FPGA Interface Design Documentation

Federico Vaga

Apr 25, 2020
1 FPGA Device Structure 1.0

The purpose of this document is to help HDL designers in the development of FPGA designs so that the exported interface can be easily used by the corresponding low-level software.

This document is tailored to the needs of the BE-CO-HT section at CERN to handle different FPGA designs on the SVEC, SPEC and similar FMC carrier boards.
The aim of this set of rules is to help the low-level software to perform a sanity check on an FPGA device, to ease debugging and to provide support for low-level software auto-configuration for byte-order and optional components.

In order to make the discussion easier to follow for the reader, we will be referring to an example design through the text, depicted in Fig. 1.1.

Fig. 1.1: A simplified diagram of the SPEC board, with an ADC mezzanine and an FPGA configured with the SPEC-ADC-WRTD design.

The discussion that follows focuses on the coloured tables in the figure, their structure and why they are useful.
1.1 Device Stability

Every design change requires a new release. It is impossible to handle different designs with the same software without being aware of the fact that they are different. It is really important that a device does not change its nature without formalising a new release.

This is the key ingredient for a simpler software administration and design.

**Rule 01.01.00** Any change requires a new release. The release must be versioned using the **semantic versioning** schema.

1.2 Device Grouping

It is important to identify FPGA devices that we want to drive and clearly draw their borders. For example, all our FMC FPGA designs have a clear set of IP-cores which are used to manage carrier components, the mezzanines or provide services to the user. They need to be grouped using a clear and fixed memory map that uniquely identifies the device as such. Otherwise it will be impossible for the supporting software to correctly handle them, especially after a number of versions. Supporting software often comes in the form of a Linux device driver running in an external host or in a dedicated processor inside the same SoC, but it can also be running in bare-metal mode (no OS) in a soft-CPU core elsewhere in the programmable logic.

**Rule 02.01.00** A set of IP-cores or devices which are used together to offer a service must be grouped together in a device.

**Observation 02.01.01** A device could be composed of a single IP-core.

**Observation 02.01.02** A device can be made of other devices.

**Observation 02.01.03** The offsets of internal components (IP-cores, devices) are fixed. Within the same release they do not change.

**Rule 02.02.00** The device must have all **metadata** registers described in this document.

**Observation 02.02.01** Typical FPGA designs on FMC carriers have a common part which is application-agnostic and an application-specific part. Ideally, there will be one FPGA device in the design for the common part and one or more devices for the application-specific part. In the example in Fig. 1.1 we have two devices: one which takes care of the basic functionality of the SPEC board (e.g. I2C bus to identify the FMC, temperature, interrupt management . . . ) and one (the top level) specifically designed to turn the combination of SPEC and ADC mezzanine into a digitiser.

**Rule 02.03.00** Any change to the device behaviour, its interface or its composition must be done in a new device release.
1.3 Device Metadata

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size (bit)</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>32</td>
<td>Vendor ID</td>
</tr>
<tr>
<td>0x00000004</td>
<td>32</td>
<td>Device ID</td>
</tr>
<tr>
<td>0x00000008</td>
<td>32</td>
<td>Version</td>
</tr>
<tr>
<td>0x0000000C</td>
<td>32</td>
<td>Byte Order Mark</td>
</tr>
<tr>
<td>0x00000010</td>
<td>128</td>
<td>Source ID</td>
</tr>
<tr>
<td>0x00000020</td>
<td>32</td>
<td>Capability Mask</td>
</tr>
<tr>
<td>0x00000030</td>
<td>128</td>
<td>Vendor UUID</td>
</tr>
</tbody>
</table>

1.3.1 Vendor ID and Device ID

The vendor ID and the device ID are used to identify a device. They will be used by the driver for sanity-checking FPGA designs and ease of debugging.

The device driver at probe() time will read these registers and refuse to drive the device if it does not recognise it.

**Rule 03.01.00** A device must export at offset 0x00 a 32-bit register containing the vendor ID.

**Rule 03.02.00** The vendor ID format must be one of the following:

- 0x0000 followed by a valid 16 bit PCI vendor ID (e.g. ‘0x000010DC’)
- 0x01 followed by a valid 24 bit MA-L vendor ID (e.g. 0x0180D336)
- 0xFF000000 for an auto-generated UUID (RFC4122) that will be accessible at offset 0x30

**Rule 03.03.00** A device can export at offset 0x30 a 128-bit register containing an auto-generated vendor UUID (RFC4122).

**Observation 03.03.01** The vendor UUID (RFC4122) is meaningful only if the vendor ID is 0xFF000000.

**Rule 03.04.00** A device must export at offset 0x04 a 32-bit register containing the device ID.

**Observation 03.04.01** The device ID content is vendor-specific.

**Observation 03.04.02** The device ID is meaningful only in combination with the vendor ID.

**Observation 03.04.03** The vendor must put in place a database of device ID in order to avoid internal conflicts.

**Observation 03.05.01** The vendor ID and device ID registers are used for identification. Today (April 2020), there is no widely-used internal FPGA interconnect which supports auto-discovery, and this document does not aim to fill that gap. Instead, it leaves options open to designers as to how to ensure modularity and software reuse.

**Observation 03.05.02** In the BE-CO-HT section at CERN, we use the Device Metadata blocks in each device and we follow what we affectionately call “The Convention”, which is
nothing else than a tacit agreement on the fact that all FPGA designs contain a “base” device sitting at offset \(0\times0\) in the global memory map of the FPGA. So the interplay between software and gateware at host boot time roughly goes like this:

1. The Linux host enumerates the PCIe bus and finds a SPEC board (PCIe vendor and device id in white in the figure, nothing to do with the Device Metadata which is the main focus of this document).

2. A SPEC driver is loaded because it matches these PCIe vendor and device ids.

3. The SPEC driver assumes The Convention is used for the FPGA design in this board. It checks for a vendor and device id in the Metadata block at offset \(0\times0\). It finds a SPEC BASE device and agrees to manage that device.

4. In the SPEC BASE memory map, known by the SPEC driver, there is a register which points to a possible application-specific device inside the FPGA. The SPEC driver probes that metadata block and finds the vendor and device ids of the WRTD ADC device. It then registers a platform device in the kernel and the kernel finds a matching driver for that device.

Note that the SPEC driver needn’t know anything about the WRTD ADC driver and vice-versa. Also, a similar mechanism could be used by the designers of the WRTD ADC gateware and software support to continue scanning for more blocks, and this would be their own private affair. In this way, we preserve the freedom of gateware and software developers to structure things as they want. Following the Convention is a very lightweight requirement which only applies to the SPEC BASE device and the device it points to through its offset register. Users are then free to continue using the convention in other devices if they deem it useful.

Note also that gateware and software don’t need to follow the same hierarchical structure. In our example gateware design, the WRTD ADC device is the top-level, and that device contains a SPEC BASE device. So far as the software is concerned, the two drivers are at the same hierarchical level. The SPEC BASE device is discovered first and managed by the SPEC driver. Then the WRTD ADC device is discovered and the WRTD ADC driver starts managing it.

**1.3.2 Version**

FPGA-based designs are evolutionary by nature, be it through bug fixes or feature requests. This implies that both FPGA code and drivers must be versioned to enable users to choose versions. The immediate consequence is that both driver and FPGA code must explicitly exhibit a version tag.

The driver can use this information to automatically handle interface differences.

**Rule 04.01.00** A device must export at offset \(0\times08\) a 32-bit register containing the version number according to the semantic versioning schema where:

- bits 31-24: major number
- bits 23-16: minor number
- bits 15-00: patch number
Observation 04.01.01 Any development effort should be versioned according to the release version in which it will be officially included. For example, if the current release is 4.1, and there is an ongoing development that will add new functionality; then, any development version should be distributed as version 4.2.0 without increasing the patch number until it becomes an official release. This obligation does not apply to development of backwards-compatible bug fixes, which would qualify as patch-level upgrades according to semantic versioning.

1.3.3 Byte Order Mark

FPGA devices can be used on different buses, where endianness can be different.

On the software side, drivers, must be able to dynamically identify endianness before driving the device. This is achievable with a BOM (Byte Order Mark) register in each device at a known offset.

Rule 05.01.00 A device must export the BOM register at offset 0x0C and it must contain the constant value 0xFFFExxxxx with little-endian byte order.

Observation 05.01.01 If the software reads the value 0xFFFExxxxx from the BOM register it means that the access is little-endian.

Observation 05.01.02 If the software reads the value 0xxxxxFEFF from the BOM register it means that the access is big-endian.

Rule 05.02.00 The 16 least-significant bits are used to identify the version of this convention. The value for this version (1.0) is: 0x0000.

Note: The value 0xFFFEO000 comes from UTF-32-LE.

1.3.4 Source ID

The source ID field identifies the sources from which the corresponding FPGA bitstream has been synthesised. The content depends on how the source code is managed (e.g. it could the git commit ID of a git repository).

Rule 06.01.00 A device must export at offset 0x10 a 128-bit register containing a source identifier.

Rule 06.02.00 The source identifier must be able to identify the source code from which the FPGA device has been synthesised.

Observation 06.02.01 The source identifier is project-dependent.

Observation 06.02.02 If your source ID is part of the HDL code you have committed in your repository, there is of course a chicken-and-egg problem. As you soon as you commit a new version of the file with your new source ID the commit changes. The solution is to generate a small HDL file in the scripts running your synthesis and P&R,
prior to synthesis. This file is generated each time you synthesise and is out-of-tree so far as the repository is concerned.

### 1.3.5 Capability Mask

The process of generating and deploying a new driver every time there is a tiny change in an FPGA device can be unnecessarily heavy. In order to cater for different variants of the device which do not grant a change in drivers, we introduce the concept of Capability Mask. The device is assumed to be made of subsystems providing basic functionality and a set of optional capabilities, which may or may not be present in a given variant. The driver can then find out at run time if each capability is present and react accordingly.

**Rule 07.01.00** A device must export at offset 0x20 a 32-bit register containing the capability mask.

**Rule 07.02.00** The capability mask is device-specific. Each bit represents the presence of an *optional* IP-core or device: a bit set to 1 means that the corresponding component is present, otherwise it is not. Each bit represents a specific component release with a specific configuration.

**Observation 07.02.01** The capability mask describes *only* optional capabilities. The device must work in any case, even if all optional capabilities are missing.

**Observation 07.02.02** For any component modification it will be necessary to produce a new device release because its behaviour changed (*rule 02.03.00*).

**Observation 07.02.03** Since this field is device-specific, its interpretation must be described in the device manual.

**Rule 07.03.00** All components described by the capability mask are at fixed offsets within the device memory map.

**Observation 07.03.01** A device that does not have optional components must have a capability mask with a value of 0x0.

**Observation 07.03.02** A component offset change is an interface change and for this reason it needs a new release (*rule 02.03.00*).