Scalable Computing: Practice and Experience
Volume 10, Number 3, pp. 325–336. http://www.scpe.org
ISSN 1895-1767

OPEN ENVIRONMENT FOR PROGRAMMING SMALL CONTROLLERS ACCORDING TO IEC 61131-3 STANDARD

DARIUSZ RZOŃCA, JAN SADOLEWSKI, ANDRZEJ STEC, ZBIGNIEW ŚWIDER, BARTOSZ TRYBUS, AND LESZEK TRYBUS*

Abstract. A control engineering environment called CPDev for programming small controllers in ST, FBD and IL languages of IEC 61131-3 standard is presented. The environment consists of a compiler, simulator and hardware configurer. It is open in the sense that: (1) code generated by the compiler can be executed by different processors, (2) low-level components of the controller runtime program are developed by hardware designers, (3) control programmers can define libraries with functions, function blocks and programs.

Of the three IEC languages, ST Structured Text is a basis for CPDev. FBD diagrams are translated to ST. IL compiler uses the same code generator. The runtime program has the form of virtual machine which executes universal code generated by the compiler. The machine is an ANSI C program with some platform-dependent components. The machines for AVR, ARM, MCS51 and x86 processors have been developed so far. Applications include two controllers for small DCS systems and PC equipped with I/O boards. CPDev may be downloaded from `http://cpdev.prz-rzeszow.pl/demo`.

Key words: control engineering tool, IEC 61131-3 standard, ST language compiler, multi-platform virtual processor

1. Introduction. Remarkable number of small-and-medium-scale companies in Europe manufacture transmitters, actuators, drives, PID and PLC controllers, and other control-and-measurement equipment. Engineering tools for programming such devices are often fairly simple and do not correspond to IEC 61131-3 standard [4], required by growing number of customers. The problem may be solved to some extent by developing open engineering environments for programming small control devices based on AVR, ARM, MCS51 or other microcontrollers according to IEC languages (61131-3 will be dropped for brevity). Development of such environment called CPDev (*Control Program Developer*) was initiated by the authors at the end of 2006.

The CPDev is open in the following sense:

- code generated by the compiler can be executed by different processors,
- low-level components of runtime program are provided by hardware designers,
- control programmers create their own libraries with reusable program units.

The CPDev compiler generates an intermediate, universal code executed by runtime interpreter at the controller side. Different processors require different interpreters. This resembles somewhat the concept of Java virtual machines [7] capable of executing programs on different platforms. Hence the interpreters of the CPDev universal code are also called *virtual machines*.

The same approach was adapted earlier in ISaGRAF package from ISC Triplex [5] (now in Rockwell). ISaGRAF universal code is called TIC (*Target Independent Code*) and may be executed on platforms supporting Windows, Linux, VxWorks, QNX and RTX. Much simpler CPDev does not impose such requirements, however. Another open environment called Beremiz [11] compiles IEC language code into C/C++ program, to be translated further into processor code. In this case commercial restrictions on the use of C/C++ compilers may matter sometimes.

This paper follows a few earlier publications, e.g. [9, 10], which reported on CPDev development. The content is organized as follows. For the reader not familiar with IEC standard, Sec. 2 provides some information on programming in high-level ST language. Components of CPDev, user interface, standard functions and libraries with function blocks are described in Sec. 3. Section 4 characterizes scanner, parser and code generator of ST compiler, written in C# at Ms .NET platform. Some instructions of the universal code called VMASM (*Virtual Machine Assembler*) are also presented. Section 5 describes operation and structure of the virtual machine. The machine is written in industry standard C and consists of universal and platform-dependent modules. Platform-dependent modules are written by hardware designers. Section 6 characterizes development of user function blocks, both in ST and C languages. Blocks written in C become components of the virtual machine. Programming in graphical FBD and textual IL languages is described in Sec. 7. FBD diagram is translated to ST and then compiled. Applications of CPDev for programming a small control-and-measurement

*Department of Computer and Control Engineering, Faculty of Electrical and Computer Engineering, Rzeszow University of Technology, 35-959 Rzeszow, ul. W. Pola 2, Poland, (`{drzonca, js, astec, swiderzb, btrybus, ltrybus}@prz-rzeszow.pl`).

distributed system, controllers of ship control-and-positioning system and a softcontroller based on PC with I/O boards are presented in Sec. 8.

2. A few notes on IEC 61131-3. The IEC 61131-3 standard [4] defines five programming languages, LD, IL, FBD, ST and SFC, allowing the user to choose the 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 book [6] is a good source to learn IEC programming.

ST is a high-level language originated from Pascal, especially suitable for complicated algorithms. Equivalent code for a program written in any of the other four 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, ST has been selected as a base language for the CPDev environment.

2.1. Data types. Data types, literals (constants) and variables are common components of IEC languages. Names (identifiers) are typical, although there is no distinction between capital and small characters. The standard defines twenty elementary data types, several of which are listed in Table 2.1 together with memory sizes and ranges (in CPDev). `BOOL`, `INT`, `REAL` and `TIME` are most common. `FALSE`, `13`, `-4.1415` and `T#1m25s` are examples of corresponding constants.

Table 2.1
Several elementary IEC data types

Type	Size (range)	Type	Size (range)
`BOOL`	1B (0, 1)	`LREAL`	8B IEEE-754 format
`BYTE`	1B (0 ... 255)	`TIME`	4B (-T#24d20h31m23s648ms ... T#24d20h31m23s647ms)
`WORD`	2B (0 ... 65535)		
`INT`	2B (−32768 ... 32767)	`TIME_OF_DAY`	4B (00:00:00.00 ... 23:59:59.99)
`REAL`	4B IEEE-754 format		

The standard defines three levels for accessing variables, `LOCAL`, `GLOBAL` and `ACCESS`. `LOCAL`s are available in the program, function block or function. `GLOBAL`s can be used in the whole project, but programs, function blocks or functions must declare them as `EXTERNAL`. `ACCESS` variables exchange data between different systems.

2.2. POU units. Programs, function blocks and functions, called jointly Program Organization Units (POUs), are components of IEC projects. Function blocks, designed for reuse in different parts of program, are of crucial importance. A block involves inputs, outputs and memory for data from previous executions. Therefore the blocks must be declared as instances. The IEC defines small set of standard blocks, such as flip-flops, edge detectors, timers and counters. Three of them are shown in Fig. 2.1.

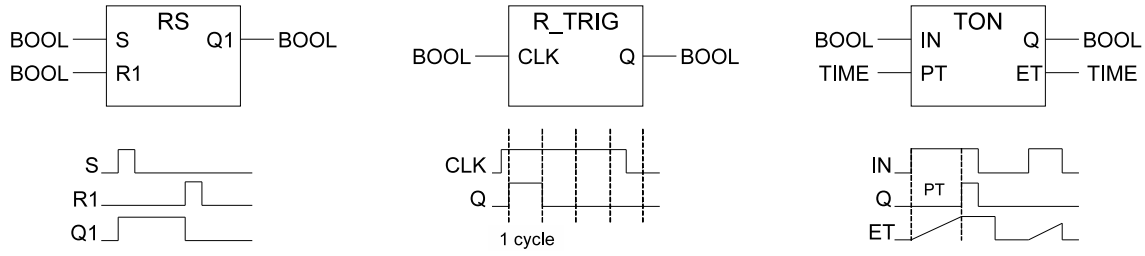


Fig. 2.1. *Examples of standard function blocks:* `RS` *flip-flop,* `R_TRIG` *rising edge detector,* `TON` *on-delay timer*

2.3. Programming. Programs written in ST or other languages begin with declarations of variables and instances of function blocks placed between `VAR_EXTERNAL` or `VAR` and `END_VAR` keywords. `GLOBAL` variables are declared before programs or separately. The declarations are followed by list of statements. The statements involve expressions which, when evaluated, yield results in one of defined data types, i. e. elementary (Table 2.1) or derived, such as alias, array or structure. The following operators are available (in descending priority): parenthesis, function evaluation, negation, power, arithmetic operators, Boolean operators.

ST language provides five types of statements:

- assignment := (Pascal symbol),
- selections IF, CASE,
- loops FOR, WHILE, REPEAT,
- early exits RETURN, EXIT,
- function and function block invocations.

Simple examples are presented in the following sections. Typical program looks like a sequence of function and function block invocations (calls).

3. CPDev environment. The CPDev consists of three programs executed by PC and one by the controller (Fig 3.1). The PC programs are as follows:

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

The programs have dedicated interfaces and exchange data through files in appropriate formats. The CPDev compiler (the same name as the package) generates universal code executed by virtual machine (VM) run by the controller. The VM operates as an interpreter. The universal code is a list of instructions of VM language called VMASM assembler. VMASM is not related to any particular processor, but close to typical assemblers. The compiler employs ST syntax rules, list of VMASM instructions and POUs from libraries. Besides the universal code the compiler generates some information for debugging and simulation by CPSim.

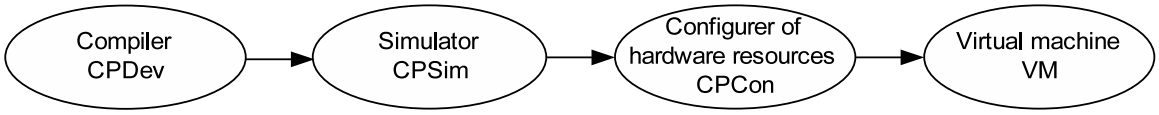


FIG. 3.1. *Components of CPDev environment*

Configuration of hardware resources by means of CPCon involves memory, input/output and communication interfaces. User specifications define memory types and sizes, numbers and types of I/Os and communication channels, validity flags, etc. Allocation of hardware resources has the form of a *map* that assigns symbolic addresses from ST programs to physical ones. By using it, the compiled code can be assembled for a particular platform to create final, universal executable code. From CPDev viewpoint, hardware platforms differ only in hardware allocation maps, whereas the compiled code is identical.

The CPDev environment has been recently extended by graphic editor of FBD diagrams and compiler of IL language. FBD diagram is automatically converted into ST code and compiled as above. Compilers of ST and IL differ in details only.

3.1. User interface. Main window of CPDev ST compiler is shown in Fig. 3.2. The window consists of three areas:

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

Frames of the areas can be adjusted and the contents scrolled.

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. The program is written according to ST language rules. The first part involves declarations VAR_EXTERNAL of the use of global variables. Local declarations of the instances ON_DELAY and OFF_DELAY of the blocks TON, TOF are the second part. Program body consists of four statements. The first one turns a MOTOR on if START is pressed, provided that STOP or ALARM are not. Next three statements turn a PUMP on and off five seconds after the MOTOR (FBD diagram corresponding to this project is shown in Fig 7.1).

Global variables and the task are defined using separate windows (not shown). According to IEC standard the variables can be assigned CONSTANT and RETAIN attributes, and logical addresses. Task can be executed once, cyclically with a given period, or as soon as previous execution is completed. There is no limit on the number of programs assigned to a task.

Text of the project represented by the tree is kept in an XML file. Compilation is executed by calling Project->Build from the main menu. Messages appear in the lower area of the interface window. If there

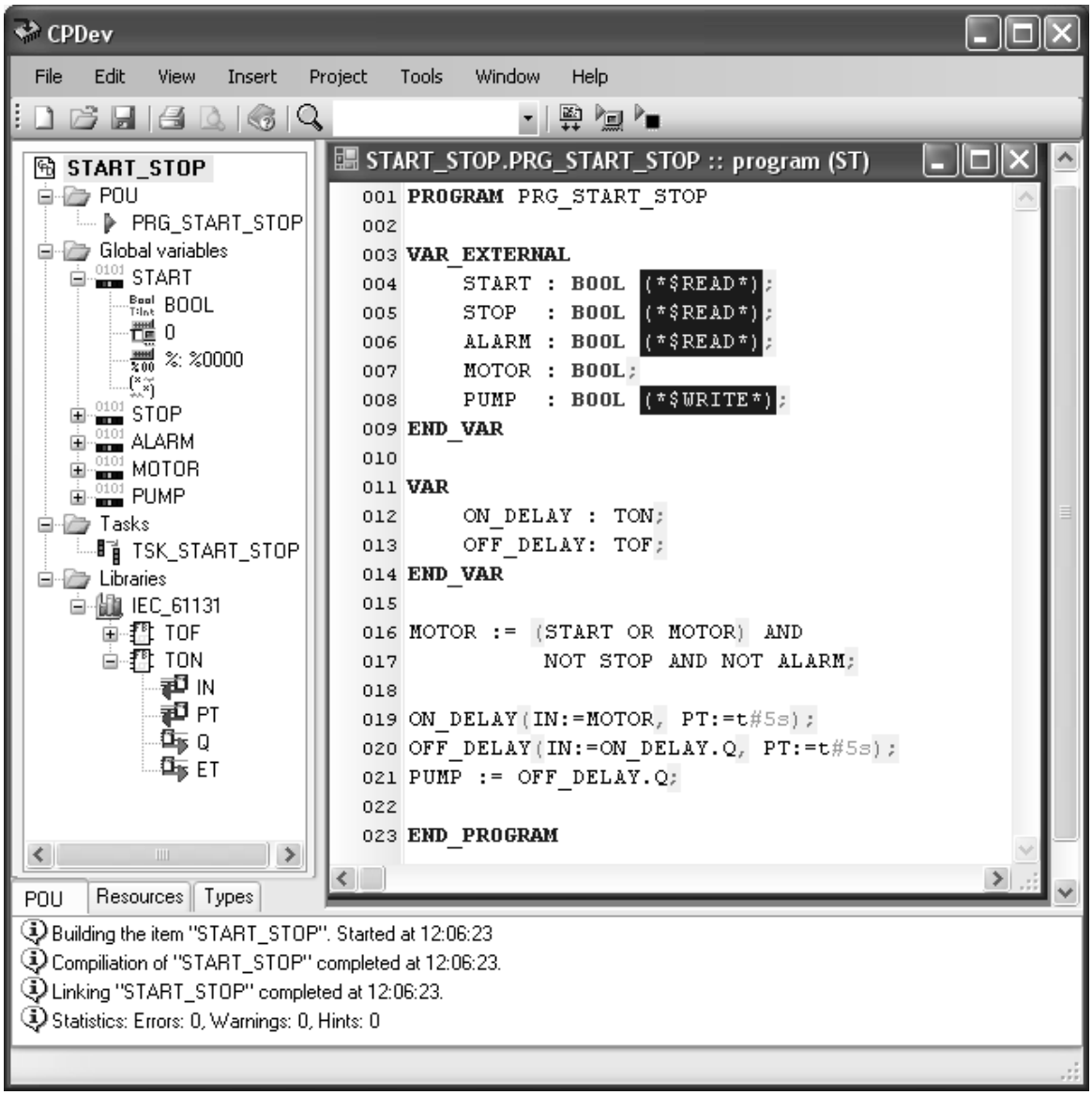


FIG. 3.2. *User interface of ST compiler (*`START_STOP` *project)*

are no mistakes, the compiled project is stored in two files. The first one contains universal executable code in binary format for the virtual machine. The second one stores mnemonic code, together with some information for simulator and hardware configurer (variable names, etc.).

3.2. Functions and function blocks. The CPDev compiler provides most of standard functions defined in IEC. Five 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`,
- 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 comparision group are particularly useful. Variables of any numerical type, i. e. `INT`, `DINT`, `REAL` and `LREAL` are arguments in most of relevant functions.

Two libraries of function block are available, namely:

- `IEC_61131` standard library,

- `Basic_blocks` library with simple blocks supplementing the standard.

The first one involves: (1) flip-flops and semaphore `RS`, `SR`, `SEMA`, (2) rising and falling edge detectors `R_TRIG`, `F_TRIG`, (3) up, down, up-down counters `CTU`, `CTD`, `CTUD`, (4) pulse, on-delay, off-delay timers `TP`, `TON`, `TOF`. Blocks typical for small multifunction controllers are in the second library, i. e. integrator, filters, max/min over time, memories, time measurement, etc.

4. ST language compiler. The task of the compiler is to convert XML source file with the project in ST language into a file with universal code in binary format. General diagram of the compiler operation involving scanner, parser and code generator is shown in Fig. 4.1.

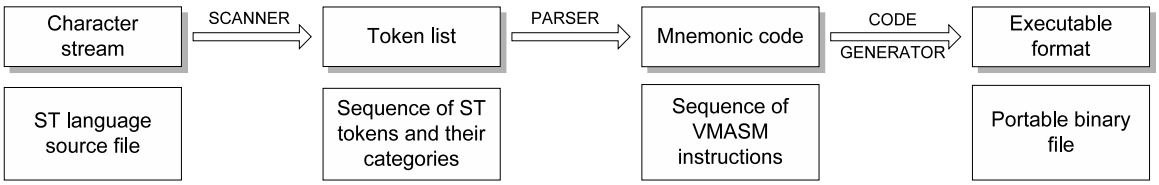


Fig. 4.1. *ST compiler components*

4.1. Scanner, parser and code generator. The scanner (lexical analyser) analyses character stream from ST source file and decomposes it into lexical units, i. e. tokens. The tokens are classified into categories such as identifiers, keywords, operators, constants (a few categories), delimiters, directives, comments, white spaces and invalid characters. The tokens with categories are collected on a list passed to the parser.

The parser operates according to top-down scheme with syntax directed translation [3]. By employing the ST syntax the parser recognizes consecutive token constructions from the scanner list. White spaces and comments are dropped. When correct construction is recognized the parser replaces it by a set of mnemonic instructions of the VMASM assembler. To do so, the parser employs built in elementary data types (Table 2.1) and list of VMASM instructions. Examples of these instructions are presented in Table 4.1.

Table 4.1
Examples of VMASM assembler instructions

Instruction	Meaning	Instruction	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	`MEMCP`	Memory copy
`NOT`	Negation	`RETURN`	Return from function

Normally a single ST statement is translated into several VMASM instructions. Some translations require introduction of auxiliary variables and labels. Derived data types and POUs from libraries (functions, function blocks and programs) are also parsed. The mnemonic code is written in a special text format. The code can be consolidated with other mnemonic codes.

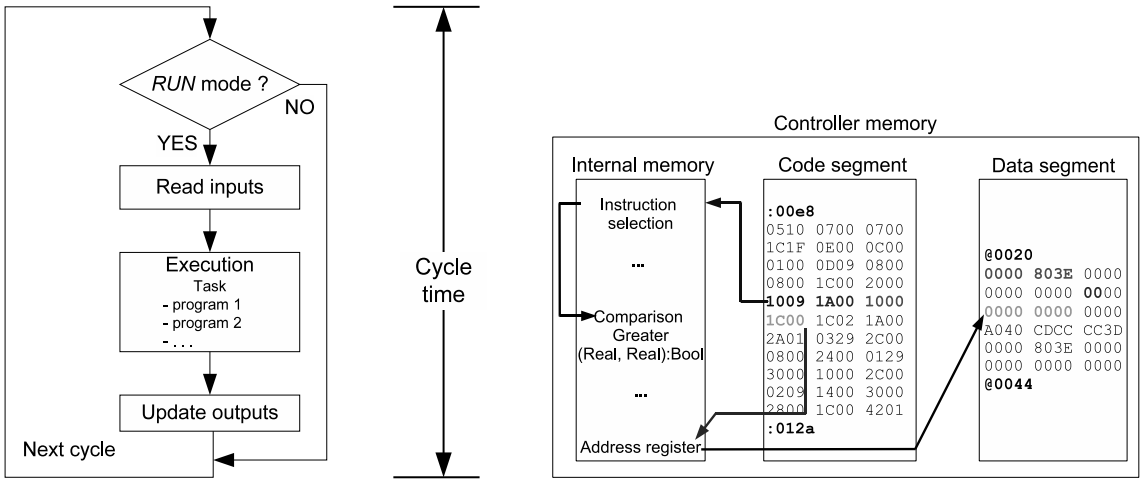
In the third step the code generator converts the consolidated mnemonic code into universal executable code in binary format. Mnemonics of the VMASM instructions, names of the variables and labels are replaced by corresponding number identifiers. To do so, the generator employs a *Library Configuration File* (LCF) with the identifiers of the instructions, numbers and types of the operands, and information how the operands are acquired (operand identifier may be an index to variable or a direct value). Each implementation of virtual machine is defined by specific LCF configuration file. Besides binary file with the executable code the compiler generates a text file with mnemonic code, some additional information for CPSim simulator and CPCon configurer (variable names, etc.) and compilation report (HTML).

4.2. Parser and code generator classes. Essential components of the compiler are designed as classes in C# language [1, 2]. Each token of ST language is encapsulated into an object of corresponding class. The classes inherit from an abstract `STIdentificator` class. During compilation, identifiers are collected into lists. The lists employ predicates for finding appropriate identifiers, what eliminates the need for hash tables. There is a list of global identifiers and local lists which store identifiers of functions, function blocks, programs, etc.

Identifiers in a list are checked for uniqueness. When identical names are found compilation is stopped and error reported. If local identifier hides a global one, the compiler produces a warning.

The parser generates text sequence of VMASM instructions for the code generator. Each instruction is represented by a mnemonic followed by operand names. Code generator replaces mnemonics and variable names with appropriate number identifiers (indexes). While processing an instruction, the generator extracts some information from libraries, e.g. operand size, type and passing method. The number identifier can be interpreted as a pointer to variable or as immediate value. Instructions resulting from compilation are represented by instances of `VMInstruction` class. The operand list `VMOperand` is also stored as a member of this class. By using lists of operands typical problems with fixed-size operand tables are avoided.

5. Multi-platform virtual machine. Binary file with the universal code and hardware allocation map from the CPCon configurer are downloaded into the controller, to be processed by virtual machine. Main features of the processing are characterized below.



a) Operation of VM b) Memory organization

FIG. 5.1. *Virtual Machine*

5.1. Operation cycle. Virtual machine is an automaton operating according to Fig. 5.1a. As indicated before, the machine is specific for a particular processor and works as an interpreter. The task consists of programs executed consecutively. The binary code involves number identifiers of the instructions and addresses of operands. The machine, similarly as a real processor, maintains program counter with the address of instruction to be executed, and base address of the data area with operands (specified for each POU). Given the instruction address, the machine fetches the identifier, decodes it, fetches the operands, and executes the instruction. Stack emulation and update of the base addresses permit multiple, concurrent calls of functions and function blocks. The machine monitors time cycle of the task and sets alarm flag if timeout appears. It also triggers input/output procedures responsible for external variables.

Allocation of software to memory segments is shown in Fig. 5.1b. The instructions and their operands are in the code segment (*read only*). Data segment contains global, local and auxiliary variables, some of them with constant values. The data segment can be accessed directly or indirectly by special virtual registers. The machine's internal memory keeps code of the interpreter, stacks and registers. There is no way of accessing internal memory from the program level. The machine is able to execute multiple instances of programs.

As shown in Fig. 5.2, the virtual machine consists of a few universal and platform-dependent modules to simplify implementation. The universal modules remain unchanged (if one neglects compilation of the source code for a given processor). The platform-dependent modules interface the machine to particular hardware, executing VM requests to low-level procedures. For instance, the module `Time&Clock` is associated with hardware, as it employs time interrupts to handle `TIME` data. `DATE_AND_TIME` data require real-time clock (RTC) on board. I/O functions provide interface to analog and binary inputs and outputs, and to communication fieldbus or network. The multitasking module is optional (not implemented yet), since it employs mechanisms of the host operating system.

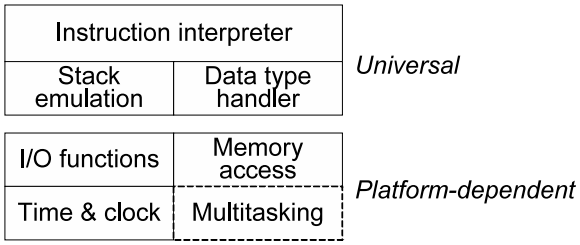


FIG. 5.2. *Universal and platform-dependent VM software modules*

The universal part of the virtual machine has been written in ANSI C, so it can be directly applied to different processors. As indicated in Sec. 4.1, the number of data types and the way in which the machine instructions are executed are defined by the LCF configuration file. For example, one can limit the number of elementary data types or define a subset of VMASM instruction to be used. A set of general specifications has been developed in CPDev for handling processor components (interrupt system, RTC) and external interfaces (I/O, communications). The specifications are in the form of prototypes of corresponding procedures (names, types of inputs and returned outputs). The prototypes do not depend on processor and hardware solutions.

The file with the prototypes is compiled together with the universal modules of the virtual machine. The contents (bodies) of the specification procedures can be prepared by hardware designers and, as a binary file, consolidated with the compiled universal modules. This gives the complete code of the virtual machine for given platform. Till now, the machines for AVR, ARM, MCS-51 and PC platforms have been developed.

We stress that the contents of low-level procedures dependent on hardware solutions may be written by designers themselves. This makes the CPDev package *open* in the hardware sense.

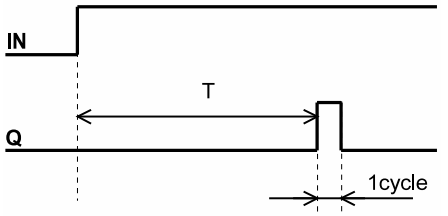
6. User defined function blocks. The CPDev environment allows the user to define function blocks both at PC side and at controller side, i. e. as components of virtual machine. The PC side blocks are written in ST, whereas the VM side ones are in C. However, the C blocks are still invoked in the main ST program compiled and downloaded from PC. So, as far as invocations are concerned, there is no difference between ST and C blocks.

6.1. ST blocks. User libraries are created in CPDev as typical projects which may include all kinds of POU units of IEC standard, i. e. programs, functions and function blocks. Declarations `VAR_INPUT` and `VAR_OUTPUT` determine input/output structure of functions and function blocks. There is no difference between programming of a project directly for controller implementation and programming a library. However, the library project is semi-compiled to VMASM mnemonics and not to binary form. So the last component of ST compiler, code generator (Fig. 4.1), is not needed. The file with mnemonics becomes user-defined library and is exported to *Libraries* folder.

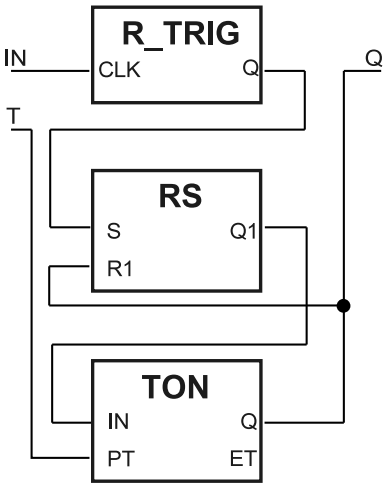
Example of user function block `FB_PULSE` is shown in Fig. 6.1. The block generates single pulse at the output `Q` after time `T`, since rising edge has appeared at the input `IN`. The program of the block may implement FBD diagram of Fig. 6.1b, with standard blocks `R_TRIG`, `RS` and `TON` from CPDev `IEC_61131` library (Fig. 2.1). Corresponding ST code is shown in Fig. 6.1c, with `FB_PULSE` belonging to the project `PROJ_MY_BLOCK` (top of Fig. 6.1c). XML file with `PROJ_MY_BLOCKS` source code should be saved for future extensions and modifications. Semi-compilation of the project yields a file with VMASM mnemonics, called, for instance, `My_Library`. This file must be exported to *Libraries*. If `FB_PULSE` is needed in a new project, both `My_Library` and `IEC_61131` must be imported (the latter to support the former).

6.2. C-language blocks. Such blocks are needed at hardware level to handle I/O and communication channels. Inputs and outputs are declared in ST, but the block body is implemented in C, at virtual machine side (declarations are also repeated). Directive `(*$HARDWARE_BODY_CALL...*)` informs CPDev compiler that the block is a component of VM.

Table 6.1 presents initial parts of the code of `GPS_GGA` block which provides serial communication with a GPS device according to NMEA protocol (GGA is a command in NMEA). Identifier `ID:0003` in the `(*$HARDWARE...)` directive means that `GPS_GGA` is the third of C language blocks at VM side. `Align:4` tells the compiler to locate the variables at addresses divided by 4.



a) Time diagram



b) FBD realization

```
PROJ_MY_BLOCKS.FB_PULSE :: function
001 FUNCTION_BLOCK FB_PULSE
002
003 VAR_INPUT
004    IN : BOOL;
005    T  : TIME;
006 END_VAR
007
008 VAR_OUTPUT
009   Q    : BOOL;
010 END_VAR
011
012 VAR
013   TRIG_B: R_TRIG;    RS_B: RS;
014   TON_B: TON;
015 END_VAR
016
017 TRIG_B(CLK:=IN);
018 RS_B(S:=TRIG_B.Q, R1:=Q);
019 TON_B(IN:=RS_B.Q1, PT:=T);
020 Q:=TON_B.Q;
021
022 END_FUNCTION_BLOCK
```

c) ST code

Fig. 6.1. *Function block* FB_PULSE

Table 6.1
Declaration of C language block for GPS interfacing

ST declaration	C declaration in VM
FUNCTION_BLOCK GPS_GGA (*$HARDWARE_BODY_CALL ID:0003; Align:4 *) VAR_INPUT PORT : BYTE; END_VAR VAR_OUTPUT UTC : TIME_OF_DAY; LAT : LREAL; LON : LREAL; ALT : LREAL; QUALITY : BYTE; END_VAR END_FUNCTION_BLOCK	typedef struct __declspec(align(4)) tagIO_GPS_GGA { /*inputs*/ VM_BYTE Port; /*outputs*/ VM_TIME_OF_DAY Utc; VM_LREAL Lat; VM_LREAL Lon; VM_LREAL Alt; VM_BYTE Quality; } IO_GPS_GGA, *PIO_GPS_GGA;
Structure of the body	
switch(ID) {... case 0x0003: { PIO_GPS_GGA arg = (PIO_GPS_GGA)GET_PARAM_POINTER();...}...}	

The block's PORT input specifies communication channel. The outputs determine UTC time, LATitude, LONgitude and ALTitude of actual position, together with QUALITY of GPS reading. We stress that besides the declarations there is no body in ST component of the block.

Structure tagIO_GPS_GGA defined at VM side repeats ST declarations with alignment, specifies type name and pointer type. Executions of C blocks are implemented by switch(ID) statement with bodies entered at successive cases. So the body of GPS_GGA is entered at case 0x0003. Function GET_PARAM_POINTER() returns pointer to the structure determined for the blocks instance in declaration VAR ... END_VAR in the main ST program. The pointer is of general type void*, so must be converted to the type PIO_GPS_GGA. The resulting

pointer is saved in `arg` variable, sufficient for further processing. Other C language blocks are implemented in the same way. Given such template, hardware designers can prepare C blocks themselves.

7. FBD and IL compiler. The CPDev environment has been extended recently with simple graphic editor of FBD diagrams and compiler of IL textual language, mainly for teaching purposes. ST compiler remains basic platform of the environment.

7.1. Programming in FBD. The graphic editor, called Blockers (Fig. 7.1), provides basic editing functions, i. e. inserting blocks into diagram, connecting inputs and outputs of the blocks, selecting and removing objects, zooming, etc. The blocks are chosen from CPDev libraries. Global input/output variables and constant values are also placed in the diagram. Built-in syntax checker verifies correctness. Resulting FBD diagram is saved in XML text file whose structure follows recommendations of PLCopen [13]. The XML file is then converted into ST language by means of `FBD2CPDev` translator. Connections between the blocks and instances of the blocks are represented by automatically created local variables of corresponding types. Convention of variable names is based on types of blocks in the diagram and on execution order.

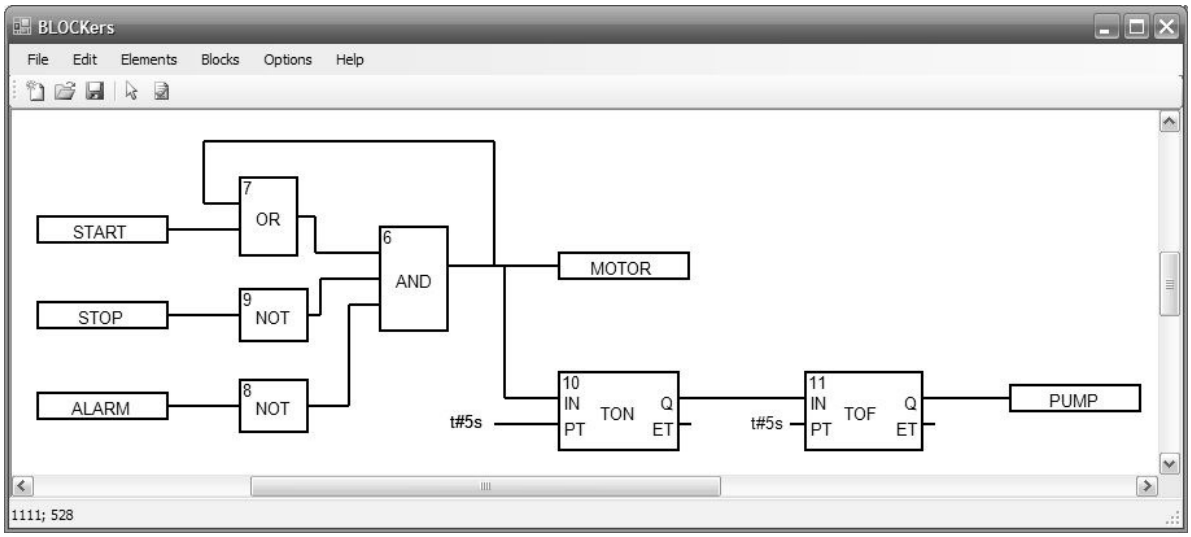


Fig. 7.1. *FBD diagram of the* `START_STOP` *system*

Figure 7.1 shows FBD diagram of the `START_STOP` system drawn using the Blockers editor. Numbers in the upper left corners of the blocks indicate execution order. Notice that in case of the function blocks `TON`, `TOF` the numbers may be used to distinguish instances. The variables placed in narrow rectangles on the left and right are interpreted as global. Equivalent ST code generated by `FBD2CPDev` translator is shown in Table 7.1 (compare Fig. 3.2).

Table 7.1
ST program converted from FBD

```
PROGRAM START_STOP

VAR_EXTERNAL
START : BOOL;
STOP : BOOL;
ALARM : BOOL;
MOTOR : BOOL;
PUMP : BOOL;
END_VAR

VAR
var_OR7_0 :  BOOL;
var_NOT9_0 :  BOOL;
var_NOT8_0 :  BOOL;
var_AND6_0 :  BOOL;
TON10 :  TON;
TOF11 :  TOF;
END_VAR

var_AND6_0 := AND(var_OR7_0,var_NOT9_0,var_NOT8_0);
var_OR7_0 := OR(var_AND6_0,START);
var_NOT8_0 := NOT(ALARM);
var_NOT9_0 := NOT(STOP);
TON10(IN := var_AND6_0, PT := t#5s);
TOF11(IN := TON10.Q, PT := t#5s);
MOTOR := var_AND6_0;
PUMP := TOF11.Q;

END_PROGRAM
```

It is seen that:

- connections between the blocks are represented by local variables `var_OR7_0` to `var_AND6_0`; name of a variable indicates source block of that variable,
- two instances `TON10`, `TOF11` are created, with names involving the block type and execution order.

Outputs of the instances, i. e. `TON10.Q` and `TOF11.Q`, are denoted in the standard way (compare Fig. 3.2).

7.2. Programming in IL. Since declaration parts of programs written in ST and IL are the same, and outcome of each compilation is a file with VMASM code, the compiler of IL language has been developed by extending the original ST compiler. The ST compiler generates the VMASM code from expression trees built of tokens acquired from ST code. By analysing a sequence of IL instructions one can create similar trees and employ them in succesive stages of compilation, in the same way as while compiling ST. This gives more efficient VMASM code than direct translation of IL instructions into VMASM, since VMASM, unlike IL, does not rely on the notion of accumulator. Accumulator is not needed in expression trees, typical for high-level languages.

Table 7.2
IL program for `START_STOP` *project*

```
PROGRAM PRG_START_STOP
                                    LD START
VAR_EXTERNAL                        OR MOTOR
    START : BOOL;                   ANDN STOP
    STOP  : BOOL;                   ANDN ALARM
    ALARM : BOOL;                   ST MOTOR
    MOTOR : BOOL;
    PUMP  : BOOL;                   CAL ON_DELAY(IN:=MOTOR, PT:=t#5s)
END_VAR                             CAL OFF_DELAY(IN:=ON_DELAY.Q, PT:=t#5s)

VAR                                 LD OFF_DELAY.Q
    ON_DELAY : TON;                 ST PUMP
    OFF_DELAY: TOF;
END_VAR                             END_PROGRAM
```

The `PRG_START_STOP` program of Fig 3.2 is rewritten in IL in Table 7.2. The instruction `LD START` loads `CR` register (*Current Result*; accumulator in IEC) with the value of `START`. Next the `CR` is `OR`ed with `MOTOR`, with the result in `CR`. The following `ANDN` negates `STOP`, `AND`s it with CR, always with the result in `CR`. Similarly for another ANDN. `ST MOTOR` saves `CR` in the variable `MOTOR`. `CAL` instructions invoke function blocks.

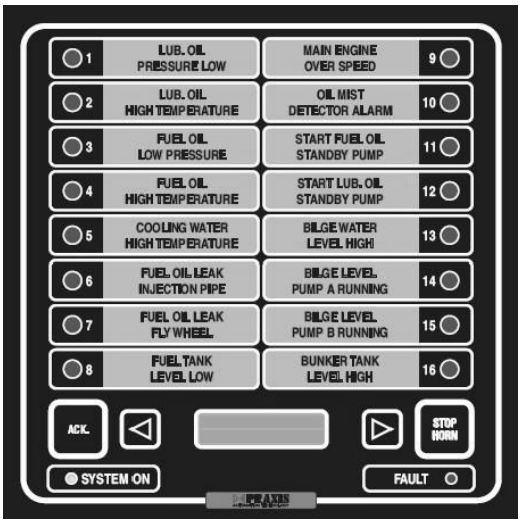
8. CPDev applications. The CPDev package is currently applied for programming new SMC controller from LUMEL, Zielona Góra, Poland. SMC operates as a central unit in small DCS systems involving distributed I/O modules, intelligent transmitters, PID controllers, etc. [12]. Development of another application in forthcoming version of MINI-GUARD Ship Control & Positioning System from Praxis Automation Technology, Leiden, The Netherlands, is in progress [8]. For lab and teaching applications PC-based softcontrollers can be used.

8.1. SMC controller. The SMC shown in Fig. 8.1a is based on Atmel AVR 8-bit microcontroller. Platform-dependent modules of virtual machine, i. e. interrupts, RTC and communication interfaces, have been written by LUMEL engineers, and sent to the authors in binary format. Consolidation of universal and LUMEL modules has resulted in a VM-SMC machine which, as SMC firmware, executes ST program compiled and downloaded from PC. The controller is equipped with two serial ports, one (*master*) for distributed I/Os and field devices, another (*slave*) for host PC or HMI panel. Modbus RTU protocol is applied (up to 230.4 kbaud). Third `Complex_blocks` library to implement self-tuning PID control loops is provided.

8.2. MINI-GUARD controllers. The MINI-GUARD system consists of seven types of controllers (Fig. 8.1b) involving NXP ARM7 16/32-bit microcontrollers. The controllers have application dedicated faceplates. Virtual machine for Atmel ARM7 has been sent to Praxis A.T., to be adapted for the NXP ARM7. The software to handle C language blocks described in Sec 6.2 has been developed especially for MINI-GUARD. The controllers communicate over Ethernet, external devices are connected via universal serial interface or OPC.

8.3. Softcontrollers with NI and InTeCo boards. A PC equipped with I/O board and executing a control program is called softcontroller. Two such boards can be used so far, namely NI-DAQ USB 6008 from National Instruments and RT-DAC/USB from InTeCo, Cracow, Poland (Fig. 8.1c,d). A common interface

a) SMC controller



b) Alarm Panel of MINI-GUARD

c) NI-DAQ I/O board

d) RT-DAC I/O board

Fig. 8.1. *Applications of CPDev package*

`CPDev.CPCom.ICommDev` has been developed, with provision for other types. Softcontroller is configured in two steps. First a board is selected from menu and I/O channels defined. Then global variables of the project are linked to the channels. Binary channels become `BOOL`s and analog one `REAL`s. Softcontrollers can be connected into DCS system by means of Modbus TCP protocol.

9. Conclusions and future work. CPDev environment for programming small controllers in ST, FBD and IL languages of IEC 61131-3 standard has been presented. The environment is considered open because compiled code can be executed by different processors, low-level software components are provided by hardware designers, and control programmers can create their own libraries with reusable program units. The compiler produces universal executable code processed by runtime virtual machine operating as interpreter. The machine is an ANSI C program composed of universal and platform-dependent modules. The machines for AVR, ARM, MCS51 (core) and x86 processors have been developed so far. User function blocks can be programmed in ST and C. The ST blocks are kept in CPDev libraries, whereas C blocks become components of virtual machine. FBD diagram is translated to ST and then compiled. CPDev has been used for programming controllers in two small DCS systems and for PC-based softcontroller with I/O boards.

Future work on CPDev will be motivated primarily by needs of the users. Next version will include structured data types and global arrays, at least two-dimensional (local arrays are available now). Current

simple FBD editor should be upgraded to more professional level. Depending on ST statements the compiled code is longer or shorter, as in the expression `x1 AND x2` vs. function `AND(x1, x2)`. Templates indicating more efficient solutions are important for the users. Virtual machine for FPGA platform with simple multitasking mechanism is currently under development.

REFERENCES

[1] A. Appel, J. Palsberg, *Modern compiler implementation in Java*, Cambridge University Press, Second edition, (2002).
[2] C# Language Specification, `http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx`, (2007).
[3] K. Cooper, L. Torczon, *Engineering a Compiler*, Morgan Kaufmann, San Francisco, (2003).
[4] IEC 61131-3 standard: Programmable Controllers—Part 3, Programming Languages, *IEC*, (2003).
[5] ISaGRAF User's Guide, *ICS Triplex Inc.*, (2005).
[6] K. H. John, M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems* Berlin—Heidelberg, Springer-Verlag, (2001).
[7] T. Lindholm, F. Yellim, *Java Virtual Machine Specification - Second Edition*, Java Software, Sun Microsystems Inc, (2004).
[8] Mini-Guard Ship System, *Praxis Automation Technology B. V.*, `http://www.praxis-automation.com`, (2009).
[9] 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).
[10] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, L. Trybus, Programming controllers in Structured Text language of IEC 61131-3 standard, *Journal of Applied Computer Science*, Vol. 16, No 1, (2008).
[11] E. Tisserant, L. Bessard, M. de Sousa, *An Open Source IEC 61131-3 Integrated Development Environment*, 5^{th} Int. Conf. Industrial Informatics, Piscataway, NJ, USA, (2007).
[12] SMC, *Lumel S.A.*, `http://www.lumel.com.pl/en`, (2009).
[13] XML Formats for IEC 61131-3 ver. 1.01 Official Release, `http://www.plcopen.org`, (2007).

Edited by: Janusz Zalewski
Received: September 30, 2009
Accepted: October 19, 2009