MINI-DCS SYSTEM PROGRAMMING IN IEC 61131-3 STRUCTURED TEXT

Received 25th February; accepted 15th May 2008.

Dariusz Rzońca, Jan Sadolewski, Andrzej Stec, Zbigniew Świder, Bartosz Trybus, Leszek Trybus

Abstract:

A prototype environment called CPDev for programming small-distributed control-and-measurement systems in Structured Text language of IEC 61131-3 standard is presented. The environment is open what means that the code generated by the compiler can be executed on different hardware platforms. However, an interpreter, another words - a virtual machine, must process such universal code similarly as programs written in Java. The CPDev environment consists of the compiler, simulator and configurer of hardware resources (i.e. communications). They are developed in C# at MS.NET Framework 2.0 platform. CPDev is open allowing the user to create function blocks and libraries. External interface procedures (drivers) can be written by hardware designers and linked with the universal code. Free selection of data types required by different applications is provided. Virtual machine written in ANSI C is dedicated for a particular processor. So far the machines for AVR, MCS-51 and PC have been developed. Programming a mini-DCS system from LUMEL Zielona Góra has been the first application of CPDev.

Keywords: *mini-DCS system, control program execution, ST language, compiler, IEC 61131-3 standard, virtual processor.*

1. Introduction

Domestic control-and-measurement industry manufactures transmitters, actuators, drives, PID and PLC controllers, recorders, etc. connected with distributed systems, are used for automation of small and medium scale processes. However, engineering tools applied for programming such devices are rather limited and do not correspond to IEC 61131-3 standard [1] (Polish law since 2004). This restricts effectiveness of competition with foreign products. The problem may be solved to some extent by developing an universal, open engineering environment for programming control devices, particularly small PID, PLC and multifunction controllers according to IEC 61131-3 (further denoted IEC for brevity).

Basic task of control engineering tool is to compile source program written in one of IEC languages into machine code of a given processor. Change of the processor requires a new compiler. A tool, to be called *universal*, should be able to generate a code executable on different platforms, particularly such as AVR, ARM, MCS-51 and PC, if domestic devices are considered. However, such universal code must be executed by an interpreter that translates instructions of this code into instructions of the machine language. Each of the platforms must have its own interpreter. So the universal code is in fact a kind of intermediate code into which the source program is compiled. Generally speaking, it resembles somewhat the concept of Java virtual machines capable of executing programs on different platforms [2]. Therefore the interpreters of the universal code will also be called *virtual machines* here.

Engineering environment can be considered *open* if it provides the following:

- tools for development of user functions and function blocks, which are program units suitable for reuse,
- specifications of I/O and communication interfaces in the form of prototypes of driver procedures, which are common for different processors and hardware solutions (the same way of procedure call; internal bodies can be different).

The environment may be called *flexible*, if of all data types available in the IEC standard, the user can freely choose the ones suitable for particular application. For instance, PID and PLC algorithms need somewhat different sets of data types.

This paper presents current state of work on engineering environment for programming small control-and-measurement devices and distributed mini-systems according to the IEC standard. An example of such mini-DCS involving instruments from LUMEL Zielona Góra is shown in Fig.1. The environment, called CPDev (*Control Program Developer*), satisfies universality, openness and flexibility requirements stated above. Initial information on CPDev has been presented on recent Real Time Systems conference [3, 4]. CPDev is being developed at Microsoft. NET Framework 2.0 [5].

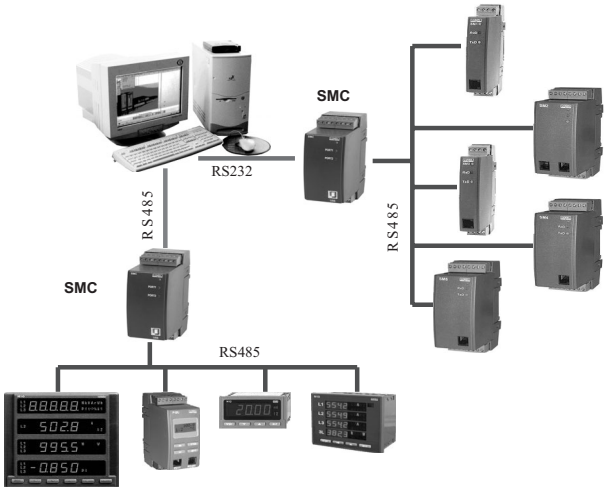

Fig.1. Example of mini-distributed system with equipment from LUMEL Zielona Góra.

The paper is organized as follows. Structure of CPDev and the way in which the code for different platforms is generated are explained in Sec.2. The environment consists of ST language compiler (also called CPDev), simulator CPSim and configurer CPCon of hardware resources. The compiler produces a file with the universal code for virtual machine. User interface is presented in Sec.3 together with simple example. Functions and function blocks available in CPDev libraries are listed in Sec.4. The user can create his function blocks and libraries. Simulator CPSim and configurer CPCon are described in Secs.5 and 6, respectively. The configurer generates a file with hardware allocation map, which provides another data needed by virtual machine. Configuration of a mini-DCS test system from LUMEL, involving a new SMC programmable controller, is given as an example. Language of virtual machine, operation cycle, and the way in which program interfaces are adapted to different processors and hardware solutions are explained in Sec.7.

2. General characteristic of CPDev environment

2.1. Programming languages

The IEC standard defines five programming languages, i.e. LD, IL, FBD, ST and SFC, allowing the user to choose one suitable for particular application. Instruction List IL and Structured Text ST are text languages, whereas Ladder Diagram LD, Function Block Diagram FBD and Sequential Function Chart SFC are graphical ones (SFC is not an independent language, since it requires components written in the other languages). Relatively simple languages LD and IL are used for small applications. FBD, ST and SFC are appropriate for medium-scale and large applications. John and Tiegelkamp's and Kasprzyk's books [6,7] are good sources to learn IEC programming.

ST is a high level language originated from Pascal, Ada and C, especially suitable for complicated algorithms (e.g. for PID self-tuning). Equivalent code for a program written in any of the other languages can be developed in ST, but not *vice versa*. Hence most of engineering packages use ST as a default language for programming user function blocks. Due to such reasons it has been assumed that ST will be employed as base language in CPDev environment. In future, programs written in the other IEC languages will be converted into ST (graphic editor for FBD is being developed).

2.2. CPDev environment

The CPDev environment (called also package) involves four programs shown in Fig.2. At PC side we have:

- CPDev compiler of ST language,
- CPSim software simulator,
- CPCon configurer of hardware resources.

The programs exchange data through files in appropriate formats. The CPDev compiler generates universal code executed by virtual machine VM at the controller side. The machine operates as an interpreter. The executable code is a list of primitive instructions of the virtual machine language called VMASM assembler. VMASM is not related to any particular processor, however it is close to somewhat extended typical assemblers. Brief characteristic of VMASM is given in Sec.7. CPSim simulator also involves the virtual machine (in this case at the PC side).

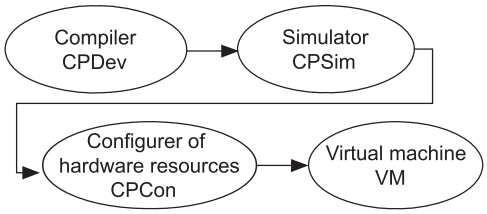

Fig. 2. Components of CPDev environment.

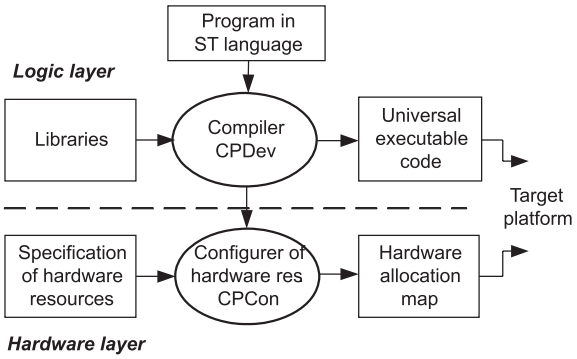

Fig. 3. Logic and hardware layers of CPDev environment.

Fig.3 shows how the CPDev compiler and CPCon configurer cooperate. Separation of program compilation at the logic layer from hardware configuration at the hardware layer simplifies generation of the code for different platforms. The ST source program is compiled into universal executable code applying relative addresses defined in ST (called local addresses here). The compiler employs functions, function blocks and programs stored in libraries. Configuration of hardware resources at the second layer involves memory, input/output and communication interfaces. This includes memory types and areas, numbers and types of inputs, outputs and communication channels, physical addresses, validity flags, etc. Allocation of hardware resources has the form of a map that assigns local addresses to physical ones. Virtual machine at the target platform, given the code and hardware allocation map, is able to execute calculations.

2.3. Different hardware platforms

From CPDev viewpoint hardware platforms differ in terms of hardware allocation maps and not in executable code. The code remains the same, hence it is called *universal*. Some diversification of binary code is possible for optimization of execution by particular processor (see Sec.7).

Software deployment at different platforms is illustrated in Fig.4. Virtual machine is dedicated to a particular processor. So far the machines for AVR, MCS-51 and PC have been developed (PC machine is a part of CPSim). ARM 7 is considered as the next one. While developing the machines attention has been given to decrease time overhead for code interpretation. Similarity of the VMASM assembler to machine languages is the advantage. Indexing mechanism for instruction interpretation has been employed [3].

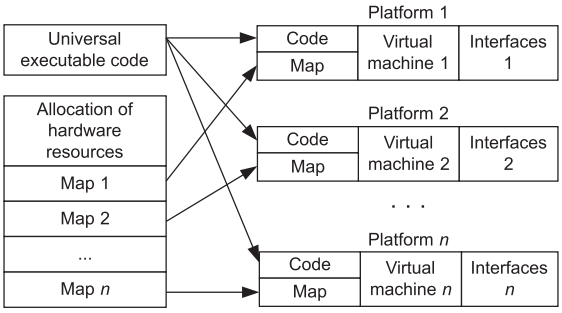

Fig. 4. Software deployment at different hardware platforms.

3. User interface

3.1. Data types

The CPDev compiler is able to process twenty elementary data types defined in the IEC standard [1, 6, 7]. However, only a part is needed while programming a specific device or a system. For an SMC controller presented further ten data types listed in Table 1 have been selected (the same types are used in Industrial IT 800xA DCS from ABB). The types available in particular instance of the compiler are selected by means of an XML configuration file (Sec.7). The elementary types may be used to define derived types such as alias, arrays and structures [6, 7].

Table 1. Data types for SMC programmable controller.

Type	Size (range)	Type	Size
BOOL	1B (0, 1)	WORD	2B
INT	2B (-32768...32767)	DWORD	4B
DINT	4B (-2^{31} ... $2^{31}-1$)	STRING	variable length
UINT	2B (0 ... 65535)	TIME	4B
REAL	4B, IEEE-754 format	DATE_AND_TIME	8B

3.2. Program in ST language

Main window of user interface in CPDev is shown in Fig.5. It consists of three areas:

- tree of project structure, on the left,
- program in ST language, center,
- message list, bottom.

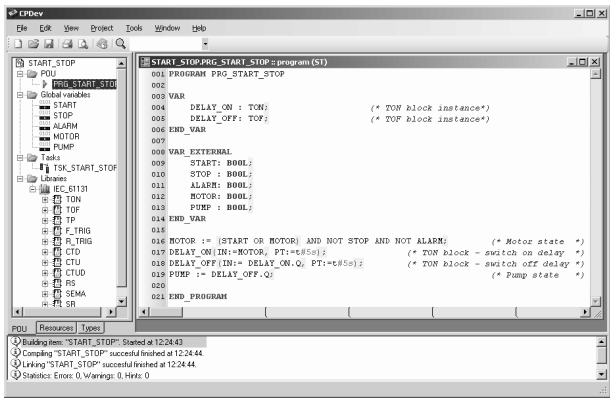

Fig.5. User interface in CPDev environment.

Tree of the START_STOP project shown in the figure includes POU unit with the program PRG_START_STOP, five global variables from START to PUMP, task TSK_START_STOP, and two standard function blocks TON and TOF from IEC_61131 library.

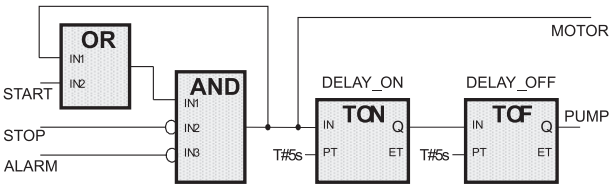

Fig. 6. START_STOP system for control of a motor and pump (with delay of 5 seconds).

The PRG_START_STOP program seen in the main area is written according to ST language rules. The first part involves declarations of instances DELAY_ON, DELAY_OFF of the blocks TON and TOF. Declarations of the global variables (EXTERNAL) are the second part, and four instructions of the program body, the third one. The instructions correspond to FBD diagram shown in Fig.6. So one can expect that certain MOTOR is turned on immediately after pressing a button START and the PUMP five seconds later. Pressing STOP or activation of an ALARM sensor triggers similar turn off sequence.

3.3. Global variables and task

Global variables can be declared in CPDev either using individual windows or collectively at variable list. The list for the START_STOP project is shown in Fig.7.

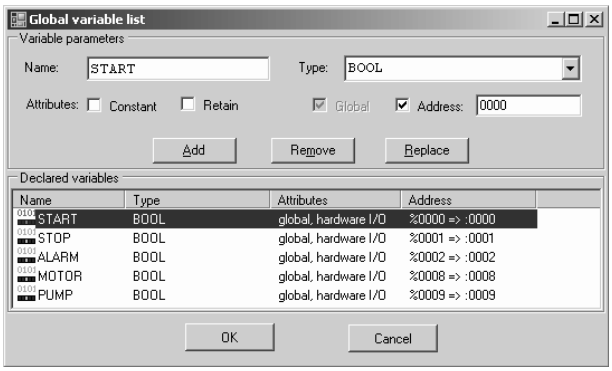

Fig. 7. Global variable list for the START_STOP project.

All variables have RETAIN attribute. The addresses specify *directly represented variables* [6, 7] and denote relative location in controller memory (keyword AT declares the address in individual window). Here these addresses are called *local*. As explained before, correspondence of local addresses to physical ones is defined by hardware allocation map. Variables without addresses (not used in this project) are located automatically by the compiler.

Window with declaration of the TSK_START_STOP task is shown in Fig.8. A task can be executed once, cyclically or continuously (triggered immediately after completing, as in small PLCs). There is no limit on the number of programs assigned to a task, however a program can be assigned only once.

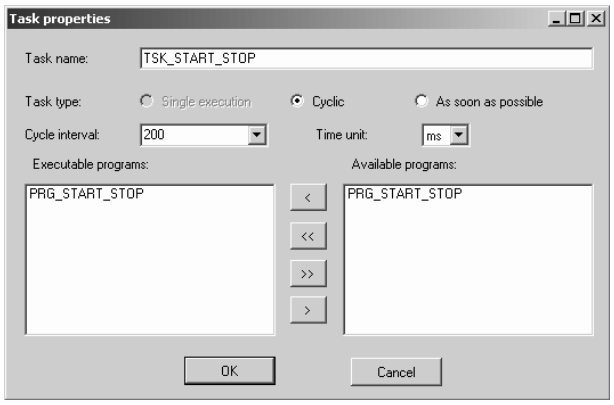

Fig. 8. Declaration of TSK_START_STOP task.

Text of the project represented by the tree is kept in an XML text file. Compilation is executed by calling Project->Build from the main menu. Messages appear in the lower area of the interface display (Fig.5). If there are no mistakes, the compiled project is stored in two files. The first one (*.xcp extension) contains universal executable code in binary format for the virtual machine. The second one (*.dcp) contains mnemonic code (Sec.7), together with some information for simulator and hardware configurer (variable names, etc.).

4. Functions and libraries

4.1. Standard functions

The CPDev compiler provides most of standard functions defined in IEC standard. Six groups of them followed by examples are listed below:

- type conversions: INT_TO_REAL, TIME_TO_DINT, TRUNC,
- numerical functions: ADD, SUB, MUL, DIV, SQRT, ABS, LN,
- Boolean and bit shift functions: AND, OR, NOT, SHL, ROR,
- selection and comparison functions: SEL, MAX, LIMIT, MUX, GE, EQ, LT,
- character string functions: LEN, LEFT, CONCAT, INSERT,
- functions of time data types: ADD, SUB, MUL, DIV (IEC uses the same names as for numerical functions).

Selector SEL, limiter LIMIT and multiplexer MUX from selection and comparison group are particularly useful. Variables of any numerical type, i.e. INT, DINT, UINT and REAL (called ANY_NUM in IEC [6, 7]) are arguments in most of relevant functions.

4.2. Function block libraries

Typical program in ST language is a list of function block calls, where inputs to successive blocks are outputs from previous ones (see Fig.6). So far the CPDev package provides two libraries:

- IEC_61131 standard library,
- Basic_blocks library with simple blocks supplementing the standard.

Table 2a lists blocks from the first library. Source programs for three of them are presented in Table 2b. The programs for the SR flip-flop and R_TRIG rising edge detector are obvious (CLKp denotes previous value of CLK). The output ET (*Elapsed Time*) of the timer TON is the difference between current value of the system time counter read by CUR_TIME() function, and the value of local variable sTime set at the rising edge of the input IN. The output Q is set to TRUE when ET becomes equal to the input PT (*Preset Time*).

Table 2. (a) Standard blocks from the IEC_61131 library; (b) programs of SR, R_TRIG and TON.

a)

Bistable elements		**Edge detectors**	
flip-flop	RS	rising	R_TRIG
flip-flop	SR	falling	F_TRIG
semaphore	SEMA		
Counters		**Timers**	
up	CTU	pulse	TP
down	CTD	on-delay	TON
up-down	CTUD	off-delay	TOF
		real time clock	RTC

b)

```
FUNCTION_BLOCK SR
 VAR_INPUT
  S1: BOOL;
  R: BOOL;
 END_VAR
 VAR_OUTPUT
  Q1: BOOL;
 END_VAR
Q1 := S1 OR (NOT R AND Q1);
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK R_TRIG
 VAR_INPUT
  CLK: BOOL;
 END_VAR
 VAR_OUTPUT
  Q: BOOL;
 END_VAR
 VAR
  CLKp: BOOL := FALSE;
 END_VAR
Q := CLK AND NOT CLKp;
CLKp := CLK;
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK TON
 VAR_INPUT
  IN: BOOL;
  PT: TIME;
 END_VAR
 VAR_OUTPUT
  Q: BOOL;
  ET: TIME;
 END_VAR

IF NOT IN THEN
  Q := FALSE;
  ET := t#0ms;
ELSE
  IF(ET < PT) THEN
    Q := FALSE;
    ET := ET+TASK_CYCLE;
    PT_x := PT;
  ELSE
    Q := TRUE;
    ET := PT_x;
    PT_x:= CUR_TIME();
  END_IF
END_IF
END_FUNCTION_BLOCK
```

The second library involves blocks of Table 3 (names are dropped). They are similar to the blocks available in multifunction instruments such as PSW-166 from ZPDA Ostrów Wlkp., Sipart DR24 from Siemens or Protronic 550 from ABB. The blocks have up to four inputs and one or two outputs. The integrator and totalizer execute calculations on double precision numbers (LREAL in IEC).

Table 3. Simple blocks of Basic_Blocks library.

Mathematics linear function division with non-zero divisor square root with linear origin difference amplifier integrator pseudo-random numbers	**Flop-flops, pulsers** D flop-flop T flip-flop JK flop-flop one cycle delay pulse duration time totalizer (integration, pulse) square wave triangle wave
Memories analog memory binary memory	**Filters** lag filter (1st order) lead filter
Signal analyzers maximum over time minimum over time	

The user can develop functions, function blocks and programs, and store them in his libraries. Functionality of the compiler with respect to tables and structured variables follows the IEC standard.

5. CPSim simulator

The compiled project may be verified by simulation before downloading into the controller. The CPSim simulator can be used in two ways:

- before configuration of hardware resources (simulation of the algorithm),
- after configuration of the resources (simulation of the whole system).

The first way involves logic layer of the CPDev environment (Fig.3). PC computer operates as virtual machine executing universal code. Simulation window (not shown) is a table with variables, their types, simulated values and local addresses.

The second way requires configuration of hardware resources, so it is application dependent. In case of mini-DCS of Fig.1, the CPCon configurer generates hardware allocation map (*.xcp file) that assigns local addresses to physical ones (remote) and specifies conversion of ST data formats (Table 1) into formats accepted by hardware. The objective is to bring simulation close to hardware level, so CPSim uses both the code (*.xcp) and the map (*.xmc). Simulation window of the START_STOP project is shown in Fig.9. The two faceplates on the left present values of three inputs and two outputs (TRUE is marked). The user can select faceplates, arrange them on the screen and assign variables. Simulated values can be set both in group and in individual faceplates.

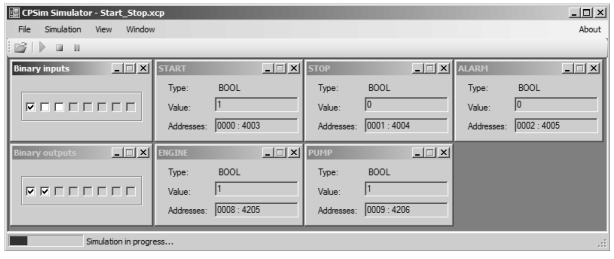

Fig. 9. Simulation of the START_STOP project.

So far the window of Fig.9 is used for simulation only. In future it will also be involved in on-line tests (*commissioning*).

6. Communication configurer CPCon

6.1. Mini-distributed system

The CPCon configurer defines hardware resources for particular application. The example considered here involves mini-DCS system with SMC programmable controller, I/O modules of SM series and other devices from LUMEL Zielona Góra (Fig.1). Modbus RTU protocol is employed [8] on both sides of SMC.

Fig.10 shows test realization of the system with SMC controller (on the left), SM5 binary input module (middle), and SM4 binary output module (on the right). The console with pushbuttons and LEDs (below) is used for testing. The PC runs CPDev package (and eventually SCADA). PC and SMC are connected *via* USB channel configured as a virtual serial port.

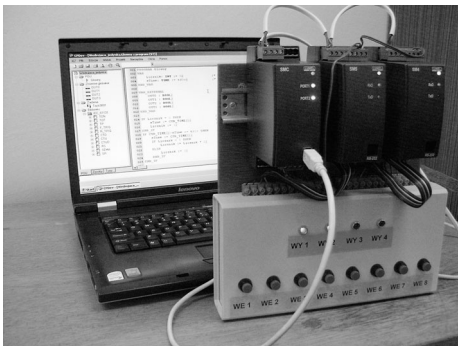

Fig. 10. Test set-up of mini-DCS system with SMC controller and SM I/O modules.

6.2. Functions of CPCon configurer

The CPCon functions are as follows:

- configuration of communication between SMC and SM I/O modules,
- creation of file with hardware resource allocation map (*.xmc),
- downloading the files with executable code (*.xcp) and map (*.xmc) to the SMC.

Recall that having the map the CPSim can be used in the second mode.

Main window of the CPCon configurer is shown in Fig. 11. The Transmission slot sets speed, parity and stop bits for PC↔SMC and SMC↔SM communications. Communication task table determines what question↔answer and command↔acknowledgment transactions take place bet-ween SMC controller (*master*) and SM modules (*slaves*). The transactions are called *communication tasks* and rep-resented by the rows of the table. The DCS system is configured by filling the rows, either directly in the table or interactively through a few windows of Creator of com. tasks (bottom).

The first row specifies communication between SMC and SM5 binary input module (remote). SM5 is connected to pushbuttons in the console (Fig.10), which, in case of the START_STOP project, set the variables START, STOP

and ALARM (Figs.5, 6). In SMC these variables have consecutive addresses beginning from 0000 (Fig.7). SM5 places the inputs in consecutive 16-bit registers beginning from 4003. So all variables can be read in a single Modbus transaction with the code FC3 (read group of registers [8]). However, since BOOL occupies single byte in CPDev, the interface of the virtual machine has to perform 16→8 bit conversion.

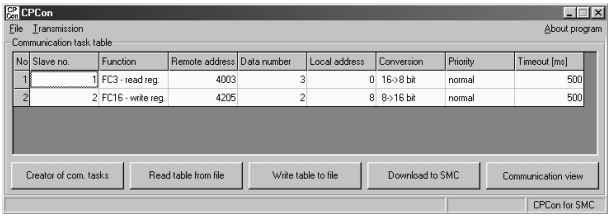

Fig. 11. Communication configuration of the START_STOP project.

Communication tasks are handled by SMC during pauses that remain before end of the cycle, after execution of the program. Single transaction takes 10 to 30 ms, depending on speed (max. 115.2 kbit/s). If the pause is large, the task can be executed a few times. It has been assumed that the task with NORMAL priority is executed twice slower than the task with HIGH priority, and the task with LOW priority three times slower. As seen in Fig.11, the communication with SM5 module has NORMAL priority. The Timeout within which transaction must be completed is 500 ms.

Second row of the Communication task table defines communication with the SM4 binary output module. SM4 controls the console LEDs. Two consecutive variables, MOTOR and PUMP, the first one with the local address 0008, are sent to SM4 by single message with the code FC16 to remote addresses beginning from 4205 (write group of registers). This time 816-bit conversion is needed.

7. Compiler and virtual machine

7.1. VMASM assembler

General task of the CPDev compiler is to transform text of the program in ST language into executable form for the virtual machine. The compiler consists of three parts that perform the following functions:

- *scanner* decomposes text of the program into tokens (lexical units),
- *parser* translates the list of tokens into mnemonic code of the VMASM assembler,
- *code generator* converts the mnemonics into executable code in binary format for the virtual machine.

Examples of primitive instructions of the VMASM assembler are given in Table 4 [3]. VMASM does not involve the notion of accumulator. Data are stored in memory as constants, variables or stacks. The constants begin with the hash sign #, the labels with the colon:. For example, the instruction MCD Q, #01, #00 initialises the variable Q with one byte (#01) of zero value (#00). Names of auxiliary variables and labels created automatically during compilation of complicated expressions or IF instructions contain question mark? (ST names are without question marks). Function instructions such as ADD, SUB or NOT are called directly by their names. A signature-involving name of the function and types of the arguments defines function execution. The same name, e.g. GE (greater-or-equal), can be used for all types from ANY_NUM group. Functions involving internal instructions are preceded by the keyword CALF (*Call Function*) and address of the argument set.

Table 4. Example of VMASM assembler instructions.

Instructions	Meaning	Instructions	Meaning
MCD	Constant initialization	GE	Greater or equal
MEMCP	Assignment	SHL	Bit shift to the left
ADD	Addition	JMP	Unconditional jump
SUB	Subtraction	JZ	Conditional jump
AND	Logic product	CONCAT	String concatenation
NOT	Negation	RETURN	Return from function

Depending on resources and applications, the virtual machine can handle all or only some of the twenty data types defined in IEC standard. This is determined by a library configuration file LCF (XML format), whose elements <deny-type> (of the TYPES section) eliminate unwanted data types. This makes the compiler flexible.

The LCF has restricted the number of types for SMC controller to ten (Table 1). LCF files may also be used to assign different binary codes for the same VMASM instruction at different processors. This opens the way to some form of code optimisation. So by means of the LCFs one can create any number of dedicated compilers.

7.2. Creating the virtual machine

Main part of the virtual machine for interpretation of the universal code has been written in ANSI C, so it can be directly adapted to different processors. The machine is an automaton operating according to Fig.12.

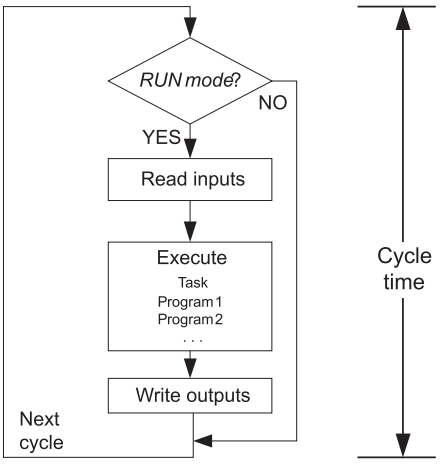

Fig. 12. Phases of virtual machine cycle.

The task consists of programs executed consecutively. The code involves binary identifiers of instructions and operands. The machine, similarly as any real processor, maintains program counter with the index (address) to the next instruction. The machine fetches the identifier of the instruction, decodes it, fetches the operands and executes the instruction. It also triggers I/O and communication procedures (drivers) and monitors time cycle of the task.

The main part of virtual machine depends on the type of the processor but does not depend on hardware solutions of particular platform. CPDev environment provides general specifications of external interfaces in the form of prototypes of I/O and communication procedures (names, types of inputs and returned outputs). The prototypes are kept in a file with *.h extension (similarly as stdio.h in C). The contents of corresponding binary file, e.g. with *.obj extension, can be written by hardware designers (as done by LUMEL for SMC controller). By linking the main part with this *.obj file the virtual machine for particular platform is created. This makes the CPDev environment open also in the hardware sense.

8. Conclusions

CPDev environment for programming industrial controllers and other control-and-measurement devices according to IEC 61131-3 standard has been presented. So far only the ST language is available. The environment is universal, since the compiled code may be executed on different platforms. However, the execution must be carried out by virtual machines dedicated for particular processors. This corresponds to the concept of Java virtual machines. Hardware allocation map defines available resources. The environment is open in terms of software and hardware. The user can program external interfaces (drivers). Mini-DCS system from LUMEL is the first application of the package.

One assumes that programs written in future in other languages (e.g. FBD) will be converted into ST before compilation. XML format for data exchange between languages has already been defined by PLCOpen [9].

ACKNOWLEDGMENTS

Support of MNiSzW under the grant R02 058 03 is gratefully acknowledged.

AUTHORS

Dariusz Rzońca, Jan Sadolewski, Andrzej Stec, Zbigniew Świder, Bartosz Trybus, Leszek Trybus* - Rzeszów University of Technology, Division of Computer Science and Control, 35-959 Rzeszów, W. Pola 2, Poland. E-mail: ltrybus@prz-rzeszow.pl

* Corresponding author

References

[1] IEC 61131-3 standard: *Programmable Controllers -Part 3, Programming Languages. IEC*. (2003).

[2] T. Lindholm, F. Yellim, *Java Virtual Machine Specification - Second Edition,* Java Software, Sun Microsystems Inc., 2004.

[3] D. Rzońca, J. Sadolewski, B. Trybus, "IEC 61131-3 standard ST compiler into universal executable code". In: *Real-Time Systems. Methods and Applications*, WKŁ, Warsaw, 2007, pp. 189-198 (in Polish).

[4] A. Stec, Z. Świder, L. Trybus, Functional characteristic of the prototype system for embedded systems programming. In: *Real-Time Systems. Methods and Applications*, WKŁ, Warsaw, 179-188 (2007) (in Polish).

[5] *C# Language Specification* - http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx (November 2007).

[6] K.H. John, M. Tiegelkamp, IEC 61131-3: *Programming Industrial Automation Systems*. Berlin-Heidelberg, Springer-Verlag, 2001.

[7] J. Kasprzyk, *Programming Industrial Controllers*. WNT, Warsaw, 2006 (in Polish).

[8] *Modicon MODBUS Protocol Reference Guide*. MODICON, Inc., Industrial Automation Systems, Massachusetts, 1996. www.modbus.org/docs/ PI_MBUS_300.pdf

[9] *XML Formats for IEC 61131-3 ver. 1.01 -Official Release*. www.plcopen.org/