DiveBits User Manual
Release 1.0

Ruediger Willenberg

May 17, 2022

Contents

1 Introduction **3**

2 Installation **4**

2.1 Prerequisites 4

2.2 Download/Installation 4

3 Quick Start **5**

3.1 Building a demo system for Zedboard or Arty A7-35 5

3.2 Adapting the demo for other targets 7

4 Using DiveBits in your design **9**

4.1 Calling DiveBits 9

4.2 Set path for project-specific DiveBits data 10

4.3 Design Entry with DiveBits IP 10

4.3.1 Adding *divebits_config* 11

4.3.2 Adding data-receiving ***DiveBits*** components 13

4.4 Extraction of DiveBits component data, YAML template file 14

4.5 Bitstream implementation 15

4.6 Bitstream copy and extraction of config BRAM location(s) 15

4.7 Provision/generation of YAML configuration files 16

4.8 Generation of diversified bitstreams 16

5 Overview of data-receiving DiveBits components 17
5.1 Block and RTL components . . . 17
5.1.1 divebits_constant_vector . . . 17
5.1.2 divebits_4_constant_vectors . . . 18
5.1.3 divebits_16_constant_vectors . . . 19
5.1.4 divebits_BlockRAM_init . . . 19
5.2 Block-only components . . . 21
5.2.1 divebits_AXI_4_constant_registers . . . 21
5.2.2 divebits_AXIS_Master . . . 22
5.2.3 divebits_AXI_Master_ReadWrite . . . 23
5.2.4 divebits_AXI_Master_WriteOnly . . . 25

1 Introduction

DiveBits is an open-source framework that enables the fast generation of bitstream copies with individualized (*diversified*) constant data. This could be individual cryptographic keys, network IDs or routing data, or any other data that has to be diversified in functionally identical bitstreams.
This functionality is provided in two parts:

- A set of low-resource footprint IP components for block-based or RTL design entry. These can be added to a design to provide various forms of constant data in the required places.
- An extension of the Xilinx Vivado design flow in the form of a set of Tcl functions. With these functions, copies of the original bitstream with *diversified* constant data can be generated in a matter of seconds. The FPGA designer provides the datasets for the diversified bitstreams in the human-readable YAML data format.

When a DiveBits-enhanced FPGA bitstream is loaded, the diversified data, which is stored in a central BlockRAM location, gets distributed to the required locations over a low-resource serial interconnect (consider it “a bitstream inside a bitstream”). After the data distribution, the main design is released out of reset and begins operation with the diversified data available.

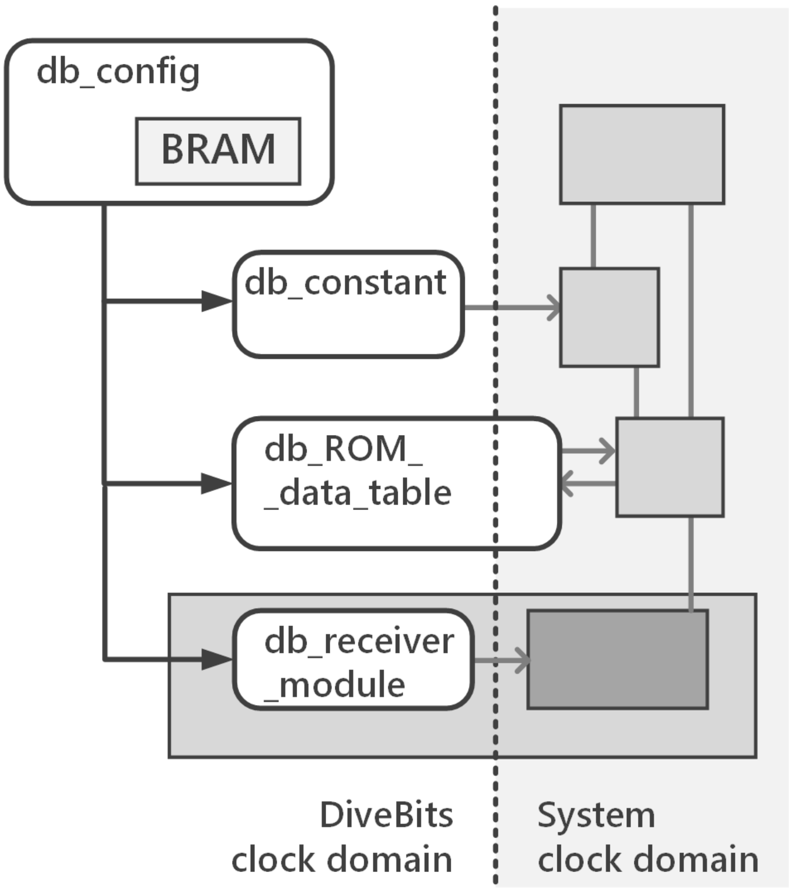


Figure 1: Constant data distribution with DiveBits components

2 Installation

2.1 Prerequisites

DiveBits works with Vivado installations under Windows and Linux. Specifically, it has been tested with

- Vivado 2017.2 (Windows 10 and Ubuntu Linux 18.04 LTS)
- Vivado 2020.1 (Windows 10 and Ubuntu Linux 18.04 LTS)
- Vivado 2021.2 (Windows 10 and Ubuntu Linux 20.04 LTS)

It is likely to work with versions in-between and hopefully even with newer versions, as well as on other Linuxes that run Vivado. Good luck with Windows 11.

Divebits requires **`python`** version 3.6.* or newer, the Python package installer **`pip`** and the python packages **`PyYAML`** and **`bitstring`**. The tools needs to be in the system's **`PATH`** variable.

Under Windows, download and install the most recent Python; **`pip`** is automatically installed. On a command-line, install the extra packages with

```
> pip install pyyaml bitstring
```

Under Ubuntu Linux, install python and pip with

```
> sudo apt-get install python3 python3-pip
```

and the packages with

```
> pip3 install pyyaml bitstring
```

2.2 Download/Installation

DiveBits is provided on ***Github.com*** and can be retrieved and installed in one of two ways:

- Download and extract ZIP file
  `https://github.com/rw-hsma-fpga/divebits/archive/refs/heads/main.zip`
- Clone the git repository with
  `git clone https://github.com/rw-hsma-fpga/divebits.git`

Using **`git`** enables you to upgrade to newer versions with a simple **`git pull`**. Of course, if you want to adapt and modify DiveBits code, you can also fork the repository to your own Github account first.
DiveBits does not have to be installed/unzipped/cloned in a specific system location, it can be started and used from anywhere. We still recommend a location in your work directory, but it does not have to be in relation to a specific Vivado project.

3 Quick Start

3.1 Building a demo system for Zedboard or Arty A7-35

DiveBits comes with am automatically-building design for the ***Digilent Zedboard*** and the ***Digilent Arty A7-35***. See the next subsection to adapt the demo to other targets.
To activate the demonstration flow, execute the Tcl file **$DIVEBITS/divebits_demo_project.tcl**, assuming **$DIVEBITS/** is the directory that you cloned or unzipped to. There are two ways to do this:

- In the Vivado startup menu, select Tools » Run Tcl Script... and select the file *or*
- Type the following into the Tcl console input at the bottom of the Vivado window:
  `> source $DIVEBITS/divebits_demo_project.tcl` (Replace $DIVEBITS)

The script prepares several Tcl functions and lists them in the Console window. You can recall that list anytime by calling

```
> DEMODB_HELP
```

Resize or scroll through the console window to see the complete list of commands.
Make sure that no project is open in Vivado. A fully automatic build can now be achieved by calling either

```
> DEMODB_0_AUTOMATIC_BUILD ZEDBOARD
```

or

```
> DEMODB_0_AUTOMATIC_BUILD ARTY_A7_35
```

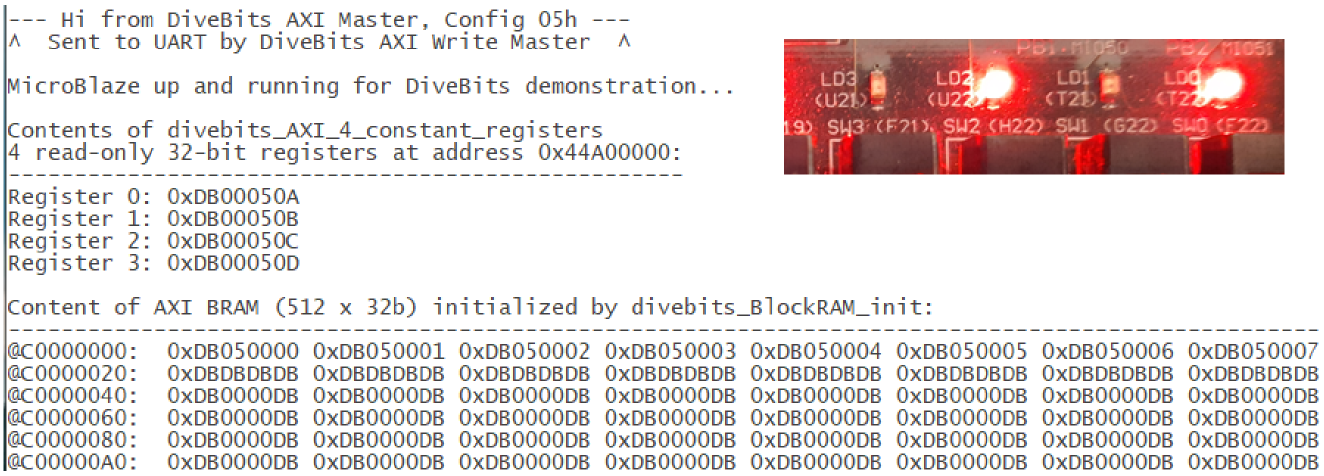
This command will run automatically through all the steps to:

- Activate DiveBits functionality
- Generate a new project
- Build a block design with a Microblaze processor and a selection of DiveBits IP blocks
- Implement the hardware design and build a bitstream
- Extract all the information from the design that is required to provide diversified data through the DiveBits components
- Add pre-compiled Microblaze binary code to the bitstream so that the processor can read data from certain connected DiveBits components and output it via UART.
- Call a Python script that will generate individual DiveBits data for 8 bitstreams in YAML format
- Generate the 8 bitstreams in a few seconds.

By default, the Vivado project will be generated in the directory **$DIVEBITS/DEMO_PROJECT/**. The directory that DiveBits uses to store all its project-specific data is set to **DEMO_PROJECT/DiveBits_data**. The project path can be set to any other location by specifying it as a second parameter after the board parameter when calling `DEMODB_0_AUTOMATIC_BUILD`
The 8 configuration files generated by script are named **config_0.yaml** through **config_7.yaml** and can be found in
DEMO_PROJECT/DiveBits_data/3_bitstream_config_files.
They result in corresponding bitstream files **config_0.bit** through **config_7.bit** in
DEMO_PROJECT/DiveBits_data/7_output_bitstreams
which can be downloaded to the board through the Vivado Hardware manager.
The significant number `0..7` in the filename is used in the DiveBits configuration data and turns up in various places in the bitstream's output:

- `LED[2..0]` and `LED[6..4]` display the number in binary. For control purposes, `LED[7]` is always 1 and `LED[3]` is always 0. The LEDs are controlled by two ***divebits_constant_vector*** components with 4 output bits each.
- A simple ***divebits_AXI_Write_Master*** that can be used as a low-resource CPU replacement writes a short text message containing the config number to the UART.
- The Microblaze processor reads from the AXI Subordinate ***divebits_AXI_4_constant_registers*** and prints the four 32-bit constants provided in this way.
- The Microblaze processor dumps the complete content of a dual-port BlockRAM that was initialized via its second port by a ***divebits_BlockRAM_init*** component.

Figure 2 shows part of the output for the bitstream **config_5.bit**.



```
--- Hi from DiveBits AXI Master, Config 05h ---
^  Sent to UART by DiveBits AXI Write Master  ^

MicroBlaze up and running for DiveBits demonstration...

Contents of divebits_AXI_4_constant_registers
4 read-only 32-bit registers at address 0x44A00000:
--------------------------------------------------
Register 0: 0xDB00050A
Register 1: 0xDB00050B
Register 2: 0xDB00050C
Register 3: 0xDB00050D

Content of AXI BRAM (512 x 32b) initialized by divebits_BlockRAM_init:
------------------------------------------------------------------------------------------------------
@C0000000:  0xDB050000 0xDB050001 0xDB050002 0xDB050003 0xDB050004 0xDB050005 0xDB050006 0xDB050007
@C0000020:  0xDBDBDBDB 0xDBDBDBDB 0xDBDBDBDB 0xDBDBDBDB 0xDBDBDBDB 0xDBDBDBDB 0xDBDBDBDB 0xDBDBDBDB
@C0000040:  0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB
@C0000060:  0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB
@C0000080:  0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB
@C00000A0:  0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB 0xDB0000DB
```

Figure 2: Ausgabe **config_5.bit**

To receive the UART output

- For the Arty Board, connect a terminal at ***115200 Baud*** to the UART available as part of the USB programming connection.

- For the Zedboard, the UART TX output is available at 3.3V LVCMOS level at Pin 2 of its Pmod A connector. The simplest way to access the data is through a PmodUSBUART device plugged into the connector's upper row.

3.2 Adapting the demo for other targets

To use the demo for another target board and FPGA, the demo generation must be executed step by step instead of through the single-command flow, so that the FPGA device and various pin constraints can be modified:

- Call `DEMODB_HELP` in the Tcl console to list all the required commands again.
- Call `DEMODB_1_create_project_and_block_design ZEDBOARD`
  Again a different project path can be specified, too.
- Now that the project has been built, it has to be adjusted to the new target:
  - In the Vivado menu, click Tools > Settings > Project Settings > General and change the setting Project device to the correct FPGA device.
  - Open the constraints file **db_demo_zedboard** and adapt the pin constraints for the clock input, reset button, UART TX and RX(unused) as well as the 8 LEDs. Also adapt the clock period and waveform pulse length if different from 100MHz. Save with Ctrl+S.
  - In the block diagram, adjust the External Port Properties of the `reset_rtl` port if the polarity of the reset button is ***Active Low***. Run Validate (F6) to propagate the change to the **Reset Manager**. Validate informs you that **Clock Wizard** now has a mismatched reset level. Open the **Clock Wizard** settings. Surprisingly, you find the reset polarity setting on the Output Clocks tab.
  - If the input clock is different from 100MHz adjust this, too, in the Clocking Options tab of the **Clock Wizard**.
  - Check your block design again with Validate (F6) and save it with Ctrl+S
- Call `DB_1_component_extraction`, a general DiveBits function that extracts structural data from the block diagram, adjusts block diagram settings and generates some initial files for DiveBits configuration.
- Call `DEMODB_2_build_bitstream`; the name is self-explanatory.
- Call `DEMODB_3_add_microblaze_binary`. A pre-compiled ELF binary for the Microblaze processor is integrated into the code memory locations in the bitstream.
- Call `DB_2_get_memory_data_and_bitstream $ELF_BIT_PATH`. Do not change $ELF_BIT_PATH to something else, it is a Tcl variable that has been set by the previous call. This general DiveBits command extracts the location of the BRAM that is later filled with

DiveBits data and makes a copy of the project's generated bitstream. Since we have already generated another bitstream with the Microblaze binary included, we need to specify its path; otherwise DiveBits would copy the original implementation flow result.

- Call `DB_PT_run_python3_config_generator $DEMOFILES/generate_demo_configs.py -n 8`

  $DEMOFILES is another pre-set Tcl variable, but you change the number of files generated if you want to. Python scripts should only be called from the Vivado Tcl console with the DB command shown, otherwise Vivado will use its own internal copy of Python and things WILL go wrong.
- Finally, `DB_3_generate_bitstreams` will make your bitstreams, which you will find in **DEMO_PROJECT/DiveBits_data/7_output_bitstreams** and can download at will.
- `DEMODB_4_clean_project` will delete the entire demo project folder (CAUTION: That includes your modified target data).

4 Using DiveBits in your design

4.1 Calling DiveBits

To use the DiveBits framework in your Vivado project, call the Tcl script **$DIVEBITS/divebits_tools/divebits.tcl**, assuming **$DIVEBITS/** is the directory that you cloned or unzipped to. There are two ways to do this:

- In the Vivado startup menu, select Tools » Run Tcl Script... and select the file *or*
- Type the following into the Tcl console input at the bottom of the Vivado window:
  `> source $DIVEBITS/divebits_tools/divebits.tcl` (Replace $DIVEBITS)

With the execution of the script, all DiveBits commands required for the extension of the design flow are established as commands, as listed by the console output:

```
*****************************************************************************************
**** DiveBits tools - function overview: ************************************************

 call  DB_set_DiveBits_data_path $DATA_PATH           to set non-standard DB data path

 call  DB_0a_add_block_ip_repo                        to add DiveBits IP repository

 call  DB_0b_add_rtl_ip_repo                          to add DiveBits RTL to project

 call  DB_1a_block_component_extraction               after block design is finished

 call  DB_1b_rtl_component_extraction                 after HDL design is finished

 call  DB_2_get_memory_data_and_bitstream $BIT_PATH   after implementation
          ($BIT_PATH  only needs to specified
           when another tool (e.g. Vitis, SDK) has
           modified the bitstream post-implementation)

 call  DB_PT_run_python3_config_generator $SCRIPT     to make config files

 call  DB_3_generate_bitstreams                       to make diversified bitstreams

 call  DB_HELP                                        for this list
*****************************************************************************************
```

Figure 3 illustrates how these calls integrate into the regular Vivado flow.

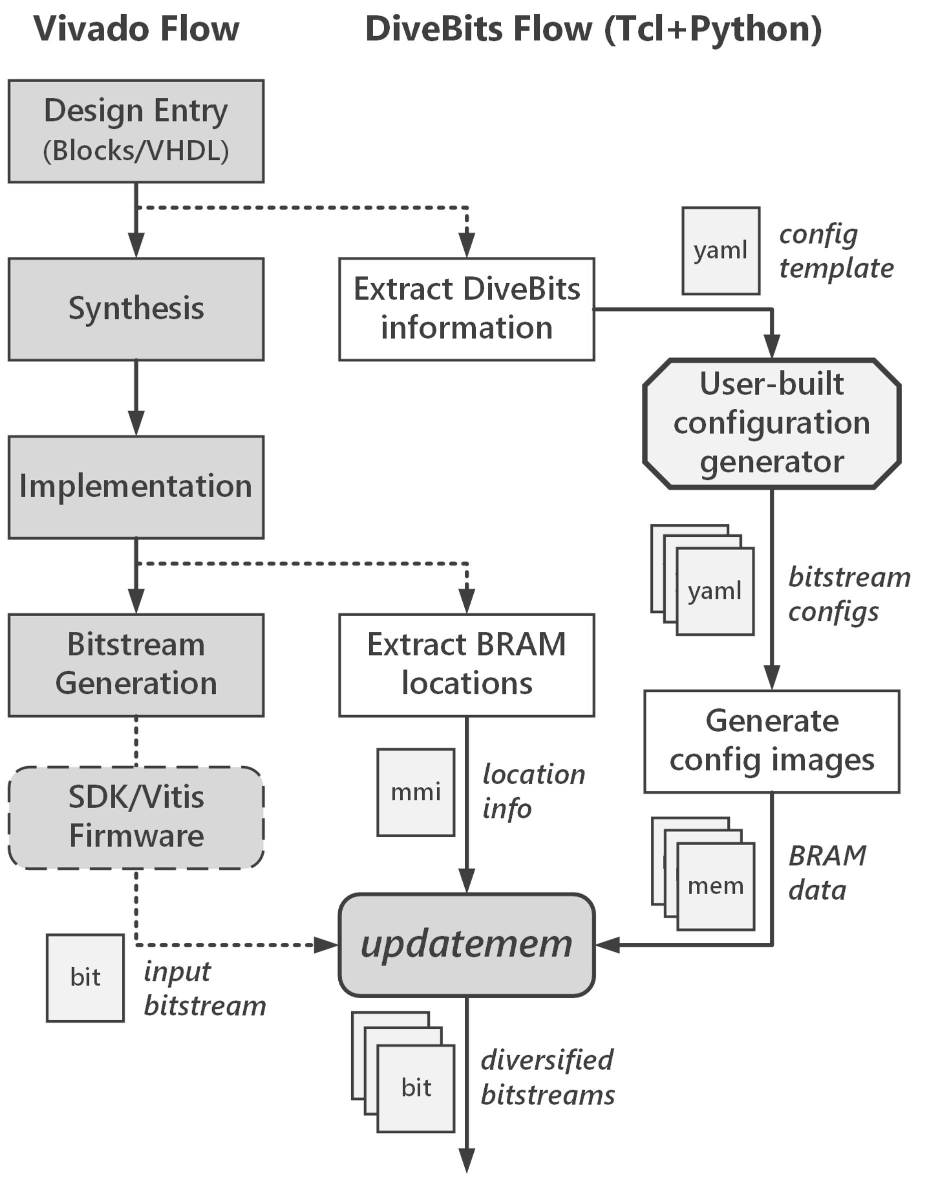


Figure 3: DiveBits-enhanced flow

4.2 Set path for project-specific DiveBits data

By default, when using DiveBits inside a newly created Vivado project, DiveBits will store all its work data in the directory
$PROJECT_FOLDER/DiveBits_data/, i.e. as a subfolder of the Vivado project path. If you'd like the data to be stored further up in the hierarchy (e.g. for versioning purposes) or somewhere else, you can specify that path with

```
> DB_set_DiveBits_data_path $DATA_PATH
```

4.3 Design Entry with DiveBits IP

DiveBits supports two types of design entry:

- ***Block Design*** (preferred): DiveBits provides all its components as block IP for the Vivado IP integrator. This is the recommended way of using DiveBits, as it allows simpler connectivity between DiveBits components and a higher degree of automation of the DiveBits IP flow. To add the complete repository of Block IP components to your project's IP catalog, call `DB_0a_add_block_ip_repo`.

- ***RTL***: DiveBits provides the majority of its components as VHDL entities that can be instantiated in a design's top-level IP file (see below for integrating a DiveBits HDL component lower in the hierarchy). To add the DiveBits VHDL sources to your Vivado project, call `DB_0b_add_rtl_ip_repo`.

4.3.1 Adding *divebits_config*

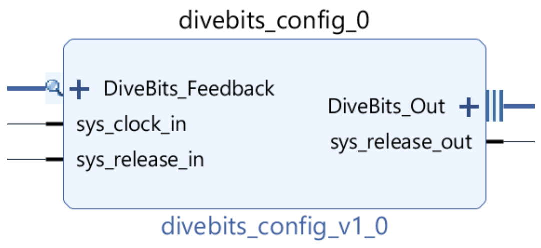


Figure 4: divebits_config

Every DiveBits design, no matter if block design or RTL flow, needs exactly 1 ***divebits_config*** component (see Figure 4). This component holds the BlockRAM that stores an individual bitstream's diversified data and distributes it to the data-receiving DiveBits components. Make sure to add it first to your design. While all configuration parameters for this block/entity are preset with default values, you need to make a few important design considerations:

Release wiring and polarity
DiveBits configuration only operates once at FPGA startup. After the bitstream has been loaded and FPGA operation begins, the ***divebits_config*** component distributes data over the DiveBits serial interconnect. Only after all DiveBits data has been distributed, other components should start operating. For this purpose, ***divebits_config*** provides as signal `sys_release_out` which changes state to release system operation.
divebits_config can start operation automatically after the FPGA begins operation (Global Set/Reset), however in most designs, it will itself wait for a release on its input `sys_release_in`. There are two common usage scenarios:

- `sys_release_in` receives a ***Locked*** signal from a clock manager, which is usually directly connected to a system reset manager. The clock manager indicates with it that the system clock is stable and system operation can safely begin. Instead, ***divebits_config*** now consumes this signal and begins operation, and its `sys_release_out` hands down the locked state to the reset manager after the DiveBits configuration is done.

- `sys_release_in` receives a system ***Reset*** signal and holds the rest of the system in reset with its `sys_release_out` until it is done.

divebits_config holds the `sys_release_out` for a configured number of cycles after completing operation, thereby ensuring that all data-receiving components have completed processing the data, as well as that constant signals have reached a stable state if the DiveBits clock domain is asynchronous to the system clock domain (see below). The block IP option ***Release Delay Cycles*** and the corresponding VHDL generic `DB_RELEASE_DELAY_CYCLES` can be set between `20` and `259` cycles. The default of `20` is considered safe.
The meaning and polarity of ***Release*** is opposite to ***Reset***: When the signal is ACTIVE, the system is released to operate. The default polarity of `ACTIVE_HIGH` (i.e. set for a high-active clock manager locking signal or a low-active reset signal) can be changed in the block parameter ***Release Active High*** or the VHDL generic ***DB_RELEASE_ACTIVE_HIGH***.

DiveBits topology and CRC check

DiveBits components can be connected in two types of topologies:

- In a classic ***bus*** topology, the ***DiveBits_out*** port of ***divebits_config*** is connected to the ***DiveBits_in*** port of all data-receiving components. This is the least resource-intensive topology, however depending on the number of receiving components, a slightly higher fanout results.

- The components can also be connected in a ***daisy-chain*** topology, where each component's ***DiveBits_in*** is connected to another component's ***DiveBits_out***. The data is routed through a receiving shift register in each component. The order of connection of the components does not matter. The last component in the chain needs to connect back to ***divebits_config***'s ***DiveBits_Feedback*** port.

  The main purpose of this topology is that ***divebits_config*** can check the integrity of the data stream after it has passed all components. It computes a ***CRC32*** checksum on the received data and only releases the `sys_release_out` when the checksum matches. Otherwise, it restarts the DiveBits configuration process.

  To active the CRC32 check, select the block IP option ***Daisy chain input with CRC check*** or set the VHDL generic `DB_DAISY_CHAIN_CRC_CHECK` to `true`.

Clocking considerations

All DiveBits component need to be connected to the same clock as the interconnect communication is clock-synchronous.
In the most straightforward usage, the DiveBits components use the same clock as all the design components connected to them. However, complex designs might use multiple independent clocks that could be phase-synchronous or asynchronous.
Furthermore, if timing closure is hard, DiveBits components can be put on a comparably slow clock to ease timing pressure. As they are only communicating hundreds, maybe low thousands, of bits once at FPGA startup, the incurred startup delay is often non-critical.

If the DiveBits components have different clocks than some or all components connected to them, make sure to declare the corresponding clocks asynchronous to each other. The delayed release mechanism will ensure that all output signals are settled and no metastability ensues.

4.3.2 Adding data-receiving DiveBits components

All other DiveBits components can be added and connected as required in the bus or daisy-chain topologies. While the graphical block design only requires a mouse click-and-drag to connect to components, in the RTL flow two separate signals **`db_clock`** and **`db_data`** need to connected between components. Currently, the following components are available:

- ***divebits_constant_vector***: (Block IP and VHDL) Provides a single constant of configurable width (1-64 bits) to other system components.
- ***divebits_4_constant_vectors***: (Block IP and VHDL) Provides four constants of the same configurable width (1-64 bits) to other system components.
- ***divebits_16_constant_vectors***: (Block IP and VHDL) Provides 16 constants of the same configurable width (1-64 bits) to other system components.
- ***divebits_BlockRAM_init***: (Block IP and VHDL) Operates as a master for a native BlockRAM interface and initializes a BlockRAM of arbitrary size and data width through one of its two ports. The other port can therefore be used by the system to retrieve constant ROM data.
- ***divebits_AXI_4_constant_registers***: (Block IP only) Constitutes a memory-mapped AXI Subordinate with 4 read-only 32-bit data registers. Can provide diversified data to a CPU system without re-compiling the software.
- ***divebits_AXIS_Master***: (Block IP only) An AXI Stream master that can write up to 256 data words into an AXI stream. Data width is configurable between 32 and 1024 in power-of-two steps.
- ***divebits_AXI_Master_ReadWrite***: (Block IP only) An AXI master that can replace a CPU for simple configuration tasks. Can execute diversified programs composed of a simple set of opcodes to:
  - Write values to memory-mapped AXI subordinates.
  - Copy values between AXI subordinates.
  - Repeatedly read from an AXI subordinate address (polling) until specific bits are set or cleared. Wait cycles can be configured so the polling doesn't clog the interconnect.
- ***divebits_AXI_Master_WriteOnly***: (Block IP only) A simpler AXI master that can only write to memory-mapped AXI addresses. Can repeat writes on bus errors, so for example a full ***AXI Uartlite*** FIFO can be handled without explicit reads.

Connecting the system-side of these components obviously differs for each component, as does the YAML configuration. See Section 5 for detailed descriptions.
All data-receiving components share a set of configuration parameters/VHDL generics:

- ***DB Address***/`DB_ADDRESS`: The unique DiveBits address of this component in the design. ***divebits_config*** is always address `0`, all other components can be assigned addresses between `1` and `65534/0xFFFE`.

  Addresses don't *have* to be set explicitly in the block design flow, as the next flow step can assign unique addresses automatically. You don't have to know the address to specify diversified data, that is done by block name.

  The RTL flow is not able to do automatic re-assignment of VHDL generics because of Vivado's limitations. You have to assign each DiveBits component a unique `DB_ADDRESS` value, or the DiveBits flow will abort and ask you to fix it.

- ***Daisy-Chaining possible***/`DB_DAISY_CHAIN`: This parameter is somewhat cosmetic as it shuts off the `DiveBits_Out` port when unchecked/set to `false`. It doesn't have any effect in the RTL flow. Since Synthesis will optimize away any unused outputs anyway, no resources are saved by changing the option.

- ***Enable Local Release Signal***/`DB_LOCAL_RELEASE`: In large-scale designs, global synchronous resets might not be feasible. Therefore, each data-receiving component can provide a local reset/release signal named `local_release_out` which releases a defined number of cycles after the component's data was received. The number of wait cycles is specified with ***Local Release Delay Cycles***/`DB_RELEASE_DELAY_CYCLES` and the polarity can be set with ***Local Release High Active***/`DB_RELEASE_HIGH_ACTIVE`.

4.4 Extraction of DiveBits component data, YAML template file

After design entry has finished, call either

```
> DB_1a_block_component_extraction
```

for the block design flow or

```
> DB_1b_rtl_component_extraction
```

for the RTL flow. The call accomplishes the following steps:

- It identifies all DiveBits components added during design entry by type, name, DiveBits address and component-specific parameters.

- If a DB Address has been used twice, the block design flow will automatically re-assign a unique address.

  The RTL flow can't do that, so it aborts with a suggestion to the designer to re-assign addresses and re-execute the command.

- Based on the component parameters, DiveBits calculates how many configuration bits of diversified data need to be stored in BlockRAM and how many BlockRAMs are required. The number of BlockRAMs required is an implementation parameter for ***divebits_config*** and needs to be changed before synthesis if it is different from the default value of `1`.

  For the block design flow, DiveBits can automatically adjust the requisite parameter, `DB_NUM_OF_32K_ROMS`.

  The RTL flow, again, cannot do this. It aborts and informs the user to set the VHDL generic `DB_NUM_OF_32K_ROMS` to the required value and re-execute the command.

- All structural data that is required for later processing is stored in the file **$DIVEBITS_DATA/1_extracted_components/db_components.yaml**

- A template YAML file that illustrates how to specify parameters for the various components is placed in
  $DIVEBITS_DATA/2_config_file_template/db_template.yaml

  This file is also later provided to user-written Python programs and can be used to import a skeleton data structure that can then be populated with diversification data.

4.5 Bitstream implementation

After the data extraction is finished (without RTL flow abort due to address conflict or wrong number of config BlockRAMs), regular time-intensive bitstream generation can be initiated.
If you are using Vitis or another tool to build processor binaries that will be included in BlockRAM locations in the bitstream, please continue with that step. You will be able to specify the binary-augmented bitstream instead of the implementation-result bitstream in the next step.

4.6 Bitstream copy and extraction of config BRAM location(s)

After bitstream implementation, call

```
> DB_2_get_memory_data_and_bitstream $BIT_PATH
```

This call accomplishes two tasks:

- It opens the implementation result to retrieve the location of the BlockRAM(s) inside the ***divebits_config*** component. With that information, an ***MMI*** file is built. The MMI file is required later to insert the configuration bits into the correct bitstream location.

- DiveBits makes a private copy of the bitstream produced by the implementation run. with the optional $BIT_PATH parameter you can specify a different bitstream than the one in the implementation run folder. As mentioned before, the purpose is to specify a bitstream that has been augmented with other data (like software binaries) after implementation.

4.7 Provision/generation of YAML configuration files

Before initiating the generation of diversified bitstreams, the corresponding YAML configuration files need to be provided in
$DIVEBITS_DATA/3_bitstream_config_files/
(one YAML file for each bitstream). The YAML files have to follow the basic structure shown in the generated template file and provide component-specific data structures as detailed in Section 5.
How the files are provided and generated is up to the FPGA designer, but the collection of DiveBits Tcl commands include one to call a Python script from the console:

```
> DB_PT_run_python3_config_generator $SCRIPT
```

CAUTION: This wrapper command is the only way that Python scripts should be called from the Vivado Tcl console, as it bypasses the inadequate Python installation that Vivado uses internally and instead calls the system-installed Python 3 with the added ***PyYAML*** and ***bitstring*** packages.
Furthermore, it automatically adds the two call parameters

```
-t $TEMPLATE_FILE_DIR -c $BITSTREAM_CONFIG_DIR
```

that tell the script where to find the template and where to put the configuration files.

4.8 Generation of diversified bitstreams

Based on the provided or generated configuration files, diversified bitstreams can now be generated by calling

```
> DB_3_generate_bitstreams
```

The resulting files will be placed in
$DIVEBITS_DATA/7_output_bitstreams/
and can be used for download to the target(s).

5 Overview of data-receiving DiveBits components

In this section, you find all data-receiving DiveBits components with their function-specific parameters/generics, ports and YAML configuration structures.

5.1 Block and RTL components

5.1.1 divebits_constant_vector

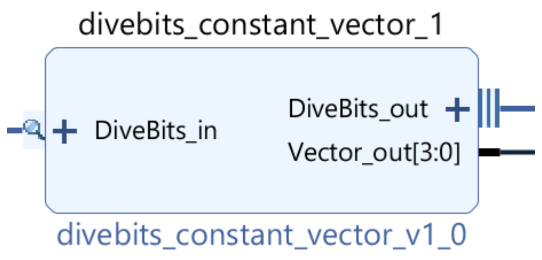


Figure 5: divebits_constant_vector

Provides a single constant bit vector of configurable length.

Parameters/Generics

- ***Vector Width***/`DB_VECTOR_WIDTH` specifies the width of the vector between 1 and 64 bit.
- ***Default Value***/`DB_DEFAULT_VALUE` specifies a default value shown if the component is not reconfigured. As this is of type integer, only 31-bit wide positive values can be specified at maximum.

Signals

- `Vector_Out` is the configurable-width constant vector output

YAML configuration

```
- BLOCK_PATH: /divebits_constant_vector_1
  CONFIGURABLE:
    VALUE: 0x42
```

A decimal or hexadecimal value can be specified.

5.1.2 divebits_4_constant_vectors

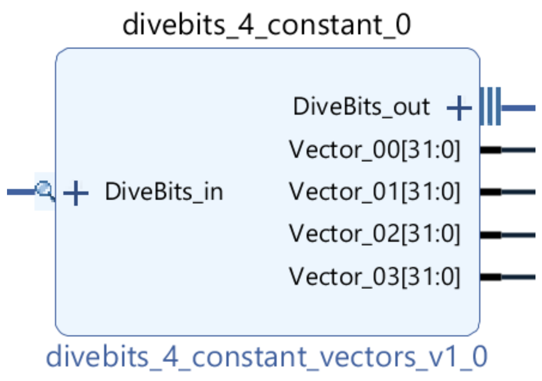


Figure 6: divebits_4_constant_vectors

Provides 4 constant bit vectors of (identical) configurable length.

Parameters/Generics

- ***Vector Width***/`DB_VECTOR_WIDTH` specifies the width of the vectors between 1 and 64 bit.
- ***Default Value All***/`DB_DEFAULT_VALUE_ALL` specifies a default values shown by all vectors if the component is not reconfigured. As this is of type integer, only 31-bit wide positive values can be specified at maximum.
- ***Default Value 00..03***/`DB_DEFAULT_VALUE_[00..03]` specify default values for each vector shown if the component is not reconfigured. The same integer limitation applies

Signals

- `Vector_[00..03]` are the configurable-width constant vector outputs

YAML configuration

```
- BLOCK_PATH: /divebits_4_constant_0
  CONFIGURABLE:
    VALUE_00: 0xDEADBEEF
    VALUE_01: 0x0815
    VALUE_02: 0
    VALUE_03: 4711
```

Decimal or hexadecimal values can be specified.

5.1.3 divebits_16_constant_vectors

Provides 16 constant bit vectors of (identical) configurable length.

Parameters/Generics

- ***Vector Width***/`DB_VECTOR_WIDTH` specifies the width of the vectors between 1 and 64 bit.
- ***Default Value All***/`DB_DEFAULT_VALUE_ALL` specifies a default values shown by all vectors if the component is not reconfigured. As this is of type integer, only 31-bit wide positive values can be specified at maximum.
- ***Default Value 00..15***/`DB_DEFAULT_VALUE_[00..15]` specify default values for each vector shown if the component is not reconfigured. The same integer limitation applies

Signals

- `Vector_[00..15]` are the configurable-width constant vector outputs

YAML configuration

```
- BLOCK_PATH: /divebits_4_constant_0
  CONFIGURABLE:
    VALUE_00: 0xA0000000
    VALUE_01: 42
      [..]
    VALUE_15: 0xA000000F
```

Decimal or hexadecimal values can be specified.

5.1.4 divebits_BlockRAM_init

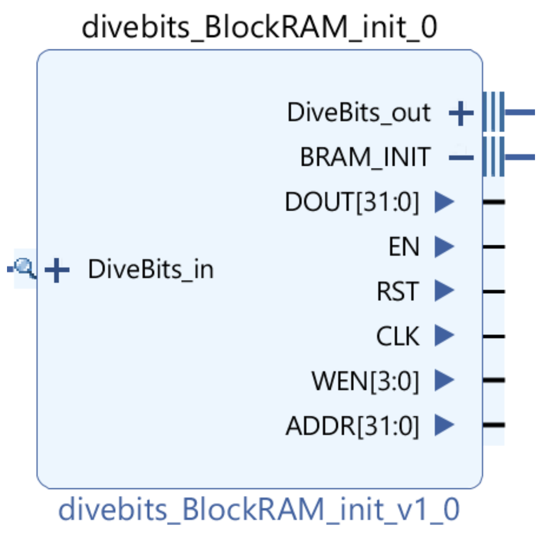


Figure 7: divebits_BlockRAM_init

A write master that can be connected to one side of a dual-port BlockRAM to initialize it with data.

Parameters/Generics

- ***BRAM Port Data Width***/`DB_BRAM_DATA_WIDTH` specifies the actual data width being stored
- ***BRAM Port Address Width***/`DB_BRAM_ADDRESS_WIDTH` specifies the actual address width and consequently the memory depth.
- ***BRAM Controller Mode***/`DB_BRAMCTRL_MODE` is a boolean that, when selected, extends the actual address width to 32bit and the actual data width to the next power of 2. This is required in block designt for the IP Integrator to recognize the component as a BRAM Controller.

Signals

- `CLK`,`ADDR`,`DOUT`,`WEN`,`RST` are the common output signals for synchronous BlockRAM write control and compose the BRAM interface.

YAML configuration

```
- BLOCK_PATH: /divebits_BlockRAM_init_0
  CONFIGURABLE:
    default_value: 0xffffffff
    ranges:
    - from: 0x200
      to: 0x2ff
      value: 0xAA000077
    words:
      0x0: 42
      0x1: 747
      0x1ff: 0xDEADBEEF
```

`CONFIGURABLE` for the ***divebits_BlockRAM_init*** component is a dictionary (key-value store) with the three keys **`default_value`**, **`ranges`** and **`words`**. All addresses and data words can be specified in decimal or hexadecimal.
`default_value` holds the value for all addresses that are not otherwise specified.
`words` is a nested dictionary where specific data word addresses can be assigned data values.
`ranges` is a list, where each list element is a dictionary with the three keys three keys **`from`**, **`to`** and **`value`**. Each set allows to specify an address range that is initialized with the same word.

5.2 Block-only components

5.2.1 divebits_AXI_4_constant_registers

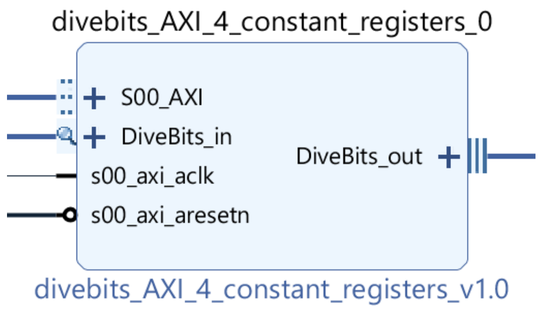

divebits_AXI_4_constant_registers_0
S00_AXI
DiveBits_in
s00_axi_aclk
s00_axi_aresetn
DiveBits_out
divebits_AXI_4_constant_registers_v1.0

Figure 8: divebits_AXI_4_constant_registers

This is a an AXI Subordinate component that provides 4 constant, memory-mapped 32-bit registers. It can provided diversified data to a CPU system without re-compiling the software.

Parameters/Generics

- ***Default Value***/`DB_DEFAULT_VALUE` specifies an identical 32-bit default value for all 4 registers.

Signals

- `S00_AXI` is an ***AXI4Lite*** Subordinate interface with all the requisite signals.

YAML configuration

```
- BLOCK_PATH: /divebits_AXI_4_constant_registers_0
  CONFIGURABLE:
    REGISTER_00_VALUE: 0xA380
    REGISTER_01_VALUE: 0xB787
    REGISTER_02_VALUE: 314159
    REGISTER_03_VALUE: 2718
```

Values for all four 32-Bit registers can be specified in decimal or hexadecimal.

5.2.2 divebits_AXIS_Master

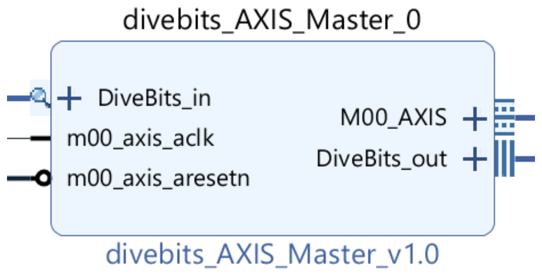


Figure 9: divebits_AXIS_Master

An AXI Stream Master that can write a configurable number of words into an AXI Stream.

Parameters/Generics

- ***Data Word Width***/`DB_DATA_WIDTH` can be set to any power of 2 from 32 to 1024.
- ***Number Of Data Words***/`DB_DATA_WIDTH` can be set to any power of 2 from 32 to 256. This defines the storage depth, however less words than set here can be configured.

Signals

- `M00_AXIS` is an ***AXI4 Stream*** Master interface with all the requisite signals.

YAML configuration

```
- BLOCK_PATH: /divebits_AXIS_Master_0
  CONFIGURABLE:
    WORD_COUNT: 3
    DATA:
      0:
        TDATA: 0xdeadbeef
        TLAST: false
      1:
        TDATA: 0xc0ffee77
        TLAST: false
      2:
        TDATA: 0xbadc0c0a
        TLAST: true
```

`WORD_COUNT` has to be explicitly set to a number smaller or equal than the specified storage size. `DATA` is a dictionary with keys numbered from `0` to `WORD_COUNT`-1. Each key is itself a dictionary with the values for `TDATA` (decimal or hexadecimal) and `TLAST` (boolean).

5.2.3 divebits_AXI_Master_ReadWrite

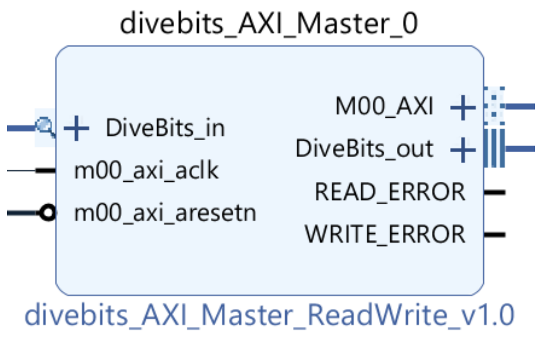


Figure 10: divebits_AXI_Master_ReadWrite

An AXI master that can replace a CPU for simple configuration tasks. Can execute diversified programs composted of a simple set of opcodes to:

- Write values to memory-mapped AXI subordinates.
- Copy values between AXI subordinates.
- Repeatedly read from an AXI subordinate address (polling) until specific bits are set or cleared. Wait cycles can be configured so the polling doesn't clog the interconnect.

Parameters/Generics

- ***Number Of AXI Master Instruction Words***/`DB_NUM_CODE_WORDS` can be set to 64, 128 or 256 instruction words.
- ***Wait And Repeat Access After Bus Error***/`DB_REPEAT_AFTER_BUS_ERROR` is a boolean that determines if an operation is tried again after a bus error or if the Master freezes.
- ***Bitwidth of Repeat Waitcycles counter***/`DB_REPEAT_WAITCYCLES_WIDTH` can be specified from 3 to 20. If the previous option is activated, the master waits for (2ŴIDTH) cycles before trying again.

Signals

- `M00_AXI` is an ***AXI4Lite*** Master interface with all the requisite signals.
- `READ_ERROR` signals a bus read error happened.
- `WRITE_ERROR` signals a bus write error happened.

YAML configuration

```
- BLOCK_PATH: /divebits_AXI_Master_0
  CONFIGURABLE:
    OPCODE_COUNT: 5
    CODE:
      0:
        OPCODE: SET_BASE_ADDR
        ADDR: 0x40600000
      1:
        OPCODE: WRITE_FROM_CODE
        ADDR: 4
        DATA: 65
      2:
        OPCODE: READ_TO_BUFFER
        ADDR: 0
      3:
        OPCODE: WRITE_FROM_BUFFER
        ADDR: 4
      4:
        OPCODE: READ_CHECK_WAIT
        ADDR: 8
        CHECK_MASK: 0x0000000F
        CHECK_DATA: 0x00000003
```

`OPCODE_COUNT` has to be set to the amount of following opcodes.
`CODE` is a dictionary of multiword opcode keys numbered from `0` to `OPCODE_COUNT`-1. Each entry is another dictionary of opcode data. The available opcodes are:

- **`SET_BASE_ADDR`** sets an address base for all following relative address operations with the key **`ADDR`**. This is useful to read and configure peripherals with a base address and register offsets.
- **`READ_TO_BUFFER`** reads from the specified **`ADDR`** offset into a buffer register.
- **`WRITE_FROM_BUFFER`** writes to the specified **`ADDR`** offset from the buffer register.
- **`WRITE_FROM_CODE`** writes **`DATA`** to the specified **`ADDR`** offset.
- **`READ_CHECK_WAIT`** reads from the specified **`ADDR`** offset and checks if all bits set in **`CHECK_MASK`** are identical with the same bits in **`CHECK_DATA`**. In the example shown above, the master reads from address offset 8 and then only checks the lower 4 bit of that read against 0x3. If they are not identical, it loops an reads again until they are.

5.2.4 divebits_AXI_Master_WriteOnly

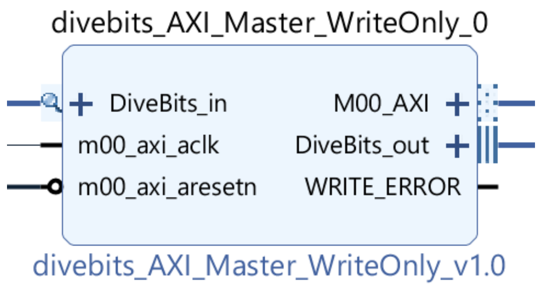


Figure 11: divebits_AXI_Master_WriteOnly

A simpler AXI master that can only write to memory-mapped AXI addresses. Can repeat writes on bus errors, so for example a full ***AXI Uartlite*** FIFO can be handled without explicit reads.

Parameters/Generics

- ***Number Of AXI Master Instruction Words***/`DB_NUM_CODE_WORDS` can be set to 64, 128 or 256 instruction words.
- ***Wait And Repeat Access After Bus Error***/`DB_REPEAT_AFTER_BUS_ERROR` is a boolean that determines if an operation is tried again after a bus error or if the Master freezes.
- ***Bitwidth of Repeat Waitcycles counter***/`DB_REPEAT_WAITCYCLES_WIDTH` can be specified from 3 to 20. If the previous option is activated, the master waits for (2ŴIDTH) cycles before trying again.

Signals

- `M00_AXI` is an ***AXI4Lite*** Write Master interface with all the requisite signals.
- `WRITE_ERROR` signals a bus write error happened.

YAML configuration

```
- BLOCK_PATH: /divebits_AXI_Master_WriteOnly_0
  CONFIGURABLE:
    OPCODE_COUNT: 3
    CODE:
      0:
        OPCODE: SET_BASE_ADDR
        ADDR: 0x40600000
      1:
        OPCODE: WRITE_FROM_CODE
        ADDR: 4
        DATA: 65
      2:
        OPCODE: WRITE_FROM_BUFFER
        ADDR: 4
```

`OPCODE_COUNT` has to be set to the amount of following opcodes.
`CODE` is a dictionary of multiword opcode keys numbered from **`0`** to **`OPCODE_COUNT`**-1. Each entry is another dictionary of opcode data. The available opcodes are:

- **`SET_BASE_ADDR`** sets an address base for all following relative address operations with the key **`ADDR`**. This is useful to read and configure peripherals with a base address and register offsets.
- **`WRITE_FROM_CODE`** writes **`DATA`** to the specified **`ADDR`** offset.
- **`WRITE_FROM_BUFFER`** writes the last written **`DATA`** again to the specified **`ADDR`** offset from the buffer register.