MORDICUS-HW: FRAMEWORK FOR BACK-END ELECTRONICS CONTROL AND CONFIGURATION*

Georgiy Stelmakh

Applied Mathematics Faculty, National Technical University of Ukraine
Kyiv Polytechnic Institute, Kyiv, Ukraine
georgiy.stelmakh@cea.fr

(Received August 20, 2014)

The paper addresses the architectural patterns and programming principles of *Mordicus-hw*, a framework designed to optimize collaborative development between electronics and software engineers by providing them with software tools that are adapted to their respective activities. When designing specific modules, electronics engineers are often led to write small "quick and dirty" C/C++ programs in order to test and debug their design. But the actual software that they have developed to run and test their hardware designs is often discarded afterwards because it lacks the re-usability required by the application modules developed by software engineers. The *Mordicus-hw* framework addresses these issues, favours collaborative design and development between electronics, and software engineers by providing a simplified, high-level, script-based register programming platform.

DOI:10.5506/APhysPolBSupp.7.695
PACS numbers: 07.05.Hd, 29.85.Ca, 89.20.Ff

1. Introduction

When designing their specific modules, electronics engineers are often led to write small "quick and dirty" C/C++ programs in order to test and debug their design. To compile and run these programs, they need to install and maintain tool-chains, write makefiles, or learn specialized APIs that are outside their expertise. The valuable part of their work, *i.e.* the actual software that they have developed to test their hardware designs is often discarded afterwards because it lacks the re-usability required by the application modules developed by software engineers.

* Presented at the Workshop on Picosecond Photon Sensors for Physics and Medical Applications, Clermont-Ferrand, France, March 12–14, 2014.

The *Mordicus-hw* framework has been designed to address this issue and more generally, to optimize collaborative design and development between electronics and software engineers by providing the former with a simplified, high-level, script-based register programming platform and the latter with modular C++ classes and templates capturing the patterns and best practices that ensure quality software for handling hardware devices over a distributed application.

A prototype version [1] was implemented and used in the test bench of the AGET chip developed for the GET generic electronics for TPC-based medium size nuclear physics experiments. *Mordicus-hw* is part of the more general *Mordicus* framework designed by CEA IRFU for the development of TDAQ systems.

2. Register access policies

In *Mordicus-hw*, the software layers closest to the hardware are modelled as sequences of register read/write. Registers belong to devices representing the different electronic entities. Every device is associated to a "register access policy" representing the protocol through which hardware registers are read from or written to.

The most common way to access a register through software is to have it mapped over the memory space of the processor, other more complex, standardized (such as I2C [2]) or proprietary protocols must often be used for different kinds of electronic devices. The framework architecture confines the specification of the register access policy to a single Policy class that basically implements 4 elements that fully characterize the way a register is accessed:

- the type that represents a register reference;
- the data type that is read from/written to the register;
- the register write function: `poke()`;
- the register read function: `peek()`.

3. Remote register access policy

The major goal of *Mordicus-hw* is to allow users to read from and write to hardware registers remotely from a general workstation. This calls for a client–server architecture (Fig. 1) in which a client sends read and write requests to a register access server on the embedded system containing the devices hosting the registers. For each request, the server determines the local policy through which the target registers are accessed before executing the actual read or write operation (Bus, I2P, AGET or other one protocol).

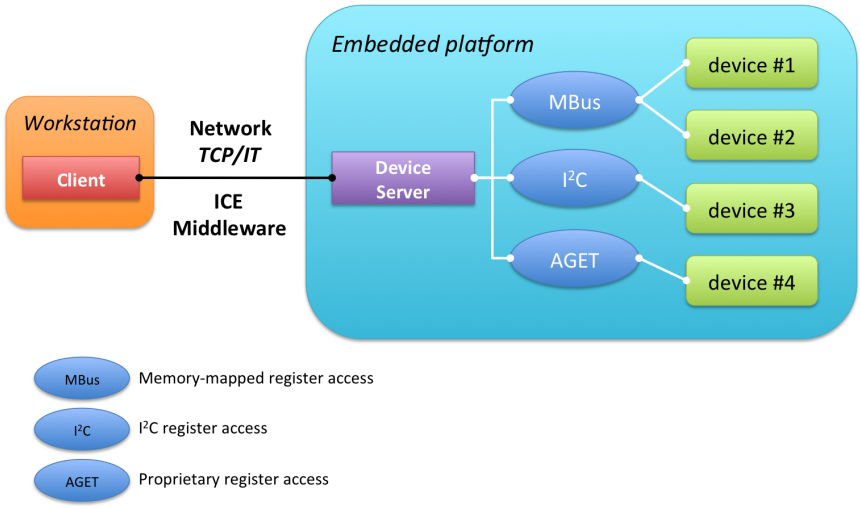

Fig. 1. Client–server architecture.

From the client on the general-purpose workstation the server on the embedded system can be accessed though the network (TCP/IP/UDP *etc.*) with the help of ICE. The server interacts with the devices in the embedded system with the help of some protocol, which should be implemented by firmware engineers, and then it becomes easy to modify any register in any device. Also, the advantage is that *Mordicus-hw* allows engineers to easily upgrade (for example, adding more devices, remove them *etc.*) their system without modification their configuration program.

3.1. Internet Communication Engine (ICE)

The mechanism which has been described above is based on the Internet Communication Engine (ICE), developed and maintained by ZeroC [3]. An additional advantage of using such middleware is to allow clients to be written in programming languages that are not the same as the one used for servers, for example: client is in Python while embedded server has been developed in C++.

If there is distributed process and the client–server architecture, there also should be an interaction between client and server with the help of some network protocol. And, of course, the best way is to spend as less efforts as it possible. ICE will help with it.

If there is a client–server model, there also should be the contract between client and server to determine how they should interact. This contract is the interface definition. Here comes the ICE to provide cross-platform and

cross-language capability (Fig. 2). It generates code for client and server languages, named the interaction language. Then the only thing that remains to software engineers is to implement server code.

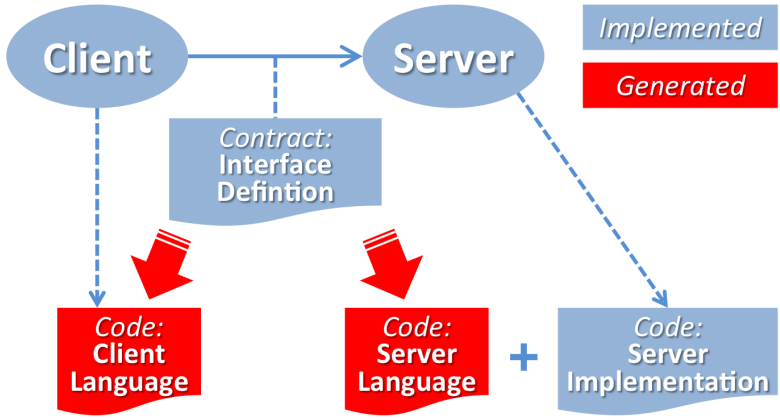

Fig. 2. Client–server architecture over ICE middleware.

4. Optimization issues

The main problem of interaction between the client and the target server is latency. If script contains a huge amount of read/write operations then loading configuration to the device will take some time.

Bit-field access optimization was actually implemented in *Mordicus-hw* resulting in significant acceleration of control sequences. The caching mechanism uses C++ objects which accomplish the single register read in their constructor and the final write-back in their destructor, doing all the bit-field access operations in the form of chained method (bit-fields are referenced here as strings):

```
Reg.poke("control", 1).poke("status", 11);
```

In this example, `Reg` is a register object and the first call to the `poke()` returns the object on which, the subsequent `poke()` calls are made.

There are some plans for the future to implement the possibility to send whole script (not by the command) to the embedded system in the single network operation, and then process it in the embedded system.

5. Conclusions

The basic mechanisms that provide a reliable distributed control and configuration framework have been already implemented in *Mordicus-hw*. However, there are three major goals that should be achieved in future.

The first goal is to optimize network performance — to implement the "Batch" objects — series of remote register access instructions (or a whole configuration script) that would be transported in a single network operation to the target node and then locally interpreted and executed. This feature could provide more possibilities for automatically reusing scripts developed by firmware engineers.

Next step is advanced device parametrization — the possibility to instantiate register devices of any kind with an arbitrary number of parameters.

And the last goal is to implement parameters in more than 64-bit values. It means the possibility to transfer in a single network operation the whole configuration database.

REFERENCES

[1] J. Chavas *et al.*, *Mdaq-D3, a C++ Distributed Driver Development Framework Used in a Nuclear Physics Experiment*, Proceedings of Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2011.

[2] *Everything about I2C*, `http://i2c.info/`

[3] M. Henning, *A New Approach to Object-Oriented Middleware*, IEEE Internet Computing, January 2004, pp. 66–75.