Analysis and implementation of the IEC 61131-3 software model under POSIX Real-Time operating systems

Inmaculada Plaza *, Carlos Medrano, Alfonso Blesa

Department of Electronic Engineering and Communications, EUPT—Teruel University of Zaragoza, Spain

Available online 30 June 2006

Abstract

In this paper, we present a proposal to implement the IEC 61131-3 software model. This model establishes the high-level elements of the Programmable Logic Controllers. We discuss some of the points of the standard that should be clarified before implementing it, such as concurrency, variable scope and cyclic operation. We show the implementation guidelines using POSIX compliant Real-Time Operating Systems, in particular RTLinux running on PCs. A translator has been implemented accepting textual descriptions following the IEC 61131-3 standard.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Programmable Logic Controller; IEC 61131-3; POSIX

1. Introduction

Traditional programming languages for PLCs (Programmable Logic Controller), such as Ladder programming, had several drawbacks, including [1]: weak software structure, poor facility to handle complex data structures, limited control over program execution, limited facilities for software reuse and the use of different symbols between PLC products. The IEC 61131-3 standard is a global standard for control programming that tries to overcome these problems in order to improve software quality. In this paper, we will focus on the software model and the programming organization units (POU) defined in the standard. Among other advantages, the use of the standard elements promotes the development of well-structured programs, code reuse and the application of software engineering principles such as encapsulation and data hiding.

Linux has already been used in control systems, such as in the MatPLC project [2,3], which has also been ported to the QNX POSIX compliant Real-Time operating system and includes its own IEC61131-3 compiler [4]. However, its manual states that it still lacks some features such as semantic checking, error messages and multiple configurations and tasks [2]. In addition, MatPLC specifies a much more general framework than PLC programming. Formal specification for a subset of IEC PLC languages can be found in [5]. Johansson and Öhmann [6,7] considered the implementation of IEC 61131-3, but focused on graphical languages and used a very specific tool, the real-time expert system G2 from the Gensym Corporation.

In the present paper, we consider the case of a single PLC, so communication functions are not considered, and we will centre on IEC 61131-3 standard programming methods. The standard is often understood as a set of programming languages to describe algorithms. However, it also includes configuration elements and allows the user to define its own POUs or data types. Therefore, in the present work, we review the main concepts of the IEC 61131-3 software model, highlighting its weak points. Then, we will explain an implementation proposal using a general programming language, C, and POSIX operating systems, in short POSIX/C. As a specific Real-Time operating system we will use RTLinux [8], which fulfils the minimal Real Time system profile (POSIX 1013.13). A translator from IEC 61131-3 to POSIX/C has been built following the
implementation guidelines which we briefly describe. Then, the performance of our model will be compared with that of some commercial products. Finally, we discuss the main conclusions and future areas of research derived from the present paper.

2. IEC 61131-3 software model

The software model is shown in Fig. 1 [9,10]. Fig. 2 shows an example of a configuration following the IEC syntax. We will briefly review the fundamental concepts of the standard that are used through this paper. Descriptions are supposed to be textual and the first language implemented is Instruction List (IL), both choices are based on the simplicity to handle them. For a discussion of the implementation of graphical languages see [6,7].

- **Configuration**: A language element that corresponds to the programmable controller system. The configuration allows the declaration of global variables and resources.
- **Resource**: A processing facility that is able to execute IEC programs. The resource can declare global variables, tasks and program instances associated with tasks. It can be assigned to a given processor.
- **Task**: A task controls the execution of programs and function blocks.
- **Program organization units (POU)**: Functions, function blocks (FB) and programs. Functions accept inputs and return a result and additional values by means of output variables. Function block (FB) type definition can contain input, output, external, internal and input/output variables. The body of the FB type definition is an algorithm acting on the declared variables in order to provide new output values. External and input/output variables do not represent any data directly but reference other data. FBs can be instantiated inside programs or other FBs, each instance corresponding to a different data memory region. Since FBs have also internal variables, invocation of a given FB instance with the same arguments (inputs) does not always yield the same output values. FB instances can be passed as parameters to another FB or function, and their variables are accessed like a structured data type, but some rules are needed to be coherent [9,10]. Programs are similar to FBs, except they can declare global variables. Programs are instantiated in resources, where their inputs and outputs are connected to directly represented variables, resource or configuration global variables.
- **Global variables**: They can be declared in configurations, resources or programs, and used inside programs or FBs provided they are declared as external and they are in the scope of the element where the program or FB is declared.
- **Directly represented variables**: They represent the association of a data element with physical or logical locations in the programmable controller’s input, output, or memory structure. The identifier of directly represented variables starts with a “%” character, followed by a location prefix, a size prefix, and the position. They can be associated with other symbolic identifiers in the declarative part of an element. Their use is inevitable when mapping variables to I/O.
- **Data types**: The standard also defines elementary data types, but allows the user to define its own data types: arrays, structures, enumerations, subranges and direct derivations.

It is worth discussing some aspects that are not well explained in the standard:
(1) **The use of directly represented variables.** These variables are located in positions of the memory or I/O of the PLC. However, this can give rise to indirect communication between the programs, which is not explicitly stated in the standard. This communication could happen if two programs access the same memory position. It is also unclear whether or not these variables must be declared in configurations or resources since there are some examples in the standard where they are used as inputs of programs without previous declaration (Fig. 20 of [9]), in contrast with the strict rule that applies to variables used inside POUs.

(2) **Concurrency problems.** In IEC 61131-3 tasks are used to express concurrent execution. However, there is a lack of constructs to define task synchronization and intertask communication. There are global areas but there is no easy way to get mutual exclusion or a similar facility that is present in concurrent languages (the semaphore function block has been removed in the last version of the standard). There are only two rules that apply to FB execution:

- If a function block receives more than one input from another function block, then when the former is executed, all inputs from the latter shall represent the results of the same evaluation.

---

Fig. 2. Example of IEC configuration, resource, program and their associated C-elements (shown schematically).
• If two or more function blocks receive inputs from the same function block, and if the ‘destination’ blocks are all explicitly or implicitly associated with the same task, then the inputs to all such ‘destination’ blocks at the time of their evaluation shall represent the results of the same evaluation of the ‘source’ block.

(3) PLC cyclic operation. While traditional PLC can scan the program in a periodic way, triggered by a clock, or cyclically, IEC 61131-3 defines several kinds of tasks: periodic, non-periodic—triggered by an event—and what we could call a ‘default task’ that controls all the programs that have no explicit task association. In part 8 of the standard it is suggested that the ‘default task’ may be in charge of scanning physical inputs and outputs [10]. While this part gives recommendations and not compliance requirements, it is worth stating that we are doubtful about the practical use of this last recommendation. In fact, control tasks usually sample inputs and set outputs at regular intervals, which is not the case when this job is done by the default task, since it has the lowest priority and is only executed if the other tasks are not active.

3. Proposed implementation

3.1. Implemented features and simplifications

A summary of the implemented features is presented in Table 1. Fig. 3 shows the implementation of the Software Model.

Following the above discussion, our implementation is based on these criteria:

(1) Directly represented variables can only be used in the declarative part of configurations and resources, in the connection of input and outputs of programs and in the connection of the trigger variable of task declarations.

(2) A given location in memory, input or output can only be accessed by a single variable. This is a non-normal restriction that we impose in order to promote good software practices.

(3) Every task can perform the scan of physical inputs and outputs for the directly represented variables it uses. Each time the task is executed, it will read the input, execute its associated programs and set the outputs.

(4) Data concurrency is achieved if programs use only local variables. At the beginning of the task execution the local variables are refreshed, while at the end some of the resource or configuration global variables get a new output value. Both operations are atomic. In fact, the standard refers only to concurrency in FB execution, but we extended the rule to programs, which in turn cover FBs.

In this way, programs use only symbolic variables, thus making their reuse easy. In addition, the only global variables that can be seen in the whole configuration are those explicitly declared in the declarative part of the configuration or resource. Two programs can communicate via some of the global variables. As a consequence of rule 2, a resource or a program can get exclusive access to some positions of input or output.

In addition, further simplifications have been done in the first version. Among them, we point out:

(1) Since our implementation is based on a high level language POSIX/C, and using PCs, we restrict the use of directly represented variables to locations in input or output space.

(2) The task will not control the execution of FBs, but only the execution of programs.

(3) Aperiodic tasks are only allowed to be triggered by variables located at some physical inputs; see Section 3.3.
(4) We consider a single configuration.
(5) In practice, we have worked with single processor systems. Thus, resources are always assigned to the same processor. However, we will mention how this assignment could be done for multi-processor systems under RTLinux.

It is also worth noting that IEC 61131-3 allows multitasking in either a preemptive or non-preemptive environment. By using a Real-Time Operating system we are selecting the preemptive case. Most part of PLCs have a fixed number of tasks. Our implementation has no software restriction on this number and is flexible with the priority and period of the task. On the other hand, it is true that some PLCs offer so many predefined types of tasks that they can fit most part of application needs, for instance Siemens’ Organization Blocks [11].

3.2. Programming organization units and POSIX/C

The concepts introduced by IEC 61131-3 standard can be implemented using a general-purpose language. POSIX/C has been chosen [12,13] since its Real Time extensions allow the PLC to get the desired temporal response in control systems. As pointed out by one of the referees, the use of C++ would make implementation easier. However, when this project started there was a question about the use of C++ in RTLinux FAQ section [14]. The answer stated that the use of C++ is a “hazardous effort”, even though some code example is included in the RTLinux sources. This and the fact that the end user does not have to modify or even read the generated code, led us to use C.

In this and next sections, the notation c-function, c-type or c-structure (in general c-something) will be used to distinguish normal C programming language concepts from IEC 61331 concepts named with the same word. Task will refer always to IEC 61131 tasks. Pthread means POSIX thread, as usual. We will consider RTLinux as a specific Operating System.

We start our discussion with FBs, since there is no equivalent in procedural programming languages such as Pascal or C. Data can be defined using a new c-type, a c-structure. The algorithm is a c-function, which receives a pointer to the c-type (Fig. 4). Thus, the FB type can be compiled to an

---

**IEC SINTAXIS**

```c
Function Block Average

FUNCTION_BLOCK AVERAGE
VAR_INPUT
  xnew: INT;
END_VAR
VAR_OUTPUT
  xave: INT;
END_VAR
VAR
  xold: INT;
END_VAR
(*HECO NOT SHOWN*)
END_FUNCTION_BLOCK
```

**IMPLEMENTATION**

```c
typedef struct _average_structp{
  int xnew;
  int xave;
  int xold;
} average_structp;

average_structp
int do_average(average_structp *);
int do_average(average_structp *average){
  /* c-Algorithm
  * Use average->xnew, */
  /* average->xave, */
  /* and average->xold */
}
```

---

Fig. 4. FB type “AVERAGE” and its corresponding c-elements.
object file (.o). Every instance of a given FB type is a new variable of the corresponding c-type, whose fields include the internal variables (the state of the FB) and the external variables that are accessed by means of their address (a pointer in C). Apart from this last detail, input, output, external and internal variables are treated in the same way in the c-structure, regardless of its mode. The IEC to POSIX/C translator checks that the input variables are not overwritten within the algorithm. Programs can be translated in a similar way.

When calling an FB from a program, the following steps should be performed inside the corresponding c-function (Fig. 2):

- To fill the fields of the c-structure corresponding to input variables.
- To call the c-function with the proper pointer. Outputs of the FB are modified inside the c-functions.
- To update the variables that are connected to the outputs of the FB.

A remark should be made about IEC 61131-3 data types: they can also be translated into c-data types in a simple way, so we are not going to discuss this point any further.

3.3. Configuration elements and RTLinux modules

We continue our discussion by explaining how to map configuration elements into RTLinux modules so as to actually run the PLC. It is worth recalling that modules are nothing more than c-programs with an init function (to insert the module) and a cleaning function (to remove it). The program is compiled into an object file (name.o). Module insertion into the Linux kernel allows extending its functionalities, the modules become part of the kernel itself. RTLinux is based on the use of modules [15]. All the modules share the same address space with the kernel.

A configuration like the one which is shown in Fig. 2 has global variables and resources, and the resources themselves contain global variables, programs and tasks. We propose a division of the whole configuration into modules (Fig. 5), following the preliminary ideas that were presented in [16]. A single module could be produced, but the division into modules provides some flexibility, as we will see below.

Since all the kernel variables that a module uses must have been created before it is inserted, next paragraphs are listed according to insertion order.

First, several modules have to be inserted so as to make some functions available to the whole kernel. These functions are related to standard and user defined FBs as well as to elementary and user defined data types.

Then, the configuration module is inserted. The configuration module declares the configuration global variables in a single c-structure, which can then be accessed by all the resources (and in fact, by the whole kernel). It also declares a table (Global Function Table, GFT) whose entries are pointers to c-functions. They correspond to the different types of programs used in the configuration. Since RTLinux modules and the kernel share the same address space, the global area can be seen from other modules. Thus, the

Fig. 5. The IEC 61131-3 model in RTLinux. The number in the box shows the insertion order.
configuration becomes part of the kernel operating system, which provides storage area and, of course, much more abilities that will be needed for the PLC, such as the scheduling of kernel threads.

Programs are similar to FBs and have two sides: data and algorithm. The data of program instances are declared inside the Pthreads of the resource (see below). The algorithm (a c-function) is implemented in a RTLinux module. The module defines the function and has a single Pthread, whose only objective is to fill its entry in the GFT with a pointer to the c-function. A module is inserted for each program type used in a configuration.

Each resource gives rise to a RTLinux module. In this module, each task of the resource is implemented as a Pthread. For periodic tasks, the period and the priority can be associated to the Pthread, taking into account that the IEC 61131-3 priority convention is different from that of POSIX. Fig. 6 gives an outline of the code of the resource module with a periodic task. Periodic behaviour is obtained by reading the time (clock_gettime primitive), and then entering an infinite loop where function execution is done, a period is added to the previous time at each cycle, and a sleep function is called (clock_nanosleep primitive).

Aperiodic tasks can be treated in different ways and needs further explanation. In our implementation, the trigger variable is restricted to be located at some given positions of physical inputs. This is because we used I/O cards that can be configured to detect changes in I/O signals by means of the generation of an interruption. In this case, the IEC task is associated to an interrupt handler and a Pthread. The interrupt handler checks the rising edge of the variables and wakes up the corresponding thread. Thus, detection of the edge is fast (a few microseconds in typical PCs), but the actual execution of the programs depends on the priority of the task. But we point out that, in contrast to periodic tasks, this is a very specific implementation. Fig. 7 gives an outline of the code. Note that this code makes use of POSIX signals, so the patch from Vidal and Ripoll [17] must be applied. If this patch is not installed, a similar effect can be obtained using non-posix functions from RTLinux that suspend and wake up a thread.

Other possibilities for dealing with aperiodic tasks could be: (i) to provide a check of rising edges of boolean trigger variables, whenever they are changed after program execution; (ii) to provide a polling of I/O if the trigger variable is a physical input (the period of this polling determines the reaction time of the PLC).

Continuing with the explanation of the resource module (Fig. 5), the resource global variables are implemented again as a c-structure, defined as static global variable, so that they are visible to the resource module but not to other modules. Global configuration variables and GFT are defined as extern c-variables. In order to associate a resource with a given processor, RTLinux provides a non-POSIX function to set the CPU of a given Pthread.

In order to take into account the concurrency effects mentioned in the standard, we use local copies of the variables inside the resource’s Pthreads. The global variables are used only at the beginning and at the end of the algorithm. The code of a Pthread contains several parts. Let us consider a periodic Pthread (Fig. 6). It has the following parts:

- A declarative part for local variables. The Pthread will execute several programs, so declarations of instances of the programs (i.e., associated c-structures) are needed as well.
- An external variable addressing part. External variables in programs and in the FBs inside them are accessed by reference. For instance, if an external variable is declared in a program type, once a program is instantiated within a resource, the external variable should belong to resource or configuration areas. Therefore, in this part of the code, the pointers have to be set to the proper address. The IEC-to-C translator has to deal with these situations, using the concept of compilation interface [10], which holds information about the names and types of input, output and external variables of a POU.
- A third part is the periodic repetition of the following steps:
  - Copy from global to local variables. When the variables are located in I/O space (directly represented variables), they must be read from these addresses, which can imply a hardware access. This operation must be atomic.
  - Reading the pointers to c-function in the GFT and execution of the programs, that is, to fill the fields of the c-structure and to call the c-functions associated. After every c-function call, the outputs of the programs are stored in the connected variables.
  - Updating global variables. Finally, the global variables that can be modified by the programs (output or external variables) are refreshed with the local copies. If a variable addresses a physical output, the hardware writing must be done. Again, this is an atomic operation.

In all cases, the access to global areas must be done under mutual exclusion between the tasks.

In this schema, the typical scan of cyclic programs is performed independently in every task and global areas are updated at the end of the cycle. More flexible schemas should allow the user to decide when a global variable must be updated or how to synchronize the tasks [3].

Using different modules for each program type, each resource and for the configuration global data (instead of getting a single module for the whole IEC configuration) is not imperative, but allows on-line changes in programs. In our model, a program could change its algorithm, but not its declarative part. Once the new program is compiled into an RTLinux module, the manager can insert it (changing the name in order not to collide with previously inserted modules). The new module will fill the GFT with the new
Assume an IEC TASK defined in a resource as:

```c
TASK_PERIODIC {INTERVAL:=-100ms, PRIORITY:=3};
```

/* include files */

/* Declaration of external variables: global variables of the configuration and GPT */
/* (global function table), spin locks to protect access to shared variables etc */

/* Declaration of the variables of the resource and other stuff */

```c
static pthread_t threadplc_PERIODIC;
```

/* The function executed by the periodic thread */

```c
void *taskplc_PERIODIC(void *arg) {
  /* Declaration of Variables for IEC programs associated to TASK_PERIODIC */
  /* (data and pointer to functions). Also local variables for resource */
  /* and configuration variables and other stuff */
  ...
  struct timespec period, t;
  struct sched_param p;
  int prio;
  ...
  /* Set up parameters and, in case the IEC program contains an external */
  /* variable, also set up pointers to them */
  ...
  prio=100-3; /* 100 is the maximum allowed priority in RTLinux */
  period.tv_sec=0;
  period.tv_nsec=1000000; /* 10 ms = 1e7 ns */
  p.sched_priority=prio;
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
  pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, NULL);
  pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
  ...
  clock_gettime (CLOCK_REAL_TIME, &t);
  while(1){
    /* Atomic reading of GPT, so as to know the functions to be executed */
    ...
    /* Atomic copy from global variables to local variables */
    ...
    /* Execution of programs associated to tasks */
    ...
    /* Atomic update of global variables changed by program execution */
    ...
    /* Sleep until next execution */
    /* timespec_add is a macro of RTLinux that helps in making */
    /* 't = t + period' in a proper way (see rtlinux.h) */
    timespec_add(&t, &period);
    clock_nanosleep (CLOCK_REAL_TIME, TIMER_ABSTIME, &t, NULL);
  }
  return 0;
}
```

```c
int init_module(void){
  /* Create thread for IEC tasks */
  pthread_create(&threadplc_PERIODIC, NULL, taskplc_PERIODIC, NULL);
  /* Initialize variables of the resource */
  ...
  return 0;
}
```

```c
void cleanup_module(void){
  int ret;
  ...
  pthread_cancel(taskplc_PERIODIC);
  pthread_join(taskplc_PERIODIC, (void *) ret);
  ...
}
```

Fig. 6. Skeleton of a periodic task. Details for periodic execution are shown, while other needed primitives are only outlined.
pointer, so that any Pthread that needs to execute this program type will read the new one. A mechanism has been implemented so that it is possible to know when the pointer of the GFT can be safely changed and the old module can be removed.

Fig. 5 also shows a process named manager. It is a Linux process in user space which is in charge of the insertion and removal of modules, the communication with the modules for monitor or debugging purposes and the communication with external systems. Its requirements are not
real time and its description is out of the scope of the present paper.

4. IEC 61131-3 TO POSIX/C translator

4.1. Outline of translator implementation

The ideas presented above serve as guidelines for a translator that generates POSIX/C code from textual descriptions according to IEC 61131-3. Afterwards, the code generated must be compiled using a c-compiler, gcc, so as to obtain the modules. The algorithmic part of FBs and programs is written in IL. Even though this language has an assembler-like appearance, the standard points out that it is not required that IL is considered an instruction set for any real or virtual machine [10]. We will consider it a rudimentary way of defining algorithms:

• There are some load instructions that store the operand in the current result. Store operations copy the value of the current result to a variable.
• Operators replace the current result by its previous value operated upon by the operator with respect to the operand. Open parentheses defer the operations until they are closed.
• There are also jumps and function and function block calls.

We have used a single 64 bit wide accumulator for integer or bit-string types (long long int in C) and another 64 bit wide accumulator for real types (double in C). The translation of the algorithms is rather easy but cumbersome. Special care must be taken with the sign extension. For user types, wide accumulator for real types (double in C). The translator has been built using flex and bison [18,19]. Among the implemented features we point out the following:

• Numeric data types, bit-string data types and duration.
• User defined data types: arrays, structures, enumerations and derived types.
• User defined function blocks.
• Definition of configuration and resources, declaring the task and their characteristics.

4.2. Comments on the formal language definition

Formal language definition can be found in Appendix B of part 3 of the standard [9]. A few remarks should be made, some of them already treated by Sousa [2].

• Some conflicts have been found when implementing the translator. The origins of these conflicts are some ambiguities in the standard, specially the definition of literals. For instance, the string “1” can be understood as a boolean or as a integer, which entails further problems in the syntactic tree.
• Although the standard allows for declaring global variables in programs, there is no place for such a construction in the formal definition of the languages, which is probably an omission.
• IL operators are surprisingly missing from the keyword list. The present definition allows a user to define a function named LD. Thus, an instruction like “LD A” could be interpreted either as a function call or as the load operation.
• Some IL operators, like ADD, can be interpreted as function as well. In our implementation, we see them exclusively as operators.
• The connection of programs in resources is also confusing (Table 2). It seems that input or output variables that are not simple types (arrays or structures) can be connected element by element. However, when calling an FB this is not possible, even though both constructions seem very similar. On the other hand, the variables to which input and output are connected can be fields of a structure but not one field length, and they can be variables of another resource, violating the scope of resource global variables defined in the standard itself. Thus, the syntax seems to allow an unnecessary resource reference. This possibility has been restricted in our implementation.

5. Evaluation

Using the main characteristics cited by Lewis as Ref. [1], the implemented translator fulfills the following:

Table 2
Examples of program connection and FB call

<table>
<thead>
<tr>
<th>Description</th>
<th>Allowed by standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROG1 is a program instance in resource RESI. The same configuration declares another resource, RES2, which has a global variable X. A is a structured variable, ON being one of its fields</td>
<td></td>
</tr>
<tr>
<td>Program connection in a resource</td>
<td>Allowed by standard</td>
</tr>
<tr>
<td>FILTER (A.ON:=X, …)</td>
<td>…</td>
</tr>
<tr>
<td>FB call inside a program</td>
<td>Not allowed by standard</td>
</tr>
<tr>
<td>CAL FB1( A.ON:=Y, …)</td>
<td>…</td>
</tr>
<tr>
<td>Program connection in a resource, where the associated variable belongs to another resource</td>
<td>Allowed by standard</td>
</tr>
<tr>
<td>FILTER (A.ON:=RES2.X, …)</td>
<td>…</td>
</tr>
</tbody>
</table>
• Promotion of well-structured programming, due to user defined FBs and programs.
• Strong type typing, since the translator raises an error if the data types are not the same in an assignment or in a program or FB connection.
• Control execution. In the resource, the user can assign the programs to different tasks, each one having its own priority, period and trigger variable.
• Support for complex data types.
• Promotion of vendor independence. The PLC can be executed on any architecture RTLinux has been ported to, and, with slight modifications, on any POSIX operating system.

Tables in the standard are used to check system compliance. Our implementation fulfils about one-third of the features shown in these tables. More specifically, this percentage grows to 64% when considering the IL language features. The non-implemented features mainly refer to non-IL languages, SFC elements, time and string data types and variable declaration qualifiers.

We highlight some key points about our translator. First, it is an independent tool, which does not need to be integrated into other software development applications. The input to the translator is a text file. Second, the translator is focused on the standard and it does not make use of any vendor-specific instructions. Third, it is completely based on free software.

In addition, the present paper is the base for further development in order to include other languages; we only need to extend the translator to deal with them since the framework given by the IEC 61131-3 concept of configuration is the same.

Even though the performance is not the main objective of our implementation, we can get an approximate idea of the performance of the model on PCs and how it compares with commercial products. The comparison is not rigorous since he considered a 20 MHz processor, his model is also better than compiled programs for general purpose CPUs.

In this paper, we have considered the IEC 61131-3 software model and a specific implementation based on POSIX/C. The model improves software quality, but it presents several semantic shortcomings: concurrency, variable scope, use of directly represented variables, program communication and cyclic execution of PLCs. Some interpretations have been proposed for these weak points together with some comments in the formal definition of the languages.

Any architecture for which a POSIX operating system is implemented can run a PLC thanks to the implemented translator, thus extending the possibility to execute control systems based on IEC 61131-3.

Further research can be derived from the presented work:
• Code optimisation, since the first version is not focused on performance.

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Suggested model Pentium 1 GHz</th>
<th>SIMATIC WinLC in dual Pentium 400 MHz</th>
<th>SIMATIC S7-300</th>
<th>Modicon Premium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit operation</td>
<td>0.038</td>
<td>0.230</td>
<td>0.1–1.9</td>
<td>0.06–0.87</td>
</tr>
<tr>
<td>Word boolean operation (16 or 32 bits)</td>
<td>0.035</td>
<td>0.06</td>
<td>0.1–4.1</td>
<td>0.08–1</td>
</tr>
<tr>
<td>Integer math: addition and subtraction (16 or 32 bits)</td>
<td>0.029</td>
<td>0.160</td>
<td>0.8–14.8</td>
<td>1.2–149</td>
</tr>
<tr>
<td>Integer math: multiplication and division (16 or 32 bits)</td>
<td>0.116</td>
<td>0.13</td>
<td>0.6–60</td>
<td>1.4–48</td>
</tr>
<tr>
<td>Floating point addition and subtraction (32 bits)</td>
<td>0.033</td>
<td>0.160</td>
<td>1.4–60</td>
<td>1.4–47</td>
</tr>
<tr>
<td>Floating point multiplication and division (32 bits)</td>
<td>0.044</td>
<td>0.283</td>
<td>1.4–2500</td>
<td>1.4–2520</td>
</tr>
</tbody>
</table>
- Inclusion of more standard features, such as other languages, error management etc.
- Implementation on other hardware architectures, for which RTLinux has been ported. Thus, the hardware could be chosen to fit the application needs.

Acknowledgements

The authors thank the anonymous reviewers for their helpful and valuable suggestions and comments.

References