# System tests and debugging with Python

J. Brooke, R. Frazier, G. Heath, B. Huckvale, D. Newbold
H. H. Wills Physics Laboratory, University of Bristol, United Kingdom
jim.brooke@cern.ch

## *Abstract*

The methodology used to produce the CMS Global Calorimeter Trigger (GCT) control and test software is described. An interpreted interface to an object-oriented model of the system has allowed the hardware to be controlled and tested in an intuitive interactive environment. This benefits both firmware and software designers, in terms of productivity, development cycle time, and firmware/software integration.

## I. INTRODUCTION

Testing, debugging, integration and commissioning of complex custom hardware demands similarly complex software. One approach employed in high energy physics has been to produce an ad-hoc set of stand-alone executables that allow the operator to perform a pre-defined set of tests on the hardware. Other approaches employ more sophisticated single executables that can carry out a wide variety of tests and procedures, driven by a text or graphical menu system. We describe an alternative approach to software for tests and debugging, that has a variety of benefits over these methods.

Our approach is based around an object model of the system hardware and firmware, known as the *GCT Driver*, written in C++ and compiled for performance. It will become the API for final system software; this offers the advantage that the Driver will be well tested when the final software is required. Using an object model of the system provides excellent flexibility for adding functionality that may not be foreseen from the outset, and allows the Driver to be easily understood.

What distinguishes this approach from standard OO design practice, is that we produce bindings for the Driver such that it can be used within an interpreted language. This allows the Driver to be used in an interactive environment, and through scripts. Interactive use is ideal for debugging, where the operator can choose the exact sequence of operations as they are performed. The capability to run scripts is useful when particular sequences of operations are frequently used, but may require regular revision, or are not considered ready for inclusion in the Driver for other reasons. Obviously, the use of an interpreted language speeds up the development cycle, as the compilation stage is not required. In our experience with this design, the integration of software with firmware followed a cycle where initial tests were carried out interactively, then encoded in scripts. These would then undergo some cycles of modification and testing, before a mature version is included in the compiled Driver.

## II. SOFTWARE TOOLS

The GCT Driver is written in C++, which offers both the object-oriented environment we require, and the capability to produce very fast byte code when required. The Driver object model includes the GCT specific hardware. We use the HAL (Hardware Access Library) [1], another object-oriented class library written in C++, to model and access the VMEbus and PCI-VME bridge used to control the GCT.

### *A. Python*

"Python is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days." - http://www.python.org

Python [2] is used here to provide an interpreted object-oriented interface to the Driver. Being easily learnt offers the advantage that tests and debugging can be performed by anyone with some knowledge of the hardware system, and do not require detailed knowledge of the Driver software.

### *B. SWIG*

"SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages." - http://www.swig.org

SWIG [3] is used to automatically produce the python library that provides the interface to the Driver. SWIG takes as input one additional (trivial) header file per class. It is installed easily, hence producing the python interface to the Driver requires almost no additional effort.

## III. CMS GLOBAL CALORIMETER TRIGGER SOFTWARE

The CMS Global Calorimeter Trigger comprises several different modules, populated with FPGAs [4], [5]. The GCT Driver design is shown in a pseudo-UML diagram in Figure 1 (which shows the inheritance structure on the same diagram as class associations). The system is accessed via a singleton system object (an instance of GlobalCaloTrigger), which contains STL vectors of objects that represent the cards in the system. Each card object contains STL vectors of FPGA objects that represent the FPGAs on the card. Functions common to all FPGAs, including register read and write, are implemented in the FPGA base class. Classes specific to particular FPGAs in the system inherit from the FPGA base class, and provide specialised functions. Finally, functions associated with common firmware blocks are encapsulated in classes named after the firmware block (eg. LUT, Buffer) and are members of FPGA sub-classes where required.

Each class provides two types of method. First, accessor functions are provided that return member components. This allows the user to trivially navigate the system in an intuitive way. This is of particular importance during interactive use via python. The second type of method invokes some operation on the hardware. These are built up from the read and write methods provided by the FPGA class. Sub-classes of FPGA provide specific sequences of reads and writes, while the 'card' and 'system' classes provide methods that repeat a specific operation on multiple FPGAs or cards.

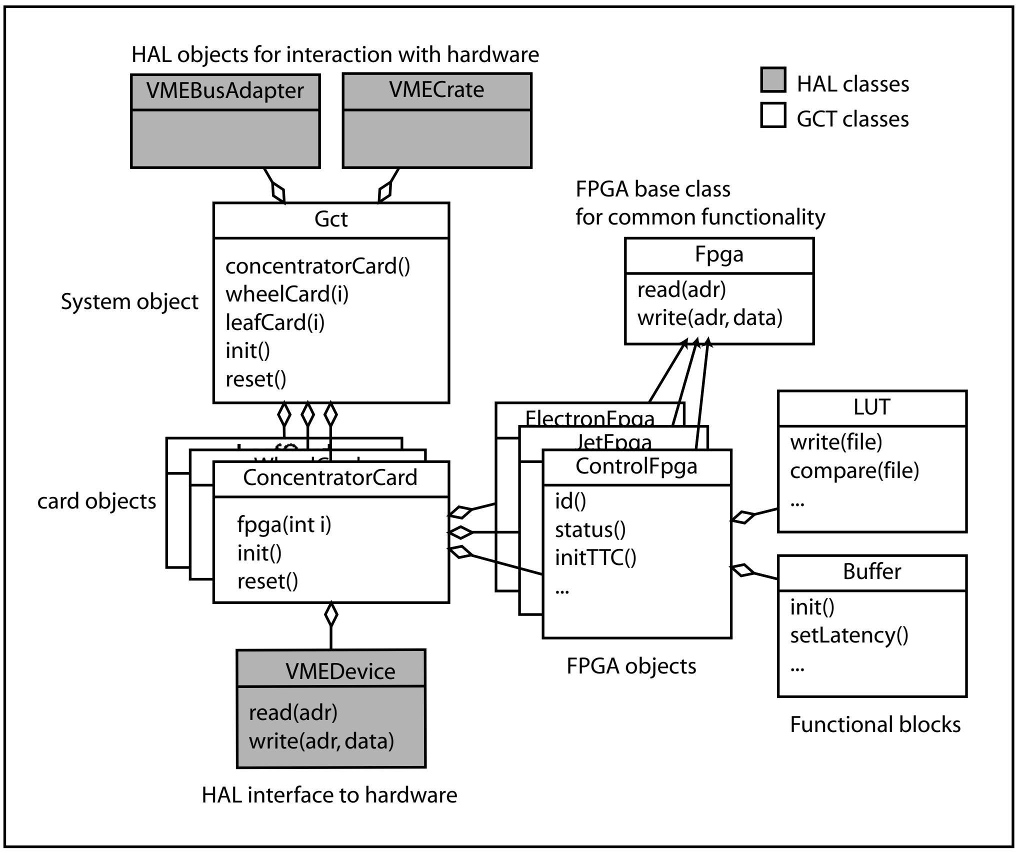

Figure 1: Pseudo-UML diagram of the GCT hardware driver

## A. *GCT software in use*

At the time of writing, the GCT hardware is under production. However, this framework was used extensively in testing and debugging a previous version of the GCT hardware. The system started with nothing more than register read/write functionality implemented in the C++ model. This allowed the user to test the software/hardware combination via the python shell, then set up a more complex procedure in a script to test some more complex function. Once this function had become established, it would be migrated to the C++ model. This migration is a simple procedure, since the python script and the new C++ method make use of the same class library, and hence class, object and method names. A new round of testing would follow this, building increasingly complex functionality on top of code that is well integrated with the hardware. It should also be noted that the python language was simple enough that anyone familiar with the system could write complex test scripts; no prior software knowledge was required, or knowledge of the internal implementation of the GCT Driver or the HAL.

## REFERENCES

[1] C. Schwick, *http://cmsdoc.cern.ch/ cschwick/software/documentation/HAL*

[2] G. van Rossum, *"Scripting Languages: Automating the Web", World Wide Web Journal* **Vol 2 Issue 2** (1997)

[3] D. M. Beazley, *"Perl Extension Building with SWIG", O'Reilly Perl Conference* **2** (1998)

[4] M. Stettler *et. al.*, these proceedings.

[5] G. Iles *et al.*, these proceedings.