration logic with meta-operations for chip control. However, in traditional platforms, developers have to make efforts on both chip control and application logic, thus leading to high time costs.

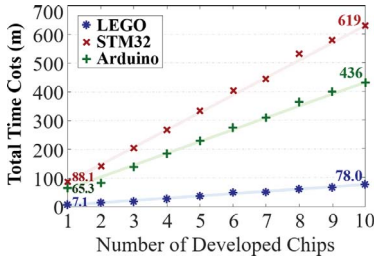


Fig. 26. LEGO greatly reduced the time cost of developing new chips for plug-and-play.

G. Time Cost for Supporting New Chips

Finally, we examined the difficulty of developing new chips on LEGO and smart end devices. We recruited 16 programmers to develop new chips with UCDL in our LEGO system, and using C++ and Python on the selected STM32-based device and Arduino device in Section VIII-E, respectively. The cumulative time cost for supporting 10 new chips is recorded in Fig. 26.

We can see that time cost for both STM32 and Arduino is much longer than on our LEGO device. The essential difference stems from the way of chip control. For LEGO, chips are directly controlled by the gateway given the corresponding UCDL description files, which directly indicate the chip configuration and control logic. However, for traditional platforms such as STM32 and Arduino, chips are controlled by an MCU, and developers need to make mutual adaptations between chips and the MCU through embedded programming. In addition, although STM32 and Arduino devices can be updated wirelessly, developers have to configure the wireless programming at the beginning, so it takes much more time when developing the first chip.

Finally, as the UCDL description files are directly oriented to the chip, so it can be commonly used for different applications. However, chip control programs for STM32 and Arduino are developed for a particular MCU, which further limits the cross-platform application of these programs.

IX. Limitations and Future Works

1) *Limitations of supported functional chips*. In the initial prototype, we mainly focused on designing LEGO with a simple architecture to support chips plug-and-play with commonly used interfaces in IoT and embedded systems, including I^2C, SPI, UART, 1-WIRE, RS485, I^2S, PWM, and PCM. The limitations of functional chips mainly lies in the interface and data rates.

Interface limitations: The supported chips are mainly in serial interfaces, and chips with parallel interfaces are not compatible, such as chips or modules with PCI-E (PCI-Express) and SATA interfaces, which are not supported in the current prototype. To support chips or modules (e.g., Solid State Disk) with these parallel interfaces, the corresponding interface kernel or conversion circuits could be integrated into our future design. However, this may also increase the complexity and power consumption.

Data rates: Limited by low-power design and the interaction data rate between LEGO devices and the gateway, the supported chip data rate in the demo is limited by 10 Mbps. To support chips with higher data rates, a high-speed cache can be integrated into the LEGO device to compensate for the data rate differences between chips and the gateway. To further enhance versatility, in the future, we may study the lightweight design and implementation of data cache for high-data rate chips.

2) *Security Issues*. LEGO could handle threats from UCDL codes and chip description files through its special control architecture, as it forms a whitelist-like validation mechanism. Specifically, since functional chips operate based on a state machine and are controlled by specific underlying signals on the target pin as defined in UCDL, only legitimate UCDL commands and description files can effectively access the target chip. Once UCDL statements are maliciously modified, they cannot drive the functional chips to work, and the anomaly can be detected.

However, similar to many wireless systems, LEGO faces security issues in communication. For this issue, lightweight encryption schemes such as PRESENT [32], KATAN [33], and CP-ABE [34] can be implemented on LEGO devices with no more than 2K logic gates. We could also integrate secure information flow methods [35] to further enhance security for the system.

3) *Error detection*. In the initial stage, LEGO had a simple error checking mechanism that could detect errors directly based on the operational states of functional chips and required no additional components. However, the system could not identify the exact types of errors (e.g., errors in meta-operation sequences, signal timing, and chip connection definitions).

To address this issue, we might design a lightweight error-checking circuit on LEGO devices and integrate domain-specific language (DSL) approaches [36], [37], [38] on the gateway as well, precisely identify errors in the UCDL codes. We might also integrate a hardware-checking circuit into LEGO devices to further enhance the robustness of the system operation. However, this may increase complexity and costs; hence, designing the circuit in a lightweight way is the direction we need to focus on in the future.

X. Conclusion

In this paper, we propose LEGO, a lightweight and chip-level re-configurable architecture for IoT end devices. To support unified plug-and-play for heterogeneous functional chips, we first decouple chip control from IoT end devices to the gateway and design a novel Unified Chip Description Language (UCDL) to access various types of chips uniformly. This makes IoT devices entirely decoupled from specific applications and does not need to make any changes when plugging in new chips. Then, we design a novel signal converting circuit on the end device to dynamically generate underlying signals for chips plug-and-play, and we also built up a layered instruction orchestrator and hierarchical scheduler to minimize communication overhead. The results show that our LEGO device have 49% to 61% lower power consumption compared with traditional IoT end devices, and can respond to chips plug-and-play within 0.13 seconds. With lightweight and easy-to-deploy features,

we believe LEGO is conducive to accelerating the large-scale deployment of IoT applications.

ACKNOWLEDGMENT

The authors sincerely thank the editor and all the reviewers for shepherding our paper.

REFERENCES

[1] B. B. Gupta and M. Quamara, "An overview of Internet of Things (IoT): Architectural aspects, challenges, and protocols," *Concurrency Comput.: Pract. Experience*, vol. 32, no. 21, 2020, Art. no. e4946.
[2] H. D. Kotha and V. M. Gupta, "IoT application: A survey," *Int. J. Eng. Technol.*, vol. 7, no. 2.7, pp. 891–896, 2018.
[3] D. Schoder, "Introduction to the Internet of Things," in *Internet of Things A to Z: Technologies and Applications*, 2018, pp. 1–50. [Online]. Available: https://doi.org/10.1002/9781119456735.ch1
[4] G. Guan, D. Wei, G. Yi, K. Fu, and Z. Cheng, "TinyLink: A holistic system for rapid development of IoT applications," in *Proc. Int. Conf. Mobile Comput. Netw.*, 2017, pp. 383–395.
[5] G. Guan, B. Li, Y. Gao, Y. Zhang, J. Bu, and W. Dong, "TinyLink 2.0: Integrating device, cloud, and client development for IoT applications," in *Proc. 26th Annu. Int. Conf. Mobile Comput. Netw. (MobiCom)*, 2020, pp. 1–13.
[6] R. Manoj and A. Fernandez, "Rapid prototyping IoT end applications using software development kits and add on plugins," in *Proc. IEEE Int. Symp. Nanoelectronic Inf. Syst. (iNIS)*, 2017, pp. 263–267.
[7] F. Yang, N. Matthys, R. Bachiller, S. Michiels, W. Joosen, and D. Hughes, "μPnP: Plug and play peripherals for the Internet of Things," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–14.
[8] K. Kerliu et al., "Secure over-the-air firmware updates for sensor networks," 2019, pp. 97–100.
[9] J. Huang, "The application of reprogramming technology in a wireless sensor network," *J. Phys.: Conf. Ser.*, vol. 2173, no. 1, 2022, Art. no. 012064.
[10] C. Dong and F. Yu, "An efficient network reprogramming protocol for wireless sensor networks," *Comput. Commun.*, vol. 55, no. C, pp. 41–50, Jan. 2015. [Online]. Available: https://doi.org/10.1016/j.comcom.2014.08.017
[11] N. Jevtic and V. Drndarevic, "Development of smart transducers compliant with the IEEE 1451.4 standard," in *Proc. IEEE Int. Symp. Instrum. Control Technol. (ISICT)*, 2012, pp. 126–131.
[12] K. Malar and N. Kamaraj, "Development of smart transducers with IEEE 1451.4 standard for industrial automation," 2014, pp. 111–114.
[13] T. Licht, "The IEEE 1451.4 proposed standard," *IEEE Instrum. Meas. Mag.*, vol. 4, no. 1, pp. 12–18, 2001.
[14] D. Borycki, "Rapid IoT development with Azure IoT central," *MSDN Mag.*, vol. 33, no. 12, pp. 34–36, 39–41, 2018.
[15] K. Tanaka and H. Higashi, "Rapid IoT software development," in *Proc. Int. Conf. Comput. Sci. Appl.*, 2017, pp. 733–742.
[16] G. Tanganelli, C. Vallati, and E. Mingozzi, "Rapid prototyping of IoT solutions: A developer's perspective," *IEEE Internet Comput.*, vol. 23, no. 4, pp. 43–52, Jul./Aug. 2019.
[17] T. Leesatapornwongsa, A. Sengupta, M. S. Ardekani, G. Petri, and C. A. Stuardo, "Transactuations: Where transactions meet the physical world," *ACM Trans. Comput. Syst.*, vol. 36, no. 4, pp. 1–31, May 2020.
[18] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, "Firmware over-the-air programming techniques for IoT networks—A survey," *ACM Comput. Surv.*, vol. 54, no. 9, pp. 1–36, 2021.
[19] W. Badawy, A. Ahmed, S. Sharf, R. A. Elhamied, M. Mekky, and M. A. Elhamied, "On flashing over the air 'FOTA' for IoT appliances—An ATMEL prototype," in *Proc. IEEE 10th Int. Conf. Consum. Electron. (ICCE-Berlin)*, 2020, pp. 1–5.
[20] A. Ahmed et al., "Wireless ATMEL AVR in-circuit serial programmer based on Wi-Fi and Zigbee," in *Proc. 16th Int. Comput. Eng. Conf. (ICENCO)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 187–190.
[21] X. He, S. Alqahtani, R. Gamble, and M. Papa, "Securing over-the-air IoT firmware updates using blockchain," in *Proc. Int. Conf. Omni-Layer Intell. Syst.*, 2019, pp. 164–171.
[22] P. Thakur, V. Bodade, A. Achary, M. Addagatla, N. Kumar, and Y. Pingle, "Universal firmware upgrade over-the-air for IoT devices with security," in *Proc. 6th Int. Conf. Comput. Sustain. Global Develop. (INDIACom)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 27–30.
[23] A. W. Malik, A. U. Rahman, A. Ahmad, and M. M. D. Santos, "Over-the-air software-defined vehicle updates using federated fog environment," *IEEE Trans. Netw. Service Manag.*, vol. 19, no. 4, pp. 5078–5089, Dec. 2022.
[24] N. Matthys et al., "μPnP-Mesh: The plug-and-play mesh network for the Internet of Things," in *Proc. IEEE 2nd World Forum Internet Things (WF-IoT)*, Piscataway, NJ, USA: IEEE Press, 2015, pp. 311–315.
[25] J. Hester and J. Sorber, "Flicker: Rapid prototyping for the batteryless Internet-of-Things," in *Proc. 15th ACM Conf. Embedded Netw. Sensor Syst.*, 2017, pp. 1–13.
[26] M. Bassoli, V. Bianchi, and I. De Munari, "A plug and play IoT Wi-Fi smart home system for human monitoring," *Electronics*, vol. 7, no. 9, pp. 1–13, Jan. 2018.
[27] D. D. McCracken and E. D. Reilly, "Backus–Naur form," 2003, doi: 10.5555/1074100.1074155.
[28] Wikipedia, "Backus–Naur form," Wikipedia. Accessed: Oct. 13, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form#Overview
[29] "QScriptEngine class," QT. Accessed: Dec. 10, 2021. [Online]. Available: https://doc.qt.io/qt-5/qscriptengine.html
[30] S. Elder, "The real cost for a custom IC," Planet Analog. Accessed: Sep. 25, 2022. [Online]. Available: https://www.planetanalog.com/the-real-cost-for-a-custom-ic/
[31] "Libero SoC v12.0 and later," Microsemi, 2021. Accessed: May 16, 2022. [Online]. Available: https://www.microsemi.com/product-directory/vectorblox-ai/5598-libero-soc
[32] S. B. Sadkhan and A. O. Salman, "A survey on lightweight-cryptography status and future challenges," in *Proc. Int. Conf. Adv. Sustain. Eng. Appl.*, 2018, pp. 105–108.
[33] S. Surendran, A. Nassef, and B. D. Beheshti, "A survey of cryptographic algorithms for IoT devices," in *Proc. IEEE Long Island Syst., Appl. Technol. Conf. (LISAT)*, 2018, pp. 1–8.
[34] E. R. Naru, H. Saini, and M. Sharma, "A recent review on lightweight cryptography in IoT," in *Proc. Int. Conf. I-SMAC (IoT in Social, Mobile, Analytics Cloud) (I-SMAC)*, 2017, pp. 887–890.
[35] X. Li et al., "Caisson: A hardware description language for secure information flow," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 109–120, 2011.
[36] L. Ryzhyk et al., "{User-guided} device driver synthesis," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2014, pp. 661–676.
[37] A. Serrano and J. Hage, "A compiler architecture for domain-specific type error diagnosis," *Open Comput. Sci.*, vol. 9, no. 1, pp. 33–51, 2019.
[38] "Validation in a domain-specific language," Microsoft, 2011. [Online]. Available: https://learn.microsoft.com/zh-tw/previous-versions/visualstudio/visual-studio-2010/bb126413(v=vs.100)?redirectedfrom=MSDN

Chong Zhang received the Ph.D. degree in computer science and engineering from the University of Electronic Science and Technology of China (UESTC), in 2022. He is a Lecturer with Southwest Petroleum University, and now serves as a Postdoctoral Fellow with UESTC. His research interests include Internet of Things, computer systems, and networking. He has published several papers in the prestigious conferences and journals, including ASPLOS, MobiCom, NSDI, and the ACM TOSN.

Songfan Li (Member, IEEE) received the Ph.D. degree in computer science and engineering from the University of Electronic Science and Technology of China (UESTC), in 2022. His research interests include wireless systems, embedded computing, and backscatter systems. He has published several papers in the prestigious conferences and journals, including MobiCom, ASPLOS, NSDI, and the IEEE TRANSACTIONS ON NETWORKING.

Yihang Song received the Ph.D. degree in computer science and engineering from the University of Electronic Science and Technology of China (UESTC), in 2022. He is a Postdoctoral Fellow with the University of Electronic Science and Technology of China. His research interests include Internet of Things, low-power wireless communication, and networking. He has published several papers in the prestigious conferences and journals, including NSDI, MobiCom, ASPLOS, and the ACM TOSN.

Qianghe Meng received the B.Eng. degree in electronic and information engineering from the University of Electronic Science and Technology of China, and the M.Eng. degree in electronic and electrical engineering from the University of Glasgow. He is working toward the D.Eng. degree in computer science and technology with the University of Electronic Science and Technology of China. His current research interests include wireless sensing, mobile computing, and statistical analysis.

Li Lu (Member, IEEE) received the Ph.D. degree from the Key Lab of Information Security, Chinese Academy of Science, in 2007. He is a Professor with the School of Computer Science and Engineering, University of Electronic Science and Technology of China. His research interests include wireless systems for low power and high performance computing, IoT and industrial control systems, and security in wireless systems. He is a member of ACM.

Hongzi Zhu (Senior Member, IEEE) received the Ph.D. degree in computer science from Shanghai Jiao Tong University, in 2009. He is a Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include Internet of Things and mobile computing. He received the Best Paper Award from the IEEE Globecom 2016. He is an Associate Editor for the IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY and the IEEE INTERNET OF THINGS JOURNAL. He is a IEEE VTC Distinguished Lecturer and a member of the IEEE Computer Society, the IEEE Communication Society, and the IEEE Vehicular Technology Society. For more information, please visit http://lion.sjtu.edu.cn.

Xin Wang received the Ph.D. degree from the University of Edinburgh, in 2013. His research interests include big data, graph databases, and network science. He has published several papers in the prestigious conferences and journals, including SIGMOD, VLDB, ICDE, CVPR, and the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, and received ICDE 2014 Best Paper Runner-up Award, WISE 2019 Best Paper Runner-up Award, etc.