embeddedworld2022
Exhibition&Conference
... it's a smarter world

A model-based approach to mastering complexity in FPGA-SoC software development

Alexander Wirthmüller (*Author*)
MPSI Technologies GmbH
Munich, Germany
aw@mpsitechnologies.com

***Abstract*—A comprehensive software development method is presented which covers the vast range of software aspects to be considered for feature-rich FPGA-SoC applications. By employing an Open Source model-based design tool, the need for profound programming skills at all, i.e. RTL, Embedded Linux and user interface / M2M communication levels is greatly reduced. The method allows developers to focus on core project functionality while repetitive/auxiliary coding tasks are automated. At the same time, clean source code structure and good coding standards can be strictly enforced.**

Keywords—software engineering; model-based design; FPGA-SoC; heterogeneous computing

I. INTRODUCTION

Increasing availability of low-cost computer-on-modules (COM's) based on FPGA-SoC's, combining compute cores and programmable logic in one package, paves the way for complex data processing and advanced control close to where the action happens, for example on the factory floor.

While hardware capabilities are exploding, embedded software developers are faced with a multitude of programming concepts and languages to be mastered in order to make good use of those features.

In this paper, the Open Source software development tool Whiznium is presented with a tabletop 3D laser scanner serving as use case. Rather than requiring manual project bring-up, Whiznium gives developers a head-start by providing modular, well-structured, auto-generated source code trees for complex applications. Here, this is highlighted in individual project aspects, ranging from RTL computer vision implemented in VHDL, to multi-core Embedded Linux data processing and session management in C++, to HTTPS and OPC UA communication, for web user interfaces and industrial-grade connectivity, respectively.

II. WHIZNIUMDBE AND WHIZNIUMSBE

A. Motivation

The principal motivation for designing Whiznium was to relieve the developer of work in multi-programming language settings typical for FPGA-SoC's, especially with regard to functionality that is repeated from project to project. The vendor-specific low-level synthesis and placement tools (for the FPGA area) or compilers (for the MCU/CPU area), which get the best possible performance out of the target hardware, were not to be replaced. Also a new "one-size-fits-all" programming language would not have done justice to the task of realizing heterogeneous embedded software projects. The aspects that occur are too different and the existing languages have prevailed not least because they guarantee optimized results for their specific application aspect.

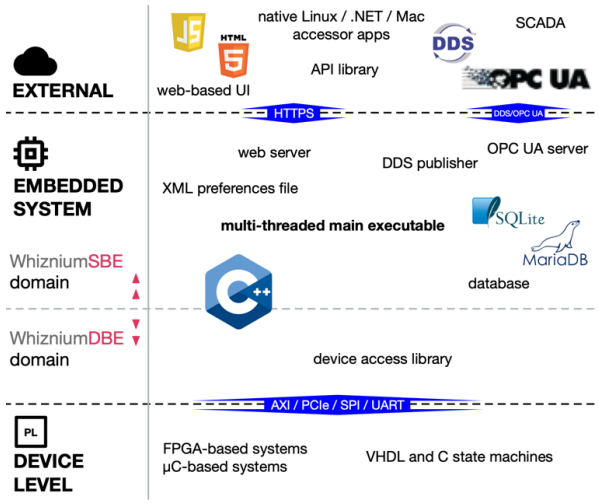

Figure 1 Overview software projects with WhizniumDBE & SBE

As a result, the chosen developer-friendly approach, which promises massive time savings while respecting existing programming languages and aspect-specific best practices, was model-based source code generation. It was subsequently implemented in WhizniumDBE and WhizniumSBE.

B. Software design process

The goal of the software design process with Whiznium is to enable the tool to write a synthesizable or compile-ready source code tree that covers all project aspects. For this purpose, Whiznium is given a software description in the form of aspect-specific model files, the level of detail of which usually increases

as the project progresses. Accordingly, the output of Whiznium at the beginning of a project is limited to a skeleton of source code files, which is filled by manual additions, classical debugging and iteration of the model information, coupled with repeated Whiznium calls, until all project requirements are met.

Whiznium is divided into WhizniumDBE (Device Builder's Edition [1]) for FPGA and MCU code, and WhizniumSBE (Service Builder's Edition [2]) for multi-core compatible embedded Linux software. Both tools (... are WhizniumSBE projects, an indication of the universality of the WhizniumSBE concept, and ...) each follow their own sequences of progressively accumulating project information, from coarse to fine, in an SQL database.

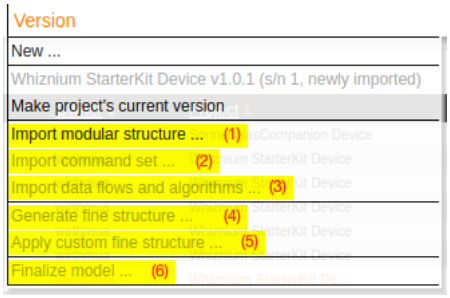

Figure 2 Sequence of import/generation steps WhizniumDBE

The beginning and end of the WhizniumDBE sequence (Figure 2) are well known from FPGA projects, but are also used here in a modified form for MCU projects: from a hierarchical module structure, there are intermediate steps such as commands that the processing system (PS) can trigger in programmable logic (PL), as well as algorithms that can be parallelized, until reaching the fine structure of ports, signals and processes.

A key feature of Whiznium is the built-in ability - besides manual imports - to automatically derive elements of the next finer level. An example of this is the automatic generation of handshake fine structure for command calls between FPGA modules. In addition, custom module templates can be incorporated, which, thanks to access to the SQL database, can engage with the WhizniumDBE sequence based on parameters

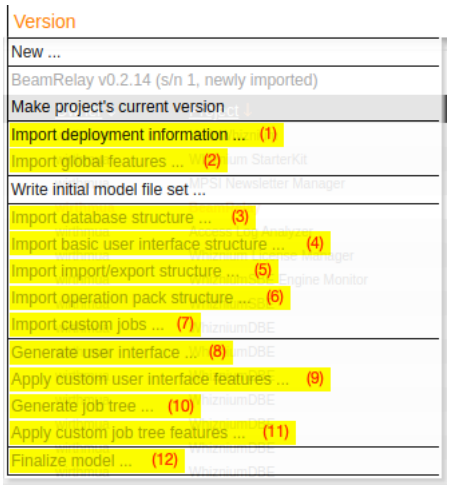

Figure 3 Sequence of import/generation steps WhizniumSBE

- in extensible, open C++ code. This possibility is significantly more powerful than just using generics in VHDL modules.

The WhizniumSBE sequence (Figure 3) has more aspects to consider than its WhizniumDBE counterpart. It begins with information on target platforms and components (see below), the structure of an underlying relational SQL database (usually SQLite [3] for embedded projects) and a rough outline of the user interface (UI). The next finer level describes, for example, the distribution of arithmetic operations to separate executables or the structure of the UI down to the level of individual controls. Finally, the hierarchical job tree describes a structure of C++ classes whose objects are responsible for individual hardware, algorithm, or UI aspects at runtime. State machines, inter-job communication and XML/JSON data blocks for the MVC-compliant UI can also be specified here.

After all relevant project information has been gathered in the WhizniumDBE or WhizniumSBE SQL databases, source code trees are generated in a simple but robust process. A large number of template files are used, which are individualized with placeholders and insertion points for resulting code files. In the case of WhizniumDBE, template files can also be specific for module templates.

As previously mentioned, hand-written code exists symbiotically within automatically generated source code trees. Therefore, starting with the second call of Whiznium for a project or with the first model iteration, manual content of the existing source code tree is extracted from insertion points (comments in the respective programming language) before being reinserted into the new version's source code tree. Comprehensive syntax checks secure this process.

C. *WhizniumDBE*

WhizniumDBE can generate the following components:

- device access library: C++ code for the PS, which enables non-blocking calling of PL commands and data transfers from/to the PL
- easy model device access library: C++ code for the above functionality, but with serial execution
- FPGA code: VHDL code and constraints for PL, divided into modules, with PS-callable commands and data transfers. Optionally with FPGA instruction cache
- MCU code: C code optimized for minimal size, which emulates the event-based logic of PL

D. *WhizniumSBE*

WhizniumSBE can generate these components:

- main engine: multi-core compatible C++ code with SQL database connection, which on the one hand reliably accesses hardware subsystems and on the other hand manages sessions for external access. A libmircohttpd-based HTTPS server and optionally an OPC UA server and a DDS publisher are included for M2M communication

- operation engines: C++ code for satellite executables that can execute remote procedure calls (RPCs). Connection to the main engine via HTTP
- combined engine: a combination of the functionalities of the main engine and operation engines if no cluster architecture or heterogeneous multi-core architecture is available
- database access library: C++ code for easy access to the project MySQL/PostgreSQL/SQLite database
- web app user interface files: HTML5 and JavaScript code for multilingual web-based UI. Detailed access management according to user levels
- Vue.js app: a more modern-looking implementation of the web-based UI
- API library: C++ code to enable machine-to-machine control based on UI communication. This simplifies integration into native Linux applications
- C# API library: same as above for integration into native Windows applications
- Swift API package: same as above for integration into native macOS applications
- Java API package: same as above for integration into Java applications

III. Demo Project

A tabletop 3D laser scanner with a turntable on the one hand and a 5MP camera and line lasers on the other (Figure 4, [4]), whose logic is implemented on an FPGA SoC [5], serves as demo project. FPGA SoC's ideally embody the fusion of low-level and high-level embedded software, as PS and PL are combined on one piece of silicon. Whiznium however is also suitable for pure PS/PL systems, as well as for those where PS and PL are found on separate chips or circuit boards.

IV. Further Whiznium Concepts

Whiznium emerged from everyday developer work, so the following requirements relevant to developers have been considered:

- Whiznium relies on few but well-established and open technologies: libxml2, libmicrohttpd, [one of] MySQL/PostgreSQL/SQLite, [optional] Vue.js
- thanks to the "all code in plain sight" principle, all functionality is easy to understand
- projects started with Whiznium can be further developed independently
- Whiznium is compatible with versioning via Git, especially for source tree iterations
- generated code is optimized for good readability and has a modular structure. This facilitates testability and maintenance; the usual debugging tools can be used

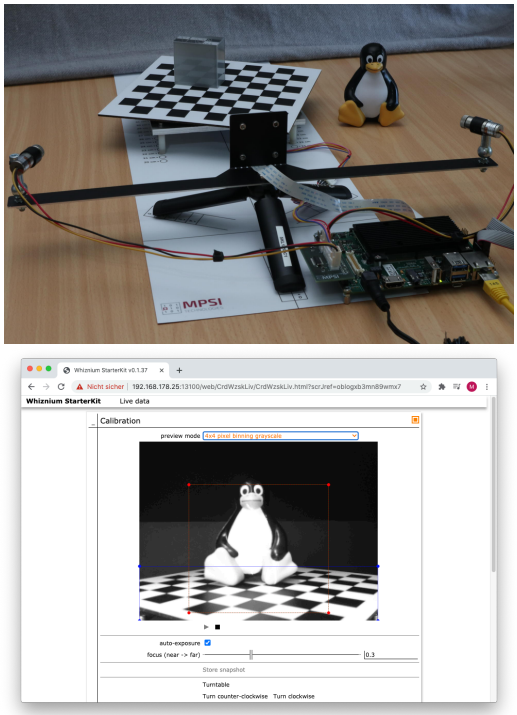

Figure 4 Tabletop 3D laser scanner hardware and software (excerpt)

- WhizniumDBE and WhizniumSBE are Open Source. This enables user-specific, also closed source, extensions

V. Conclusion and Outlook

A methodology was presented to develop powerful software for modern embedded systems in a time-saving manner. The model-based approach of the Whiznium tools covers most aspects occurring in FPGA-SoC systems with a high level of detail.

Whiznium is a living project which, thanks to its openness, can easily incorporate market trends, be it new hardware platforms or new programming approaches, into its model description and source code generation. Contributions of practical relevance from the community are always welcome.

References

[1] Whiznium Device Builder's Edition on GitHub: https://github.com/mpsitech/wdbe-WhizniumDBE

[2] Whiznium Service Builder's Edition on GitHub: https://github.com/mpsitech/wznm-WhizniumSBE

[3] SQLite project home page: https://www.sqlite.org/index.html

[4] Whiznium StarterKit on GitHub: https://github.com/mpsitech/wzsk-Whiznium-StarterKit

[5] Microchip PolarFire SoC FPGA's: https://www.microchip.com/en-us/products/fpgas-and-plds/fpgas/polarfire-fpgas/polarfire-soc-fpgas

(all links accessed on May 29, 2022)