A Customized Real-time Compilation for Motion Control in Embedded PLCs

Huifeng Wu, Yi Yan, Danfeng Sun, Rene Simon

***Abstract*—General PLCs are difficult to adapt to changeable applications, especially for those with special motion control; this has seriously affected the development efficiency. This paper presents the concept of the embedded PLC (ePLC), whose hardware structure can be customized according to actual requirements. We proposed a three-layer architecture, and its customizable application layer could be compiled in real time. Correspondingly, the PLC program was divided into an engine program, control program, and customizing program. The description and compilation method of the customizing program was provided. We presented a customized winding machine language (CWML) based on the proposed ePLC software structure. This was implemented in an automatic winding machine, which was easier to use compared with some widely used languages (e.g., G-Code).**

***Index Terms*—motion control, customizing program, embedded PLC**

I. Introduction

PLCs are used for logic control in many applications [1], [2]. In order to meet the increasingly complex control needs, many scholars initiated research to improve PLC performance, such as enhancement through algorithms and corresponding function modules [3], improving the safety and reliability of PLCs by improving certain design methods [4]–[7], and improving program quality through forecasting and finding mistakes in the control procedure [8]. Software development and verification methods for PLCs are important research subjects [9]–[14]. In addition to improving the performance of PLCs, their functionality has also been enhanced with the fast development of integrated circuit technology. Now, PLCs have motion control abilities, such as position controlling, linear interpolation, and circular interpolation, for a wide range of applications [15]–[18]. By integrating motion control into the PLC, the control system was greatly simplified because simple motion control can be realized by the PLC without a special motion controller. There are many ways to realize this integration. One is to integrate a simple motion control function into the PLC [19], [20], which combines logic control and motion control. However, this hardly ensures the accuracy and speed of sophisticated motion control functions because the motion control and the logic control operate in the same cycle. Therefore, this approach is usually used in simple motion control applications. Another way to realize integration is to use an independent motion control module. The module has an independent executive system and development environment, and it can accomplish complex motion control by means of communication and interaction with the PLCs. An example of this is seen in the Panasonic FP2 position control unit [21].

There are different ways to develop a PLC program with motion control. For motion control integrated into a PLC, the motion control program is developed in the PLC's development environment. For independent expansion motion modules, the motion control program can be developed by G-Code, such as in the MC421/221 of OMRON CS1 series PLC [22]. It can also be developed by special programming languages, such as the textual description language SYMPAS or provided by motion control instructions in ladder-diagram programming environments, such as the FP series PLC created by Panasonic [21]. In order to meet different applications needs, different controllers must be chosen, and specific development environments and development languages to accomplish the system design and development are required. This is a great challenge for designers. The motion control standard [23] provides a standard library that can be reused among different platforms, and can reduce the complexity and the costs of development and maintenance. Many present design tools adopt this development method, such as CoDeSys and MULTIPROG. CoDeSys SoftMotion combines PLC logical control with motion control [24]. It integrates a motion control function toolkit into the PLC programming system, and utilizes the IEC61131-3 standard programming language for development. It can also expand the CNC POU library to provide complex motion control functions such as interpolation.

Although the function of PLCs has been enhanced, especially in the field of complex motion control, there are still some shortcomings of PLCs. These are listed below.

1) Since PLCs are designed for general purposes, they are not the optimal choice for a control task. Additionally, the development process of motion control in PLCs is not convenient, and the PLC requires continual reprogramming to meet the requirement of a specific application. Furthermore, although the IEC61131-3 standard development languages are suitable for developing a logic control program, it is challenging to develop complex motion control programs.
2) Function development and process development are mixed together, which makes it hard to adjust the sequence and parameters. Function development refers

Huifeng Wu and Yi Yan are with the Institute of Intelligent and Software Technology, Hangzhou Dianzi University, Hangzhou, 310018, China (e-mail: yybjyyj@163.com).

Danfeng Sun are with the Institute of Industrial Internet, Hangzhou Dianzi University, Hangzhou, 310018, China.

Rene Simon is with the Department of Automation and Computer Sciences, Harz University of Applied Sciences, Wernigerode, 38855, Germany.

This work was supported by a Grant from The National Natural Science Foundation of China(No.U1609211), Science and Technology Program of Zhejiang Province(No.2018C04001)

to the development of some basic function, such as servo motor control, and the function is not related to any application. Meanwhile, process development refers to application-related development, such as the cooperation of servo motors. Function development always results in libraries. Under the traditional development architecture, developers should not only accomplish function development, but also need to be familiar with the process and technical parameters in the development of certain equipment control procedures, which greatly extends the development period.

3) End-users cannot adjust the program according to actual control needs. End-users are most familiar with the actual requirement, and they always need to adjust the sequence and parameters of motion control to achieve optimal control. However, the existing development method does not support users to customize the program.

In order to solve these problems and expand the scope of PLC applications, this paper puts forward the concept of an embedded PLC whose hardware structure can be customized according to actual requirements and whose software structure is divided into three layers, with an application layer that can be compiled in real time. The remainder of this paper is organized as follows. In section II, we introduce the basic ideology and concept of this paper. In sections III, IV , V, and VI, we describe the implementation process in detail. A case analysis is carried out in section VII. Finally, conclusions and future work is discussed in section VIII.

II. The Basic Ideology and Concept

A. The Concept of ePLC

With function enhancement of the embedded processor, it is possible to implement several control tasks in one or more embedded processors. The $ePLC$ takes the embedded processor as the core, and peripheral circuits can be added according to requirements. The $ePLC$ has more open hardware structure compared with PLCs, and its hardware can be customized on-demand, making it quite similar to a special controller. The software structure of $ePLC$ can also be customized. The development method of the $ePLC$ is similar to that of the PLC, which means that the languages of IEC61131-3 are also supported [25]. Fig. 1 shows one typical structure of the $ePLC$. It contains three cores. The $ePLC$ has been applied in some fields (e.g., CAD [26]).

B. Multilayer Architecture of ePLC

Fig. 2 shows that the general control program of PLCs includes several program blocks, and the execution flow is determined at design time. In our architecture, the control program also includes several program blocks, but they are independent from each other, which means that the execution flow is not determined at design time, and even the parameters can be adjusted, as shown in Fig. 3.

The execution flow can be changed without reprogramming the control program at runtime. To achieve this goal, a customizing program is introduced to the $ePLC$, and a customizing thread is added to the executing system of the $ePLC$ to parse and run the customizing program. The traditional PLC's executing system includes a control program driver module. This module interprets the input signals, carries out the program stored in memory, and generates the output signals; this process is an endless loop when the PLC is running. At the same time, communication, exception handling, and other functions are implemented in the loop. With the enhancement of PLC functionality and application complexity, the control program became more complicated. In order to improve the control efficiency, the PLC's executing system uses multiple threads progressively. A customizing thread can be added to the executing system to parse and run the customizing program in the multi-threads kernel.

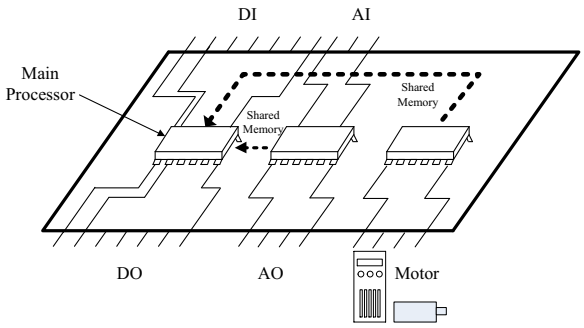

Fig. 1. Hardware structure of the $ePLC$.

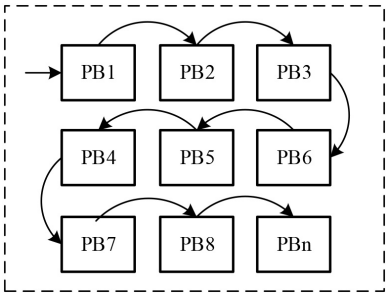

Fig. 2. General control program of the PLC includes several program blocks, and the execution flow is determined at design time.

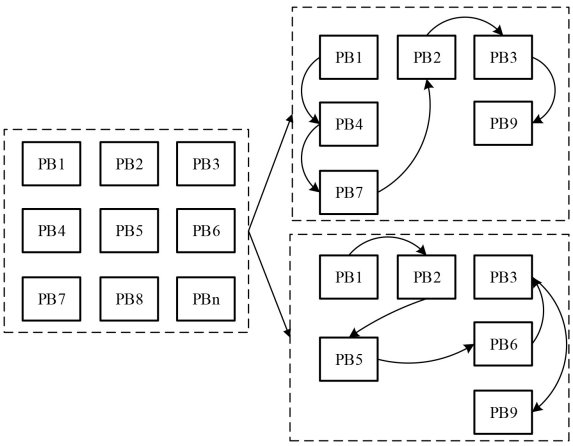

Fig. 3. Customizable control program structure of the $ePLC$, whose program blocks are independent of each other, and the execution flow is not determined at design time.

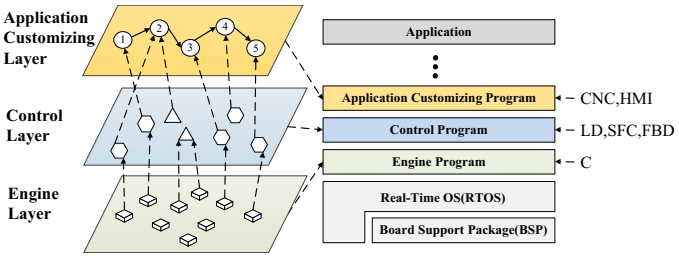

Fig. 4. Three-layer architecture of the ePLC.

With the purpose of separating function development and process development, we used a three-layer development architecture, as shown in Fig. 4:

1) Engine Layer. In the Engine Layer, the kernel and algorithms are developed through C language or an assembler language by professional developers. The program developed in this layer is called the Engine Program (EP), which includes the initialization, interruption handling, timer, communication, and all kinds of algorithms of the $ePLC$. The motion control-related algorithms are also included in the engine program.
2) Control Program Layer. In the Control Program Layer, programs can be written by the languages in IEC61131-3. Engineers with no knowledge of programming can program the PLC easily through graphical languages, such as LD, SFC, and FBD. The programs developed in this layer are called Control Program (CP), which includes logic control and motion control. The motion control-related function is realized by calling the algorithms in the EP, and the execution sequence and parameters are of no concern in this layer's program.
3) Application Customizing Layer. In the application customizing layer, users can customize the sequence and parameters of motion control. The customizing program is concerned with the special control process, and it is implemented through calling the program modules in the CP layer. Users can program in this layer, and the $ePLC$ can compile it into the CP layer in real time. The program developed in this layer is called the Application Customizing Program (ACP).

A complex control system can be divided into three comparatively simple parts through the aforementioned architecture, and the flexibility is also improved tremendously. Developers are independent of each other so that they can focus on their own task; meanwhile, the motion control part in the CP is application-independent through the introduction of the ACP layer, which can make the CP stable. An application developer can customize and adjust the motion CP via the ACP according to actual requirements. Compared with the way that the ACP directly calls algorithms in the engine layer, the CP layer can not only implement the task of logic control, but also make the adjustment of motion control more easily. Programs of each layer are stored in fixed positions of PLC's FLASH, and the programs will be moved to RAM when the power is on. The following discussions are in the case that programs are running in RAM.

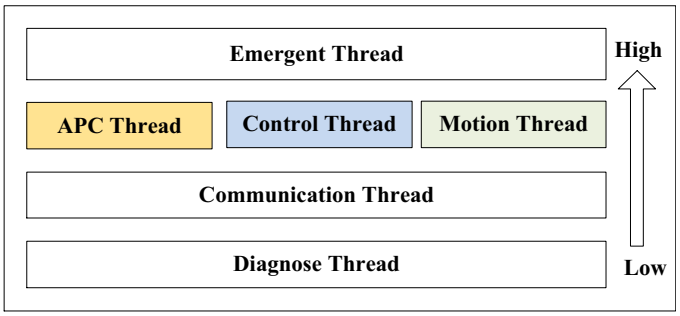

Fig. 5. Sketch map of multi-thread structure, including four priorities and six threads.

C. Thread Relationship Analysis

The $ePLC$ used a priority-based preemptive multi-thread scheduling system, with four priorities and six threads, as shown in Fig. 5. The threads at the same priority level use the time slice to get the run time.

Emergent Thread: The emergent thread has the highest priority and is used to handle emergent tasks, such as interrupts. All interrupts, including I/O interrupts, timer interrupts, exception interrupts, and so on, unless they are caused by fatal errors, will not be handled immediately when occurring. The interrupt information will be packaged and pushed to a queue that will be handled by the emergent thread.

ACP Thread, **Control Thread**, and **Motion Thread**: The ACP Thread, Control Thread, and Motion Thread have the same priority. The ACP Thread is used to handle the ACP by reading, parsing, and executing the program. The Control Thread is used to handle the CP, responding to the requests of the ACP thread and calling engine algorithms to run. The Motion Thread is used to respond to the requests of the Control Thread and to execute the algorithms in the EP to output the motor-driven signals. The purpose of the corporation of multi-threads is to avoid task blockages caused by task delays in multi-task systems, and thus to improve the system's responsiveness. For example, when the Motion Thread is waiting for the end of the actuator, I/O logical control can work as normal.

Communication Thread: The Communication Thread is used to deal with the communication task, and responds to all kinds of communication requests, such as HMI.

Diagnose Thread: A diagnosis module was embedded in the system, and the idle time of the CPU was used to diagnose nonfatal faults on the condition of not affecting control function.

D. Relation and Definition of Data Interaction

Communication, synchronization, and mutual exclusion between multi-threads must be handled correctly and efficiently to ensure that the $ePLC$ can work as expected. In our system, data interaction and parameter passing were realized by shared memory. Similar to PLCs, a public and specific data area in memory was dedicated to store the values of soft elements. Data area D is a byte data area, and is used to realize parameters passing; data area M is

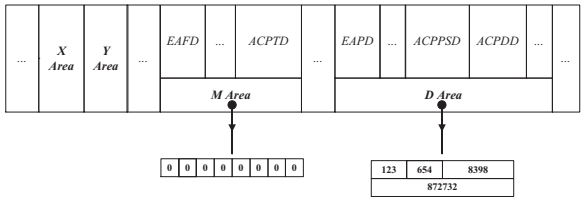

Fig. 6. Distribution of dedicated public data area.

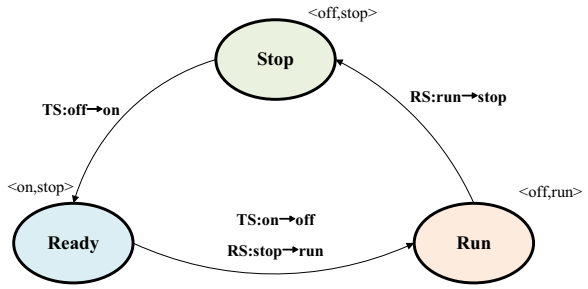

Fig. 7. State transition diagrams of engine algorithms.

a bit data area, and is used to realize synchronization and mutual exclusion. Several specific data regions in data area D and data area M were used to realize data exchange and run control, as exemplified in Fig. 6. These data sections are defined as follows:

Definition 1 $EAPD$ (Engine Algorithm Parameter Data Section): a specific section in data area D used to store the parameters of engine algorithms; the address of every parameter is fixed.

Definition 2 $EAFD$ (Engine Algorithm Flag Data Section): a specific section in data area M used to store the start flag, stop flag, and state flag of algorithms. Engine algorithms can be called to execute by setting the start flag in the CP, and the algorithm's state can be detected by reading the state flag.

Definition 3 $ACPDD$ (ACP Dedicated Data Section): a specific section in the RAM used to store the ACP from FLASH.

Definition 4 $ACPPSD$ (ACP Parameter Swap Data Section): a specific section in data area D used to store the parameters obtained from parsing the ACP. Only the parameters that are used by the next executing instruction are stored in $ACPPSD$. The CP gets parameters from $ACPPSD$, and passes them to EAPD.

Definition 5 $ACPTD$ (ACP Trigger Data Section): a specific section in data area M used to start the program block in the CP. Every program block is associated with an ACP instruction.

III. The Realization of Engine and Algorithms

The EP was responsible for initialization, interrupt handling, timing, communications, and other basic functions (BF) of the $ePLC$. It also contained several expansion algorithms (EA) that enhanced the $ePLC$'s functionality.Supposing $EP = \{BF, EA, EAP, EAF\}$, $EA = \{ea_1, ea_2, \cdots, ea_i, \cdots, ea_m\}$, which means that the EA contains m algorithms. Corresponding to m algorithms, the EAP is parameter set for EA with m subsets, $EAP = \bigcup_{i=1}^{m} EAP_i$. $EAP_i = \{eap_1, eap_2, \cdots, eap_i, \cdots, eap_n\}$ is the set of all parameters for ea_i, written as $EAP_i \bowtie ea_i$. Furthermore, $\exists EAP_i \bowtie ea_i \cdot \exists EAP_j \bowtie ea_j : i \neq j \rightarrow EAP_i \cap EAP_j = \emptyset$. $EAF = \bigcup_{i=1}^{m} EAF_i$ is the flag set for EA with m subsets. $EAF_i = \{eaf_1, eaf_2, \cdots, eaf_i, \cdots, eaf_k\}$ contains k flags used to start and indicate the state of algorithms for ea_i, written as $EAF_i \bowtie ea_i$. Additionally, $\exists EAF_i \bowtie ea_i \cdot \exists EAF_j \bowtie ea_j : i \neq j \rightarrow EAF_i \cap EAF_j = \emptyset$. The set of trigger flags $TS = \{on, off\}$, is triggered by the upper layer, and cleared by the EP. The set of running state $RS = \{run, stop\}$, is set and cleared in the EP. The state is set to run after sending running signals to the actuator, and the state is set to $stop$ when the actuator stops working. Therefore, run and $stop$ can also indicate the state of the algorithm. Cartesian product $TS \times RS$ represents all possible combination of the states; in other words, any value of EAS_i belongs to $TS \times RS$. $< off, stop >$ means the $Stop$ state of the engine algorithm; in this state, when the trigger flag switches from off to on, $< on, stop >$ is used to indicate that the algorithm is in the $Ready$ state, and can be scheduled to execute by the thread at any time. The trigger flag was cleared before executing to avoid repeated execution, and the $stop$ was switched to run at the same time to indicate that the algorithm was in a running state; then, the $Stop$ state was transited to Run. When the algorithm finished running, the run flag was switched to $stop$ again, and the state transited to $Stop$. Fig.7 illustrates the state transition.

IV. Design and Realization of the ACP in the ePLC

The ACP layer was used to customize motion sequence and parameters. It includes programming, compiling, and executing. An ACP is composed of a list of ACP instructions. An ACP instruction is an atomic unit of the ACP, and every instruction implements a single function. Each ACP is made up of several instructions that are executed in chronological order. In our system, every ACP instruction was used to pass parameters and set the program block of CP to run. Therefore, we first needed to design the ACP instruction.

A. Design and Realization of the ACP Instruction

Supposing the instruction set has m instructions, $IS = \{I_1, I_2, \cdots, I_m\}$, $I_i = (name, code, P)$. Any instruction has a name, a code, and a parameter set P. The instruction name is a mnemonic symbol to demonstrate the instruction's function. The instruction code is unique and can be used to identify an instruction, namely $\neg\exists I_i \exists I_j : I_i.code == I_j.code \wedge i \neq j$. We adopted a natural number N as the instruction code in the instruction set. In the development environment of the PC, only the instruction name and parameters were used, and the instruction format was defined as shown below:

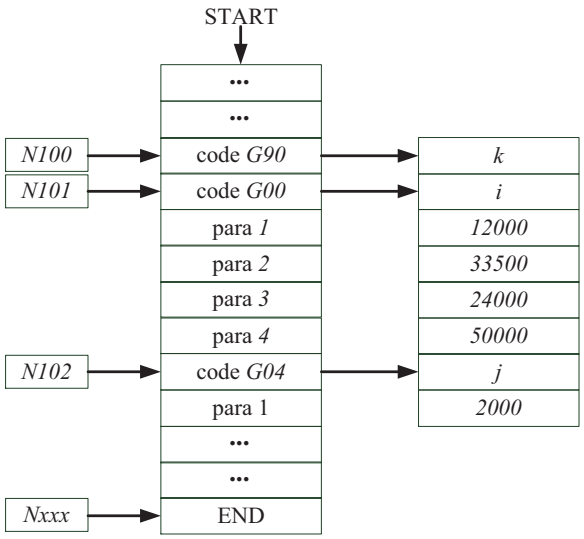

Fig. 8. Format of the ACP frame after compiling.

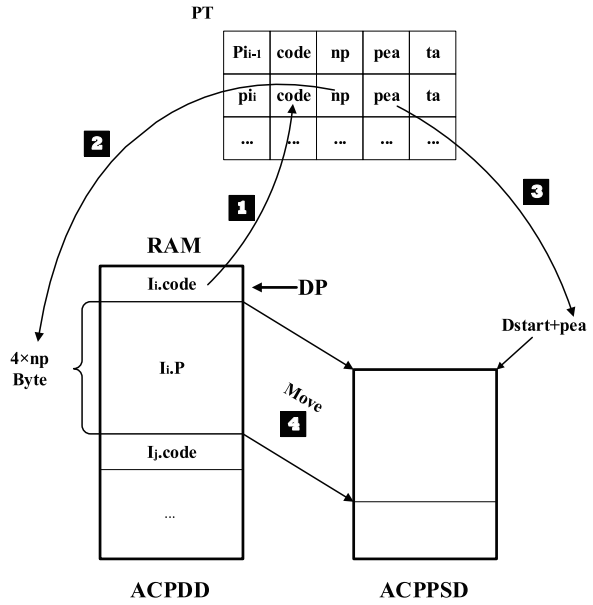

Fig. 9. Schematic of delivering ACP instruction parameters from $ACPDD$ to $ACPPSD$.

$I_i.name \quad I_i.p_1, I_i.p_2, I_i.p_3, \cdots, I_i.p_n$.
Any ACP program consisted of a finite number of chronological instructions.

B. Compilation and Execution of ACP

1) Compilation: Compiling the ACP means converting the ACP to ACP', which can be identified and executed by the $ePLC$, $f : ACP \rightarrow ACP'$. It contained two steps, $f = g \circ h$. Function g transformed the instruction name to an instruction code, $g : ACP \rightarrow T$, $g(I_i.name) = I_i.code, i = 0, 1, \cdots, m$. Obviously, g is a one-to-one function. Function h normalized parameters, $h : T \rightarrow ACP'$. The objective of h was to transform parameters to a standard format. For readability, we used a non-standardized format, which is easy for users to understand. The non-standardized format is denormalized; for example, the pulses of movement are a normalized expression of distance. In order to make it easy to understand for the user, centimeters were used as the non-standardized format for distance. Thus, parameters were normalized by transforming the parameters to the format accepted by the $ePLC$.

The data frame downloaded into the $ePLC$ was made up of compiled instructions and an END flag, as shown in fig.8. In order to parse the ACP conveniently, the length of each parameter was 4 bytes long. The data frame was parsed by the ACP driver module when the $ePLC$ was running.

2) Parsing and Executing the ACP: The ACP driver module read every ACP instruction from the $ACPDD$, and obtained parameters by parsing the instruction. The parameters were passed to $ACPPSD$. The information necessary for parsing the instruction was saved in a table PT, where each row corresponded to an instruction. $PT = \{pi_1, pi_2, \cdots, pi_m\}$, and the number of table rows was equal to the number of ACP instructions. $pi_i = (code, np, pea, ta)$, and the $code$ of pi_i was the same as the instruction code mentioned above. np is the number of parameters for instruction i. Because we denoted that the length of each parameter was 4 bytes, the length of the data block moved to $ACPPSD$ could be determined by np. pea is the address of $ACPPSD$, which is the offset of the starting address of area D. ta is the starting address of the trigger flag data section, which is the offset of the starting address of area M. Every bit in the trigger flag data section was used to control the execution of a function module in the CP, and the instruction code could be used as an offset within the byte. The parsing and execution process included the following steps:

Step 1: Reading instructions. Read a byte from the current pointer position DP of $ACPDD$ as the ACP instruction code, and judge whether it is the end of program. If yes, stop the process; if not, go to step 2.

Step 2: Parsing instruction. Get the necessary information for instruction parsing from table PT according to the instruction code.

Step 3: Transferring Data. Pass the parameters of the current instruction to $ACPPSD$ according to np and pea, which are obtained by parsing the instruction. The $4 \times np$ bytes data in $ACPDD$ following the position of the code are moved to the address of $Dstart + pea$, as illustrated in Fig.9.

Step 4: Execution instruction. Set the corresponding bit of $ACPTD$ to 1 according to ta and the instruction code, as shown in Fig.10. The bit data in area M was used as the trigger flag in the CP, and ta indicates the offset to the start address of area M. The instruction code was taken as the offset within the byte to get the bit needed to be set 1(ON) in order to execute the corresponding program block in the CP.

Step 5: Waiting for the end of execution. The ACP thread will wait until the end of the execution, and then go to step 1 again.

These steps are carried out repeatedly until all instructions are handled.

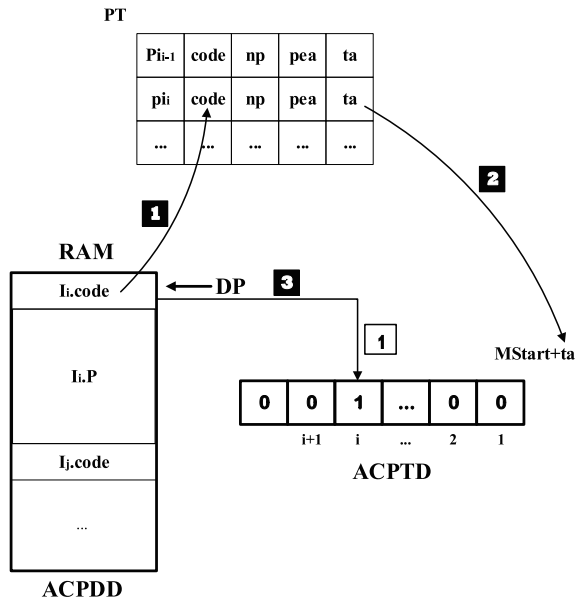

Fig. 10. Schematic of setting the execution bit in $ACPTD$.

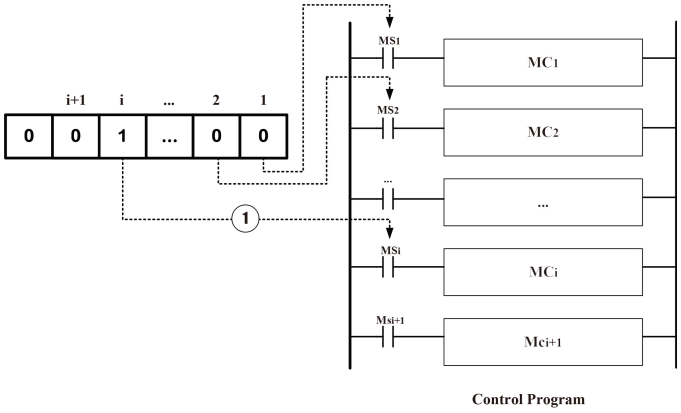

Fig. 11. Using bit data in area M to control the program module's execution.

V. Implementation of the PLC Control Program

The CP was developed using IEC61131-3 standard languages, which includes LD, SFC, FBD, ST, and IL. LD is a graphic programming language that is widely used to develop control algorithms for PLCs. In this paper, we took LD as the programming language. The CP was the middle level of our three-layer architecture, and it not only responded to the request of the top ACP layer, but also called the algorithms in the engine layer. $CP = \{IP, LP, MC, MS\}$, where IP is the initial program, which implements the initial operation of data area, port, and state. LP is the logical part of the CP.MC is the motion control part of the CP. $MC = \{mc_1, mc_2, \cdots, mc_m\}$, where mc_i is used to realize the i_{th} ACP instruction, and the instruction code is i. MS is a set of flags used to control MC, $MS = \{ms_1, ms_2, \cdots, ms_m\}$, and ms_i is used to trigger the execution of mc_i, as shown in fig.11. Any ms_i contains two states: ON and OFF. When the state is switched from OFF to ON, the corresponding mc_i will execute. In the $ePLC$, the data M area can be used as a control flag, and thus every program block of MC takes a bit of M to control it.

The mc_i implements the function of passing parameters, condition judgment, and algorithm calling. The parameters were moved to EAD, and T of EAS_j was set to ON before mc_i called the subset EA_j of EA. Besides logical control, the motion control part of the CP was called by thread cyclicly, which included the following steps:

Step 1: Traversing $ACPTD$ to judge whether there is any suspended ACP instruction. If yes, go to step 2; otherwise, finish this cycle.

Step 2: Moving data in $ACPPSD$ to $EAPD$ in accordance with the agreed format.

Step 3: Setting the corresponding bit in $EAFD$ to execute the algorithm in the EP and clear suspended bits in $ACPTD$.

Step 4: Reading the running state bit of the algorithms in $EAFD$ on each loop to wait for the end of execution. At the end of execution the loop will go to step 1 to start the next cycle.

VI. Collaboration of Threads

The synchronization and mutual exclusion of the ACP thread and the Control Thread were realized by semaphore T_1. The synchronization and mutual exclusion of the Control Thread and the algorithm thread were realized by semaphore T_2. The ACP thread contained the following five steps.

Step 1: Reading instruction (p_1).
Step 2: Parsing instruction (p_2).
Step 3: Acquiring execution semaphore T_1(p_3).
Step 4: Passing data (p_4).
Step 5: Starting up execution (p_5).

The Control Thread includes the following five steps.

Step 1: Executing logical CP (q_1).
Step 2: Traversing the flag section of the ACP instructions, and judging whether there are ACP instructions needed to be executed (q_2).
Step 3: Acquiring execution semaphore T1, and T2 (q_3).
Step 4: Passing data of $ACPPSD$ to $EAPD$ (q_4).
Step 5: Calling the engine algorithm to execute by setting the corresponding control flag, and returning T_1(q_5).

The Algorithm Thread contains the following four steps.

Step 1: Traversing the flag section of algorithm execution, and judging whether there is any algorithm needed to be executed (s1).
Step 2: Acquiring semaphore T2 (s2).
Step 3: Executing the algorithm (s3).
Step 4: Waiting for the end of execution, and returning semaphore T2 (s4).

The process was as follows. First, the ACP thread reads instructions (p_1)from $ACPDD$. When there is an ACP instruction needed to be executed, the ACP driver module will parse the instruction to get the necessary information to execute the instruction (p_2). Then, it attempts to acquire semaphore T_1 (p_3). Once done, parameters will be moved to $ACPPSD$ (p_4), and the corresponding flag bit in $ACPTD$

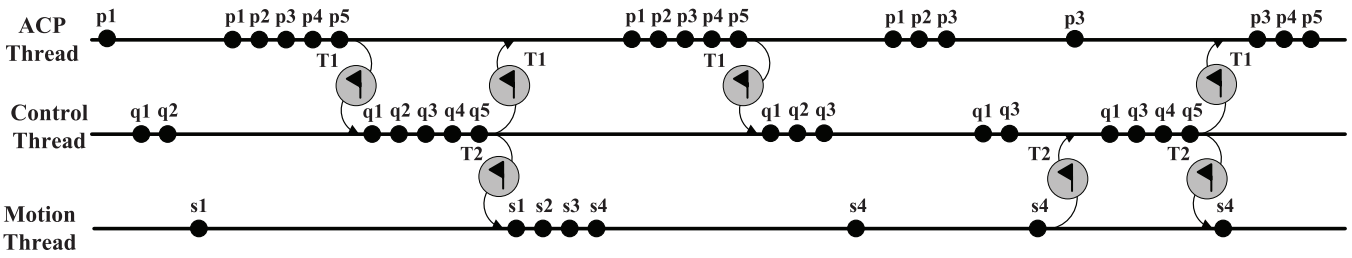

Fig. 12. Collaboration of threads.

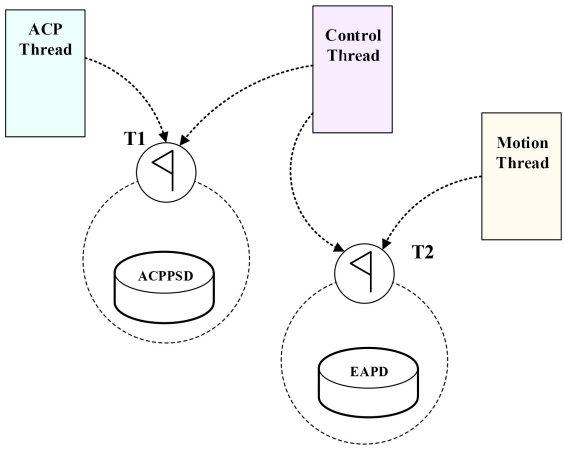

Fig. 13. Mechanism of data sharing and exclusion between threads.

will be set ON (p_5). Then, semaphore T_1 is returned to the Control Thread, and the next cycle starts. If the thread cannot get the semaphore T_1, it will keep trying until success. Besides logical control (q_1), the Control Thread traverses $ACPTD$ (q_2) to judge whether there is an ACP instruction needed to be executed; if yes,it tries to acquire the semaphores T_1, T_2 (q_3). After getting the semaphores T_1 and T_2, parameters are moved from $ACPPSD$ to $EAPD$ (q_4). Then, the corresponding flag bit in $EAFD$ is set to ON in order to execute the algorithm. Then, the semaphores T_1 and T_2 will be released. When there is a suspended ACP instruction needed to be executed and the thread cannot get the semaphores T_1 or T_2, the Control Thread continually executes q_1 and q_3 until it get the semaphores T_1 and T_2. The algorithm thread traverses $EAFD$ continuously to judge whether there is an algorithm needed to be executed (s_1); if yes, it tries to acquire the semaphore T_2 (s_2). After getting semaphore T_2, the corresponding algorithm will be executed (s_3) to generate signals for the actuator, and waits for the actuator to stop running (s_4). The semaphore T_2 will be released after the end of running. Data sharing of threads was realized through shared memory. In order to protect data and ensure their consistency, different threads were not allowed to operate on the same data area simultaneously. Due to the use of a multi-layer structure in our system, only the adjacent layer threads operated on the same data area. As shown in fig.13, semaphore T_1 was used to realize the mutual exclusion of $ACPPSD$. The ACP thread could write $ACPPSD$ only when it got the semaphore T_1; meanwhile, the Control Thread could read $ACPPSD$ only when it got the semaphore T_1.

Fig. 14. Automatic winding machine.

Similarly, the Control Thread and the algorithm thread realized the mutual exclusion of $EAPD$ via semaphore T_2.

VII. Case Analysis

A. Introduction of the Experiment Equipment

An automatic winding machine was used in our experiment, as shown in Fig.14. The machine had five cooperating axes and twelve air pumps to implement the function of wire arrangement, winding pins, trimming, loading and unloading the skeleton, and so on. The five axes were the X-axis, Y-axis, Z-axis, U-axis, and Q-axis. The X-axis, Y-axis, and Z-axis were responsible for cable feeding and coiling operation, and the executors were servo motors. As the master axis, the U-axis was responsible for winding, and the executor was also a servo motor. The Q-axis was used to control the winding angle, and the executor was a stepping motor. The controller of the winding machine was an $ePLC$ CASS-PLCA149B with motion control, which we developed ourselves, as shown in Fig.15. The controller used a dual-processor architecture, and was composed of two STM32 CortexM3 chips. The controller had 32 input ports, 32 output ports, and six motor interfaces. The main processor and the slave processor worked cooperatively to control the motor running. The main processor sent parameters and command signals to the slave processor,

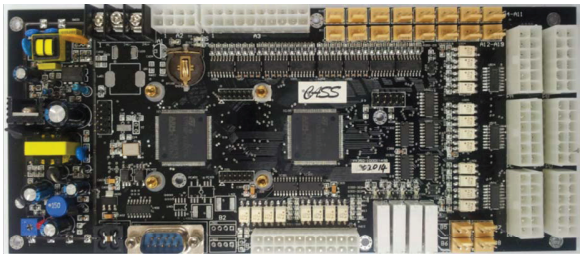

Fig. 15. Photo of ePLC CASS-PLCA149B.

which generated PWM pulses and other control signals for the motors. The ACP program, CP, and motion algorithm in the EP were executed in the main processor.

B. Action Analysis of the Winding Machine

Only the control of the motors was considered here. The movement driven by the air pumps was not discussed in this paper because they can be realized easily by I/O control in the CP. The winding machine could perform the following actions:

1) Reset: First, the X-axis, Y-axis, Z-axis, U-axis, and Q-axis search for the Z point through z-phase positioning. Then, each axis moves a given distance, and at last the location of each axis is set as its origin.
2) Loading skeletons: At the beginning of coiling, the combined movement of the air pumps carries the skeletons from the material platform to the U-axis.
3) Unloading skeletons: At the end of coiling, the combined movement of the air pumps carries the skeletons from the U-axis to the material platform.
4) Linear feed: Motors drive the X-axis, Y-axis, and Z-axis, respectively, to make the guide pin, which is used to guide a copper wire to a specified location.
5) Winding pin: This task is implemented by the X-axis, Y-axis, and Z-axis working cooperatively. The X-axis and Y-axis perform circular movements with a specified radius. Meanwhile, the Z-axis ascends or descends according to a specified distance. In this step, the copper wire is fixed on the pin of the skeleton.
6) Winding: The U-axis drives the skeleton to rotate in high-speed. The X-axis is the slave-axis, and moves back and forth following the U-axis. The copper wire winds on the skeleton when the X-axis and U-axis perform master–slave movements.
7) Trimming: After coiling another pin, the gripper pump moves forward or backward so that the copper wire is pulled apart.

C. ACP Instruction Design and Analysis

Based on the above analysis we defined the customized winding machine language ($CWML$) as shown in Table I. The $CWML$ is a customized streamlined language that included instructions with a customized format. It was primely supported by the proposed software structure on the $ePLC$. In this case, 13 instructions satisfied almost all required applications for winding. Meanwhile, the proposed three-layer structure supported adding instructions easily, and the Application Customizing Layer could compile the ACP in real time.

TABLE I
THE DEFINITION OF CWML

No.	Code	Params	Description	Function
01	R00 X_Y_Z_S_	4	X: X-axis distance Y: Y-axis distance Z: Z-axis distance S: Speed	X, Y, Z-axis high-speed feed
02	R00 U_ S_	2	U: U-axis distance S: Speed	U-axis high-speed feed
03	R00 Q_ S_	2	Q: Q-axis distance S: Speed	Q-axis high-speed feed
04	R00CW X_N_D_FD_BD_S_	4	X: X-axis distance N: Number of turns D: Wire diameter FD: Forward decreasing distance BD: Backward decreasing distance S: Speed	U-axis clockwise winding
05	R00CCW X_N_D_FD_BD_S_	4	X: X-axis distance N: Number of turns D: Wire diameter FD: Forward decreasing distance BD: Backward decreasing distance S: Speed	U-axis counterclockwise winding
06	R02CW R_N_Z_S_	4	R: Circle radius N: Number of turns Z: Z-axis distance S: Speed	Clockwise winding pin
07	R02CCW R_N_Z_S_	4	R: Circle radius N: Number of turns Z: Z-axis distance S: Speed	Counterclockwise winding pin
08	R04	1	T: delay time	Pause
09	R05	0		Absolute coordinate
10	R06	0		Incremental coordinate
11	R07	1		Gripper output
12	R08	1		Load skeleton
13	R09	1		Discarding waste wire

It would be hard work to implement our approach with some widely used languages, such as G-Code, which is shown in Table II. It includes 217 instructions, and every instruction contains several parameters.

We could develop, debug, and compile the ACP in ComEditorX, which is as an Integrated Development Environment(IDE) for ACP. ComEditorX could be used to both develop the ACP program and supply a means to define customized ACP instructions.

D. Experimental Result

The EP was divided into master and slave parts, which were executed in the master and slave processors separately. The master EP contained the motion control algorithm, CP driver module, and ACP program driver module, in addition to basic functions. The slave program was used to generate signals for the motors according to commands from the master.

TABLE II
LIST OF G-CODE FOUND ON FANUC

No.	Code	Params	Description
01	G00 X_Y_Z	3	Rapid positioning
02	G01 X_Y_Z	3	Linear interpolation
03	G02 X_Y_R_F	4	Circular interpolation, clockwise
...	...	...	...
101	G100	0	Tool length measurement
102	M00	0	Compulsory stop
103	M01	0	Optional stop
104	M03	0	Spindle on (clockwise rotation)
...	...	...	...
201	M99	0	Subprogram end
202	N01	0	Turn on load monitor
203	N02	0	Inch units. Absolute mode. Activate work offset. Activate tool offset. Deactivate tool nose radius compensation.
...	...	...	...
217	N16	0	Program stop, rewind to top of program, wait for cycle start

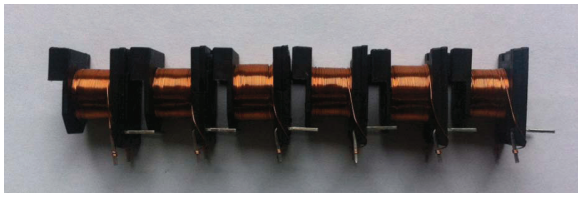

Fig. 16. Product of the winding machine.

The CP was developed in a graphical IDE for the $ePLC$ using the LD language. In the IDE we could program, debug, and compile [25] the CP. The CP was downloaded to the $ePLC$ after compiling.

Once the EP and CP were developed and downloaded to the $ePLC$, users could develop the ACP for the winding machine. Under a given skeleton and wire diameter, users could optimize the motion control sequence and parameters of the ACP repeatedly in the single step mode. Finally, the ACP was downloaded to the $ePLC$.

The skeleton was winded as in Fig.16. In this case, the diameter was 0.6 mm, the distance was 20 mm, the velocity was 40 mm/s, and the number of turns was 4. Table III compares the winding programs of G-Code and CWML. Because G-Code is designed for CNC, its velocity is low in most cases, the unit of instruction of G01 and M03 are mm/min and RPM, respectively. We found that the $CWML$ was much easier to use than G-Code in the winding machine.

Fig.17 shows the other products of the winding machine. If the skeleton, wire diameter, or position of the pin were changed, only the parameters of the ACP required adjustment. The CP and EP did not need to change.

VIII. CONCLUSION

In this paper we proposed a customized real-time compilation method to simplify special motion control. The concept of an $ePLC$ and a three-layer development architecture were put forward. The description method and compiled algorithm of the architecture were introduced in detail. Based on the software structure on the $ePLC$, we proposed the $CWML$, which was then applied to a winding machine. Using this development method, we can expand the applications of the $ePLC$ to robotics, numerical control machines, and so on.

TABLE III
COMPARISON BETWEEN G-CODE AND CWML

No.	G-Code	CWML
01	M03 S400	R00CW X20 N4 D0.6 FD0.3 BD0.3 S40
02	G01 X20 F2400	
03	G01 X0.3 F2400	
04	G01 X19.7 F2400	
05	G01 X0.6 F2400	
06	G01 X19.4 F2400	
07	G01 X0.9 F2400	
08	G01 X19.1 F2400	
09	G01 X0 F2400	
10	M05	

Fig. 17. Other winding skeletons.

REFERENCES

[1] S. Abdallah and R. Abu-Mallouh, "Heating systems with plc and frequency control," *Energy Conversion and Management*, vol. 49, pp. 3356–3361, 2008.

[2] R. Bayindir and Y. Cetinceviz, "A water pumping control system with a programmable logic controller (plc) and industrial wireless modules for industrial plants—an experimental setup," *ISA Transactions*, vol. 50, pp. 321–328, 2011.

[3] S. Gerkšič, G. Dolanc, D. Vrančić, J. Kocijan, S. Strmčnik, S. c. Blažič, I. S̄krjanc, Z. Marinšek, M. Božič ek, and A. Stathaki, "Advanced control algorithms embedded in a programmable logic controller," *Control Engineering Practice*, vol. 14, no. 8, pp. 935–948, 2006.

[4] M. Śnieżek and J. V. Stackelberg, "A fail safe programmable logic controller," *Annual Reviews in Control*, vol. 27, no. 1, pp. 63–72, 2003.

[5] S. R. Koo and P. H. Seong, "Software design specification and analysis technique (sdsat) for the development of safety-critical systems based on a programmable logic controller (plc)," *Reliability Engineering & System Safety*, vol. 91, no. 6, pp. 648–664, 2006.

[6] C. Y. Oscar Ljungkrantz, Knut Akesson and M. Fabian, "Towards industrial formal specification of programmable safety systems," *IEEE Transactions on Control Systems Technology*, vol. 20, no. 6, pp. 1567–1574, 2012.

[7] G. Kandare, S. Nik, and G. Godena, "Domain specific model-based development of software for programmable logic controllers," *Computers in Industry*, vol. 61, no. 5, pp. 419–431, 2010.

[8] I. Plaza and C. Medrano, "Exceptions in a programmable logic controller implementation based on ada," *Computers in Industry*, vol. 58, no. 4, pp. 347–354, 2007.

[9] H. Zhang, Y. Jiang, W. N. N. Hung, X. Song, M. Gu, and J. Sun, "Symbolic analysis of programmable logic controllers," *IEEE Transactions on Computers*, vol. 63, no. 10, pp. 2563–2575, 2014.

[10] J. Provost, J.-M. Roussel, and J.-M. Faure, "Generation of single input change test sequences for conformance test of programmable logic controllers," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 3, pp. 1696–1704, 2014.

[11] W. Dai, V. N. Dubinin, and V. Vyatkin, "Migration from plc to iec 61499 using semantic web technologies," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 3, pp. 277–291, 2014.

[12] M. Obermeier, S. Braun, and B. Vogel-Heuser, "A model-driven approach on object-oriented plc programming for manufacturing systems with regard to usability," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 3, pp. 790–800, 2015.

[13] B. Fernandez Adiego, D. Darvas, E. B. Vinuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. Gonzalez Suarez, "Applying model checking to industrial-sized plc programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[14] B. Caldwell, R. Cardell-Oliver, and T. French, "Learning time delay mealy machines from programmable logic controllers," *IEEE Transactions on Automation Science and Engineering*, vol. 13, no. 2, pp. 1155–1164, 2016.

[15] Nowlin and B. C, "A programmable system for motion control," in *Proceedings of the International Instrumentation Symposium*, vol. 49, 2003, pp. 87–103.

[16] R. Mamlook, S. Nijmeh, and S. M. Abdallah, "A programmable logic controller to control two axis sun tracking system," *Information Technology Journal*, vol. 5, no. 6, 2006.

[17] J. Wei-hua and Y. Dong, "Design of position system of graphite spraying machine based on plc controlling," *Modular Machine Tool & Automatic Manufacturing Technique*, pp. 52–55, 2009.

[18] X. T.-z. Sun Cheng-zhi, "Teaching and playback in two axis motion control based on plc," *Modular Machine Tool & Automatic Manufacturing Technique*, pp. 65–67, 2010.

[19] M. G. Ioannides, "Design and implementation of plc-based monitoring control system for induction motor," *IEEE Transactions on Energy Conversion*, vol. 19, no. 3, pp. 469–476, 2004.

[20] S.-R. A and S. R, "Model predictive control implementation on a programmable logic controller for dc motor speed control," in *International Conference on Electrical Engineering and Informatics*, ser. C6-1. Bandung, Indonesia: IEEE, 2011, pp. 1–4.

[21] P. Co.Ltd, *Programmable controller FP2 Positioning Unit Mannual*, 2011.

[22] O. Co.Ltd, "Cs1w-mc221(-v1)/mc421(-v1) motion control units." *Operation Mannual*, 2004.

[23] F. M. C.K. Sünder, A. Zoitl and B. Favre-Bulle, "Advanced use of plcopen motion control library for autonomous servo drives in iec 61499 based automation and control systems," *Elektrotechnik Und Informationstechnik*, vol. 123, no. 5, pp. 191–196, 2006.

[24] S. S. S. GmbH, "Logic and motion control integrated in one iec 61131-3 system:development kit for convenient engineering of motion, cnc and robot applications." 2017.

[25] Y. Yi and Z. Hangping, "Compiling ladder diagram into instruction list to comply with iec 61131-3," *Computers in Industry*, vol. 61, no. 5, pp. 448–462, 2010.

[26] W. U. Huifeng, J. Zhang, and R. Simon, "A method of introducing cad to eplc for motion control," *Ifac Papersonline*, vol. 48, no. 3, pp. 1580–1585, 2015.

Huifeng Wu received the Ph.D. degree in computer science and technology from Zhejiang university, Hangzhou, China, in 2006. He is currently a professor in the institute of intelligent and software Technology, Hangzhou Dianzi University. His research interests include software development methods and tools, software architecture, embedded system, intelligent control & automation.

Yi Yan received B.S. in automatic control engineering form Zhejiang Sci-Tech University in 1984, M.S. in computer engineering from Beijing University of Postal Telecommunications in 1990. Currently he is the director and full professor in institute of intelligent and software Technology, Hangzhou Dianzi University. His research interests include embedded system, advanced manufacturing system, intelligent control & automation, and intelligent instruments.

Danfeng Sun received M.S. in computer architecture from Hangzhou DianZi University in 2011. He is currently a research assistant in the Institute of Industrial Internet, Hangzhou DianZi University. His research interests include embeded system, motion control and IIoT.

Rene Simon obtained a doctor of engineering at the Otto-von-Guericke University Magdeburg in 2001. He is Professor of Control Systems at the Department of Automation and Computer Sciences, Harz University of Applied Sciences, Wernigerode, Germany. His major research fields include engineering of automation systems, especially industrial controllers. He is chairman of PLCopen and project leader IEC 61131-10 Ed. 1.0.