Establishing an RCF connection between experiment software and wafer control units

Paul Meehan

December 7, 2018

Contents

1 Introduction

The BrainScaleS neuromorphic hardware system [1] is a novel computing system that is used to physically emulate spiking neurons in order to simulate neural networks. Experiments are performed on it by a multitude of people with varying technical knowledge of the system. As a result, errors in the execution of the program are not necessarily understood by the experimenter, but need to be analyzed and/or fixed by one of the people who designed the system. The main goal of this project was to implement a check that would read the power state of the hardware that is required for an experiment, and throw an exception in case it is not powered on. Previously, this error would not be caught and cause errors later in the program, which were clear to a seasoned user of the software but cryptic to anyone else. To achieve this, communication with the raspberry pi controlling the wafer had to be established via the RCF framework. For ease of use and future uses, a handle and backend were implemented in the halbe framework. These include all the functions provided by the server side of the raspberry pi defined in `rcf_definitions.h`, as well as functions for specifically checking and potentially fixing the power state of HICANNs and FPGAs. Since RCF supports serialization via boost as well as SF, which are mutually only somewhat compatible, care had to be taken to link the correct RCF libraries to each peace of code, and to add compatibility hooks to code which had access to both libraries.

2 Hardware

2.1 HICANN

The HICANN (High Input Count Analog Neural Network) chip is the central building block on the wafer. It contains the neuron and synapse arrays, as well as a bit of communication logic. One wafer contains 384 HICANNs, split into 48 reticles with 8 HICANNs each. HICANNs are not individually powerable, but instead are powered in groups of 8 on their reticle.

2.2 FPGA

FPGAs (Field-Programmable Gate Array) can be used for a multitude of different things. They are a special kind of integrated circuit programmable after manufacturing. In this project, FPGAs are used to for communication between reticles/HICANNs on the wafer and other wafers or host computers. One wafer has 48 FPGAs, each of which controls a section on the wafer containing 8 HICANNs.

2.3 Reticle

A reticle is a collection of 8 HICANNs which are controlled by the same FPGA. A reticle is the smallest unit (concerning HICANNs) that can be powered on or off.

3 Software frameworks

3.1 halbe

The halbe framework [2] provides an abstraction layer between higher level software and hardware. In halbe, the different hardware devices such as FPGAs or HICANNs are defined through handles. Handles are defined for the different platforms the software may be run on, i.e. real hardware, ESS and for data dumping. In this project, a handle for the RCF-Communication between sthal and the wafer-controlling raspberry pi (also referred to as "MaCU", Main Control Unit) had to be written.

3.2 sthal

The sthal framework [3] (STateful HAL) provides higher level access to the hardware. Unlike halbe, sthal is aware of the experiment going on (it is aware of the state).

4 RCF-Communication

4.1 Basics of RCF-Communication

The Remote Call Framework [4] supplies a framework which connects two (or more) pieces of software via network. This is accomplished by client and server sharing a common definitions file containing the name(s) of interfaces (an interface being a collection of functions), as well as the names and parameters of the functions contained by the interface. After initializing the connection with a few commands, the functions can simply be called on the client side. The server side then executes the command and sends the return value back to the client. To be able to send any kind of data over a network, this data must first be serialized, i.e. converted into a byte-stream, which is later decoded on the other side of the network connection. For serialisation, several protocols exist; RCF allows the use of the boost-supplied protocol as well as its own protocol, SF. However, the use of both protocols in one project may lead to problems, as explained in section 5.3.

4.2 The sw-macu interface

For this project, only a client side interface needed to be written, as a server side interface had already been built for manual control of the wafer. From this project, every remote function needed to be incorporated into the halbe framework - this was done automatically as explained further down. `rcf_definitions.h` is the file that supplies all the RCF function definitions. To make this file compatible with hal, dummy functions for boost serialisation needed to be added. These dummies satisfy the compiler as serialisation functions are expected to be compatible with boost as well as SF, but should they ever be called, they will throw an error.
For every wafer, there is one MaCU running, which has access to several functions such as setting the fan speed, measuring the air temperature or getting the power state of various components. There are two interfaces on each MaCU: a "system control" interface

that has access to wafer-level functions like setting the fan speed, and a "reticle control" interface that has functions relating to directly controlling the reticle, like setting the power level of an FPGA and its reticle. The functions which are interesting for this project are `getFcpsOn`, `getReticlesOn` as well as `setFpgaPowerwRet`. The former two get lists containing all FPGAs or reticles that are powered on for one specific wafer, while the latter will actually turn on (or off) an FPGA including its reticle.

4.3 Handle and Backend

As according to the structure in halbe, this project was split into a handle and a backend, the handle being variable depending on the type of system the experiment is run on (hardware or ESS), and the backend containing the functions which take the handle as a parameter. The handle is split into a base struct, which is extended into a hardware, ESS and dump struct (the latter two being dummies). All functions defined by the MaCU interface are defined as member functions of the handle, potentially converting the output to `std::vector` from the custom data types used in serialisation. An RCF connection is only established within the hardware class, which contains RCF clients to both the system control and the reticle control interface (using a PIMPL, a Pointer to IMPLementation due to linking issues). The backend will call the member functions of the handle. For this project, two backend files were created: `RCFWaferControllerBackend.cpp` and `RCFWaferControllerBackendElements.cpp`, the former containing the code to check the power status and throw and exception or fix the problem, the latter containing the backend implementation of all the RCF functions defined on the MaCU. Both of these are further described in the following sections.

4.4 Generating backend files

Since the hooks for the functions defined in `rcf_definitions.h` are rather simple, a short script (`generate_backend.py`) was written to automate the generation of the files containing the elementary functions. It parses `rcf_definitions.h`, watching for the `RCF_METHOD_` tags, which identify the functions defined by the RCF-interface. These function names are then glued together with their return type and parameters, the latter either being read either from parameters.pickle or from an interactive prompt, which is then later saved (to `parameters.pickle`). The main reason of existence of `parameters.pickle` is so that the function parameters do not need to be interactively entered every time the files are generated, which makes developing `generate_backend.py` a lot easier. Assuming a stable `generate_backend.py`, this only needs to be done if `rcf_definitions.h` substantially changes, in which case parameters probably need to be entered as well. Additionally to just parsing the parameters, `generate_backend.py` also changes the return types of the structs `rcfVector_t`, `rcfFloat_t` and `rcfString_t`, which are used for transport, to `std::vector<int>`, `std::vector<float>` and `std::vector<std::string>`, respectively. This completely removes the need to deal with type conversions and gives the user the expected data structure as output. Because some functions in system control and reticle control have the same name, an

additional namespace was introduced (SystemControl and ReticleControl) to uniquely identify the functions. This will generate header and cpp files for `RCFWaferController`, `RCFWaferControllerHw` as well as `RCFWaferControllerBackendElements`.

4.5 Power Status Functions

With the baseline RCF halbe implementation established, two additional functions were written in halbe - `checkPowerStatus` and `fixPowerStatus`. While the former would just compare the FPGAs and HICANNs required to run an experiment with the hardware that was actually running, the latter had the functionality to fix problems by powering on any FPGAs or reticles that are required for the experiment but not powered on. Since `sw-macu` directly supplies functions returning all powered reticles and FPGAs, and sthal already has a function giving FPGAs and HICANNs required for the experiment, this part of the project was rather trivial to implement. The only thing that needed to be done here was the comparison of two sets. Since the RCF functions `getFcpsOn` and `getReticlesOn` return vectors of unsigned integers, and the functions supplying the required FPGAs and HICANNs return vectors of coordinates, the latter first has to be mapped to a set of unsigned integers to be able to compare them at all to the vectors received via RCF. This was done with `std::transform`. To discover whether any hardware that should be powered is not powered, the powered hardware is "substracted" from the required hardware using `std::set_difference`. If the resulting set is not empty, that means that there is hardware that is required but not turned on. In this case, an exception is thrown and the user is notified of the required, but not powered hardware. This is done for both FPGAs and HICANNs, meaning that the user will be notified of all required FPGAs as well as reticles that are not powered. Usually, one will only need to know about the FPGAs as turning them on will usually also give power to the reticles required for powering the HICANNs, but in edge cases this might not be the case.
In the same scope, a function to automatically turn on any required FPGAs (including their reticles) was implemented. Similarly to `checkPowerStatus`, `fixPowerStatus` compares the required and the powered FPGAs (but not the HICANNs/reticles). Then, it powers on any FPGAs together with their reticles using `setFpgaPowerwRet`. Should any of these calls be unsuccessful, it will immediately throw an error and notify the user of the FPGA it could not power; unlike `checkPowerStatus`, it will not give a list of all FPGAs for which it failed, but only for the first one. As there is a lot more required for a successful power state fix, this was cut from the final release, although it should be trivial to implement this function alongside the other necessary functions in the future.

4.6 Dynamic MaCU IPs

Since an experiment can be run on any wafer, another functionality needed to be added: the possibility to find the IP-address of the MaCU corresponding to the wafer. Since there was already an implementation of this call in `YAMLHardwareDatabase`, the only thing that had to be done here was allow the Wafer class access to this method. This was

done with a declaration in the base class of the hardware database, which was expanded in the database class which sthal has access to (`MagicHardwareDatabase`) to call the function already defined in `YAMLHardwareDatabase`.

5 Project Integration

5.1 ESS

The ESS (Executable System Specification) is a software system that is able to simulate the BrainScaleS hardware in most aspects. This means that an experiment can be run on either the real hardware or the ESS, meaning that any code that is part of the software project needs to be compatible with real hardware as well as the ESS. As the problem which was solved with this project does not appear within the ESS, only a dummy needed to be designed to pass the ESS tests after compile time.

5.2 Hardware Database

During runtime, the `Wafer` class, which runs the hardware check, is not "aware" of the platform it is being run on; rather, it is given a reference a `HardwareDatabase` instance, which wraps the handles used for different platforms. Therefore, a function to instantiate the handle used for the RCF connection needed to be implemented in the base class of `HardwareDatabase` as well as its children `EssHardwareDatabase`, `MagicHardwareDatabase` and `YAMLHardwareDatabase`. Additionally, since not all of the wafers (currently only wafer 17) support RCF communication via MaCU, an entry for MaCU version was added to the hardware database YAML file, labeled `macuversion`, as well as an entry in the classes handling the YAML database (`hwdb4cpp`, `hwdb4c`, `YAMLHardwareDatabase`).

5.3 Linking Issues

Since some parts of halbe (and other projects as well) use lib-rcf, but using the Boost serializer instead of the SF serializer required by this project, and there are different libraries for the different serializers, special care and configuration needed to be taken to not align the Boost libraries to the SF code or vice versa. In particular, the compile script for lib-rcf had to be changed to allow statically linking the libraries to the different projects. When incorporating the handle/backend into the halbe frame, this problem was solved by creating separate targets in the build process with different dependencies. Since the compiler would still complain about some includes, the RCF connection was hidden behind a PIMPL. In addition to this, the code I wrote had to be specifically excluded from the python wrapping, since this would throw an error due to the libraries not being correctly linked (and linking all of them together was not trivially possible, as mentioned above).

6 Summary

This project established a base for accessing any function on the wafer MaCUs. This was implemented in halbe, but can be accessed from any place in sthal (or any other project) easily, without the need to do anything except include the necessary headers. In addition to this, a simple function which would check the power state of FPGAs and HICANNs which are required for an experiment, and throw an error if any of them are not turned on, was implemented.

7 Outlook

The implementation of the power check function was a good start, but there are a lot more things that can be done starting at this point. The first thing that comes to mind is the complete implementation of a function that powers on the wafer in case it is not powered. However, the MaCU supplies many more possibilities, and many of them can be useful inside the main software stack. In case the MaCU functionality is extended on the side of the server, the new functions can also be easily added. All of this will probably not be implemented any time soon though, as the new experimental MaCU software will only be added to all wafers at some point far in the future.

8 References

[1] HBP Neuromorphic Computing Platform Guidebook, `https://electronicvisions.github.io/hbp-sp9-guidebook/pm/pm_hardware_configuration.html` [Dec 7 2018]

[2] halbe on Github, `https://github.com/electronicvisions/halbe` [Dec 7 2018]

[3] sthal on Github, `https://github.com/electronicvisions/sthal` [Dec 7 2018]

[4] Remote Call Framework documentation, `http://www.deltavsoft.com/doc/index.html` [Dec 7 2018]