The BNF grammar and Coding for HTDL

1 The BNF grammar of HTDL

To better present the design of our Hierarchical Task Description Language (HTDL), in this appendix section, we introduce its BNF grammar. The basic rule of BNF is "<symbol>::=__expression__", where the "symbol" on the left of "::=" is interpreted by the expression on the right. For a better understanding, we first summarize the symbols used in HTDL as follows:

1)"<>" represents the non-terminal symbol. In this paper, its contents are syntactic variables of HTDL.
2)"::=" is used to connect the left and right sides of a sentence, and means that the variable on the left is composed of the the content on the right.
3)"|" means or. For example, X|Y|Z means the content is selected from one of X, Y, and Z
4)"[]" means that the content is optional. For example, <A>::=<B>[<C>] means that <B> is necessary for the expression of <A>, while <C> is optional and can be defaulted under certain conditions.

Based on this foundation, we introduce the BNF grammar of HTDL at two layers, *i.e.*, App. task layer and chipset layer, as follows.

1.1 App. Task Layer

The App. task layer syntax is used to describe the data acquisition, data processing, and data upload logic of the device. This layer syntax contains eight keywords, as follows.

1.1.1 Task

The App. task definition keyword, in the syntax format `taskXXX`, defines an App. task named 'XXX'. Its BNF grammar can be summarized as:

▷**HTDL Syntax:** <task definition>::=task<task name>
1) Parameter for task name: <task name>::= <identifier>
2) Parameter for identifier[1]: <identifier>::= <letter> (<letter> | <digit> | "_")
3) Parameter for letter: <letter> ::= A | B | C | $\cdots$ | Z | a | b | c | $\cdots$ | z
4) Parameters for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of above signs [1] are as follows.

[1]<identifier> defines the naming convention for tasks in the program. Identifier must start with a letter (uppercase or lowercase) and can be followed by any number of letters, digits, and underline. Therefore, the readability and uniqueness of the identifier are guaranteed.

1.1.2 Call

Sensor control task calling keyword to activate the sensor control circuit to perform a specific sensor control task to acquire raw data. The syntax format is: `call(XXX)`, which indicates a call to the sensor control task logic name 'XXX'. A App.task can execute syntax call multiple times to realize the control and data acquisition of multiple sensor chips. The acquired data will be stored in the raw data register group in the sensor control circuit in sequence. We define the keyword $r[x]$ to represent the x-th 8-bit raw data. Its BNF grammar can be summarized as:

▷**HTDL Syntax:** <call statement>::= call<task name>
1) Parameter for task name: <task name>::= <identifier>
2) Parameter for identifier: <identifier>::= <letter> (<letter> | <digit> | "_")
3) Parameter for letter: <letter> ::= A | B | C | $\cdots$ | Z | a | b | c | ... | z
4) Parameters for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

1.1.3 Combination

The combination keyword. In some sensor chips with high sensing accuracy, data with a length of more than eight bits will be split and stored in multiple registers. In order to pre-process the sensor data, it is necessary to obtain the complete sensor data, which is the role of comb keyword. The syntax format is `comb(n,signed/unsigned, r[i][a:b],r[j][c:d],...)`, indicating that the $n-th$ sensor data is a signed/unsigned number, and its value is the splice of the $a-bits$ to $a-bits$ data of the $i-th$ data and the $c-bits$ to $d-bits$ data of the $j-th$ data in the sensor output buffer. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <comb command>::=comb(<sensor data index>, <data type>, [<register range>])
1) Parameter for sensor data index[1]: <sensor data index>::= 1 | 2 | 3 $\cdots$
2) Parameter for data type[2]: <data type>::= signed | unsigned
3) Parameter for register range[3]: <register range> ::= <register> [<start bit>:<end bit>]

The annotation of this statement are as follows.

[1]<sensor data index> is used to specify the index of sensor data and select to obtain specific sensor data in the sensor list.

[2]<data type> defines two data type: signed and unsigned. The signed data type indicates tthat the value can be positive, negative and zero, while the unsigned data type can only be positive or zero.

[3]<register range> defines the syntax structures of register, including a register identifier <register> and an optional bit range, which is specified by the start bit <start bit> and the end bit <end bit>. The bit range selects a specific bit segment in the register, e.g. [2 : 4] indicates selecting the data from $3-rd$ bit to $5-th$ bit if the register has eight bits (bit count starts at 0).

1.1.4 Between and Outside

Two interval judgment keywords, the syntax format is `between/outside(A,B,0/1)`, which means to judge whether the n_{th} sensing data is inside or outside the interval $[A, B)$. If the judgment result is true, the response pin corresponding to this data is set to low level (0) or high level (1), and its valid flag bit is set to 1, donating that this data needs to be uploaded. If the judgment result is false, the response pin is set to the opposite level and the valid bit is set to 0. The gateway is usually only interesting in the sensor data within a specific numerical interval, thereby the interval judgment function can save the system uplink communication resources and give the device real-time response capability without losing the sensing performance. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <interval check command>::=<between command> | <outside command>
1) Parameter for between command[1]: <between command>::= between(<sensor data index>, <interval>, <interval result>)
2) Parameter for outside command[2]: <outside command>::= outside(<sensor data index>,<interval>,<interval result>)
3) Parameter for sensing data index[3]: <sensor data index> ::= <integer>
4) Parameter for interval[4]: <interval> ::= <number>,<number>
5) Parameter for interval result[5]: <interval result> ::= <binary value>
6) Parameter for integer: <integer> ::= <digit> {<digit>}
7) Parameter for number: <number> ::= <integer>
8) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of this statement are as follows.

[1]<between command> is used to judge whether the data specified by the sensing data index is between two number.

[2]<outside command> is used to judge whether the data outside the specified interval $[A, B\}$.

[3]<sensor data index> defines how to reference the index of the sensor data, which is now an integer.

[4]<interval> defines a number interval that consists of two numbers divided by a comma, indicating a closed interval or an open interval.

[5]<interval result> defines the output result of the interval judgment, which is usually a binary value: 1 indicates true (the data is within or outside the interval), and 0 indicates false.

1.1.5 Sub

Difference calculation keyword, the syntax format is `sub(n)`, which means calculate the difference of the n_{th} sensor data compared to the last sent data. This keyword can be utilized in conjunction with the interval judgment keyword, adopting the syntactic structure `sub-between/sub-outside(n,A,B)`, which designates that an interval judgment of $[A, B)$ is applied to the differential value between the $(n-1)_{th}$ and n_{th} samples. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <sub command>::= sub (<sensor data index>)

1) Parameter for difference interval check command[1]: <sub interval check command>::= <sub between command> | <sub outside command>
2) Parameter for sub and between conjunction command[2]: <sub between command>::= sub-between(<sensor data index>,<interval>)
3) Parameter for sub and outside conjunction command[3]: <sub outside command> ::= sub-outside(<sensor data index>,<interval>)
4) Parameter for sensing data index: <sensor data index> ::= <integer>
5) Parameter for interval: <interval> ::= <number>,<number>
6) Parameter for number: <number> ::= <integer>
7) Parameter for integer: <integer> ::= <digit> {<digit>}
8) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of this statement are as follows.

[1]<sub interval check command> is used to judge whether the difference inside/outside a specified interval. Its a combining command which can be <sub between command> or <sub outside command>.

[2]<sub between command> is used to judge whether the difference inside a specified interval $[A, B)$.

[3]<sub outside command> is used to judge whether the difference outside a specified interval $[A, B)$.

1.1.6 Avg

Average calculation keyword, used to realize the mean value filtering of sensor data, its syntax format is `avg(n,m,k)`, which means calculate the average value of the n_{th} sensor data, the length of the queue is m, and upload the average value every k samples. Similar to the **sub** keyword, this keyword can also be used in combination with the interval judgment keyword with the syntax `avg-between/avg-outside(n,m,k,A,B)`. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <avg command>::= avg (<sensor data index>,<queue length>,<sampling interval>)

1) Parameter for average interval check command[1]: <avg interval check command>::= <avg between command> | <avg outside command>
2) Parameter for avg and between conjunction command[2]: <avg between command>::= avg-between(<sensor data index>,<queue length>,<sampling interval>,<interval>)
3) Parameter for avg and outside conjunction command[3]: <avg outside command> ::= avg-outside(<sensor data index>,<queue length>,<sampling interval>,<interval>)
4) Parameter for sensing data index: <sensor data index> ::= <integer>
5) Parameter for queue length: <queue length> ::= <integer>
6) Parameter for sample interval: <sample interval> ::= <integer>
7) Parameter for interval: <interval> ::= <number>,<number>
8) Parameter for number: <number> ::= <integer>
9) Parameter for integer: <integer> ::= <digit> {<digit>}
10) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of this statement are as follows.

[1]<avg interval check command> is used to judge whether the average inside/outside a specified interval. Its a combining command which can be <sub between command> or <sub outside command>.

[2]<avg between command> is used to judge whether the average inside a specified interval $[A, B)$.

[3]<avg outside command> is used to judge whether the average outside a specified interval $[A, B)$.

1.1.7 Upload

Data upload keyword, used to set the number of bits of each sensor data to be uploaded after the execution of this App. task. The syntax format of this keyword is `upload(a,b,c)`, which means to upload the low a bit of the first sensing data, the low b bit of the second sensing data, and so on. As the bit lengths of different sensor data are different, the device will expand each sensor data to the maximum bit length that can be processed by the UEST circuit after executing the comb command. The function of this keyword is to trim the result of data processing, thus preventing the uplink communication data volume from increasing due to the expansion of data bits. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <upload command>::= upload (<bit assignment list>)
1) Parameter for bit assignment list[1]: <bit assignment list>::= <bit assignment> (<bit assignment>)*
2) Parameter for bit assignment[2]: <bit assignment>::= <integer>
3) Parameter for integer: <integer> ::= <digit> {<digit>}
4) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of this statement are as follows.

[1]<bit assignment list> defines a list containing a series of bit assignments. each bit assignment in list consists of <bit assignment>, multiple bit assignments are divided by spaces, and can contain zero or more bit assignments.

[2]<bit assignment> defines the assignment of single bit, indicating the bit assignment or mapping of the data in the upload process.

1.2 Chipset Layer

The Chipset layer of HTDL is used to describe the Variable sensor control logic of the sensor device in a unified way. After reviewing a large number of sensor chip datasheets, it is found in this paper that, although the control logic varies from one sensor chip to another, most of the sensors use a few standardized bus communication protocols, such as SPI, I2C, etc. In addition to the interaction of bus signals, some sensor chips also output event signals as a way to indicate the occurrence of events in the environment. This layer syntax contains eight keywords, as follows.

1.2.1 Sensor

Sensor control task definition keyword, the syntax format is `sensor name`, it means to define a sensor control task named name. Its BNF grammar can be summarized as:

▷**HTDL Syntax:** <sensor task definition>::=sensor<task name>
1) Parameter for task name: <task name>::= <identifier>
2) Parameter for identifier: <identifier>::= <letter> (<letter> | <digit>)*
3) Parameter for letter: <letter> ::= $A \mid B \mid C \mid \cdots \mid Z \mid a \mid b \mid c \mid \cdots \mid z$
4) Parameters for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

1.2.2 Busconfig

Bus interface configuration keyword, the syntax format is `busconfig(n,type)`, indicates that the sensor uses the n_{th} set of pins as the bus interface, the bus type is `type`. In our prototype system, the bus types that can be specified are SPI and I2C, which can realize the data interaction with most commonly used sensor chips. Its BNF grammar can be summarized as:

▷**HTDL Syntax:** <busconfig command>::=busconfig(<pin group>, <bus type>)
1) Parameter for pin group number[1]: <pin group>::= <integer>
2) Parameter for bus type[2]:<bus type>::= SPI | I2C
3) Parameter for integer: <integer> ::= <digit> {<digit>}
4) Parameters for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of above signs [1] are as follows.

[1]<pin group> defines the number of a group of pin, which usually used for specified functions or interfaces on a microcontroller or other hardware device.

[2]<bus type> defines the type of communication bus s used, which can be SPI (Serial Peripheral Interface) or I2C (Integrated Circuit Bus).

1.2.3 Pinconfig

Single pin configuration keyword. Configuration of the input pin syntax format for `pinconfig(n,input,pull-up/pull-down)`, indicating that pin n configured for pull-up/pull-down input mode. Configuration of the output pin syntax format for `pinconfig(n,output,OD/PP)`, said pin n configured for open-drain/push-pull output mode. The single pin configuration keyword enables the device to flexibly configure the pin electrical characteristics, enhancing compatibility with different sensor chips. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <pinconfig command>::= pinconfig(<pin number>,<pin mode>,<pin configuration>)
1) Parameter for pin number[1]: <pin number>::= <integer>
2) Parameter for pin mode[2]: <pin mode>::= input/output
3) Parameter for input pin configuration[3]: <pin configuration input> ::= (pull-up | push-down)
4) Parameter for output pin configuration[4]: <pin configuration output> ::= (open-drain | push-pull)
5) Parameter for integer: <integer> ::= <digit> {<digit>}
6) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of this statement are as follows.

[1]<pin number> defines the integer of pin using to uniquely identify a pin on the hardware.

[2]<pin mode> defines the work mode of the pin, which can be input mode, used to receive signals, or output mode, used to send signals.

[3]<pin configuration input> defines the internal pull-up or pull-down resistor configuration of the pin when the pin is configured as input mode. It can be either pull-up or pull-down.

[4]<pin configuration output> defines the output type of the pin when the pin is configured as output mode, which can be open-drain or push-pull.

1.2.4 Read

Bus read keyword, with syntax format `read(n)`, indicates to read n bytes data from inside the sensor via bus. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <read command>::= read(<byte count>)
1) Parameter for byte count[1]: <byte count>::= <integer>
2) Parameter for integer: <integer> ::= <digit> {<digit>}
3) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of this statement are as follows.

[1]<byte count> defines the number of bytes to be read, which is an integer representing the amount of data to be subsequently read from the bus.

1.2.5 Write

Bus write keyword, the syntax format is `write(0xAA)`, means through the bus to write hexadecimal data to the sensor 0xAA. The write keyword supports writing unsigned 8-bit data and is compatible with both decimal and hexadecimal writing formats. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <write command>::= write(<hex data>)
1) Parameter for hexadecimal data[1]: <hex data>::= <hex digit> {<hex digit>}
2) Parameter for hexadecimal digit: <hex digit> ::= <digit> {<digit>}
3) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f

The annotation of this statement are as follows.

1.2.6 Pinread

Pin level read keyword, syntax format is `pinread(n)` , means read the pin level of pin n. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <pinread command>::= pinread(<pin number>)
1) Parameter for pin number[1]: <pin number>::= <integer>
2) Parameter for integer: <integer> ::= <digit> {<digit>}
3) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of this statement are as follows.

[1]<pin number> defines the integer of pin using to uniquely identify a pin on the hardware.

1.2.7 Pinwrite

Pin level configuration keyword, syntax format is `pinwrite(n,0/1)` , means set the pin level of pin n to low level (0) or high level (1). Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <pinwrite command>::= pinwrite(<pin number>,<pin state>)
1) Parameter for pin number[1]: <pin number>::= <integer>
2) Parameter for pin state[2]: <pin state>::= 0 | 1
3) Parameter for integer: <integer> ::= <digit> {<digit>}
4) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of this statement are as follows.

[1]<pin number> defines the integer of pin using to uniquely identify a pin on the hardware.
[2]<pin state> is used to configure the level of the specified pin, which can be 0 (Low Level) or 1 (High Level).

1.2.8 Delay

Delay keyword, the syntax format is `delay(a,b)` , indicating that the amount of delay is a, and the time unit is b. Users can choose from 1 to 4 crossover modes, representing time units of ”µs”, ”ms”, ”s” and ”min” respectively. For example, at a clock frequency of 1MHz, `delay(120,2)` indicates a delay time of 120 ms. Different sensors have different internal operation timings, and thus devices have different waiting intervals between two operations on the bus, so users can employ this keyword to wait for the transfer of the chip's internal state. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <delay command>::= delay(<duration>,<time unit>)
1) Parameter for delay length[1]: <duration>::= <integer>
2) Parameter for delay time unit[2]: <time unit>::= microseconds | milliseconds | seconds | minutes
3) Parameter for integer: <integer> ::= <digit> {<digit>}
4) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of this statement are as follows.

[1]<duration> defines the specific duration of the delay, which is an integer representing the numerical value of the delay.
[2]<time unit> defines the time unit of the delay, which can be microseconds, milliseconds, seconds, or minutes. microseconds indicates that the duration is in microseconds. milliseconds indicates that the duration is in milliseconds. seconds indicates that the duration is in seconds. minutes indicates that the duration is in minutes.

1.2.9 Wait

Event wait keyword, the syntax format is `wait(n,0/1,a,b)` , that is, wait for the level state of the nth pin to change to low or high, a and b. a and b have the same meaning as the parameters in the delay keyword, indicating the upper limit of the waiting time. Some sensor chips are inherently equipped with the ability to preprocessing sensor data. Their internal trigger conditions can be flexibly configured to suit various needs. When the sensor data satisfies these predefined trigger conditions, the chip seamlessly communicates this occurrence to the App. components through a designated pin. Upon executing the wait command, the device attentively monitors the level of the target pin. It

temporarily halts the task execution until the pin's level aligns precisely with the target condition or if the waiting duration surpasses the preset upper limit. Once either of these conditions is met, the device seamlessly resumes the suspended task, allowing it to continue its execution without interruption. Its BNF syntax can be summarized as:

▷**HTDL Syntax:** <wait command>::= wait(<pin number>,<pin state>,<duration>,<time unit>)
1) Parameter for pin number[1]: <pin number>::= <integer>
2) Parameter for pin state[2]: <pin state>::= 0 | 1
3) Parameter for delay length[3]: <duration>::= <integer>
4) Parameter for delay time unit[4]: <time unit>::= microseconds | milliseconds | seconds | minutes
5) Parameter for integer: <integer> ::= <digit> {<digit>}
6) Parameter for digit: <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The annotation of this statement are as follows.

[1]<pin number> defines the integer of pin using to uniquely identify a pin on the hardware.

[2]<pin state> is used to configure the level of the specified pin, which can be 0 (Low Level) or 1 (High Level).

[3]<duration> defines the maximum duration to wait for a specific event to occur, which is an integer representing the numerical value of the delay.

[4]<time unit> defines the time unit of the delay, which can be microseconds, milliseconds, seconds, or minutes. microseconds indicates that the duration is in microseconds. milliseconds indicates that the duration is in milliseconds. seconds indicates that the duration is in seconds. minutes indicates that the duration is in minutes.

1.3 Summary

Based on the above syntax design of the App. task layer and chipset layer, we can achieve the universal atomic expression of devices' heterogeneous task logic. Compared to programming on microprocessor platforms using other high-level programming languages (e.g., C, C++), users do not need to learn the underlying principles of microprocessors, experimental environments, language libraries, and other complex development knowledge when using HTDL for programming. Instead, they only need to consult and understand the sensor chip's datasheet to easily write sensor control and off-chip task logic. This allows users to focus more on application development on the gateway side, making IoT application development more efficient.

2 HTDL Syntax Coding

In this section, we introduce operation process of HTDL statements,*i.e.*, how HTDL meta-operations are lowered to instructions, and how these instructions are lowered to underlying signals.

2.1 Overview

In HTDL programming, each instruction is directly linked to a meta-operations of a chip. Specifically, for a given application, users will craft a profile that encapsulates the control drivers for the chipset installed on the LEGO+ device, alongside the App. control logic.For a HTDL statement, the gateway automatically maps its keywords and variables into a predefined code and connects them in series to form an independent instruction. This results in the formation of an indivisible atomic operations (AtOp). Subsequently, multiple AtOps are concatenated to form the control logic for a specific task or chipset. The coding details are presented in Section.2.2.

2.2 Coding Details

In a description file, the instructions for pin configurations (keywords: *PIN*) and logic control (keywords: *DW*, *DR*, *CW*, *SR*) are converted into gateway instructions. In contrast, the instructions for data formats decoding (keywords: *DF*) is only running on the gateway, so it does not need a coding map, and the gateway directly extracts the content for data display, we discuses it in the end of this section. The encoding scheme of the 5 gateway instructions is as follows:

2.2.1 Instruction Encoding of App. Task Layer

In the compiler design of App. task layer, each instruction begins with a 5-bit keyword code and the instructions with different keywords have different suffix encoding formats. For the convenience of readers, this appendix chapter

Table 1: HTDL Syntax Coding

	Keywords	Code	Description
Coding For Keywords	call	00001	Call a sensor control task logic
	comb	00010	Combination the sensor data for a App. task
	between	00011	Interval inside judgment for sensor data
	outside	00100	Interval outside judgment for sensor data
	sub	00101	Calculate the difference of sensor data
	avg	00110	Calculate the average of sensor data
	upload	00111	Set the bits number of uploaded sensor data
	busconfig	01000	Define bus interface for a chip
	pinconfig	01001	Set the working mode for a pin
	read	01010	Read data from a chip
	write	01011	Write data to a chip
	pinread	01100	Read pin level from a chip
	pinwrite	01101	Set the delay interval for a chip
	delay	01110	Define data format for a chip
	wait	01111	Suspend the execution of sensor tasks
	sub-between	10000	Interval inside judgment for difference of sensor data
	sub-outside	10001	Interval outside judgment for difference of sensor data
	avg-between	10010	Interval inside judgment for average of sensor data
	avg-outside	10011	Interval outside judgment for average of sensor data
Coding For Time Units	Pin Types	Code	Description
	milliseconds	001	Set the time unit to milliseconds
	microseconds	010	Set the time unit to microseconds
	seconds	011	Set the time unit to seconds
	minutes	100	Set the time unit to minutes
Coding For Data Types	Pin Functions	Code	Description
	signed	01	Set the data type to signed
	unsigned	10	Set the data type to unsigned

provides the coding map for UCDL parsing, summarized in Tables 1 and 2. The encoding scheme of the 7 gateway instructions is as follows:

Call instruction(keywords: *call*): The coding for call instruction consists of a keyword and a three bits sensor control task number. This instruction is used to call the specified sensor control task to obtain the output of the sensor.

Combination instruction(keywords: *comb*): The coding for comb instruction contains 5 fields, which are the keyword, sensor data index (2 bits), data type (2 bit), position mask of the data involved in the combination (8 bits), and effective bit mask of each data (n×8 bits). Its function is to combine n 8-register data output by the sensor into a single 24-bit sensor data. In the prototype system designed in this paper, there are at most 8 registers that can participate in the combination in each off-chip task, and at most 3 24-bit sensor data can be generated, which can meet the data processing conditions of most sensors.

Between instruction(keywords: *between*) The coding for between instruction contains 5 fields, which are the keyword, sensor data index (2 bits), left threshold (24 bits), right threshold (24 bits), and response pin level (2 bit). Its function is to make interval judgment on the specified sensor data and control the response pin level according to the judgment result.

Outside instruction(keywords: *outside*): The code format for outside instruction is the same as that of "between". The keyword encoding of the two is different, and the judgment results are opposite.

Sub instruction(keywords: *sub*): The code format for sub instruction consists of a keyword and a sensor data index (2 bits). Its function is to calculate the difference of the specified sensor data.

Avg instruction(keywords: *avg*): The code format for avg instruction contains 4 fields, which are the keyword, sensor data index (2 bits), queue length (2 bits), and upload interval (4 bits). Its function is to calculate the average value of sensor data and specify the interval for uploading the average value.

Upload instruction(keywords: *upload*): The code format for upload instruction contains 2 fields, which are the keyword and the number of bits required to upload each sensor data (15 bits). Its function is to remove invalid bits in the sensor data expanded to 24 bits and reduce the amount of uploaded data.

Table 2: HTDL Syntax Coding

	Keywords	Code	Description
Coding For Bus Type	SPI	01	Set Connections for SPI Bus logic
	I2C	10	Set Connections for I2C Bus
Coding For Pin Mode	Keywords	Code	Description
	input	01	Set input mode for pin
	output	10	Set output mode for pin
Coding For Pin Configuration Input	Keywords	Code	Description
	pull-up	01	Set connection as input with pull-up
	push-down	10	Set connection as input with push-down
Coding For Pin Configuration Output	Keywords	Code	Description
	open-drain	01	Set connection as output with open-drain
	push-pull	10	Set connection as output with push-pull
Coding For Digit	Keywords	Code	Description
	0	1111	Set the digit to 0
	1	0001	Set the digit to 1
	...	...	...
	8	1000	Set the digit to 8
	9	1001	Set the digit to 9
Coding For Hex Digit	Keywords	Code	Description
	0	10000	Set the hexadecimal digit to 0
	1	00001	Set the hexadecimal digit to 1
	...	...	...
	E	01110	Set the hexadecimal digit to E
	F	01111	Set the hexadecimal digit to F
Coding For Letter	Keywords	Code	Description
	a	000001	Set the Letter to a
	b	000010	Set the Letter to b
	...	...	...
	z	011010	Set the Letter to z
	A	011011	Set the Letter to A
	B	011100	Set the Letter to B
	...	...	...
	Z	110100	Set the Letter to Z

2.2.2 Instruction Encoding of Chipset Layer

Similar to App. task instructions, each instruction in the chipset layer begins with a 5-bit keyword code and the instructions with different keywords have different suffix encoding formats. For the convenience of readers, this appendix chapter provides the coding map for UCDL parsing, summarized in Tables 1 and 2. The encoding scheme of the 8 gateway instructions is as follows:

Busconfig instruction(keywords: $busconfig$**):** The code format for busconfig instruction contains 3 fields, which are the keyword, pin group number (2 bits), and bus type (2 bit). Its function is to configure a specified group of pins as the bus interface required by the sensor chip.

Pinconfig instruction(keywords: $pinconfig$**):** The code format for pinconfig instruction contains 4 fields, which are the keyword, pin number (4 bits), input/output type (2 bit), and electrical characteristics (2 bit). Its function is to configure the working mode of a single pin, which can be used for pin level reading, writing, and waiting.

Read instruction(keywords: *read***):** The code format for read instruction contains 2 fields, which are the keyword and the number of data to be read (3 bits). Its function is to read the specified amount of data inside the sensor chip through the bus interface.

Write instruction(keywords: *write***):** The code format for write instruction contains 2 fields, which are the keyword and the data to be written (8 bits). Its purpose is to write one byte of data to the sensor chip through the bus interface.

Pinread instruction(keywords: *pinread***):** The code format for pinread instruction contains 2 fields, which are the keyword and the pin number (4 bits) whose level is to be read. Its function is to read the level status of the specified pin.

Pinwrite instruction(keywords: *pinwrite***):** The code format for pinwrite instruction contains 3 fields, which are the keyword, the pin number (4 bits) of the level to be configured, and the level high or low (1 bit). Its function is to

configure the level status of the selected pin.

Delay instruction(keywords: *delay*): The code format for delay instruction contains 3 fields, which are the keyword, time length (8 bits), and time unit (2 bits). This command is used to suspend the terminal task for a specified time length.

Wait instruction(keywords: *wait*): The code format for wait instruction contains 5 fields, which are the keyword, waiting pin number (4 bits), target level (1 bit), time length (8 bits), and time unit (3 bits). Its function is to make the terminal suspend the execution of the task until the pin level meets the target state or the waiting time expires and then continue to execute the task.