Self-Describing Bus (SDB) Specification for Logic Cores – Version 1.0

Alessandro Rubini (Consultant for CERN)
Wesley Terpstra (GSI),
Manohar Vanga (CERN, BE/CO/HT)

July 19th 2012

Contents

List of Tables

Bugs

Even though we blessed this version as 1.0, there are still some things that ought to be changed or added:

- We should clarify that in nested buses addresses are relative to the sub-bus;
- The type description should get a more prominent place than the current description in the *product* structure;
- We need a policy for designers adding class etc, and it must be described;
- Wishbone-specific flags deserve a subsection;
- Filesystem-specific flags must be defined and documented.

Versioning

This is version 1.0, and describes version 1 of the data structures. All future **compatible** releases of this specification will be blessed 1.x, for some ever-increasing version of x. A compatible specification is one that explains better some unclear points (see "Bugs" above) or that adds record types that may be identified (and possibly ignored) by previous parsers.

If we ever need to release version 2 (or further) of the SDB structures, the specification version will be raised accordingly. New specifications will continue to describe older versions of the data structures, and parsers running on host computers will be required to still parse the older versions. (No such requirement is there for soft-cores, because the parser code is synthesized together with the data structures).

1 Introduction

This document describes a specification for a series of self description structures that can be used to provide metadata about logic blocks. This metadata

should be provided by the logic cores, like PCI or USB description records, so device drivers and other software can automatically discover the blocks and configure them during runtime.

1.1 History and Rationale

The idea of a self-description for a bus appeared while working on *White Rabbit* and related projects that make massive use of FPGA devices. Separately and concurrently, both the CERN and the GSI working groups identified the need for some way to self-detect the contents of a specific logic device after it is programmed.

We envisioned that if the internal FPGA bus could enumerate its own content, we would get the following advantages:

- Run-time validation of FPGA binary images;
- Easy matching of software and gateware;
- Automatic handling of several binaries with the same software;
- Feasibility of tools similar to `lspci` and `lsusb`;
- Automatic loading of kernel drivers on the host computer;
- Automatic setup of low-level drivers within soft cores;
- Better decoupling of gateware and software development.

As usual in engineering, we wanted a system that was as simple as possible, yet open to future extensions without introducing compatibility issues. While our internal bus is *Wishbone*, we designed the structures to be generic so other bus implementations may use them.

We are aware of the AMBA (PrimeCell) cell-id standard, but we think it is seriously under-designed: the idea is sound, but a single cell-ID field is not enough if we want to make sense of the whole bus. We think current hardware and software resources allow a richer description of logic blocks.

We are also aware of the PCI and USB data structures, but they are unsuitable for an FPGA, either. First of all, they assume devices are enumerated

by other means whereas we need to be able to scan a flat address space; then, their vendor ID space is not wide enough to allow small developers to easily participate.

This specification, thus, uses 64 bits for the vendor ID, to prevent scarcity. The vendor space is split in two parts, and all users are free to bless their own vendor number and start designing using these data structures, provided the most-significant vendor-id bit is set.

We acknowledge the usefulness of a central vendor registry, so the lower half of the vendor-ID space is reserved for numbers that are officially assigned and published. The vendor registry, however, is not part of this specification, which just lists the first few vendor-ID values that have already been used.

All multi-byte values are stored in *big endian* order. We need a well-defined endianness to allow generic scanning of the target bus; we picked bigendian because most embedded devices are big endian and because it is the format usually chosen by existing standards documents.

All data structures are 64 byte in size and they are all similar in their internal layout; the last byte in the 64-byte slot identifies the type of each structure, to allow very simple parsing code and easy extension to new types of structures. The size is a power of two in order to avoid multiplication and division in calculation of sizes, as the driver may reside in a very simple soft-core.

1.2 Scope of This Specification

This specification documents the format used by CERN and GSI. However, everyone is welcome to use the data structures defined in this specification (or customized derivatives) in their own work.

Parts of this document are written in the language of a formal specification because we need clear and sharp rules in order to make different implementations interoperate. We are sticking to those rules in our own implementations, both in gateware and in associated software.

Everybody is free to choose a vendor-ID and start coding right now; we suggest to pick a random 64 bit number and set the most significant bit (as an alternative, you can paste the bit "1" in front of a random 63 bit number).

If you want to implement your own data structures, similar to ours but

different in some way, please change the magic number, to avoid headaches if both bus implementation are instantiated within the same host computer or network.

1.3 The Overall System Structure

The bus described by the structures defined herein is set up as a flat address space. Our initial target is the *Wishbone* bus, as used in our own FPGA projects, but the overall situation is pretty general and can be applied to any bus or even a storage system for quasi-static information in flash memory

To keep variety to a minimum, this standard defines the concept of *product*, which is anything that has been *done*, and thus has its own identifiers, name, version and date. This includes meta-information, for example a record of the final build of the FPGA binary.

Every *product* that lives in some address range is called a *component*; as such it specifies its first and last address, both as 64 bit numbers.

Within the bus memory area, the address space available to bus masters is usually decoded into several blocks using the high address bits, so that the space is divided into several blocks, usually they are sized as powers of two, but this is not mandatory – some designers may use individual address lines to select blocks, to easily get a sparsely populated address space. The designer of the address demultiplexer (the *interconnect* block) is expected to describe the logic blocks living behind the interconnect, as well as the interconnect itself. Thus, the *interconnect* is a *component* (so that it owns an address range), and the associated SDB record is the first one of an array of structures; the other records on the array defines the *components* that are connected downstream of this multiplexer and optionally more abstract *products*.

Some of the blocks within the data space of an *interconnect* component can in turn be bridges to other *interconnect* components. Thus, the *bridge* component states the address where further self-description structures are to be found. This allows nesting at arbitrary levels (too deep nesting is not a good practice, but this specification is not limiting the designer's ingenuity).

Only the bus designer knows where the outer-level data structure is to be found, and such information is expected to be known by the "bus driver" software package. For example, a soft-core scanning its own bus will know

where to start from, because it is part of the same overall design; a PCI driver must know how to access internal bus memory from a PCI memory window, so it can also know where the SDB entry point is stored; an *Etherbone* bus master will comply to its own packet-format standard, so it can as well know where to start enumerating the remote bus from.

We can therefore define the following terms to build the self-description framework:

Product
: Every structure includes *product* fields, i.e. the vendor and device identifiers, version and date, and an UTF-8 name.

Component
: A component is a *product* with an associated address range. Its structure lists the first and last valid addresses within the encompassing address space and includes a *product* structure.

Interconnect
: The *interconnect* is a *component* representing an address demultiplexer. The associated data structure heads an array of *product* descriptions; its specific fields are magic number, bus type, version and the number of structures in the array.

Device
: The *device* component identifies a peripheral block, with its class, ABI version and bus-specific flags.

Bridge
: The *bridge* component marks a memory area leading to a lower-level address demultiplexer (i.e. an *interconnect*). Its data structure declares the address where the self-description for the sub-bus is to be found.

Integration
: The optional *integration* product describes an aggregate bus. It is a *product* record, not a *component*, in that it has no associated address range. This meta-information item can be used by a vendor to describe its particular combination of devices, interconnect, and address layout. For example, if an expansion card uses a number of stock devices combined with a stock interconnect, its driver can nonetheless recognize the aggregate device by the integration record.

Controller

The *controller* is a software abstraction, used in the host computer driving the bus (if any). The controller defines the methods to access its own bus and knows where the outer SDB array is found. There is no controller concept for soft-cores that self-scan their own address space.

1.4 Current Implementations

The self-description mechanisms described here are already successfully used by the Etherbone project, whereas FPGA devices equipped with an Ethernet port allow external hosts to be bus masters in the internal Wishbone bus. The host computer can completely describe and access the remote bus(es) using the structures described here; by identifying each instantiated device it can also drive the remote peripherals without prior knowledge of the specific FPGA binary it is talking to.

The same mechanism is already part of the *White Rabbit PTP Core* and the outer-level FPGA designs that are being used in our synchronized I/O boards.

2 SDB Header Material

This section includes the whole header file that defines the data structures. The header itself is included in the source code that accompanies this specification.

All fields and bits are explained in detail in later sections of this specification, but we prefer to show the meat straight at the beginning, before being lost in acronyms and gory details.

This header uses the Linux kernel coding style (e.g.: no `typedef` is used), but you can write it differently if you prefer – some of us already did – as long as the binary representation of the data matches this one.

```
/*
 * This is version 1.0 of sdb.h, as included the specification v1.0
 */
#ifndef __SDB_H__
#define __SDB_H__
```

```
#include <stdint.h>

/*
 * All structures are 64 bytes long and are expected
 * to live in an array, one for each interconnect.
 * Most fields of the structures are shared among the
 * various types, and most-specific fields are at the
 * beginning (for alignment reasons, and to keep the
 * magic number at the head of the interconnect record
 */

/* Product, 40 bytes at offset 24, 8-byte alignmed
 *
 * device_id is vendor-assigned; version is device-specific,
 * date is hex (e.g 0x20120501), name is UTF-8, blank-filled
 * and not terminated with a 0 byte.
 */
struct sdb_product {
        uint64_t                vendor_id;      /* 0x18..0x1f */
        uint32_t                device_id;      /* 0x20..0x23 */
        uint32_t                version;        /* 0x24..0x27 */
        uint32_t                date;           /* 0x28..0x2b */
        uint8_t                 name[19];       /* 0x2c..0x3e */
        uint8_t                 record_type;    /* 0x3f */
};

/*
 * Component, 56 bytes at offset 8, 8-byte aligned
 *
 * The address range is first to last, inclusive
 * (for example 0x100000 - 0x10ffff)
 */
struct sdb_component {
        uint64_t                addr_first;     /* 0x08..0x0f */
        uint64_t                addr_last;      /* 0x10..0x17 */
        struct sdb_product      product;        /* 0x18..0x3f */
};

/* Type of the SDB record */
enum sdb_record_type {
        sdb_type_interconnect = 0x00,
        sdb_type_device       = 0x01,
        sdb_type_bridge       = 0x02,
        sdb_type_integration  = 0x80,
        sdb_type_empty        = 0xFF,
};

/* Type 0: interconnect (first of the array)
 *
```

```
 * sdb_records is the length of the table including this first
 *  record, version is 1. The bus type is enumerated later.
 */
#define                         SDB_MAGIC       0x5344422d /* "SDB-" */
struct sdb_interconnect {
        uint32_t                sdb_magic;      /* 0x00-0x03 */
        uint16_t                sdb_records;    /* 0x04-0x05 */
        uint8_t                 sdb_version;    /* 0x06 */
        uint8_t                 sdb_bus_type;   /* 0x07 */
        struct sdb_component    sdb_component;  /* 0x08-0x3f */
};

/* Type 1: device
 *
 * class is 0 for "custom device", other values are
 * to be standardized; ABI version is for the driver,
 * bus-specific bits are defined by each bus (see below)
 */
struct sdb_device {
        uint16_t                abi_class;      /* 0x00-0x01 */
        uint8_t                 abi_ver_major;  /* 0x02 */
        uint8_t                 abi_ver_minor;  /* 0x03 */
        uint32_t                bus_specific;   /* 0x04-0x07 */
        struct sdb_component    sdb_component;  /* 0x08-0x3f */
};

/* Type 2: bridge
 *
 * child is the address of the nested SDB table
 */
struct sdb_bridge {
        uint64_t                sdb_child;      /* 0x00-0x07 */
        struct sdb_component    sdb_component;  /* 0x08-0x3f */
};

/* Type 0x80: integration
 *
 * all types with bit 7 set are meta-information, so
 * software can ignore the types it doesn't know. Here we
 * just provide product information for an aggregate device
 */
struct sdb_integration {
        uint8_t                 reserved[24];   /* 0x00-0x17 */
        struct sdb_product      product;        /* 0x08-0x3f */
};

/* Type 0xff: empty
 *
 * this allows keeping empty slots during development,
```

```
 * so they can be filled later with miminal efforts and
 * no misleading description is ever shipped -- hopefully.
 * It can also be used to pad a table to a desired length.
 */
struct sdb_empty {
        uint8_t                 reserved[63];   /* 0x00-0x3e */
        uint8_t                 record_type;    /* 0x3f */
};

/* The type of bus, for bus-specific flags (currently only Wishbone) */
enum sdb_bus_type {
        sdb_wishbone = 0x00
};

#define SDB_WB_WIDTH_MASK               0x0f
#define SDB_WB_ACCESS8          0x01
#define SDB_WB_ACCESS16         0x02
#define SDB_WB_ACCESS32         0x04
#define SDB_WB_ACCESS64         0x08

#define SDB_WB_LITTLE_ENDIAN            0x80

#endif /* __SDB_H__ */
```

3 Linux Kernel Model

This sections describes the plans for integration of SDB in the Linux Kernel environment. The uninterested reader can skip over to the next section where we get back to the actual structures.

In our plans this self-description standard is tightly related with Linux device drivers, because most of our FPGA devices are going to be driven by a GNU/Linux host, whether over PCI, VME, or Etherbone.

Devices may appear and disappear during system lifetime: this happens when you load and remove the PCI driver (or instantiate an Etherbone peer), but also when you reprogram the FPGA with a different binary. Another issue is that the device drivers may either be concerned with the FPGA binary as a whole or be interested in each and every individual logic block; thus, the same GPIO logic block can be either driven by the host or ignored by it – because is is directly used by the soft-core within the synthesized binary.

3.1 Wb-core and Enumeration

For the time being, let's call the bus *Wishbone* (this will definitely be the first implementation we are using, and some specifics of byte-wide access will need to be dealt with, so let's ignore generality at this point).

The bus management code will be part of a kernel module called `wb-core`. The core includes bus scanning and enumeration logic, as well as the *match* function that mates devices and drivers, like every other Linux bus is doing.

When some piece of code detects a new bus, it will register the new bus instance, claiming to be the associated controller. In addition, registration specifies the address where SDB records live. The bus and controller can be removed at any time, like you can remove a USB hub or a *Compact-PCI* bridge.

Such bus creation and removal will typically happen when the PCI driver finds its own FPGA carrier board, or when the user tells the system that at some network address the Etherbone protocol is supported.

The wishbone core will start enumerating the bus, using controller-provided operations for the individual I/O transactions; such enumeration begins at the known address the controller declared. The controller implements such transactions in the proper way, by direct reads/writes to PCI or VME memory, or by sending Etherbone frames, or whatever.

The first SDB record must be an *interconnect* record: thus, the first bytes at the SDB address are the magic number. If the magic number is not found, enumeration is aborted. The *interconnect* record is the first in an array of SDB structures, and it declares how long the array is; each device then declares its own address range. The system is designed to never generate invalid memory accesses, but the first address must be externally provided, and the controller is responsible for providing a valid address for its own bus; then, non-SDB buses are handled by simply not finding the magic number (in this case we assume the bus designer willingly placed another value at that address, by knowing the host controller is SDB-aware).

During enumeration, all SDB structures are scanned, and the core will register a device for each and any of those items. If a driver exists for the associated device, its own probe function is called, in the usual way. As an alternative, the driver may appear at a later time, or can be automatically loaded when the device is detected. Again, these operations follow sound

Linux tradition and are well known and safe.

As a special case, the probe function for a Wishbone driver can tell the controller to stop scanning the bus, by returning a specific non-zero value. When this happens, the core will stop enumerating the bus – but the driver itself is allowed to register further devices or ask to scan sub-buses.

This feature is designed to allow multi-device blocks to be handled as a whole. If the FPGA design is a single complex object, with its own CPU inside and internally-driven peripheral, the Linux driver for the associated *interconnect* or *integration* record will get ownership of everything. This prevents generic GPIO or UART drivers to be probed for by the Linux host, while still allowing generic logic blocks to be used in the internal design.

Whenever this logic multi-device complex is embedded in an outer bus, where host-accessible drivers live, such request to stop scanning will not prevent outer devices to have their own Linux driver loaded. Finally, if the driver for the logic complex wants to export some inner block to a host device driver (e.g., an internal GPIO block), it can still register some internal SDB records and keep other ones private, according to internal policies.

3.2 Accessing the Bus

After the controller registered its own self-described bus and `wb-core` is done scanning and probing drivers, applications need a way to access those resources.

Whenever a driver has taken ownership of a device, it also takes care of user access. Whether it registered GPIO pins, a tty device or a network interface, device access is not a problem of `wb-core`.

To allow generic user-space access to the bus, `wb-core` offers a char device interface, compatible with the API already in use within GSI. The char device is not created by default, but a *sysfs* attribute for the bus allows to instantiate it, with a user-provided name. With another *sysfs* attribute, user space can tell the core whether it wants complete control of the bus or only of those devices that are not yet driven, the latter behavior being the default. A `wb-core` system-wide parameter can be used to always create the char device, and even always granting whole-bus access without scanning and registering devices.

Unless it owns the whole bus, the char device will return `EBUSY` for all I/O operations that fall in an address space that belongs to a device driver. This is consistent with the user-space interfaces of *I2C-char* and *libgpio*; it is a good policy to prevent unexpected race conditions or other inconsistencies.

When user-space requests access to the whole bus, this will force unbinding of all the device drivers that are active over the bus. This is a shortcut over individually unbinding all drivers using *sysfs* device attributes. The system will also support a user-access mode by which the whole bus is available to user space without passing through the unbinding phase; this is meant as a debugging tool and must be used with great care.

3.3 Autoprobing Device Drivers

In order to allow automatic loading of device drivers, not unlike what already is in place for PCI, USB and other widespread bus interfaces, we plan to add *modalias* support for wishbone and, later, for other SDB bus versions.

3.4 Storage Support

Another use for SDB is a simple yet effective flash storage organization. SDB records can be used as a very simple file-system-like interface that can be parsed by a soft-core CPU with very little overhead.

Such a file-system is mostly-read and requires no wear-leveling; still, we sometimes need to update FPGA binary images or some calibration parameters. In our search for the state of the art, we didn't find small and simple filesystems that allow in-place replacement of files, without touching nearby files or layout information.

If the idea will fly, we'll define storage as a bus type, where each record describes a file, with both a name and easily-searched device or class identifier. Within SDB, bridge devices lend themselves to be used to represent directories, if needed, without any semantic change. This approach allows serious memory savings in soft-core programs that must both find whether they have a diagnostic channels in the form of a UART and whether they have been provided a configuration file, or other board-dependent parameters.

Another special use of this filesystem is for FMC flash devices: the standard

mandates that the leading part of the flash includes IPMI information for the card; the driver for the carrier board will thus be able to scan the trailing part of the device for the SDB magic and then register a filesystem that only spans the needed part of the flash, as defined at manufacture time (or by device-specific software).

Applications will create the SDB filesytem as an image file, will write them using normal char-device MTD operations and will be able to mount it with as a specific filesystem driver. When mounted, individual files will be replaceable in place using the normal Unix tools and system calls.

4 SDB Structures

This section defines the structures that are to be embedded in the address space of the target bus. The words **shall**, **must**, **should**, **may**, **can** have the usual normative meaning when used in bold face.

4.1 Definitions

SDB Structure
: A 64-byte memory area, located within the bus being described at a known address. The structure **must** bit 64-byte aligned and it **must** be readable with 32-bit I/O transactions. The bus **may** allow 64-bit, 16-bit and 8-bit access to the structure. Code reading the structure **should** use 32-bit transfers, and **can** use different sizes only when aware of the specifics of the bus.

SDB Record
: A synonym for *SDB structure*.

SDB Array or SDB Table
: An in-memory array of SDB records. The records **must** be contiguous with no intervening holes, and the table **must** be aligned at a 64-byte boundary. The first SDB structure in the array **must** be an *interconnect* record (for this reason, you **must** verify the *magic number* of the array before accessing any other location in the array.

Record Type
: The last byte of every SDB structure (offset 0x3f) represents its type.

When reading any SDB structures, unless it is the first in an array, software **must** check the *record type* before making sense of other fields. Designers **may** extend this specification with new record types, and software **must** ignore those structures whose type is not known.

SDB Product
A data structure hosted within some SDB records. All currently defined record types are *products*.

SDB Component
A data structure hosted within some SDB records. A *component* includes a *product* structure and defines an address range.

The following sections define the details of each structure.

4.2 SDB Product Structure

The product is embedded in all currently-defined non-empty data structures. All multi-byte fields **must** be stored in big-endian byte order.

Table 1: SDB Product Structure (40 bytes, at offset 24)

First	Last	Size	Name	Value	Description
0x18	0x1f	8	vendor_id	-	64-bit vendor ID
0x20	0x23	4	device_id	-	32-bit vendor specific device ID
0x24	0x27	4	version	-	Vendor specific device version number
0x28	0x2b	4	date	-	The release date (hex format, eg. 0x20120621)
0x2c	0x3e	19	name	-	UTF-8 device name, 0x20 filled, without terminator
0x3f	0x3f	1	record_type	-	Record type byte (see Table 2)

vendor_id
This field provides a 64 bit field that identifies the vendor of the device. The vendor may be an company, organization or an individual. The vendor name space is split in two halves; anybody **can** pick a vendor ID in the upper half (first bit set), and the 63 bits **must** be picked as

a random number and **should** be used consistently in all the vendor's designs.

A registry is still needed to prevent collisions when using community developed designs from multiple sources, and one should be set up as you read this. Entities that want a more official vendor ID than a random number, **should** apply with the current registry using a number of their choice. Small numbers **should** be avoided, preferring more meaningful strings instead. The registry **should** reject numbers smaller than 12 bits, and **may** reject numbers according to policies other than collisions with other vendors.

device_id
: This field specifies a manufacturer defined device ID for the device being described. Vendors are free to manage these 32 bits as they like, but they **should** use the same identifier for fully compatible implementations, using other fields like *version* and *date* to differentiate them.

version
: This field specifies a manufacturer defined version number for the device. Vendors **can** use the bits as they wish; for example, this **may** be used sequentially or **may** be derived from the information provided by the source code management in use for gateware source code.

date
: Design/release date of the product. This **must** be either 0 (unspecified) or a 32-bit hex format number in the format 0xYYYYMMDD. For example, 0x20120621.

name
: The UTF-8 name of the device. As long as the name fits in 19 bytes, designers are free to choose any string (e.g. both "`UART`" or "`8250-like Serial`" are valid names). The name **should** be a single word or an hyphenated word, avoiding spaces, because it **may** be used by driver software to generate pathnames. The string **must** start at offset 0 and **must** be feature value 0x20 (space) in all trailing bytes. It **must not** have a trailing zero byte.

record_type
: Since the product structure is at the end of the SDB record, it includes the type field. You can access the field from any SDB record, because all records feature the type byte at offset 0x3f. Software **must** verify this field before trying to make sense of any other field in the SDB record.

There is a record type for each kind of SDB record, and the header file gives it a symbolic name through `enum`. The currently defined record types are listed in Table 2. New record types will most likely enter this specification over time, without the need to change the SDB version or overall layout. Users adding new record types **must** choose a yet-unused value with the hight bit clear for *component* records (0-127); users adding new record types of informative value (a *product* or a completely different structure) **must** choose a yet-unused value with the high bit set (128-255). Local or temporary uses **should** fall in the ranges 0x70-0x7f and 0xf0-0xfe. Software **should** report a warning when if finds an unknown record type in the range 0x00-0x7f is found; unknown records in the range 0x80-0xff **can** be ignored silently.

Table 2: SDB Record Types

Name	Value	Description
sdb_type_interconnect	0x00	Interconnect record, first of a table
sdb_type_device	0x01	Device definition
sdb_type_bridge	0x02	Bridge to a sub-bus
	0x70-0x7f	Local/temporary use
sdb_type_integration	0x80	Informative integration structure
	0xf0-0xfef	Local/temporary use
sdb_type_empty	0xFF	Empty record

4.3 SDB Component Structure

The SDB Component is described by a data structure that includes *product* information. It provides information regarding the address space used by the component it describes.

addr_first

The field **must** represent the first byte address that belongs to this component, withing the encompassing bus. If the address bits in the bus are less than 64, the unused most significant bits must be cleared. (e.g.: 0x0000.0000.0400.0000)

Table 3: SDB Component Structure (56 bytes, at offset 8)

First	Last	Size	Name	Value	Description
0x08	0x0f	8	addr_first	-	The first valid address of the component
0x10	0x17	8	addr_last	-	The last valid address of the component
0x18	0x3f	40	product	-	SDB Product structure (see Table 1

addr_last

The field **must** represent the last byte address that belongs to this component, withing the encompassing bus. If the address bits in the bus are less than 64, the unused most significant bits must be cleared. (e.g.: 0x0000.0000.0400.ffff). Thus **must not** represent the first invalid address (e.g.: 0x0000.0000.0401.0000).

product

This is the embedded 40 byte product info structure as described in Table 1.

4.4 SDB Records

This subsection describes the currently defined SDB records that build an SDB array. These structures must be instantiated by designers for each logic block in their design and compiled into a contiguous SDB table, placed at a known address in the bus memory. Most of these structures include a *component* structure or a *product* structure, and the rules for the respective fields apply.

4.4.1 SDB Interconnect

The *interconnect* record describes the overall bus or bus subset. Every SDB table **must** feature such structure as first one in the array.

sdb_magic

The field **must** be set to 0x5344422D. If you use a similar data struc-

Table 4: SDB Interconnect Record (64 bytes, type 0x00)

First	Last	Size	Name	Value	Description
0x00	0x03	4	sdb_magic	0x5344422D	"SDB-", used to verify a table is actually there
0x04	0x05	2	sdb_records	-	Number of records in this SDB table (including this one)
0x06	0x06	1	sdb_version	1	SDB format version. Currently 1
0x07	0x07	1	sdb_bus_type	-	The bus type for all components in the table
0x08	0x3f	56	sdb_component	-	SDB Component structure (see Table 3

ture but choose to not fully comply to this standard, you **must** use a different magic number.

sdb_records
This field specifies the number of records in the table. It **must** include this very record in the count, and the whole address range (this number multplied by 64 bytes) **must** be accessible. Note that the array **may** include empty records at any position.

sdb_version
This is the record format version. In the current version of the specification this is the value 0x1. If software finds an unknown version number it **must** abort enumeration.

sdb_bus_type
This field specifies the bus type. This field is used when decoding the bus specific information inside a device record (see below). All records in the array share the same bus type, bus-specific bits in each device declare the details for data access.

Table 5 lists the currently defined types.

sdb_component
An interconnect record describes a *component*, so it embeds a component structure. The *type* field in the component is 0x00.

Table 5: SDB Bus Types

Name	Value	Description
WishBone	0x00	Specifies a Wishbone bus type, as commonly used in FPGAs
Storage	0x01	Specifies use of SDB records as a simple filesystem

4.4.2 Integration Record

An integration record is a *product* record (not a *component*, because it has no associated address range). The structure provides meta-data about the aggregate product of the bus or bus subset. For example, consider a manufacturer that takes components from various vendors and combines them with a standard bus interconnect. This aggregate product can be described by an SDB integration record, claiming a vendor ID, the release date and the other *product* information. The integrator record is is described in Table 6.

Table 6: SDB Integrator Record (64 bytes, type 0x80)

First	Last	Size	Name	Value	Description
0x00	0x1f	24	reserved	-	Reserved/unused space
0x18	0x3f	40	product	-	SDB Product Info structure

reserved
: The initial field in this record is unused, because all needed information is part of the product structure. Users **should** fill this area with all bits clear or all bit set.

product
: This is the *product* structure described in Table 1. The record type for an integration record is 0x80.

4.4.3 Device Record

This record type describes a single device or logic block mapped into the memory of the bus. In a compliant implementation, one device record **should** exist for each device that is connected to the bus. Users **may** choose to

aggregate a complex device under a single description record. The structure of the device record structure is shown below in Table 7.

Table 7: SDB Device Record (64 bytes, type 0x01)

First	Last	Size	Name	Value	Description
0x00	0x01	2	abi_class	-	The ABI class of the device (0 = Custom Device)
0x02	0x02	1	abi_ver_major	-	The ABI major version
0x03	0x03	1	abi_ver_minor	-	The ABI minor version
0x04	0x07	4	bus_specific	-	Bus specific field (flags)
0x08	0x3f	56	sdb_component	-	SDB Component Info structure

abi_class

The ABI class, if not 0, tells the kind of standard interface that the device provides. This allows a single driver to deal with compatible devices designed by different vendors, not unlikely PCI or USB classes. Currently, no ABI class is defined. Designers **should** use 0 here, at this point in time.

abi_ver_major

This is the major version number of the ABI class. Standard interfaces are not compatible between major version changes. If the class is 0, designers **can** use this field of driver-specific uses. For example, a driver can be able to deal with a number of similar devices (all with a different device-ID) and use the ABI fields as a hint to classify the various devices.

abi_ver_minor

This is the minor version number of the ABI class. Standard interfaces are compatible between minor version changes. Again, if the class is 0, developers **can** set this field for internal use.

bus_specific

This is a 4-byte field that holds bus-specific information, most likely flags. Currently, only Wishbone flags are defined; please refer to header files for details.

component

This is a standard *component* structure (see Table 3). The record type for a device is 0x01.

4.4.4 Bridge Record

A bridge record is used to describe a nested bus within the same address space. This is different from a bus controller which provides access to an entirely different address space altogether. Bus structures with nested interconnects are typical in complex projects. The structure of the bridge record is shown in Table 8.

Table 8: SDB Device Record (64 bytes, type 0x02)

First	Last	Size	Name	Value	Description
0x00	0x07	8	sdb_child	-	The relative address of the nested SDB table
0x08	0x3f	56	sdb_component	-	SDB Component structure

sdb_child
: This field gives the location of the nested bus' SDB table. This address is a relative address with respect to the start of the nested bus' address space (stored in the component info structure). The value **must** point to an SDB array that begins with an *interconnect* record.

component
: An embedded component info structure, where the type is 0x01 See Table 3.

5 Simple Real-World Examples

This section shows the details of the simplest real-world example of an SDB array, and an overlook of a more structured device.

5.1 Simple Binary Data

The FPGA binary used as the simplest example is the *boot* image to be programmed in the SPEC cards (`http://www.ohwr.org/projects/spec`); it only includes the *syscon* device, which allows generic access to the FMC mezzanine card.

The following binary dump appears at offset 0x100 of the memory window that maps to the programmable device:

```
000000 53 44 42 2d 00 02 01 00 00 00 00 00 00 00 00 00  >SDB-............<
000010 00 00 00 00 00 00 01 ff 00 00 00 00 00 00 06 51  >...............Q<
000020 e6 a5 42 c9 00 00 00 02 20 12 05 11 57 42 34 2d  >..B..... ...WB4-<
000030 43 72 6f 73 73 62 61 72 2d 47 53 49 20 20 20 00  >Crossbar-GSI   .<
000040 00 00 01 01 00 00 00 07 00 00 00 00 00 00 00 00  >................<
000050 00 00 00 00 00 00 00 ff 00 00 00 00 00 00 ce 42  >...............B<
000060 ff 07 fc 47 00 00 00 01 20 12 03 05 57 52 2d 50  >...G.... ...WR-P<
000070 65 72 69 70 68 2d 53 79 73 63 6f 6e 20 20 20 01  >eriph-Syscon   .<
```

5.2 Parsing the Data

This is the suggested parsing sequence for the data shown above. The parsing code is assumed to know where the data structure is expected to live.

- The parser verifies the magic number 0x5344422D at offset 0.
- The type byte of 0x00 at offset 0x3f confirms this is an *interconnect* record.
- The SDB version, at offset 6, confirms this is version 1 and we can parse it.
- By reading the 16-bit field at position 04-05, we know this is an array of two items.
- The second item is of type *device* (type 0x01).

What follows is the split-out view of the two structures:

```
Interconnect:
0x00:   53 44 42 2d              (Magic "SDB-")
0x04:   00 02                    (Number of records)
0x06:   01                       (SDB version)
0x07:   00                       (Bus type: wishbone)
0x08:   00 00 00 00 00 00 00 00  (First address)
0x10:   00 00 00 00 00 00 01 ff  (Last address)
0x18:   00 00 00 00 00 00 06 51  (Vendor: GSI)
0x20:   e6 a5 42 c9              (Device)
0x24:   00 00 00 02              (Version)
```

```
0x28:   20 12 05 11               (Date: 11th May 2012)
0x2c:   "WB4-Crossbar-GSI   "     (Name)
0x3f:   00                        (Type: interconnect)

Device:
0x00:   00 00                     (ABI class)
0x02:   01                        (ABI version major)
0x03:   01                        (ABI version minor)
0x04:   00 00 00 07               (Bus-specific: BE, 8,16,32 bits)
0x08:   00 00 00 00 00 00 00 00   (First address)
0x10:   00 00 00 00 00 00 00 ff   (Last address)
0x18:   00 00 00 00 00 00 ce 42   (Vendor: CERN)
0x20:   ff 07 fc 47               (Device)
0x24:   00 00 00 01               (Version)
0x28:   20 12 03 05               (Date: 5th March 2012)
0x2c:   "WR-Periph-Syscon   "     (Name)
0x3f:   01                        (Type: device)
```

The previous dump shows how the vendor identifiers in this case have been allocated in the globally-assigned space, while device identifiers are pseudo-random numbers, in charge of the respective vendor.

5.3 A More Structured Device

The following is the output of `eb-ls`, and *Etherbone* tool, when run over a complex White Rabbit device. This output comes from scanning the SDB structures:

```
root@scul007:~# eb-ls dev/pcie_wb0
BusPath   VendorID         Product   Base(Hex)  Description
1         000000000000ce42:66cfeb52         0  WB4-BlockRAM
2         0000000000000651:eef0b198    100000  WB4-Bridge-GSI
2.1       0000000000000651:35aa6b95    100000  GSI_GPIO_32
2.2       0000000000000651:8752bf44    140000  GSI_ECA_UNIT
2.3       0000000000000651:10051981    180000  GSI_TM_LATCH
3         0000000000000651:eef0b198    200000  WB4-Bridge-GSI
3.1       000000000000ce42:66cfeb52    200000  WB4-BlockRAM
3.2       0000000000000651:eef0b198    220000  WB4-Bridge-GSI
3.2.1     000000000000ce42:ab28633a    220000  WR-Mini-NIC
3.2.2     000000000000ce42:650c2d4f    220100  WR-Endpoint
3.2.3     000000000000ce42:65158dc0    220200  WR-Soft-PLL
3.2.4     000000000000ce42:de0d8ced    220300  WR-PPS-Generator
3.2.5     000000000000ce42:ff07fc47    220400  WR-Periph-Syscon
3.2.6     000000000000ce42:e2d13d04    220500  WR-Periph-UART
```

```
3.2.7     000000000000ce42:779c5443    220600  WR-Periph-1Wire
3.2.8     000000000000ce42:779c5443    220700  WR-Periph-1Wire
```

5.4 Endianness Problems

Please note that the host may have some issues reading the binary dumps. According to how the bridge between the host and FPGA is designed you may face one of the following situations:

- The host is big-endian (data is always correct).
- The host is little-endian and the bridge is byte-oriented.
- The host is little-endian and the bridge is word-oriented.

If the bridge is byte-oriented, i.e. each and every byte can be independently addressed as such, then the usual endian conversion rules apply (e.g. you can *memcpy* the records to host memory and access fields with endian-aware code).

If the bridge is word-oriented, with 32-bit words in this example, the behaviour is stranger, in a way. After you copied the data to host memory (whether one byte at a time or not), you'll find that the bytes are swapped within each word. This happens because the 32-bit word is transferred as a whole: the least significant bits remain the the least significant, but they come from offset 3 in the data structure and are stored at offset 0 in the little-endian host. If this is your case, you need to byte-swap each 32-bit word before using the structure in a little-endian host. After such swapping, the data fields live at the correct offsets and must be accessed as big endian.

5.5 Existing Tools

As part of the *Etherbone* project, you already find a number of tools that work with SDB structures (including `eb-ls` that printed the table of devices shown above). The project is at `http://www.ohwr.org/projects/etherbone-core` .