The Concept of a Dual-Core PLC

Subjects: Automation & Control Systems | Computer Science, Hardware & Architecture
Contributor: Marcin Hubacz, Bartosz Trybus

IEC 61131-3-compliant engineering environments consist of IDE (integrated development environment) with language editors, compiler of source programs into binary code, and runtime for execution of the code. In a dual-core PLC proposed here, the cores run different projects cooperating by shared memory.

Keywords: dual-core ; PLC ; IEC 61131-3 environment

1. Notes on the IEC 61131-3 Standard

There is a common view among practicing engineers and university staff that runtime engineering environments based on the IEC 61131-3 standard [1] will remain a state of industrial practice at least until the end of this decade. The standard defines five programming languages, namely textual IL, ST, graphic LD, FBD, and mixed SFC, with time-triggered scan cycle or event-driven execution. IL, LD, and SFC are preferred in manufacturing based on PLCs, whereas ST, FBD, and also SFC dominate in general automation.

The IEC 61131-3 introduces the concept of program organization units (POUs), such as programs, function blocks, and functions. Variables are local in a POU where they are declared, or global if the scope applies to the whole project. Global variables are used for communication.

Software environments implementing the standard, for instance CODESYS [2], STEP7 [3], automation builder [4], LogicLab [5], or others, consist of three essential components, namely IDE, compiler, and runtime. IDE provides editors of the IEC 61131-3 languages, and then compiler translates the source program into executable binary code transferred to the runtime in the controller processor. The runtime executes the code in real-time with a given cycle or when an event occurs. The addresses of global variables are also provided by the compiler.

2. General Architecture of a Dual-Core PLC

The dual-core PLC being developed may be viewed as equivalent to two separate PLCs with some communication link. Here, the link is provided by the shared memory. **Figure 1** shows the general architecture, where Projects 1, 2 represent software of the PLC cores. It is assumed that each of them can also operate independently using its own local memory.

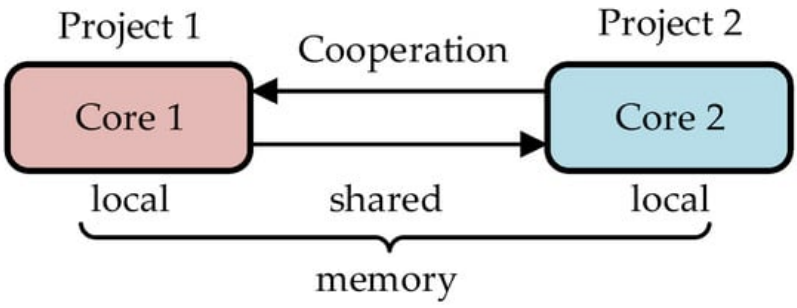

Figure 1. General architecture of a dual-core PLC.

The runtimes of the cores are copies of a single-core runtime with some minor extensions (see Section 4). Cooperation of the cores through the shared memory is provided by means of global variables.

3. Shared Global Variables

Define the following sets of global variables:

GV_1—variables updated in Project 1 and used in Project 2.

GV_2—variables updated in Project 2 and used in Project 1.

$GV = GV_1 \cup GV_2$—variables shared between Projects 1, 2.

The sets are depicted in **Figure 2**. The areas outside the $GV_1 \cup GV_2$ circles indicate other global variables declared in the projects but not shared.

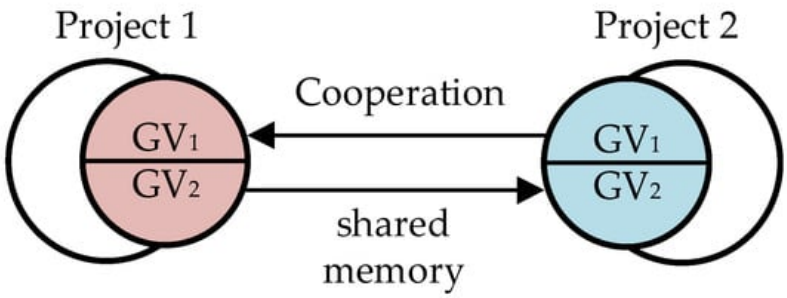

Figure 2. Sets of global variables in the two projects.

To make cooperation of the projects feasible, the original single-core designs must obey the following rule:

- Declaration of global variables in each of the two projects must contain all shared variables, i.e., the GV set.

Note that application of the rule requires also an extension to GV variable declaration in IDE, so as to indicate whether the variable is updated in the particular project or not. It may be implemented as an additional attribute of the variable, for instance, with WRITE meaning updating in the shared memory the value and READ meaning reading only. Naturally, a GV variable cannot be declared updated in both projects.

4. Improvement of Memory Organization

It seems quite probable in industrial practice that the variables from the sets GV_1, GV_2 will not be declared one after another, but will be scattered among other global variables. Transfer of such scattered variables to/from the shared memory would have to be performed individually, one by one. To make the transfer more time-efficient, the scattered GV_1, GV_2 variables may be collected into compact memory sectors transferred by fast memory copying. An extension of the single-core compiler for the dual-core application is needed to make such an ordered arrangement. The recommended memory organization after the arrangement is shown in **Figure 3**.

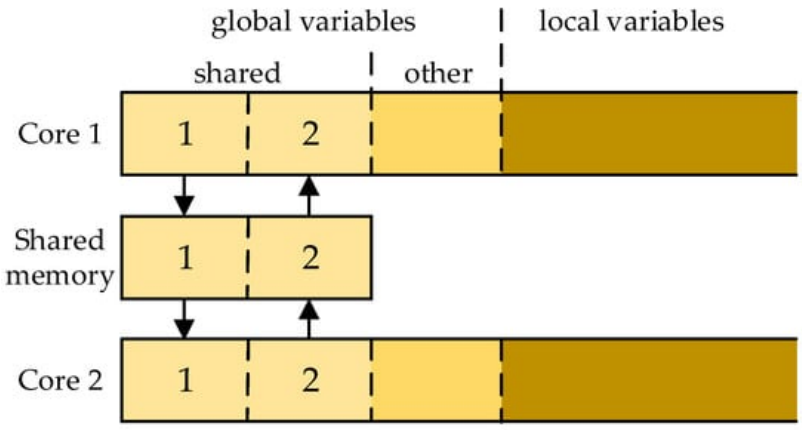

Figure 3. Memory organization of the dual-core PLC.

After such an arrangement, the relevant sectors of Core 1 memory may be denoted as:

- $CM1_1$—sector for GV_1 (updated WRITE, another words output);
- $CM1_2$—sector for GV_2 (received READ—input).

Likewise, we have $CM2_1$ (input), $CM2_2$ (output) for Core 2, and SH_1, SH_2 for the shared memory.

5. Operation of a Dual-Core PLC

During execution of the binary code, the runtime programs of typical PLCs apply read-execute-write semantics for the scan cycle. This means that all input values are read into internal local copies once a program starts its execution at the beginning of the cycle. During the remainder of the execution, only those local values are used (the mechanism is sometimes called a process image). At the end of the execution, the output values are written into global copies to be used in the next cycle.

In case of Core 1, the content of $CM1_2$ memory becomes the input (see **Figure 3**). However, so far, it is stored in SH_2, where it has been copied earlier from $CM2_2$, once Core 2 finished its execution. Hence, at the beginning of the Core 1 cycle, SH_2 must be copied into $CM1_2$. At the end of the execution, the results updated in $CM1_1$ are copied into SH_1. So, the beginning of execution and the end of the cycles of two cooperating cores may be illustrated as in **Figure 4**. To implement such behavior, the original single-core runtime must be extended by the copy-from the shared memory at the beginning of the cycle and copy-to at the end. Note that operations executed at the beginning may be called precycle, whereas those executed at the end may be called postcycle.

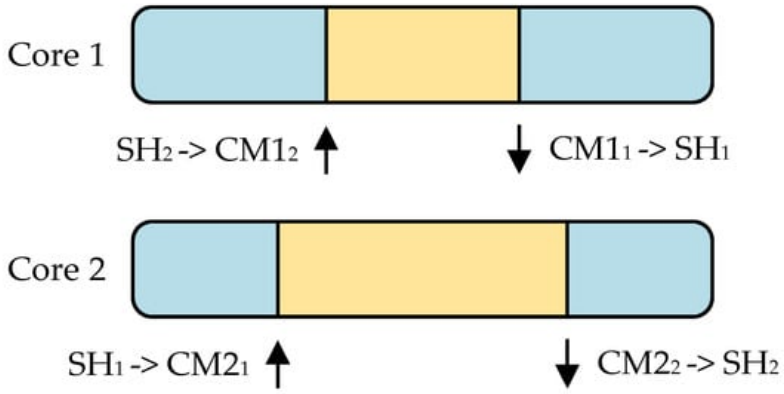

Figure 4. Scan cycles of the dual-core PLC.

To avoid conflicts between the cores while accessing the shared memory, multi-core processors have built-in hardware semaphores to check whether access is currently possible. Runtime software of each core must involve an algorithm for operating the semaphore, so raising it (releasing) when the data transfer is completed. An example of such an algorithm is shown in Section 5.

To summarize, three extensions of a typical single-core IEC 61131-3 runtime engineering environment are needed for implementation into dual-core PLC running two projects:

- IDE: an additional attribute of a shared global variable to indicate whether it is updated in the actual core or received from the other one.
- Compiler: memory arrangement so as to have updated and received shared variables in compact sectors (optional).
- Runtime: copy-from the shared memory at the beginning of the cycle (precycle) and copy-to at the end (postcycle).

References

1. John, K.H.; Tiegelkamp, M. IEC 61131-3: Programming Industrial Automation Systems; Springer: Berlin/Heidelberg, Germany, 2010.
2. CODESYS. Available online: https://www.codesys.com/ (accessed on 11 October 2023).
3. SIMATIC STEP 7 (TIA Portal). SIEMENS. Available online: https://www.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal/software/step7-tia-portal.html (accessed on 11 October 2023).
4. Automation Builder. Available online: https://new.abb.com/plc/automationbuilder (accessed on 11 October 2023).
5. LogicLab. Available online: https://www.axelsoftware.it/en/logiclab/ (accessed on 11 October 2023).