ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA115

CODE GENERATION TOOLS AND EDITOR FOR MEMORY MAPS

P. Plutecki*, B. Bielawski, A. Butterworth
CERN, Geneva, Switzerland

Abstract

Cheburashka, a toolset created in the Radio Frequency Group at CERN, has become an essential part of our hardware and software developments. Due to changing requirements, this toolset has been recently rewritten in C++ and Python. A hardware developer, using the graphical editor, defines a memory map, which is subsequently used to ensure consistency between software and hardware. The memory map file is an input for a variety of tools used by the hardware engineers, such as VHDL code generators. In addition to aiding the firmware development, our tools generate C++ wrapper libraries. The wrapper provides a simple interface on top of a Linux device driver to read and write registers by exposing memory map nodes in a hierarchical way, performing all low-level bit manipulations and checks internally. To interact with the hardware, a software that runs on a front-end computer is needed. Cheburashka allows us to generate FESA (Front-End Software Architecture) classes with parts of the operational interface already present. This paper describes the evolution of the graphical editor and the Python tools used for C++ code generation, along with a description of their main features.

INTRODUCTION

Cheburashka [1], developed in Java, has been serving both as an editor of XML memory maps and a code generation tool for FESA [2] (Front-End Software Architecture) and the driver wrapper. For the VHDL code generation, Gena, a tool written in Python, has been used. At some point it has been decided to rewrite the code generators in Python, with extensive use of templates. The file format of the memory maps has been changed to YAML [3] (YAML Ain't Markup Language), also to work with a new tool called Cheby [4], a VHDL generator developed by the Hardware and Timing section, Controls group at CERN, the successor of Gena. Despite YAML being more human-readable than the original format (XML), considering the complexity of memory maps for devices which have been developed at the Radio Frequency group and users' habits, a new memory map editor was needed.

MEMORY MAP EDITOR

The new GUI (Graphical User Interface) has been developed to work with new file format (YAML) for memory maps. Since many code generation tools written in Python were already production ready, instead of having a monolithic program for editing memory maps and generating code, it has been decided to have the editor split into two parts:

- a generic C++ core, that, based on a schema file, can work with a single document YAML file,
- a set of Python scripts that will provide features specific for the memory map editor and possibility of running external Python tools, such as code generators.

To achieve this, a Python 3.6 interpreter has been embedded in the editor, using pybind11 [5] library.

As it is not a trivial task to design an intuitive interface, the GUI is composed of dockable widgets (see Fig. 1), which can be freely rearranged by the user. These widgets can be hidden, detached, tabbed and resized. All changes done to the layout are saved in the user's configuration file, so they are persistent.

Core

The core of the editor has been written in modern C++, using Qt5 [6] libraries. For parsing YAML files, yaml-cpp [7] has been selected, as it is a C++11 library that supports the latest YAML standard (1.2).

The editor needs a schema file, which defines the structure of a YAML file that is being edited. The schema, also a YAML file, specifies allowed mappings and sequences. Moreover it defines basic validation rules for scalars:

- if it is required,
- its type:
  - boolean,
  - integer,
  - hexadecimal integer,
  - floating-point,
  - string,
  - enumeration,
  - file.
- regular expression pattern matching,
- its default value,
- in case of numerical types, a range (minimum and/or maximum value).

In most cases, these validation rules are sufficient, although sometimes it is necessary to define more sophisticated checks. For this, a validation function written in Python can be bound to a scalar or a mapping.

In addition to the validation, auxiliary attributes can be defined in the schema, such as a tooltip text, if a scalar can be added, removed or edited by the user. The latter are

* p.plutecki@cern.ch

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA115

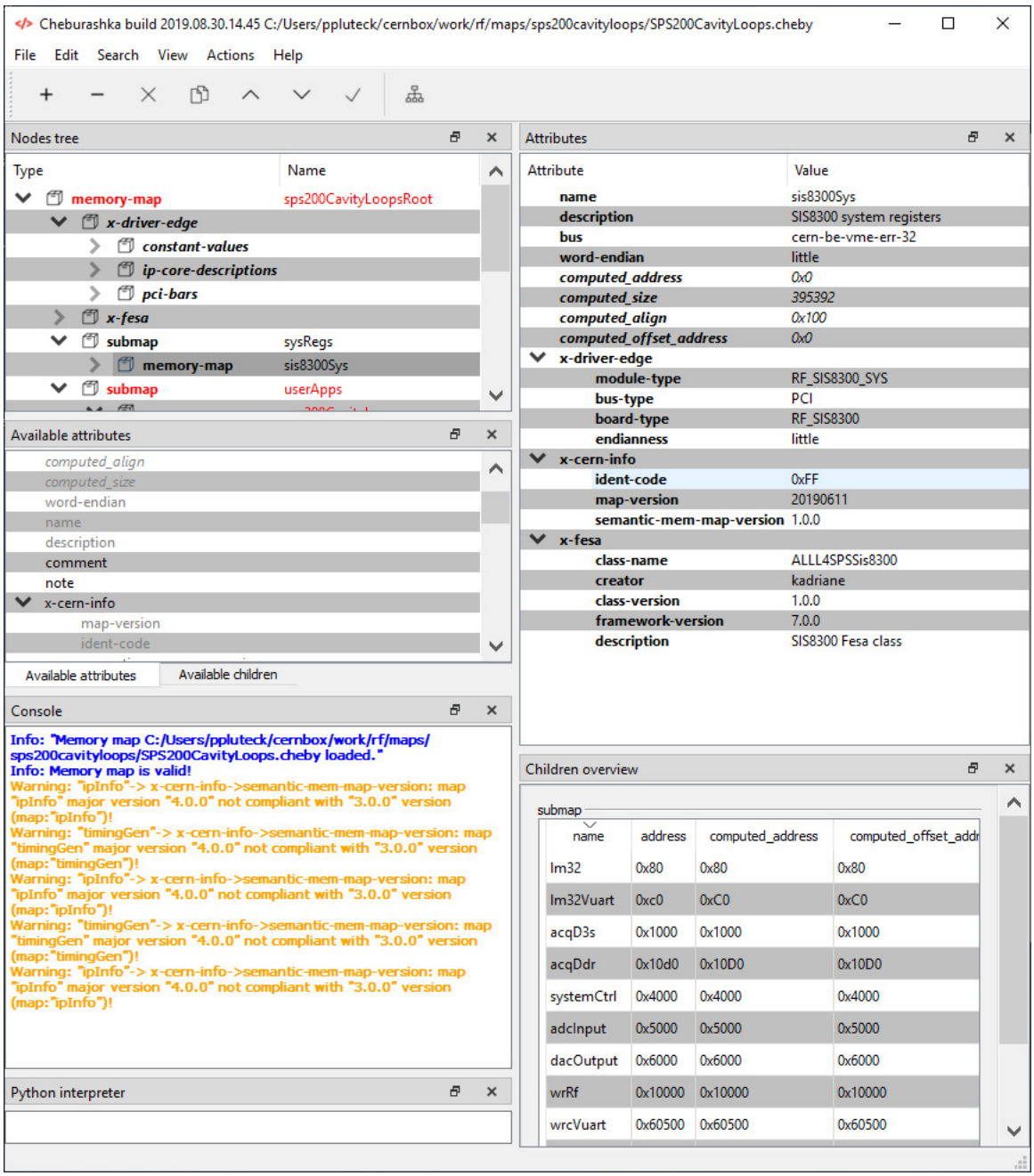

Figure 1: Main window of the memory map editor, with all widgets docked.

especially useful when some data need to be calculated at runtime and presented to the user.

By default, when opening the application, the standard schema is loaded, but the user can select a custom schema file in the settings menu.

The GUI has standard features such as:

- a search menu, working also with regular expressions,
- an autosave functionality with configurable interval and file rotation,
- an advanced undo/redo mechanism, which covers all user actions.

The debugging process might be difficult, especially when a developer can't reproduce steps leading to an application crash. As a precaution, breakpad [8], an open source library, has been used as the crash dump generator, mainly because of its availability on target platforms (Windows, Linux). If the application crashes, a dump file is created, which later can be used by the developer to see stack traces, combined with source code references, that lead to the crash.

Using Gitlab's CI/CD (Continuous Integration/Continuous Deployment/Delivery) mechanism, the Continuous Deployment strategy has been put in place. After the developer commits changes to the repository, the

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA115

application is built for Windows and Linux and deployed to CERN's network file systems, becoming accessible to the users. Symbol files, required to create references to source files in stack traces, are automatically created and stored.

C++ and Python Bindings

The C++ core exposes the major part of its interface (also partially Qt's API) to the embedded Python interpreter, but the bindings themselves are not sufficient for the editor to work properly, as it has to react to changes in a memory map made by the user. To solve this, a configuration file needs to be provided. It allows to define hooks to a Python module and function, that will be automatically called when a defined event occurs. Currently, hooks can be defined to such events as:

- an attribute has been inserted, modified, removed,
- a node has been inserted, moved, removed,
- a file has been loaded,
- a save or exit action has been requested,
- the opened file has been changed externally.

The configuration file also allows adding a custom button or a menu entry inside the editor, which, when clicked, will trigger a defined Python function. If a function needs user input, for instance to run an external tool with specific options, a list of arguments with their data types can be configured as well. For every defined argument, a dialog will pop up, with a default value from the previous call.

By default, the embedded interpreter adds the binary path to its search path (PYTHONPATH), but often it is necessary to specify additional ones. Platform dependent paths can be defined in the configuration file and, during runtime, the user can set alternative ones using the editor's settings menu. This is extremely useful as the GUI can be extended by existing Python software.

Advanced Validation

Many memory map nodes and their attributes need special validation, which is not achievable using standard rules that can be defined in the schema. A good example of such mechanism is a conversion factor validator. A read conversion factor is a formula that takes a raw value of a register and converts it to a high-level value, usually using basic algebra. A write conversion factor does the opposite, takes a high-level value and converts it to a raw value, that can be written to the hardware. Since our tools need the formula to work "as is" in both C++ and Python, it has to be syntactically and semantically correct, which checked by a Python validation function, using AST (Abstract Syntax Tree) parsing. Another example might be verifying that fields (sequences of bits) inside a register don't overlap. Any inconsistencies found by validators will be immediately highlighted by the editor allowing the user to fix them right away.

Special Functionalities Fone in Python

Some features specific to the memory map editor have been implemented in Python. The first one is the memory node addressing. An address attribute can be set as a relative (to the node's parent), hexadecimal value, or "next", which indicates that the address should be calculated based on the node's predecessor. Addresses are automatically recalculated when the memory map layout is altered. The same happens when some attributes (memory size, register width) are changed.

A memory map can contain submaps, which are memory nodes that point to a different memory map file. Contents of submaps are seamlessly loaded and presented as if they were parts of the top-level map. The editor can detect if any changes have been made inside a particular submap and upon saving or exiting the application ask the user what to do with these. When multiple nodes are referencing the same submap file, only the first one is editable. Also the GUI can detect if a submap file has been changed externally and inform the user.

CODE GENERATION

A tool called PyCheb is used as an entry point for our code generators. This tool is responsible for opening a memory map file and initial parsing, providing a comprehensive interface for other scripts. To make sure that any required changes and improvements in the generated code are easy to achieve, the Jinja2 [9] templating engine has been used.

Driver Wrapper

The first main feature of the driver wrapper generator is to create a CSV (Comma-Separated Values) file, which serves as an input for EDGE (Encore Driver GEnerator), developed by the Hardware and Timing section of the Controls group at CERN.

The second one is to generate a C++ library, which closely interacts with the EDGE Linux driver. The library provides a hierarchical interface over every memory node that is defined in a memory map. The interface allows software developers to read or write to registers and their fields, having all low-level bit-shifting and masking operations done by the wrapper. Before setting values, raw or high-level, they are automatically converted to the proper data type (depending on the register's signedness and its width), by rounding the value and clamping it to the limits of the underlying data type, avoiding accidental overflows. For blocks of memory, the wrapper offers an array-like interface, maintaining the possibility of accessing the raw pointer to the data.

Another interesting functionality is a proxy object, which can be obtained from an object representing a register. The proxy, offering an identical interface to its parent, caches all write operations to the register itself or its children (fields) and can prepare a register to be written with a single driver call. A proxy object can be initialized with a value from the hardware and then all its children can be accessed without the

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA115

need to read the value from the hardware again, preventing mismatches in the readout.

FESA

Based on a complete memory map, some parts of a FESA class, software that runs on a front-end computer, can be automatically generated. The main file describing a FESA class is the FESA design document. This XML document defines the real time behaviour, the data store and the operational interface provided by the software to the clients.

One of the biggest parts of the design is the definition of data store fields:

- configuration fields — read only fields,
- setting fields — data to be written to the hardware,
- acquisition fields — values read from the hardware.

This tool generates FESA fields based on memory map nodes and their attributes, such as access mode or persistence. The name of a field is constructed using the names of its parents, joining them with underscores. In the case of setting fields, it is the high level value that is being set, but, as shown by experience, sometimes it might be useful to set a low level value as well. To achieve that, two additional fields per field concerned are created:

- overwrite value,
- overwrite mask, or overwrite enable in case of memory nodes with conversion factors. If this field is set (or is different from zero), an alarm is raised.

For acquisition fields, an additional field is created for memory nodes with conversion factors, containing a raw, low level value.

Another part of the generated FESA design are properties, which represent the user interface of a FESA class. Inside the memory map file, a user can define a property which references existing memory map nodes explicitly or using a regular expression. If needed, a specialized filter can be applied. The filter is a body of Python's lambda expression, which will be evaluated during the generation. Inside that filter, PyCheb's interface can be used freely, so a user can, for example, create a property containing all readable memory nodes.

During the development process, it often happens that the memory layout has been changed and some changes have to be backported. Usually this requires regeneration of the design file and then merging of the generated version with the old one, but retaining changes introduced by the developer. To solve this issue, parts of the FESA design, such as custom types, properties of device data fields can be placed in the memory map file, allowing us to generate the complete design file.

In addition to the design file, C++ code is being generated. For memory nodes that are multiplexed based on a cycle (beam purpose and client) currently being played in the accelerator complex, two RT (Real Time) actions are created:

- read action, which reads data from the hardware and sets it in the data store,
- write action, which writes the data from the data store to the hardware.

These RT actions extensively use the Driver Wrapper inter-face. Its proxy mechanism is especially useful, since a single register can have several corresponding fields in the data store. If the generation tool detects the presence of certain registers inside the memory map, it creates several proper-ties and an RT action needed for handling the acquisition buffers, using a standardized approach to read out data from the hardware developed in the RF group. In case a FESA class is restarted (for example when the front-end computer is rebooted), the persistent data store needs to be written to the hardware. The code responsible for this functionality is generated as well.

CONCLUSION

Opting for the extensive usage of Python has proven to be the right design choice, due to the rapid development process and the possibility of combining all existing and new tools in a single entry point, the memory map editor. As the com-plete toolset has been used by the software developers, new requirements and feature requests emerged. Many of those caused modification of the memory map schema, which, thanks to the highly configurable GUI, have been imple-mented in an efficient manner. Memory map attributes often require special validation rules, which now can be written in Python. The code generators, with the logic separated from the view using a template engine, have been extremely easy to customize. Since the software and hardware developers at CERN often work on different platforms, the editor and the code generators are easily accessible for both Linux and Windows users.

REFERENCES

[1] A. Rey et al., "Cheburashka: a tool for consistent memory map configuration across hardware and software", *in Proc. 17th Int. Conf. on Accelerator and Large Experimental Physics Control Systems* (ICALEPCS'13), San Francisco, CA, USA, Oct. 2013, paper TUPPC116, pp. 848–851

[2] M. Arruat et al., "Front-End Software Architecture", ICALEPCS07, Knoxville, Tennessee, USA, 2007, https://accelconf.web.cern.ch/accelconf/ica07/PAPERS/WOPA04.PDF

[3] YAML, https://yaml.org

[4] Cheby, https://gitlab.cern.ch/cohtdrivers/cheby

[5] pybind11, https://github.com/pybind/pybind11

[6] Qt, https://www.qt.io

[7] yaml-cpp, https://github.com/jbeder/yaml-cpp

[8] breakpad, https://chromium.googlesource.com/breakpad/breakpad

[9] Jinja2, https://palletsprojects.com/p/jinja