Theoretical and Applied Informatics
ISSN 1896–5334
Vol.23 (2011), no. 1
pp. 21–35
DOI: 10.2478/v10179-011-0002-z

Development and Implementation of IEC 61131-3 Virtual Machine

BARTOSZ TRYBUS

Department of Electrical Engineering and Informatics
Rzeszów University of Technology
ul. W.Pola 2, Rzeszów, Poland
btrybus@prz.edu.pl

Received 19 November 2010, Revised 2 January 2011, Accepted 23 February 2011

Abstract: Virtual machine described in the paper is a runtime program for controllers in small distributed systems. The machine executes intermediate universal code similar to an assembler, compiled in CPDev engineering environment from source programs written in control languages of IEC 61131-3 standard. The machine is implemented as a C program, so it can run on different target platforms. Data formats and commands of the machine code are presented, together with the machine's Petri-net model, C implementation involving universal and platform-dependent modules, target hardware interface, input/output programming mechanisms, and practical applications.

Keywords: multi-platform virtual machine, assembler, IEC 61131-3, programmable controllers

1. Introduction

CPDev (*Control Program Developer*) is an engineering environment[1] for programming controllers in small distributed systems according to IEC 61131-3 standard [3]. Main goal of the standard is to increase quality of control software by defining dedicated programming languages and implementation procedures. CPDev integrates tools for programming, simulation, hardware configuration, on-line testing and running control applications on different platforms. Programs can be written in ST and IL textual languages (*Structured Text*, *Instruction List*) and in FBD graphical one (Function Block Diagram) [8,9].

CPDev compiles control programs into intermediate, universal code executed by runtime interpreter at the controller side. Following Java nomenclature, the interpreter is called virtual machine (VM), and its intermediate language, Virtual Machine Assembler

[1]Developed at Rzeszów University of Technology with support from MNiSzW R02 058 03 grant.

(VMASM). The VM machine is written in C, so it may run on different hardware platforms, from 8-bit microcontrollers up to 32/64-bit general purpose processors. So far, the CPDev VM has been used in control systems from LUMEL, PL [12] and Praxis A.I., NL [7], and in lab computers with a few versions of Windows [9]. FPGA implementation of VM has appeared recently [2].

Similar approach has been used earlier in ISaGRAF environment [4], where the intermediate code is called TIC (*Target Interpreted Code*). It can run on platforms supporting Windows, Linux, VxWorks, QNX and RTX. Much simpler CPDev does not impose such requirements, however. Another environment called Beremiz [13] compiles IEC programs into C/C++, translated further into processor code. Execution of such code is quicker than VMASM or TIC, however any change in IEC source program requires another translation.

Publications of the team working on development of CPDev so far do not contain extensive description of the VM virtual machine (see e.g. [11,9]). Hence the purpose of this paper is to describe the VM with sufficient details, beginning from initial assumptions, thorough data formats and VMASM commands, Petri-net model of operation, structure of C/C++ implementation with universal and platform-dependent modules, interface to target hardware, data input and output, up to some remarks on applications. At first however, CPDev programming interface is briefly characterized.

2. Programming in CPDev

Main window of CPDev interface is shown in Fig. 1. Project tree is on the left, program code in the center (here in ST), and message window at the bottom. Tree of the START_STOP project being presented consists of POU section (*Program Organization Unit*) with the program PRG_START_STOP, five global variables from START to PUMP, task TSK_START_STOP, and two function blocks TOF, TON (*timer-off*, *-on*) from IEC_61131 library. IN and PT (*Preset Time*) are the blocks' inputs, Q and ET (*elapsed time*) the outputs.

PRG_START_STOP program is written according to ST language rules. First part, from VAR_EXTERNAL, declares the use of global variables. Local declarations (VAR) of the block instances DELAY_ON, DELAY_OFF are the second part. Program body consists of four statements, where the first one turns MOTOR on if START is pressed, provided that STOP and ALARM are not set. MOTOR continues running after releasing START. Next three statements turn PUMP on and off by the two timers, 5 seconds after the MOTOR (PT:=t#5s).

Global variables and the task are defined in separate windows (not shown). Task can be executed cyclically with a given period, continuously (begins anew as soon as previous execution is completed) or just once. There is no limit on the number of programs in POU section assigned to the task.

In addition to the timers the IEC_61131 library includes other blocks of IEC standard, i.e. flip-flops, edge detectors and counters [3]. Besides programs, the POU section may involve function blocks and functions (FUNCTION_BLOCK and FUNCTION declarations). CPDev allows the user to create his own library with reusable POUs.

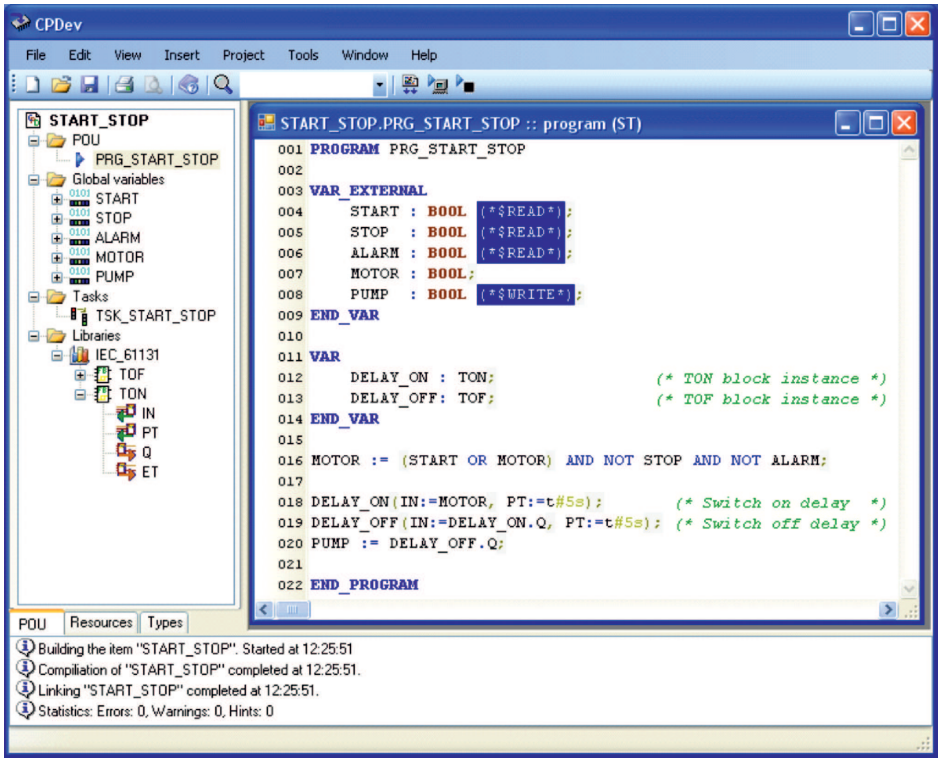


Fig. 1. User interface of CPDev environment

3. Basic assumptions on CPDev virtual machine

The concept of virtual machine has been introduced to CPDev following two essential assumptions [11,9]:

- relatively simple adaptation for different hardware platforms
- common execution layer for different IEC languages.

Recall that JavaME and .NET Compact Framework are general purpose solutions based on virtual machine concept. The specific CPDev VM is focused in IEC languages, with strong orientation towards microcontrollers.

Stages of translation of ST, FBD and IL programs into form executable by VM are shown in Fig. 2. During compilation (*Project→Build* in the main menu of Fig. 1) the programs are first converted into common form involving mnemonics of VMASM assembler. Next the VMASM code is assembled into binary VM code. The VM code is transferred to a target controller, where it is executed by virtual machine as a single task. As indicated above, the task consists of POU units executed in prescribed order. Some POUs may be imported from libraries.

Fig. 2 shows that programs created as FBD diagrams are first converted into ST language and then compiled into VMASM.

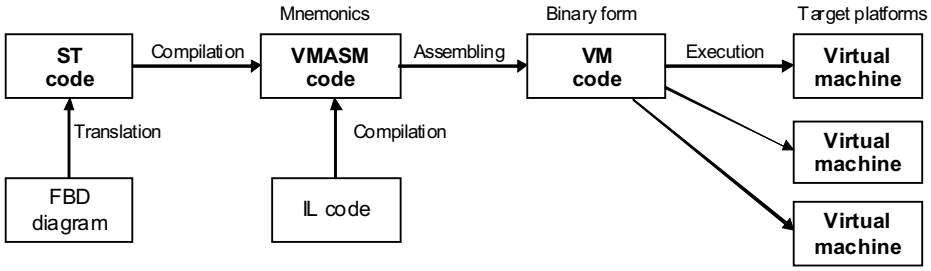


Fig. 2. Translation stages of IEC programs for CPDev virtual machine

4. VMASM data formats and commands

CPDev virtual machine is an automaton processing data of IEC types using commands of VMASM assembler. Functionality of VM involves the following issues:

- Handling IEC data types: Boolean BOOL, integer BYTE, SINT, INT, WORD, DINT, LINT, DWORD, LWORD, real REAL, LREAL, time and date TIME, DATE, TIME_OF_DAY, DATE_AND_TIME; Tab.1 presents VM implementation of these types.
- Execution of functions (examples): arithmetics ADD, SUB, MUL, DIV, MOD, numerical SQRT, LOG, SIN, ASIN, EXP, Boolean NOT, AND, OR, XOR, bit shift SHL, ROL, comparison GT, GE, LT, EQ and others.
- Program flow control by means of jumps JMP, JZ, JNZ, calls of function block CALB and function CALF, early exit RETURN, memory handling MCD, MEMCP (*Move from Code to Data*, *Memory Copy*).

The notion of accumulator does not exist in VM. Results of commands are stored in variables, temporary or declared. Temporary variables are created automatically by the compiler.

Name	Implementation	Name	Implementation
BOOL	1B (0, 1)	LINT	8B (-2^{63} .. 2^{63}-1)
SINT	1B (-128 .. 127)	LWORD	8B (0 .. 2^{64}-1)
BYTE	1B (0 .. 255)	LREAL	8B (IEEE-754)
INT	2B (-32768 .. 32767)	DATE	4B
WORD	2B (0 .. 65536)	TIME_OF_DAY	4B
DINT	4B (-2^{31} .. 2^{31}-1)	DATE_AND_TIME	8B
DWORD	4B (0 .. 2^{32}-1)	TIME	4B
REAL	4B (IEEE-754)	STRING	Variable length string

Tab. 1. Elementary data types of virtual machine

The IEC standard (61131-3 dropped for brevity) also defines multi-element variable types, i.e. arrays and structures (one-dimensional arrays are available in CPDev). VM handles these two types by means of a few dedicated commands, e.g. AURD/AUWD read/write data from/to indexed array.

To become more familiar with what VM actually does, translation of the four ST instructions from Fig. 1 into VMASM code is presented in Tab. 2. The code consists of command mnemonics (JNZ, MCD, NOT, etc.), declared (START, MOTOR) and temporary variables (?LR? at the beginning), constants (#01, #00), and labels (:? at the beginning). Execution of the first ST instruction begins with testing values of START and MOTOR. If they are nonzero, JNZ jumps to :?OR0046 label, where MCD command sets ?LR?AND0045 temporary variable to 1 (#01). Otherwise ?LR?AND0045 is set to 0 (#00 in MCD after second JNZ), followed by JMP to :?EOR004A. From this label, if ?LR?AND0045 is nonzero, NOT, JZ and MCD set another ?LR?AND0043 variable to 0, provided that STOP is 1. In such case the first JZ on the right side in Tab. 2 jumps to :?AND0042, where MCD sets MOTOR to 0. If STOP is 0 but ALARM is not, NOT, JZ and MCD (on the right side) also set MOTOR to 0. Otherwise, i.e. when both STOP and ALARM are 0, MOTOR is set to 1 (at MCD MOTOR, #01, #01).

Note that the compiler has replaced OR, AND instructions by conditional jumps JZ, JNZ and dropped calculation of AND NOT ALARM in the ST instruction after finding that the first part gives 0 when STOP is 1 (*lazy evaluation feature*).

ST instructions	
MOTOR := (START OR MOTOR) AND NOT STOP AND NOT ALARM; DELAY_ON(IN:=MOTOR, PT:=t#5s); DELAY_OFF(IN:= DELAY_ON.Q, PT:=t#5s); PUMP := DELAY_OFF.Q	
VMASM commands	
JNZ START, :?OR0046 JNZ MOTOR, :?OR0046 MCD ?LR?AND0045, #01, #00 JMP :?EOR004A :?OR0046 MCD ?LR?AND0045, #01, #01 :?EOR004A JZ ?LR?AND0045, :?AND0044 NOT ?LR?AND004B, STOP JZ ?LR?AND004B, :?AND0044 MCD ?LR?AND0043, #01, #01 JMP :?EAND004E :? AND0044 MCD ?LR?AND0043, #01, #00 :?EAND004E	JZ ?LR?AND0043, :?AND0042 NOT ?LR?AND004F, ALARM JZ ?LR?AND004F, :?AND0042 MCD MOTOR, #01, #01 JMP :?EAND0052 :?AND0042 MCD MOTOR, #01, #00 :?EAND0052 MEMCP DELAY_ON.IN, MOTOR, #0100 MCD DELAY_ON.PT, #04, #88130000 CALB DELAY_ON, :?TON?CODE MEMCP DELAY_OFF.IN, DELAY_ON.Q, #0100 MCD DELAY_OFF.PT, #04, #88130000 CALB DELAY_OFF, :?TOF?CODE MEMCP PUMP, DELAY_OFF.Q, #0100

Tab. 2. ST program and its VMASM translation

From :?EAND0052 label, MEMCP and MCD set IN and PT inputs of the blocks DELAY_ON, DELAY_OFF (#88130000 denotes 5 seconds). CALB is followed by block instance and label to its code. The last MEMCP assigns DELAY_OFF.Q output to variable PUMP.

As seen, the VMASM translation looks rather lengthy. This is in fact the price for multi-platform functionality of CPDev generated code.

5. Petri-net task execution model

CPDev virtual machine can be modelled by a Hierarchical Time Coloured Petri net (HTCP) [5,14], whose upper level specifies activities performed within task cycle loop (superpage) and lower level describes details of those activities (subpages). The upper level for VM implementation in small controllers is shown in Fig. 3.

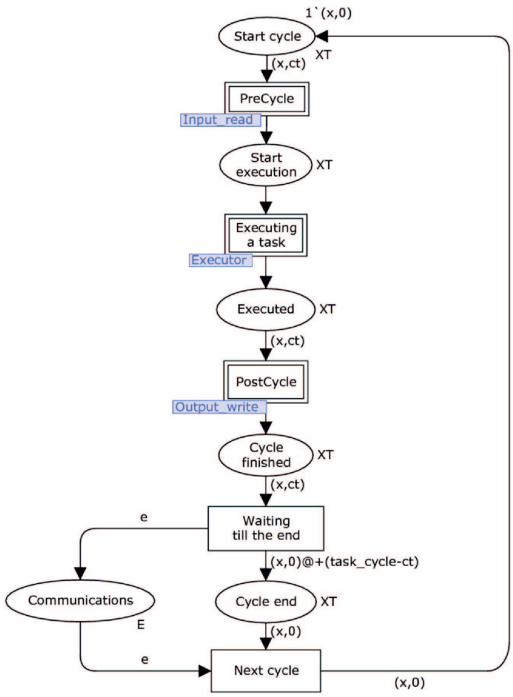


Fig. 3. Petri-net model of CPDev virtual machine cycle (superpage)

Substitution transitions *PreCycle*, *Executing a task* and *PostCycle* involve subpages *Input_read*, *Executor* and *Output_write*, respectively. Token (*x,ct*) of the type *XT* passing through the net represents activity flow. The token holds extra integer variable *ct* for collection of time delays introduced by successive activities. The amount of time left till the end of the cycle is modelled by *time stamp* @+(*task_cycle–ct*). The transition *Waiting till the end* does not pass the token until *task_cycle–ct* time elapses. While waiting, some other activities can be performed, for instance communications or testing. In Fig. 3 the former is triggered by a token e of type E. A Petri-net communications model for VM implementation has been described in [10].

A simplified model of *Executing a task* substitution transition is shown in Fig. 4 (subpage). Task consists of programs executed sequentially. Execution begins when the token is passed from the superpage (Fig. 3) via the input port *Start execution*. *Programs* place holds tokens (*pn*, *pt*) with the number *pn* and execution time *pt*. The model of Fig. 4 contains a sample initial marking with three programs on the left. For simplicity, their execution times are given apriori as 10, 20 and 10 abstract units. In more realistic scenario the values of *pt* would not be constant, but should be determined online during the model simulation (they may also vary from cycle to cycle depending on some conditions). Token of the fourth dummy program with *pt*=0 indicates end of the

task. Collected time *ct* is adjusted in the expression (x, *ct*+*pt*)@+*pt* at the transition *Run program*. The expression also increases time stamp @+*pt*, keeping the token at the *Processor* place until *pt* time elapses. *Programs done* collects tokens of executed programs. The token with zero execution time is a guard [*pt*=0] of the transition *Completing task*. The output port *Executed* returns the token back to the superpage of Fig. 3.

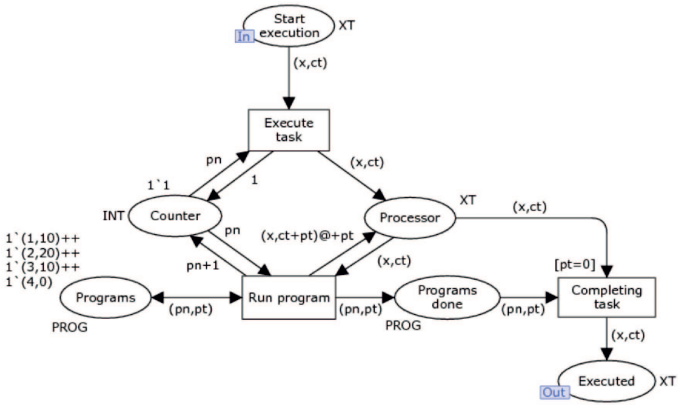


Fig. 4. Petri-net model of task execution (subpage)

After some extensions, the HTCP model described above has been simulated using CPNTools [1]. The model determines time aspects of task execution and may be used to examine interactions between VM and other components of controller software.

6. Structure of the virtual machine

Following the original assumption on simple adaptation for different hardware platforms, the VM machine has been written as a program in C. Most of this program is universal, independent of platforms, some parts need adjustment however.

Simplified structure of the VM is shown in Fig. 5, with universal and platform-dependent modules indicated. The former consists of command interpreter responsible for execution of VMASM code, elementary and multi-element data type handling and stack emulation. There are over 200 commands, although some of them differ only in data types and number of parameters (e.g. ADD_INT, ADD_REAL). While designing a VM machine for simple microprocessor some types (and functions) may be dropped, e.g. LREAL, LWORD, STRING, using a configuration file. Two stacks, i.e. data stack and call stack, are implemented. The latter supports calls of programs, functions and function blocks (POUs).

Platform-dependent modules connect VM with particular hardware through a set of low-level functions. Details are described in Secs.7 and 8.

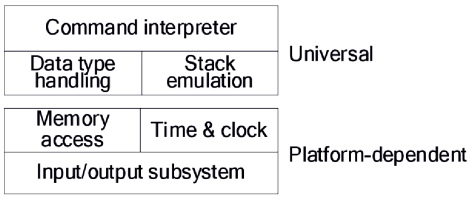


Fig. 5. Basic modules of virtual machine

The VM needs three areas of memory, i.e. code, data and internal memories. The area involving program code is logically separated from data area, as in processors with Harvard architecture. It is read-only memory, so no VMASM command can alter its content. Data area can be addressed directly or indirectly via registers. Internal memory contains registers, stacks and command interpreter (firmware).

Basic registers (logical) of VM are listed in Tab. 3. As indicated before, accumulator does not exist, so the commands store results in variables (as in Tab. 2). *Task cycle* (configured) is used by VM (Fig. 3). *Actual task cycle* (last value) is particularly useful for on-line testing (*commissioning*). *Status1* stores exception flags, therefore appropriate reactions can be programmed.

Register name	**Function**
Program counter	Indicates next VMASM command
Data offset	Index to data area being used
Call stack pointer Data stack pointer	Pointers to call stack (POUs) and data stack
Task cycle Actual task cycle	Configured and measured task cycle
Cycle counter	Counts cycles (with reset)
Status1	VM status word (array index faulty, time cycle exceeded, cold start, etc.)
RTC clock	Absolute time

Tab. 3. Basic registers of virtual machine

7. Target hardware interface

Platform-dependent modules of Fig. 5 connect VM with particular hardware by implementing interface of low-level functions. Such functions can be written by hardware designers and consolidated with universal part of the C program. Naturally, a C compiler for the target processor is needed.

Interface to Virtual Machine Platform (VMP) involves a set of functions, most important of which are given below:

- *VMP_LoadConfiguration* – loads task parameters (cycle, number and order of POUs, etc.), binary code of POUs, and allocates memory for data; memory size is determined by the compiler.
- *VMP_PreRunConfiguration* – initializes hardware and stores its initial state (time, parameters).
- *VMP_PostRunConfiguration* – relieves hardware and resources used by the task.
- *VMP_PreCycle* – updates program variables from signal input readings (Sec.8), stores initial value of system clock.
- *VMP_PostCycle* – sets values of signal outputs according to variables with results, determines actual task cycle; if time is left, triggers another activities (communications, tests).
- *VMP_ReadRTC* – returns reading of real-time clock (if available); the reading may be used in the program.
- *VMP_CurrentTime* – returns actual value of system clock (in milliseconds beginning from the start; RTC is not involved).

Memory access module in Fig. 5 is also platform-dependent. Some architectures other than x86, for instance ARM, impose restrictions on accessing data from any address. *Flash* memory in industrial controllers stores program code and some parameters of function blocks (e.g. PID settings). Special algorithm is required to write into *flash*.

8. Data input and output

Reading signal inputs and writing outputs depends on hardware solutions. To incorporate related low-level functions into VM, two mechanisms are available in CPDev:

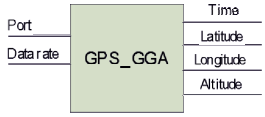
- configurer of hardware resources
- native blocks.

The configurer assigns program variables to inputs and outputs, or to communication interfaces. The assignment is performed during configuration and is independent from the program. Operation of *Input/output subsystem* of the VM (Fig. 5) depends on the assignment. Separation of control program from hardware configuration is typical for multi-module PLCs and DCS systems.

Contrary to the configurer, native blocks are components of the user program, i.e. POUs, providing hardware dependent functions internally. They may be also called hardware blocks. Native blocks are usually written in C/C++ and linked to CPDev by a library. They are used in the program in standard way, i.e. as timers TOF, TON in

Fig. 1. *Java Native Interface* and *Platform Invoke* in .NET technology involve similar mechanisms. Small multifunction instruments for measurement and control also employ native blocks (e.g. Sipart 24 from Siemens).

Native blocks available in CPDev provide read/write of program variables into non-volatile memory (*flash*), communicate over serial and CAN busses [6], handle LCD displays. Tab. 4 shows declaration and use of GPS_GGA block which reads data from a GPS module using NMEA serial protocol (GGA is a command in NMEA). The block has two inputs, PORT number and DATA_RATE, declared as constants. The outputs provide UTC time (absolute), LATitude, LONgitude, ALTitude of actual position, together with QUALITY of readings. The user program (right side) assigns outputs of the block instance to program variables.

Symbol	**Declaration**	**Use in ST**
	FUNCTION_BLOCK GPS_GGA (*$HARDWARE_BODY_CALL ID:0003*) (*$COMMENT The block is implemented in C *) VAR_INPUT PORT (*$CONST*) : BYTE; DATA_RATE (*$CONST*) : INT; END_VAR VAR_OUTPUT UTC : TIME_OF_DAY; LAT : LREAL; LON : LREAL; ALT : LREAL; QUALITY : BYTE; END_VAR END_FUNCTION_BLOCK	PROGRAM GPS_PRG VAR GPS_POS : GPS_GGA; END_VAR GPS_POS(PORT:=BYTE#9, DATA_RATE:=9600); UTC_TIME:=GPS_POS.UTC; LONGITUDE:=GPS_POS.LON; LATITUDE:=GPS_POS.LAT; ALTITUDE:=GPS_POS.ALT; END_PROGRAM

Tab 4. GPS_GGA native block

9. Applications

The CPDev virtual machine has been implemented first in SMC industrial controller from LUMEL, Zielona Góra, PL (Fig. 6a) [12]. SMC is equipped with Atmel AVR 8-bit microcontroller and operates as a central unit in small DCS systems involving distributed I/Os, intelligent transmitters, displays, etc., with PC or HMI panel as a host (Fig. 6b). Modbus RTU protocol is applied at both sides (up to 230.4 kbaud). Since SMC does not have I/Os of its own, so communication subsystem is especially important part of CPDev software. Hardware configurer described above defines Modbus transactions. Native blocks handling communications with particular field devices are also available.

So far SMC controller has been used in several prototype applications involving measurements, control, monitoring and diagnostics. Function block developed by the

user has turned out particularly useful. Two dedicated mini-systems are currently being prepared by LUMEL for off-the-shelf applications. They consists of SMC controller, distributed I/O modules (SM series) and an HMI touch-panel. The first mini system controls heating substation.(“hot spot”) in a municipal heating network. Besides temperature measurements, DATE_AND_TIME variable affects PID control. The other system belongs to simple Manufacturing Execution Systems (MES). On, off and alarm signals from machine tools, casters, injection moulding machines, etc., are monitored, recorded, displayed at plant floor, and used for statistics. It is oriented towards production lines in small and medium scale enterprises.

Mini-Guard Ship Control and Positioning System from Praxis A.I., Leiden, NL is another application [7] (tested first on a Chinese ship). Mini-Guard consists of seven types of dedicated controllers (Fig. 6c) involving NXP ARM7 16/32-bit microcontrollers. The controllers communicate over Ethernet, external devices are connected through serial port (as GPS_GGA in Tab. 4) or OPC interface. Other native blocks handle LCD displays.

Some lab, diagnostic and teaching applications require VM machine operating as a *soft-controller*, i.e. PC equipped with I/O boards. NI-DAQ USB 6008 and RT-DAC/USB boards from National Instruments and InTeCo, Cracow, respectively, have been interfaced to VM so far [9]. CPDev VM has been also implemented on computers with Windows Embedded, CE. NET and QNX6. FPGA Verilog version has been developed recently [2].

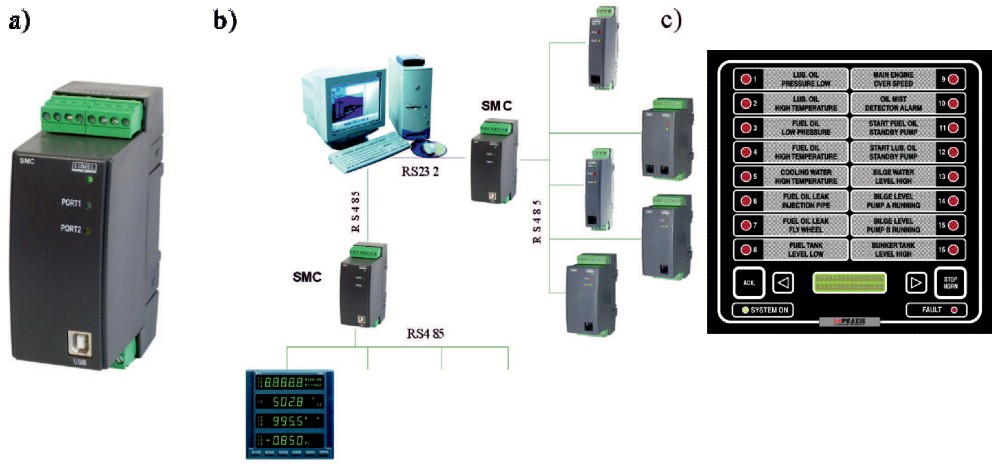


Fig. 6. a) SMC controller in b) distributed system; c) one of Mini-Guard controllers

10. Conclusions

CPDev environment is available for programming controllers in small distributed systems according to IEC 61131-3 standard. The environment is considered open because the compiled code can be executed by different processors, low-level software may be written by hardware designers, and control programmers can create their own libraries with reusable program units. VMASM universal code produced by CPDev compiler is executed by runtime virtual machine operating as an interpreter. The machine is a C program composed of universal and platform-dependent modules. It has been implemented in AVR, ARM and x86 processors, and applied in two small DCS systems and in PC-based soft-controller with I/O boards.

Details of VM development and implementation has been described here, including VMASM data formats and commands, Petri-net model of task execution, structure of C implementation, function interface to target hardware, programming mechanisms of data input/output, and practical applications. Future work will concentrate on extension of VM to multitasking, beginning from Windows CE and a simple RTOS system (e.g. FreeRTOS).

References

1. *CPN Tools: Computer Tool for Coloured Petri Nets*, http://www.daimi.au.dk/cpntools/.
2. Z. Hajduk, B. Trybus, J. Sadolewski: *Hardware implementation of virtual machine for programmable controllers* (in Polish), In: Metody wytwarzania i zastosowania systemów czasu rzeczywistego, WKŁ, pp.327-336, Warszawa, 2010.
3. IEC 61131-3 Standard: *Programmable Controllers. Part 3. Programming Languages*, IEC, 2003.
4. *ISaGRAF User's Guide*, ICS Triplex Inc., 2005.
5. K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Springer –Verlag, 1997.
6. W. Mikluszka, B. Trybus: *Modelling and implementation of CAN bus in CPDev environment* (in Polish), In: Metody wytwarzania i zastosowania systemów czasu rzeczywistego, WKŁ, pp.293-302, Warszawa, 2010.
7. *Mini-Guard Ship Control & Positioning System*, Praxis Automation Technology B.V., http://www.praxis-automation.com, 2010.
8. D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, L. Trybus: *Mini-DCS System Programming in IEC 61131-3 Structured Text*, Journal of Automation, Mobile Robotics & Intelligent Systems, Vol.2, No.3, 2008.

9. D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, L. Trybus: *Open environment for programming small controllers according to IEC 61131-3 standard.* Scalable Computing: Practice and Experience, Vol.10, No.3, pp.325-336, 2009.

10. D. Rzońca, B. Trybus: *Application of coloured Petri net for design of SMC controller communication subsystem*, Studia Informatica, Vol.27, No.1, pp.1-12, 2008.

11. J. Sadolewski, B. Trybus: *Multiplatform virtual machine for control systems* (in Polish), In: Modele i zastosowania systemów czasu rzeczywistego, WKŁ, pp.293-302, Warszawa, 2008.

12. *SMC programmable controller*, Lumel S.A., http://www.lumel.com.pl/en, 2010.

13. E. Tisserant, L. Bessard, M. Sousa: *An Open Source IEC 61131-3 Integrated Development Environment*, 5th Int. Conf. Industrial Informatics, Piscataway, NJ, USA, 2007.

14. B. Trybus: *Introduction to Conversion of Control Software Structured Models into Coloured Petri Nets*, Theoretical and Applied Informatics, Vol. 19, No.1, pp.57-70, 2007.

Projektowanie i implementacja maszyny wirtualnej normy IEC 61131-3

Streszczenie

W artykule przedstawiono projekt i implementację maszyny wirtualnej będącą elementem środowiska wykonawczego dla sterowników. Przeznaczona jest przede wszystkim do małych, rozproszonych systemów sterowania. Maszyna współpracuje z pakietem CPDev, opracowanym na Politechnice Rzeszowskiej, który służy do programowania w językach normy IEC 61131-3 (PN-EN 61131-3) (Rys.1). Programy w ST, IL lub FBD są kompilowane do kodu pośredniego VMASM, który w postaci binarnej może być wykonywany przez maszynę na platformie docelowej (Rys. 2 i Tab. 2). Zestaw instrukcji maszyny wirtualnej oraz obsługiwane przez nią typy danych zostały dostosowane do normy IEC (Tab. 1).

Działanie maszyny zostało zamodelowane za pomocą hierarchicznej czasowej kolorowanej sieci Petriego. Elementami tego modelu jest strona przedstawiająca cykl zadania (nadrzędna, Rys. 3) oraz podrzędna, reprezentująca moduł wykonawczy (Rys. 4). Symulacja modelu pozwoliła zweryfikować przyjęte założenia projektowe.

Maszyna wirtualna została zaimplementowana jako program w języku C. Jej strukturę wewnętrzną przedstawiono na Rys. 5. Część modułów jest uniwersalna, pozostałe zależą od platformy docelowej sterownika. Dzięki takiemu układowi, maszyna może być przystosowana do różnego sprzętu. Dostosowanie maszyny polega na przygotowaniu funkcji wchodzących w skład interfejsu sprzętowego, określających m.in. sposób ładowania programu, obsługę cyklu zadania i zegara czasu rzeczywistego. Współpraca ze sprzętem obejmuje także odczyt wejść i zapis wyjść procesowych. Konfigurator zasobów sprzętowych pozwala przypisać zmienne programu do określonych wejść/wyjść.

Mechanizm bloków sprzętowych pozwala natomiast bezpośrednio korzystać z mechanizmów niskopoziomowych w kodzie programu. W ten sposób zrealizowano m.in. obsługę protokołu NMEA (Rys. 4).

Dwa pierwsze zastosowania maszyny wirtualnej ze środowiskiem CPDev to sterownik SMC polskiej firmy Lumel, będący centralnym węzłem małego rozproszonego systemu sterowania (mini-DCS, Rys. 6a,b) oraz system Mini-Guard z Praxis Automation (Holandia) stosowany do monitorowania systemów na statku i jego pozycjonowania (Rys. 6c).

Dzięki maszynie wirtualnej programy tworzone w środowisku CPDev w językach normy IEC 61131-3 (ST, IL, FBD) mogą być uruchamiane na różnych sterownikach, wyposażonych w procesory AVR, ARM, x86 i inne. Przedmiotem dalszych prac będzie możliwość jednoczesnego wykonywania przez maszynę kilku zadań sterujących (wielozadaniowość).