ARTIQ Documentation

Release 7.8190.db79100

M-Labs and contributors

Mar 20, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

ARTIQ (Advanced Real-Time Infrastructure for Quantum physics) is the next-generation control system for quantum information experiments. It is maintained and developed by M-Labs and the initial development was for and in partnership with the Ion Storage Group at NIST. ARTIQ is free software and offered to the entire research community as a solution equally applicable to other challenging control tasks, including outside the field of ion trapping. Several other laboratories (e.g. at the University of Oxford, the Army Research Lab, and the University of Maryland) have later adopted ARTIQ as their control system and have contributed to it.

The system features a high-level programming language that helps describing complex experiments, which is compiled and executed on dedicated hardware with nanosecond timing resolution and sub-microsecond latency. It includes graphical user interfaces to parametrize and schedule experiments and to visualize and explore the results.

ARTIQ uses FPGA hardware to perform its time-critical tasks. The Sinara hardware, and in particular the Kasli FPGA carrier, is designed to work with ARTIQ. ARTIQ is designed to be portable to hardware platforms from different vendors and FPGA manufacturers. Several different configurations of a high-end FPGA evaluation kit are also used and supported. FPGA platforms can be combined with any number of additional peripherals, either already accessible from ARTIQ or made accessible with little effort.

ARTIQ and its dependencies are available in the form of Nix packages (for Linux) and Conda packages (for Windows and Linux). Packages containing pre-compiled binary images to be loaded onto the hardware platforms are supplied for each configuration. Like any open source software ARTIQ can equally be built and installed directly from source.

ARTIQ is supported by M-Labs and developed openly. Components, features, fixes, improvements, and extensions are funded by and developed for the partnering research groups.

Core technologies employed include Python, Migen, Migen-AXI, Rust, MiSoC/VexRiscv, LLVM/llvmlite, and Qt5.

Website: https://m-labs.hk/artiq

Cite ARTIQ as `Bourdeauducq, Sébastien et al. (2016). ARTIQ 1.0. Zenodo. 10.5281/zenodo.51303`.

CHAPTER

TWO

INSTALLING ARTIQ

ARTIQ can be installed using the Nix (on Linux) or Conda (on Windows or Linux) package managers.

Nix is an innovative, robust, fast, and high-quality solution that comes with a larger collection of packages and features than Conda. However, Windows support is poor (using it with Windows Subsystem for Linux still has many problems) and Nix can be harder to learn.

Conda has a more traditional approach to package management, is much more limited, slow, and lower-quality than Nix, but it supports Windows and it is simpler to use when it functions correctly.

In the current state of affairs, we recommend that Linux users install ARTIQ via Nix and Windows users install it via Conda.

2.1 Installing via Nix (Linux)

First, install the Nix package manager. Some distributions provide a package for the Nix package manager, otherwise, it can be installed via the script on the Nix website. Make sure you get Nix version 2.4 or higher.

Once Nix is installed, enable Flakes:

```
$ mkdir -p ~/.config/nix
$ echo "experimental-features = nix-command flakes" > ~/.config/nix/nix.conf
```

The easiest way to obtain ARTIQ is then to install it into the user environment with `$ nix profile install git+https://github.com/m-labs/artiq.git\?ref=release-7`. Answer "Yes" to the questions about setting Nix configuration options. This provides a minimal installation of ARTIQ where the usual commands (`artiq_master`, `artiq_dashboard`, `artiq_run`, etc.) are available.

This installation is however quite limited, as Nix creates a dedicated Python environment for the ARTIQ commands alone. This means that other useful Python packages that you may want (pandas, matplotlib, ...) are not available to them.

Installing multiple packages and making them visible to the ARTIQ commands requires using the Nix language. Create an empty directory with a file `flake.nix` with the following contents:

```
{
  inputs.artiq.url = "git+https://github.com/m-labs/artiq.git?ref=release-7";
  inputs.extrapkg.url = "git+https://git.m-labs.hk/M-Labs/artiq-extrapkg.git?ref=release-
↪7";
  inputs.extrapkg.inputs.artiq.follows = "artiq";
  outputs = { self, artiq, extrapkg }:
    let
      pkgs = artiq.inputs.nixpkgs.legacyPackages.x86_64-linux;
```

(continues on next page)

(continued from previous page)

```
      aqmain = artiq.packages.x86_64-linux;
      aqextra = extrapkg.packages.x86_64-linux;
    in {
      defaultPackage.x86_64-linux = pkgs.buildEnv {
        name = "artiq-env";
        paths = [
          # ========================================
          # EDIT BELOW
          # ========================================
          (pkgs.python3.withPackages(ps: [
            # List desired Python packages here.
            aqmain.artiq
            #ps.paramiko  # needed if and only if flashing boards remotely (artiq_flash -
↪H)
            #aqextra.flake8-artiq

            # The NixOS package collection contains many other packages that you may find
            # interesting. Here are some examples:
            #ps.pandas
            #ps.numpy
            #ps.scipy
            #ps.numba
            #ps.matplotlib
            # or if you need Qt (will recompile):
            #(ps.matplotlib.override { enableQt = true; })
            #ps.bokeh
            #ps.cirq
            #ps.qiskit
          ]))
          #aqextra.korad_ka3005p
          #aqextra.novatech409b
          # List desired non-Python packages here
          #aqmain.openocd-bscanspi  # needed if and only if flashing boards
          # Other potentially interesting packages from the NixOS package collection:
          #pkgs.gtkwave
          #pkgs.spyder
          #pkgs.R
          #pkgs.julia
          # ========================================
          # EDIT ABOVE
          # ========================================
        ];
      };
    };
  nixConfig = {  # work around https://github.com/NixOS/nix/issues/6771
    extra-trusted-public-keys = "nixbld.m-labs.hk-
↪1:5aSRVA5b320xbNvu30tqxVPXpld73bhtOeH6uAjRyHc=";
    extra-substituters = "https://nixbld.m-labs.hk";
  };
}
```

Then spawn a shell containing the packages with `$ nix shell`. The ARTIQ commands with all the additional packages should now be available.

You can exit the shell by typing Control-D. The next time `$ nix shell` is invoked, Nix uses the cached packages so the shell startup is fast.

You can create directories containing each a `flake.nix` that correspond to different sets of packages. If you are familiar with Conda, using Nix in this way is similar to having multiple Conda environments.

If your favorite package is not available with Nix, contact us using the helpdesk@ email.

2.2 Installing via Conda (Windows, Linux)

Warning: For Linux users, the Nix package manager is preferred, as it is more reliable and faster than Conda.

First, install Anaconda or the more minimalistic Miniconda.

After installing either Anaconda or Miniconda, open a new terminal (also known as command line, console, or shell and denoted here as lines starting with $) and verify the following command works:

```
$ conda
```

Executing just `conda` should print the help of the `conda` command. If your shell does not find the `conda` command, make sure that the Conda binaries are in your `$PATH`. If `$ echo $PATH` does not show the Conda directories, add them: execute `$ export PATH=$HOME/miniconda3/bin:$PATH` if you installed Conda into `~/miniconda3`.

Controllers for third-party devices (e.g. Thorlabs TCube, Lab Brick Digital Attenuator, etc.) that are not shipped with ARTIQ can also be installed with this script. Browse Hydra or see the list of NDSPs in this manual to find the names of the corresponding packages, and list them at the beginning of the script.

Set up the Conda channel and install ARTIQ into a new Conda environment:

```
$ conda config --prepend channels https://conda.m-labs.hk/artiq
$ conda config --append channels conda-forge
$ conda create -n artiq artiq
```

Note: The board-specific files containing bitstream and firmware for the FPGA board can be obtained through AFWS, and are only required when flashing. Controllers for third-party devices (e.g. Thorlabs TCube, Lab Brick Digital Attenuator, etc.) that are not shipped with ARTIQ can also be installed with Conda. Browse Hydra or see the list of NDSPs in this manual to find the names of the corresponding packages.

Note: On Windows, if the last command that creates and installs the ARTIQ environment fails with an error similar to "seeking backwards is not allowed", try to re-run the command with admin rights.

Note: For commercial use you might need a license for Anaconda/Miniconda or for using the Anaconda package channel. Miniforge might be an alternative in a commercial environment as it does not include the Anaconda package channel by default. If you want to use Anaconda/Miniconda/Miniforge in a commercial environment, please check the license and the latest terms of service.

After the installation, activate the newly created environment by name.

```
$ conda activate artiq
```

This activation has to be performed in every new shell you open to make the ARTIQ tools from that environment available.

Note: Some ARTIQ examples also require matplotlib and numba, and they must be installed manually for running those examples. They are available in Conda.

2.3 Upgrading ARTIQ (with Nix)

Run `$ nix profile upgrade` if you installed ARTIQ into your user profile. If you used a `flake.nix` shell environment, make a back-up copy of the `flake.lock` file to enable rollback, then run `$ nix flake update` and re-enter `$ nix shell`.

To rollback to the previous version, respectively use `$ nix profile rollback` or restore the backed-up version of the `flake.lock` file.

You may need to reflash the gateware and firmware of the core device to keep it synchronized with the software.

2.4 Upgrading ARTIQ (with Conda)

When upgrading ARTIQ or when testing different versions it is recommended that new Conda environments are created instead of upgrading the packages in existing environments. Keep previous environments around until you are certain that they are not needed anymore and a new environment is known to work correctly.

To install the latest version, just select a different environment name and run the installation command again.

Switching between Conda environments using commands such as `$ conda deactivate artiq-6` and `$ conda activate artiq-5` is the recommended way to roll back to previous versions of ARTIQ.

You may need to reflash the gateware and firmware of the core device to keep it synchronized with the software.

You can list the environments you have created using:

```
$ conda env list
```

2.5 Flashing gateware and firmware into the core device

Note: If you have purchased a pre-assembled system from M-Labs or QUARTIQ, the gateware and firmware are already flashed and you can skip those steps, unless you want to replace them with a different version of ARTIQ.

You need to write three binary images onto the FPGA board:

1. The FPGA gateware bitstream
2. The bootloader
3. The ARTIQ runtime or satellite manager

2.5.1 Installing OpenOCD

Note: This version of OpenOCD is not applicable to Kasli-SoC.

OpenOCD can be used to write the binary images into the core device FPGA board's flash memory.

With Nix, add `aqmain.openocd-bscanspi` to the shell packages. Be careful not to add `pkgs.openocd` instead - this would install OpenOCD from the NixOS package collection, which does not support ARTIQ boards.

With Conda, install `openocd` as follows:

```
$ conda install -c m-labs openocd
```

2.5.2 Configuring OpenOCD

Note: These instructions are not applicable to Kasli-SoC.

Some additional steps are necessary to ensure that OpenOCD can communicate with the FPGA board.

On Linux, first ensure that the current user belongs to the `plugdev` group (i.e. `plugdev` shown when you run `$ groups`). If it does not, run `$ sudo adduser $USER plugdev` and re-login.

If you installed OpenOCD on Linux using Nix, use the `which` command to determine the path to OpenOCD, and then copy the udev rules:

```
$ which openocd
/nix/store/2bmsssvk3d0y5hra06pv54s2324m4srs-openocd-mlabs-0.10.0/bin/openocd
$ sudo cp /nix/store/2bmsssvk3d0y5hra06pv54s2324m4srs-openocd-mlabs-0.10.0/share/openocd/
↪contrib/60-openocd.rules /etc/udev/rules.d
$ sudo udevadm trigger
```

NixOS users should of course configure OpenOCD through `/etc/nixos/configuration.nix` instead.

If you installed OpenOCD on Linux using Conda and are using the Conda environment `artiq`, then execute the statements below. If you are using a different environment, you will have to replace `artiq` with the name of your environment:

```
$ sudo cp ~/.conda/envs/artiq/share/openocd/contrib/60-openocd.rules /etc/udev/rules.d
$ sudo udevadm trigger
```

On Windows, a third-party tool, Zadig, is necessary. Use it as follows:

1. Make sure the FPGA board's JTAG USB port is connected to your computer.
2. Activate Options → List All Devices.
3. Select the "Digilent Adept USB Device (Interface 0)" or "FTDI Quad-RS232 HS" (or similar) device from the drop-down list.
4. Select WinUSB from the spinner list.
5. Click "Install Driver" or "Replace Driver".

You may need to repeat these steps every time you plug the FPGA board into a port where it has not been plugged into previously on the same system.

2.5.3 Obtaining the board binaries

If you have an active firmware subscription with M-Labs or QUARTIQ, you can obtain firmware that corresponds to the currently installed version of ARTIQ using AFWS (ARTIQ firmware service). One year of subscription is included with most hardware purchases. You may purchase or extend firmware subscriptions by writing to the sales@ email.

Run the command:

```
$ afws_client [username] build [afws_directory] [variant]
```

Replace `[username]` with the login name that was given to you with the subscription, `[variant]` with the name of your system variant, and `[afws_directory]` with the name of an empty directory, which will be created by the command if it does not exist. Enter your password when prompted and wait for the build (if applicable) and download to finish. If you experience issues with the AFWS client, write to the helpdesk@ email.

Without a subscription, you may build the firmware yourself from the open source code. See the section *Developing ARTIQ*.

2.5.4 Writing the flash

Then, you can write the flash:

- For Kasli:

  ```
  $ artiq_flash -d [afws_directory]
  ```

The JTAG adapter is integrated into the Kasli board; for flashing (and debugging) you simply need to connect your computer to the micro-USB connector on the Kasli front panel.

- For Kasli-SoC:

  ```
  $ artiq_coremgmt [-D 192.168.1.75] config write -f boot [afws_directory]/boot.bin
  ```

If the Kasli-SoC won't boot due to corrupted firmware and `artiq_coremgmt` cannot access it, extract the SD card and replace `boot.bin` manually.

- For the KC705 board:

  ```
  $ artiq_flash -t kc705 -d [afws_directory]
  ```

  The SW13 switches need to be set to 00001.

2.6 Setting up the core device IP networking

For Kasli, insert a SFP/RJ45 transceiver (normally included with purchases from M-Labs and QUARTIQ) into the SFP0 port and connect it to an Ethernet port in your network. If the port is 10Mbps or 100Mbps and not 1000Mbps, make sure that the SFP/RJ45 transceiver supports the lower rate. Many SFP/RJ45 transceivers only support the 1000Mbps rate. If you do not have a SFP/RJ45 transceiver that supports 10Mbps and 100Mbps rates, you may instead use a gigabit Ethernet switch in the middle to perform rate conversion.

You can also insert other types of SFP transceivers into Kasli if you wish to use it directly in e.g. an optical fiber Ethernet network.

If you purchased a Kasli device from M-Labs, it usually comes with the IP address `192.168.1.75`. Once you can reach this IP, it can be changed with:

```
$ artiq_coremgmt -D 192.168.1.75 config write -s ip [new IP]
```

and then reboot the device (with `artiq_flash start` or a power cycle).

In other cases, install OpenOCD as before, and flash the IP (and, if necessary, MAC) addresses directly:

```
$ artiq_mkfs flash_storage.img -s mac xx:xx:xx:xx:xx:xx -s ip xx.xx.xx.xx
$ artiq_flash -t [board] -V [variant] -f flash_storage.img storage start
```

For Kasli devices, flashing a MAC address is not necessary as they can obtain it from their EEPROM.

Check that you can ping the device. If ping fails, check that the Ethernet link LED is ON - on Kasli, it is the LED next to the SFP0 connector. As a next step, look at the messages emitted on the UART during boot. Use a program such as flterm or PuTTY to connect to the device's serial port at 115200bps 8-N-1 and reboot the device. On Kasli, the serial port is on FTDI channel 2 with v1.1 hardware (with channel 0 being JTAG) and on FTDI channel 1 with v1.0 hardware.

If you want to use IPv6, the device also has a link-local address that corresponds to its EUI-64, and an additional arbitrary IPv6 address can be defined by using the `ip6` configuration key. All IPv4 and IPv6 addresses can be used at the same time.

2.7 Miscellaneous configuration of the core device

Those steps are optional. The core device usually needs to be restarted for changes to take effect.

- Load the idle kernel

The idle kernel is the kernel (some piece of code running on the core device) which the core device runs whenever it is not connected to a PC via Ethernet. This kernel is therefore stored in the *core device configuration flash storage*.

To flash the idle kernel, first compile the idle experiment. The idle experiment's `run()` method must be a kernel: it must be decorated with the `@kernel` decorator (see *next topic* for more information about kernels). Since the core device is not connected to the PC, RPCs (calling Python code running on the PC from the kernel) are forbidden in the idle experiment. Then write it into the core device configuration flash storage:

```
$ artiq_compile idle.py
$ artiq_coremgmt config write -f idle_kernel idle.elf
```

Note: You can find more information about how to use the `artiq_coremgmt` utility on the *Utilities* page.

- Load the startup kernel

The startup kernel is executed once when the core device powers up. It should initialize DDSes, set up TTL directions, etc. Proceed as with the idle kernel, but using the `startup_kernel` key in the `artiq_coremgmt` command.

For DRTIO systems, the startup kernel should wait until the desired destinations (including local RTIO) are up, using `artiq.coredevice.Core.get_rtio_destination_status()`.

- Load the DRTIO routing table

If you are using DRTIO and the default routing table (for a star topology) is not suitable to your needs, prepare and load a different routing table. See *Using DRTIO*.

- Select the RTIO clock source (KC705 and Kasli)

The KC705 may use either an external clock signal, or its internal clock with external frequency or internal crystal reference. The clock is selected at power-up. Setting the RTIO clock source to "ext0_bypass" would bypass the Si5324 synthesiser, requiring that an input clock be present. To select the source, use one of these commands:

```
$ artiq_coremgmt config write -s rtio_clock int_125  # internal 125MHz clock (default)
$ artiq_coremgmt config write -s rtio_clock ext0_bypass  # external clock (bypass)
```

Other options include:

- `ext0_synth0_10to125` - external 10MHz reference clock used by Si5324 to synthesize a 125MHz RTIO clock,
- `ext0_synth0_100to125` - exteral 100MHz reference clock used by Si5324 to synthesize a 125MHz RTIO clock,
- `ext0_synth0_125to125` - exteral 125MHz reference clock used by Si5324 to synthesize a 125MHz RTIO clock,
- `int_100` - internal crystal reference is used by Si5324 to synthesize a 100MHz RTIO clock,
- `int_150` - internal crystal reference is used by Si5324 to synthesize a 150MHz RTIO clock.
- `ext0_bypass_125` and `ext0_bypass_100` - explicit aliases for `ext0_bypass`.

Availability of these options depends on the board and their configuration - specific setting may or may not be supported.

CHAPTER

THREE

DEVELOPING ARTIQ

Warning: This section is only for software or FPGA developers who want to modify ARTIQ. The steps described here are not required if you simply want to run experiments with ARTIQ. If you purchased a system from M-Labs or QUARTIQ, we normally provide board binaries for you.

The easiest way to obtain an ARTIQ development environment is via the Nix package manager on Linux. The Nix system is used on the M-Labs Hydra server to build ARTIQ and its dependencies continuously; it ensures that all build instructions are up-to-date and allows binary packages to be used on developers' machines, in particular for large tools such as the Rust compiler. ARTIQ itself does not depend on Nix, and it is also possible to compile everything from source (look into the `flake.nix` file and/or nixpkgs, and run the commands manually) - but Nix makes the process a lot easier.

- Download Vivado from Xilinx and install it (by running the official installer in a FHS chroot environment if using NixOS; the ARTIQ flake provides such an environment). If you do not want to write to `/opt`, you can install it in a folder of your home directory. The "appropriate" Vivado version to use for building the bitstream can vary. Some versions contain bugs that lead to hidden or visible failures, others work fine. Refer to Hydra and/or the `flake.nix` file from the ARTIQ repository in order to determine which version is used at M-Labs. If the Vivado GUI installer crashes, you may be able to work around the problem by running it in unattended mode with a command such as `./xsetup -a XilinxEULA,3rdPartyEULA,WebTalkTerms -b Install -e 'Vitis Unified Software Platform' -l /opt/Xilinx/`.
- During the Vivado installation, uncheck `Install cable drivers` (they are not required as we use better and open source alternatives).
- Install the Nix package manager, version 2.4 or later. Prefer a single-user installation for simplicity.
- If you did not install Vivado in its default location `/opt`, clone the ARTIQ Git repository and edit `flake.nix` accordingly.
- Enable flakes in Nix by e.g. adding `experimental-features = nix-command flakes` to `nix.conf` (for example `~/.config/nix/nix.conf`).
- Enter the development shell by running `nix develop git+https://github.com/m-labs/artiq.git\?ref=release-7`, or alternatively by cloning the ARTIQ Git repository and running `nix develop` at the root (where `flake.nix` is).
- You can then build the firmware and gateware with a command such as `$ python -m artiq.gateware.targets.kasli`. If you are using a JSON system description file, use `$ python -m artiq.gateware.targets.kasli_generic file.json`.
- Flash the binaries into the FPGA board with a command such as `$ artiq_flash --srcbuild -d artiq_kasli/<your_variant>`. You need to configure OpenOCD as explained *in the user section*. OpenOCD is already part of the flake's development environment.

- Check that the board boots and examine the UART messages by running a serial terminal program, e.g. `$ flterm /dev/ttyUSB1` (`flterm` is part of MiSoC and installed in the flake's development environment). Leave the terminal running while you are flashing the board, so that you see the startup messages when the board boots immediately after flashing. You can also restart the board (without reflashing it) with `$ artiq_flash start`.
- The communication parameters are 115200 8-N-1. Ensure that your user has access to the serial device (e.g. by adding the user account to the `dialout` group).

Warning: Nix will make a read-only copy of the ARTIQ source to use in the shell environment. Therefore, any modifications that you make to the source after the shell is started will not be taken into account. A solution applicable to ARTIQ (and several other Python packages such as Migen and MiSoC) is to prepend the ARTIQ source directory to the `PYTHONPATH` environment variable after entering the shell. If you want this to be done by default, edit the `devShell` section of `flake.nix`.

CHAPTER

FOUR

RELEASE NOTES

4.1 ARTIQ-7

Highlights:

- **New hardware support:**
  - Kasli-SoC, a new EEM carrier based on a Zynq SoC, enabling much faster kernel execution (see: https://arxiv.org/abs/2111.15290).
  - DRTIO support on Zynq-based devices (Kasli-SoC and ZC706).
  - DRTIO support on KC705.
  - HVAMP_8CH 8 channel HV amplifier for Fastino / Zotinos
  - Almazny mezzanine board for Mirny
  - Phaser: improved documentation, exposed the DAC coarse mixer and `sif_sync`, exposed upconverter calibration and enabling/disabling of upconverter LO & RF outputs, added helpers to align Phaser updates to the RTIO timeline (`get_next_frame_mu()`).
  - Urukul: `get()`, `get_mu()`, `get_att()`, and `get_att_mu()` functions added for AD9910 and AD9912.
- Softcore targets now use the RISC-V architecture (VexRiscv) instead of OR1K (mor1kx).
- Gateware FPU is supported on KC705 and Kasli 2.0.
- Faster compilation for large arrays/lists.
- Faster exception handling.
- Several exception handling bugs fixed.
- Support for a simpler shared library system with faster calls into the runtime. This is only used by the NAC3 compiler (nac3ld) and improves RTIO output performance (test_pulse_rate) by 9-10%.
- Moninj improvements: - Urukul monitoring and frequency setting (through dashboard) is now supported. - Core device moninj is now proxied via the `aqctl_moninj_proxy` controller.
- The configuration entry `rtio_clock` supports multiple clocking settings, deprecating the usage of compile-time options.
- Added support for 100MHz RTIO clock in DRTIO.
- Previously detected RTIO async errors are reported to the host after each kernel terminates and a warning is logged. The warning is additional to the one already printed in the core device log immediately upon detection of the error.
- Extended Kasli gateware JSON description with configuration for SPI over DIO.

- TTL outputs can be now configured to work as a clock generator from the JSON.
- On Kasli, the number of FIFO lanes in the scalable events dispatcher (SED) can now be configured in the JSON.
- `artiq_ddb_template` generates edge-counter keys that start with the key of the corresponding TTL device (e.g. `ttl_0_counter` for the edge counter on TTL device `ttl_0`).
- `artiq_master` now has an `--experiment-subdir` option to scan only a subdirectory of the repository when building the list of experiments.
- Experiments can now be submitted by-content.
- The master can now optionally log all experiments submitted into a CSV file.
- Removed worker DB warning for writing a dataset that is also in the archive.
- Experiments can now call `scheduler.check_termination()` to test if the user has requested graceful termination.
- ARTIQ command-line programs and controllers now exit cleanly on Ctrl-C.
- `artiq_coremgmt reboot` now reloads gateware as well, providing a more thorough and reliable device reset (7-series FPGAs only).
- Firmware and gateware can now be built on-demand on the M-Labs server using `afws_client` (subscribers only). Self-compilation remains possible.
- Easier-to-use packaging via Nix Flakes.
- Python 3.10 support (experimental).

Breaking changes:

- Due to the new RISC-V CPU, the device database entry for the core device needs to be updated. The `target` parameter needs to be set to `rv32ima` for Kasli 1.x and to `rv32g` for all other boards. Freshly generated device database templates already contain this update.
- Updated Phaser-Upconverter default frequency 2.875 GHz. The new default uses the target PFD frequency of the hardware design.
- `Phaser.init()` now disables all Kasli-oscillators. This avoids full power RF output being generated for some configurations.
- Phaser: fixed coarse mixer frequency configuration
- Mirny: Added extra delays in `ADF5356.sync()`. This avoids the need of an extra delay before calling `ADF5356.init()`.
- The deprecated `set_dataset(..., save=...)` is no longer supported.
- The `PCA9548` I2C switch class was renamed to `I2CSwitch`, to accommodate support for PCA9547, and possibly other switches in future. Readback has been removed, and now only one channel per switch is supported.

4.2 ARTIQ-6

Highlights:

- **New hardware support:**
  - Phaser, a quad channel 1GS/s RF generator card with dual IQ upconverter and dual 5MS/s ADC and FPGA.

  - Zynq SoC core device (ZC706), enabling kernels to run on 1 GHz CPU core with a floating-point unit for faster computations. This currently requires an external repository (https://git.m-labs.hk/m-labs/artiq-zynq).
  - Mirny 4-channel wide-band PLL/VCO-based microwave frequency synthesiser
  - Fastino 32-channel, 3MS/s per channel, 16-bit DAC EEM
  - Kasli 2.0, an improved core device with 12 built-in EEM slots, faster FPGA, 4 SFPs, and high-precision clock recovery circuitry for DRTIO (to be supported in ARTIQ-7).
- **ARTIQ Python (core device kernels):**
  - Multidimensional arrays are now available on the core device, using NumPy syntax. Elementwise operations (e.g. `+`, `/`), matrix multiplication (`@`) and multidimensional indexing are supported; slices and views are not yet.
  - Trigonometric and other common math functions from NumPy are now available on the core device (e.g. `numpy.sin`), both for scalar arguments and implicitly broadcast across multidimensional arrays.
  - Failed assertions now raise `AssertionError`s instead of aborting kernel execution.
- **Performance improvements:**
  - SERDES TTL inputs can now detect edges on pulses that are shorter than the RTIO period (https://github.com/m-labs/artiq/issues/1432)
  - Improved performance for kernel RPC involving list and array.
- Coredevice SI to mu conversions now always return valid codes, or raise a `ValueError`.
- Zotino now exposes `voltage_to_mu()`
- **`ad9910`:**
  - The maximum amplitude scale factor is now `0x3fff` (was `0x3ffe` before).
  - The default single-tone profile is now 7 (was 0).
  - Added option to `set_mu()` that affects the ASF, FTW and POW registers instead of the single-tone profile register.
- Mirny now supports HW revision independent, human readable `clk_sel` parameters: "XO", "SMA", and "MMCX". Passing an integer is backwards compatible.
- **Dashboard:**
  - Applets now restart if they are running and a ccb call changes their spec
  - A "Quick Open" dialog to open experiments by typing part of their name can be brought up Ctrl-P (Ctrl+Return to immediately submit the selected entry with the default arguments).
  - The Applets dock now has a context menu command to quickly close all open applets (shortcut: Ctrl-Alt-W).
- Experiment results are now always saved to HDF5, even if `run()` fails.
- Core device: `panic_reset 1` now correctly resets the kernel CPU as well if communication CPU panic occurs.
- NumberValue accepts a `type` parameter specifying the output as `int` or `float`
- A parameter `--identifier-str` has been added to many targets to aid with reproducible builds.
- Python 3.7 support in Conda packages.
- *kasli_generic* JSON descriptions are now validated against a schema. Description defaults have moved from Python to the schema. Warns if ARTIQ version is too old.

Breaking changes:

- `artiq_netboot` has been moved to its own repository at https://git.m-labs.hk/m-labs/artiq-netboot
- Core device watchdogs have been removed.
- The ARTIQ compiler now implements arrays following NumPy semantics, rather than as a thin veneer around lists. Most prior use cases of NumPy arrays in kernels should work unchanged with the new implementation, but the behavior might differ slightly in some cases (for instance, non-rectangular arrays are not currently supported).
- `quamash` has been replaced with `qasync`.
- Protocols are updated to use device endian.
- Analyzer dump format includes a byte for device endianness.
- To support variable numbers of Urukul cards in the future, the `artiq.coredevice.suservo.SUServo` constructor now accepts two device name lists, `cpld_devices` and `dds_devices`, rather than four individual arguments.
- Experiment classes with underscore-prefixed names are now ignored when `artiq_client` determines which experiment to submit (consistent with `artiq_run`).

4.3 ARTIQ-5

Highlights:

- **Performance improvements:**
  - Faster RTIO event submission (1.5x improvement in pulse rate test) See: https://github.com/m-labs/artiq/issues/636
  - Faster compilation times (3 seconds saved on kernel compilation time on a typical medium-size experiment) See: https://github.com/m-labs/artiq/commit/611bcc4db4ed604a32d9678623617cd50e968cbf
- **Improved packaging and build system:**
  - new continuous integration/delivery infrastructure based on Nix and Hydra, providing reproducibility, speed and independence.
  - rolling release process (https://github.com/m-labs/artiq/issues/1326).
  - firmware, gateware and device database templates are automatically built for all supported Kasli variants.
  - new JSON description format for generic Kasli systems.
  - Nix packages are now supported.
  - many Conda problems worked around.
  - controllers are now out-of-tree.
  - split packages that enable lightweight applications that communicate with ARTIQ, e.g. controllers running on non-x86 single-board computers.
- **Improved Urukul support:**
  - AD9910 RAM mode.
  - Configurable refclk divider and PLL bypass.
  - More reliable phase synchronization at high sample rates.
  - Synchronization calibration data can be read from EEPROM.

- A gateware-level input edge counter has been added, which offers higher throughput and increased flexibility over the usual TTL input PHYs where edge timestamps are not required. See `artiq.coredevice.edge_counter` for the core device driver and `artiq.gateware.rtio.phy.edge_counter/ artiq.gateware.eem.DIO.add_std` for the gateware components.
- With DRTIO, Siphaser uses a better calibration mechanism. See: https://github.com/m-labs/artiq/commit/cc58318500ecfa537abf24127f2c22e8fe66e0f8
- Schedule updates can be sent to influxdb (artiq_influxdb_schedule).
- Experiments can now programatically set their default pipeline, priority, and flush flag.
- List datasets can now be efficiently appended to from experiments using `artiq.language.environment.HasEnvironment.append_to_dataset`.
- The core device now supports IPv6.
- To make development easier, the bootloader can receive firmware and secondary FPGA gateware from the network.
- Python 3.7 compatibility (Nix and source builds only, no Conda).
- Various other bugs from 4.0 fixed.
- Preliminary Sayma v2 and Metlino hardware support.

Breaking changes:

- The `artiq.coredevice.ad9910.AD9910` and `artiq.coredevice.ad9914.AD9914` phase reference timestamp parameters have been renamed to `ref_time_mu` for consistency, as they are in machine units.
- The controller manager now ignores device database entries without the `command` key set to facilitate sharing of devices between multiple masters.
- The meaning of the `-d/--dir` and `--srcbuild` options of `artiq_flash` has changed.
- Controllers for third-party devices are now out-of-tree.
- `aqctl_corelog` now filters log messages below the `WARNING` level by default. This behavior can be changed using the `-v` and `-q` options like the other programs.
- On Kasli the firmware now starts with a unique default MAC address from EEPROM if *mac* is absent from the flash config.
- The `-e/--experiment` switch of `artiq_run` and `artiq_compile` has been renamed `-c/--class-name`.
- `artiq_devtool` has been removed.
- Much of `artiq.protocols` has been moved to a separate package `sipyco`. `artiq_rpctool` has been renamed to `sipyco_rpctool`.

4.4 ARTIQ-4

4.4.1 4.0

- The `artiq.coredevice.ttl` drivers no longer track the timestamps of submitted events in software, requiring the user to explicitly specify the timeout for `count()/timestamp_mu()`. Support for `sync()` has been dropped.

  Now that RTIO has gained DMA support, there is no longer a reliable way for the kernel CPU to track the individual events submitted on any one channel. Requiring the timeouts to be specified explicitly ensures consistent API behavior. To make this more convenient, the `TTLInOut.gate_*()` functions now return the cursor position at the end of the gate, e.g.:

```
ttl_input.count(ttl_input.gate_rising(100 * us))
```

In most situations – that is, unless the timeline cursor is rewound after the respective `gate_*()` call – simply passing `now_mu()` is also a valid upgrade path:

```
ttl_input.count(now_mu())
```

The latter might use up more timeline slack than necessary, though.

In place of `TTL(In)Out.sync`, the new `Core.wait_until_mu()` method can be used, which blocks execution until the hardware RTIO cursor reaches the given timestamp:

```
ttl_output.pulse(10 * us)
self.core.wait_until_mu(now_mu())
```

- RTIO outputs use a new architecture called Scalable Event Dispatcher (SED), which allows building systems with large number of RTIO channels more efficiently. From the user perspective, collision errors become asynchronous, and non- monotonic timestamps on any combination of channels are generally allowed (instead of producing sequence errors). RTIO inputs are not affected.
- The DDS channel number for the NIST CLOCK target has changed.
- The dashboard configuration files are now stored one-per-master, keyed by the server address argument and the notify port.
- The master now has a `--name` argument. If given, the dashboard is labelled with this name rather than the server address.
- `artiq_flash` targets Kasli by default. Use `-t kc705` to flash a KC705 instead.
- `artiq_flash -m/--adapter` has been changed to `artiq_flash -V/--variant`.
- The `proxy` action of `artiq_flash` is determined automatically and should not be specified manually anymore.
- `kc705_dds` has been renamed `kc705`.
- The `-H/--hw-adapter` option of `kc705` has been renamed `-V/--variant`.
- SPI masters have been switched from misoc-spi to misoc-spi2. This affects all out-of-tree RTIO core device drivers using those buses. See the various commits on e.g. the `ad53xx` driver for an example how to port from the old to the new bus.
- The `ad5360` coredevice driver has been renamed to `ad53xx` and the API has changed to better support Zotino.
- `artiq.coredevice.dds` has been renamed to `artiq.coredevice.ad9914` and simplified. DDS batch mode is no longer supported. The `core_dds` device is no longer necessary.
- The configuration entry `startup_clock` is renamed `rtio_clock`. Switching clocks dynamically (i.e. without device restart) is no longer supported.
- `set_dataset(..., save=True)` has been renamed `set_dataset(..., archive=True)`.
- On the AD9914 DDS, when switching to `PHASE_MODE_CONTINUOUS` from another mode, use the returned value of the last `set_mu` call as the phase offset for `PHASE_MODE_CONTINUOUS` to avoid a phase discontinuity. This is no longer done automatically. If one phase glitch when entering `PHASE_MODE_CONTINUOUS` is not an issue, this recommendation can be ignored.

4.5 ARTIQ-3

4.5.1 3.7

No further notes.

4.5.2 3.6

No further notes.

4.5.3 3.5

No further notes.

4.5.4 3.4

No further notes.

4.5.5 3.3

No further notes.

4.5.6 3.2

- To accommodate larger runtimes, the flash layout as changed. As a result, the contents of the flash storage will be lost when upgrading. Set the values back (IP, MAC address, startup kernel, etc.) after the upgrade.

4.5.7 3.1

No further notes.

4.5.8 3.0

- The `--embed` option of applets is replaced with the environment variable `ARTIQ_APPLET_EMBED`. The GUI sets this enviroment variable itself and the user simply needs to remove the `--embed` argument.
- `EnvExperiment`'s `prepare` calls `prepare` for all its children.
- Dynamic `__getattr__`'s returning RPC target methods are not supported anymore. Controller driver classes must define all their methods intended for RPC as members.
- Datasets requested by experiments are by default archived into their HDF5 output. If this behavior is undesirable, turn it off by passing `archive=False` to `get_dataset`.
- `seconds_to_mu` and `mu_to_seconds` have become methods of the core device driver (use e.g. `self.core.seconds_to_mu()`).
- AD9858 DDSes and NIST QC1 hardware are no longer supported.
- The DDS class names and setup options have changed, this requires an update of the device database.

- `int(a, width=b)` has been removed. Use `int32(a)` and `int64(a)`.
- The KC705 gateware target has been renamed `kc705_dds`.
- `artiq.coredevice.comm_tcp` has been renamed `artiq.coredevice.comm_kernel`, and `Comm` has been renamed `CommKernel`.
- The "collision" and "busy" RTIO errors are reported through the log instead of raising exceptions.
- Results are still saved when `analyze` raises an exception.
- `LinearScan` and `RandomScan` have been consolidated into RangeScan.
- The Pipistrello is no longer supported. For a low-cost ARTIQ setup, use either ARTIQ 2.x with Pipistrello, or the future ARTIQ 4.x with Kasli. Note that the Pipistrello board has also been discontinued by the manufacturer but its design files are freely available.
- The device database is now generated by an executable Python script. To migrate an existing database, add `device_db = `` at the beginning, and replace any PYON identifiers (``true`, `null`, ...)` with their Python equivalents (`True`, `None` ...).
- Controllers are now named `aqctl_XXX` instead of `XXX_controller`.
- In the device database, the `comm` device has been folded into the `core` device. Move the "host" argument into the `core` device, and remove the `comm` device.
- The core device log now contains important information about events such as RTIO collisions. A new controller `aqctl_corelog` must be running to forward those logs to the master. See the example device databases to see how to instantiate this controller. Using `artiq_session` ensures that a controller manager is running simultaneously with the master.
- Experiments scheduled with the "flush pipeline" option now proceed when there are lower-priority experiments in the pipeline. Only experiments at the current (or higher) priority level are flushed.
- The PDQ(2/3) driver has been removed and is now being maintained out-of tree at https://github.com/m-labs/pdq. All SPI/USB driver layers, Mediator, CompoundPDQ and examples/documentation has been moved.
- The master now rotates log files at midnight, rather than based on log size.
- The results keys `start_time` and `run_time` are now stored as doubles of UNIX time, rather than ints. The file names are still based on local time.
- Packages are no longer available for 32-bit Windows.

4.6 ARTIQ-2

4.6.1 2.5

No further notes.

4.6.2 2.4

No further notes.

4.6.3 2.3

- When using conda, add the conda-forge channel before installing ARTIQ.

4.6.4 2.2

No further notes.

4.6.5 2.1

No further notes.

4.6.6 2.0

No further notes.

4.6.7 2.0rc2

No further notes.

4.6.8 2.0rc1

- The format of the influxdb pattern file is simplified. The procedure to edit patterns is also changed to modifying the pattern file and calling: `artiq_rpctool.py ::1 3248 call scan_patterns` (or restarting the bridge) The patterns can be converted to the new format using this code snippet:

```
from artiq.protocols import pyon
patterns = pyon.load_file("influxdb_patterns.pyon")
for p in patterns:
    print(p)
```

- The "GUI" has been renamed the "dashboard".
- When flashing NIST boards, use "-m nist_qcX" or "-m nist_clock" instead of just "-m qcX" or "-m clock" (#290).
- Applet command lines now use templates (e.g. $python) instead of formats (e.g. {python}).
- On Windows, GUI applications no longer open a console. For debugging purposes, the console messages can still be displayed by running the GUI applications this way:

```
python3.5 -m artiq.frontend.artiq_browser
python3.5 -m artiq.frontend.artiq_dashboard
```

  (you may need to replace python3.5 with python) Please always include the console output when reporting a GUI crash.

- The result folders are formatted "%Y-%m-%d/%H instead of "%Y-%m-%d/%H-%M". (i.e. grouping by day and then by hour, instead of by day and then by minute)
- The `parent` keyword argument of `HasEnvironment` (and `EnvExperiment`) has been replaced. Pass the parent as first argument instead.
- During experiment examination (and a fortiori repository scan), the values of all arguments are set to `None` regardless of any default values supplied.
- In the dashboard's experiment windows, partial or full argument recomputation takes into account the repository revision field.
- By default, `NumberValue` and `Scannable` infer the scale from the unit for common units.
- By default, artiq_client keeps the current persist flag on the master.
- GUI state files for the browser and the dashboard are stores in "standard" locations for each operating system. Those are `~/.config/artiq/2/artiq_*.pyon` on Linux and `C:\Users\<username>\AppData\Local\m-labs\artiq\2\artiq_*.pyon` on Windows 7.
- The position of the time cursor is kept across experiments and RTIO resets are manual and explicit (inter-experiment seamless handover).
- All integers manipulated by kernels are numpy integers (numpy.int32, numpy.int64). If you pass an integer as a RPC argument, the target function receives a numpy type.

4.7 ARTIQ-1

4.7.1 1.3

No further notes.

4.7.2 1.2

No further notes.

4.7.3 1.1

- TCA6424A.set converts the "outputs" value to little-endian before programming it into the registers.

4.7.4 1.0

No further notes.

4.7.5 1.0rc4

- setattr_argument and setattr_device add their key to kernel_invariants.

4.7.6 1.0rc3

- The HDF5 format has changed.
  - The datasets are located in the HDF5 subgroup `datasets`.
  - Datasets are now stored without additional type conversions and annotations from ARTIQ, trusting that h5py maps and converts types between HDF5 and python/numpy "as expected".
- NumberValue now returns an integer if `ndecimals` = 0, `scale` = 1 and `step` is integer.

4.7.7 1.0rc2

- The CPU speed in the pipistrello gateware has been reduced from 83 1/3 MHz to 75 MHz. This will reduce the achievable sustained pulse rate and latency accordingly. ISE was intermittently failing to meet timing (#341).
- set_dataset in broadcast mode no longer returns a Notifier. Mutating datasets should be done with mutate_dataset instead (#345).

4.7.8 1.0rc1

- Experiments (your code) should use `from artiq.experiment import *` (and not `from artiq import *` as previously)
- Core device flash storage has moved due to increased runtime size. This requires reflashing the runtime and the flash storage filesystem image or erase and rewrite its entries.
- `RTIOCollisionError` has been renamed to `RTIOCollision`
- the new API for DDS batches is:

```
with self.core_dds.batch:
    ...
```

  with `core_dds` a device of type `artiq.coredevice.dds.CoreDDS`. The dds_bus device should not be used anymore.
- LinearScan now supports scanning from high to low. Accordingly, its arguments `min/max` have been renamed to `start/stop` respectively. Same for RandomScan (even though there direction matters little).

CHAPTER

FIVE

ARTIQ REAL-TIME I/O CONCEPTS

The ARTIQ Real-Time I/O design employs several concepts to achieve its goals of high timing resolution on the nanosecond scale and low latency on the microsecond scale while still not sacrificing a readable and extensible language.

In a typical environment two very different classes of hardware need to be controlled. One class is the vast arsenal of diverse laboratory hardware that interfaces with and is controlled from a typical PC. The other is specialized real-time hardware that requires tight coupling and a low-latency interface to a CPU. The ARTIQ code that describes a given experiment is composed of two types of "programs": regular Python code that is executed on the host and ARTIQ *kernels* that are executed on a *core device*. The CPU that executes the ARTIQ kernels has direct access to specialized programmable I/O timing logic (part of the *gateware*). The two types of code can invoke each other and transitions between them are seamless.

The ARTIQ kernels do not interface with the real-time gateware directly. That would lead to imprecise, indeterminate, and generally unpredictable timing. Instead the CPU operates at one end of a bank of FIFO (first-in-first-out) buffers while the real-time gateware at the other end guarantees the *all or nothing* level of excellent timing precision. A FIFO for an output channel hold timestamps and event data describing when and what is to be executed. The CPU feeds events into this FIFO. A FIFO for an input channel contains timestamps and event data for events that have been recorded by the real-time gateware and are waiting to be read out by the CPU on the other end.

5.1 The timeline

The set of all input and output events on all channels constitutes the *timeline*. A high resolution wall clock (`rtio_counter`) counts clock cycles and causes output events to be executed when their timestamp matches the clock and input events to be recorded and stamped with the current clock value accordingly.

The kernel runtime environment maintains a timeline cursor (called `now`) used as the timestamp when output events are submitted to the FIFOs. This timeline cursor can be moved forward or backward on the timeline relative to its current value using *artiq.language.core.delay()* and *artiq.language.core.delay_mu()*, the later being a delay given in *machine units* as opposed to SI units. The absolute value of `now` on the timeline can be retrieved using *artiq.language.core.now_mu()* and it can be set using *artiq.language.core.at_mu()*. RTIO timestamps, the timeline cursor, and the `rtio_counter` wall clock are all relative to the core device startup/boot time. The wall clock keeps running across experiments.

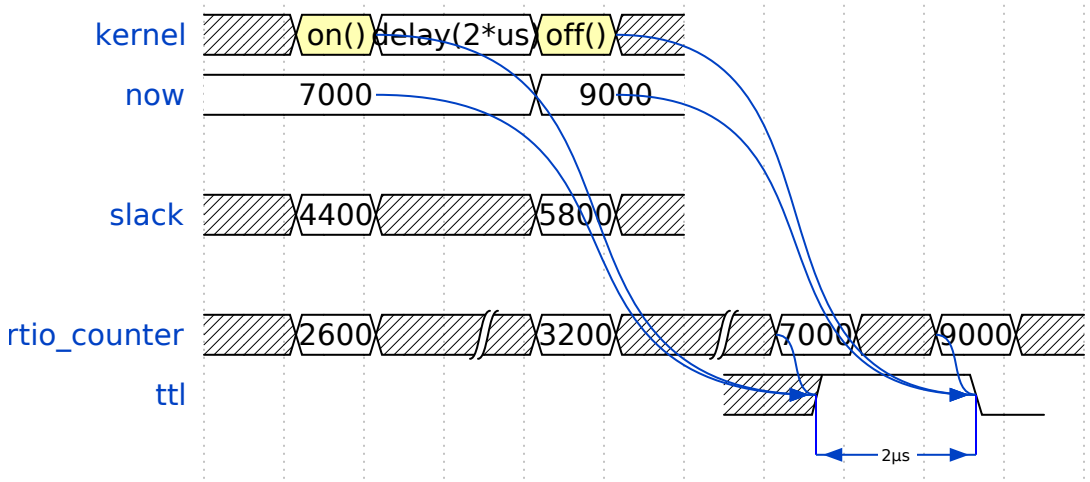
Absolute timestamps can be large numbers. They are represented internally as 64-bit integers with a resolution of typically a nanosecond and a range of hundreds of years. Conversions between such a large integer number and a floating point representation can cause loss of precision through cancellation. When computing the difference of absolute timestamps, use `self.core.mu_to_seconds(t2-t1)`, not `self.core.mu_to_seconds(t2)-self.core.mu_to_seconds(t1)` (see `artiq.coredevice.Core.mu_to_seconds()`). When accumulating time, do it in machine units and not in SI units, so that rounding errors do not accumulate.

The following basic example shows how to place output events on the timeline. It emits a precisely timed 2 µs pulse:

```
ttl.on()
delay(2*us)
ttl.off()
```

The device `ttl` represents a single digital output channel (*artiq.coredevice.ttl.TTLOut*). The *artiq.coredevice.ttl.TTLOut.on()* method places an rising edge on the timeline at the current cursor position (`now`). Then the cursor is moved forward 2 µs and a falling edge event is placed at the new cursor position. Then later, when the wall clock reaches the respective timestamps the RTIO gateware executes the two events.

The following diagram shows what is going on at the different levels of the software and gateware stack (assuming one machine unit of time is 1 ns):



The sequence is exactly equivalent to:

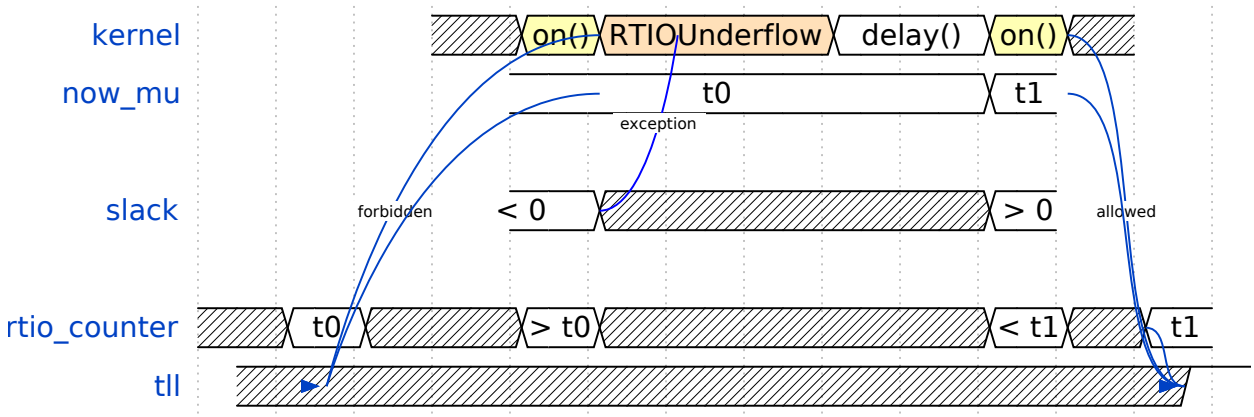
```
ttl.pulse(2*us)
```

The *artiq.coredevice.ttl.TTLOut.pulse()* method advances the timeline cursor (using `delay()`) while other methods such as *artiq.coredevice.ttl.TTLOut.on()*, *artiq.coredevice.ttl.TTLOut.off()*, `artiq.coredevice.ad9914.set()`. The latter are called *zero-duration* methods.

5.2 Underflow exceptions

An RTIO event must always be programmed with a timestamp in the future. In other words, the timeline cursor `now` must be after the current wall clock `rtio_counter`: the past can not be altered. The following example tries to place an rising edge event on the timeline. If the current cursor is in the past, an *artiq.coredevice.exceptions.RTIOUnderflow* exception is thrown. The experiment attempts to handle the exception by moving the cursor forward and repeating the programming of the rising edge:

```
try:
    ttl.on()
except RTIOUnderflow:
    # try again at the next mains cycle
    delay(16.6667*ms)
    ttl.on()
```



To track down `RTIOUnderflows` in an experiment there are a few approaches:

- Exception backtraces show where underflow has occurred while executing the code.
- The *integrated logic analyzer* shows the timeline context that lead to the exception. The analyzer is always active and supports plotting of RTIO slack. RTIO slack is the difference between timeline cursor and wall clock time (`now - rtio_counter`).

5.3 Sequence errors

A sequence error happens when the sequence of coarse timestamps cannot be supported by the gateware. For example, there may have been too many timeline rewinds.

Internally, the gateware stores output events in an array of FIFO buffers (the "lanes") and the timestamps in each lane must be strictly increasing. If an event with a decreasing or equal timestamp is submitted, the gateware selects the next lane, wrapping around if the final lane is reached. If this lane also contains an event with a timestamp beyond the one being submitted then a sequence error occurs. See this issue for a real-life example of how this works.

Notes:

- Strictly increasing timestamps never cause sequence errors.
- Configuring the gateware with more lanes for the RTIO core reduces the frequency of sequence errors.
- The number of lanes is a hard limit on the number of simultaneous RTIO output events.
- Whether a particular sequence of timestamps causes a sequence error or not is fully deterministic (starting from a known RTIO state, e.g. after a reset). Adding a constant offset to the whole sequence does not affect the result.
- Zero-duration methods (such as *`artiq.coredevice.ttl.TTLOut.on()`*) do not advance the timeline and so will consume additional lanes if they are scheduled simultaneously. Adding a tiny delay will prevent this (e.g. `delay_mu(np.int64(self.core.ref_multiplier))`, at least one coarse rtio cycle).

The offending event is discarded and the RTIO core keeps operating.

This error is reported asynchronously via the core device log: for performance reasons with DRTIO, the CPU does not wait for an error report from the satellite after writing an event. Therefore, it is not possible to raise an exception precisely.

5.4 Collisions

A collision happens when more than one event is submitted on a given channel with the same coarse timestamp, and that channel does not implement replacement behavior or the fine timestamps are different.

Coarse timestamps correspond to the RTIO system clock (typically around 125MHz) whereas fine timestamps correspond to the RTIO SERDES clock (typically around 1GHz). Different channels may have different ratios between the coarse and fine timestamp clock frequencies.

The offending event is discarded and the RTIO core keeps operating.

This error is reported asynchronously via the core device log: for performance reasons with DRTIO, the CPU does not wait for an error report from the satellite after writing an event. Therefore, it is not possible to raise an exception precisely.

5.5 Busy errors

A busy error happens when at least one output event could not be executed because the channel was already busy executing a previous event.

The offending event was discarded.

This error is reported asynchronously via the core device log.

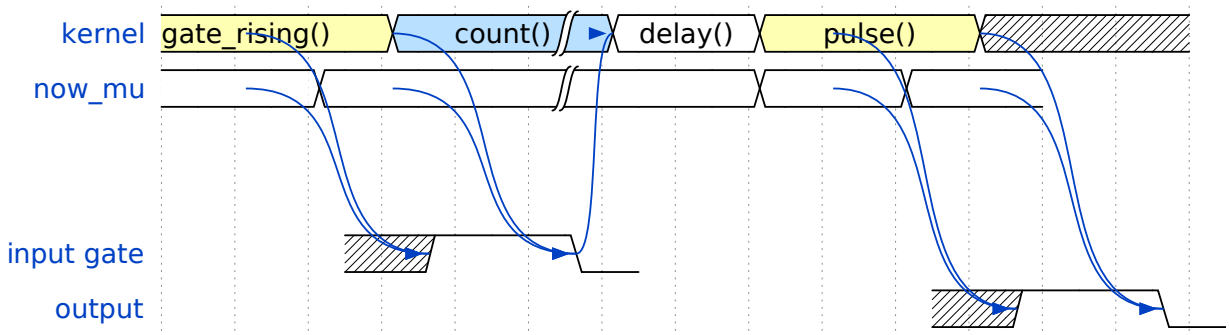
5.6 Input channels and events

Input channels detect events, timestamp them, and place them in a buffer for the experiment to read out. The following example counts the rising edges occurring during a precisely timed 500 ns interval. If more than 20 rising edges are received, it outputs a pulse:

```
if input.count(input.gate_rising(500*ns)) > 20:
    delay(2*us)
    output.pulse(500*ns)
```

The *artiq.coredevice.ttl.TTLInOut.count()* method of an input channel will often lead to a situation of negative slack (timeline cursor `now` smaller than the current wall clock `rtio_counter`): The *artiq.coredevice.ttl.TTLInOut.gate_rising()* method leaves the timeline cursor at the closing time of the gate. `count()` must necessarily wait until the gate closing event has actually been executed, at which point `rtio_counter > now`: `count()` synchronizes timeline cursor (`now`) and wall clock (`rtio_counter`). In these situations, a `delay()` is necessary to re-establish positive slack so that further output events can be placed.

Similar situations arise with methods such as *artiq.coredevice.ttl.TTLInOut.sample_get()* and *artiq.coredevice.ttl.TTLInOut.watch_done()*.



5.7 Overflow exceptions

The RTIO input channels buffer input events received while an input gate is open, or at certain points in time when using the sampling API (*artiq.coredevice.ttl.TTLInOut.sample_input()*). The events are kept in a FIFO until the CPU reads them out via e.g. *artiq.coredevice.ttl.TTLInOut.count()*, *artiq.coredevice.ttl.TTLInOut.timestamp_mu()* or *artiq.coredevice.ttl.TTLInOut.sample_get()*. If the FIFO is full and another event is coming in, this causes an overflow condition. The condition is converted into an *artiq.coredevice.exceptions.RTIOOverflow* exception that is raised on a subsequent invocation of one of the readout methods (e.g. `count()`, `timestamp_mu()`, `sample_get()`).

5.8 Seamless handover

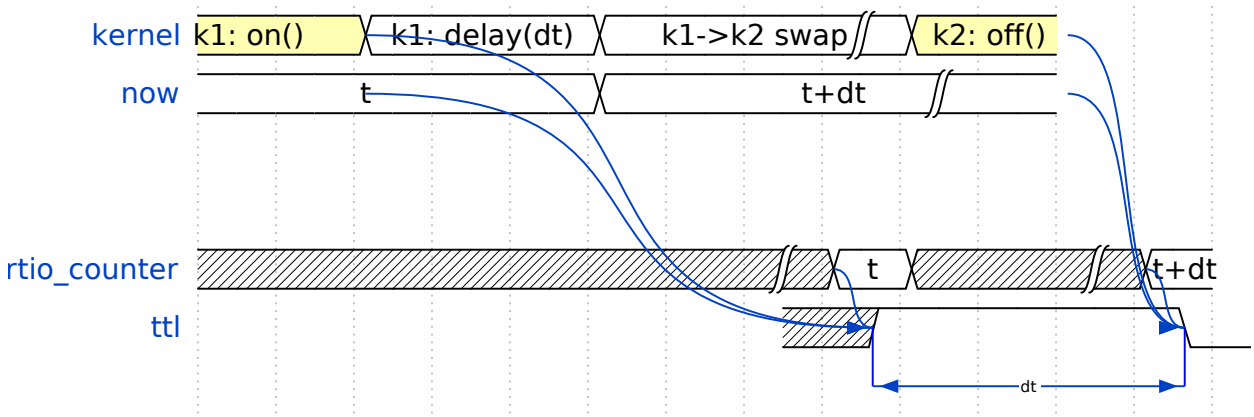
The timeline cursor persists across kernel invocations. This is demonstrated in the following example where a pulse is split across two kernels:

```
def run():
  k1()
  k2()

@kernel
def k1():
  ttl.on()
  delay(1*s)

@kernel
def k2():
  ttl.off()
```

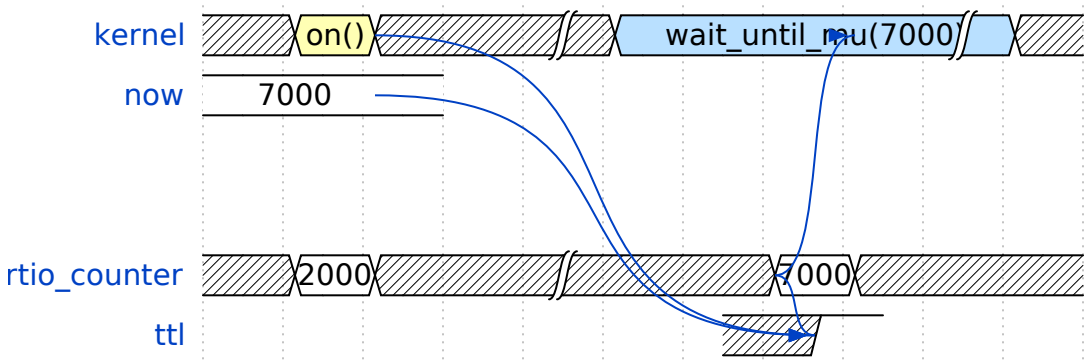
Here, `run()` calls `k1()` which exits leaving the cursor one second after the rising edge and `k2()` then submits a falling edge at that position.



5.9 Synchronization

The seamless handover of the timeline (cursor and events) across kernels and experiments implies that a kernel can exit long before the events it has submitted have been executed. If a previous kernel sets timeline cursor far in the future this effectively locks the system.

When a kernel should wait until all the events have been executed, use the *artiq.coredevice.core.Core.wait_until_mu()* with a timestamp after (or at) the last event:



In many cases, *now_mu()* will return an appropriate timestamp:

```
self.core.wait_until_mu(now_mu())
```

5.10 RTIO reset

The seamless handover also means that a kernel is not guaranteed to always be executed with positive slack. An experiment can face any of these circumstances (large positive slack, full FIFOs, or negative slack). Therefore, when switching experiments it can be adequate to clear the RTIO FIFOs and initialize the timeline cursor to "sometime in the near future" using *artiq.coredevice.core.Core.reset()*. The example idle kernel implements this mechanism. Since it never waits for any input, it will rapidly fill the output FIFOs and would produce a large positive slack. To avoid large positive slack and to accommodate for seamless handover the idle kernel will only run when no other experiment is pending and the example will wait before submitting events until there is significant negative slack.

CHAPTER

SIX

GETTING STARTED WITH THE CORE LANGUAGE

6.1 Connecting to the core device

As a very first step, we will turn on a LED on the core device. Create a file `led.py` containing the following:

```
from artiq.experiment import *


class LED(EnvExperiment):
    def build(self):
        self.setattr_device("core")
        self.setattr_device("led")

    @kernel
    def run(self):
        self.core.reset()
        self.led.on()
```

The central part of our code is our `LED` class, which derives from *artiq.language.environment.EnvExperiment*. Among other features, *EnvExperiment* calls our `build()` method and provides the *setattr_device()* method that interfaces to the device database to create the appropriate device drivers and make those drivers accessible as `self.core` and `self.led`. The *kernel()* decorator (`@kernel`) tells the system that the *run()* method must be compiled for and executed on the core device (instead of being interpreted and executed as regular Python code on the host). The decorator uses `self.core` internally, which is why we request the core device using *setattr_device()* like any other.

Copy the file `device_db.py` (containing the device database) from the `examples/master` folder of ARTIQ into the same directory as `led.py` (alternatively, you can use the `--device-db` option of `artiq_run`). You will probably want to set the IP address of the core device in `device_db.py` so that the computer can connect to it (it is the `host` parameter of the `comm` entry). See *The device database* for more information. The example device database is designed for the `nist_clock` hardware adapter on the KC705; see *FPGA board ports* for RTIO channel assignments if you need to adapt the device database to a different hardware platform.

Note: To obtain the examples, you can find where the ARTIQ package is installed on your machine with:

```
python3 -c "import artiq; print(artiq.__path__[0])"
```

Run your code using `artiq_run`, which is part of the ARTIQ front-end tools:

```
$ artiq_run led.py
```

The process should terminate quietly and the LED of the device should turn on. Congratulations! You have a basic ARTIQ system up and running.

6.2 Host/core device interaction (RPC)

A method or function running on the core device (which we call a "kernel") may communicate with the host by calling non-kernel functions that may accept parameters and may return a value. The "remote procedure call" (RPC) mechanisms handle automatically the communication between the host and the device of which function to call, with which parameters, and what the returned value is.

Modify the code as follows:

```
def input_led_state() -> TBool:
    return input("Enter desired LED state: ") == "1"

class LED(EnvExperiment):
    def build(self):
        self.setattr_device("core")
        self.setattr_device("led")

    @kernel
    def run(self):
        self.core.reset()
        s = input_led_state()
        self.core.break_realtime()
        if s:
            self.led.on()
        else:
            self.led.off()
```

You can then turn the LED off and on by entering 0 or 1 at the prompt that appears:

```
$ artiq_run led.py
Enter desired LED state: 1
$ artiq_run led.py
Enter desired LED state: 0
```

What happens is the ARTIQ compiler notices that the `input_led_state()` function does not have a `@kernel` decorator (*kernel()*) and thus must be executed on the host. When the core device calls it, it sends a request to the host to execute it. The host displays the prompt, collects user input, and sends the result back to the core device, which sets the LED state accordingly.

RPC functions must always return a value of the same type. When they return a value that is not `None`, the compiler should be informed in advance of the type of the value, which is what the `-> TBool` annotation is for.

Without the *break_realtime()* call, the RTIO events emitted by `self.led.on()` or `self.led.off()` would be scheduled at a fixed and very short delay after entering *run()*. These events would fail because the RPC to `input_led_state()` can take an arbitrary amount of time and therefore the deadline for submission of RTIO events would have long passed when `self.led.on()` or `self.led.off()` are called. The *break_realtime()* call is necessary to waive the real-time requirements of the LED state change. It advances the timeline far enough to ensure that events can meet the submission deadline.

6.3 Real-time Input/Output (RTIO)

The point of running code on the core device is the ability to meet demanding real-time constraints. In particular, the core device can respond to an incoming stimulus or the result of a measurement with a low and predictable latency. We will see how to use inputs later; first, we must familiarize ourselves with how time is managed in kernels.

Create a new file `rtio.py` containing the following:

```
from artiq.experiment import *


class Tutorial(EnvExperiment):
    def build(self):
        self.setattr_device("core")
        self.setattr_device("ttl0")

    @kernel
    def run(self):
        self.core.reset()
        self.ttl0.output()
        for i in range(1000000):
            delay(2*us)
            self.ttl0.pulse(2*us)
```

In its `build()` method, the experiment obtains the core device and a TTL device called `ttl0` as defined in the device database. In ARTIQ, TTL is used roughly synonymous with "a single generic digital signal" and does not refer to a specific signaling standard or voltage/current levels.

When *`run()`*, the experiment first ensures that `ttl0` is in output mode and actively driving the device it is connected to. Bidirectional TTL channels (i.e. *`TTLInOut`*) are in input (high impedance) mode by default, output-only TTL channels (*`TTLOut`*) are always in output mode. There are no input-only TTL channels.

The experiment then drives one million 2 µs long pulses separated by 2 µs each. Connect an oscilloscope or logic analyzer to TTL0 and run `artiq_run.py rtio.py`. Notice that the generated signal's period is precisely 4 µs, and that it has a duty cycle of precisely 50%. This is not what you would expect if the delay and the pulse were implemented with register-based general purpose input output (GPIO) that is CPU-controlled. The signal's period would depend on CPU speed, and overhead from the loop, memory management, function calls, etc, all of which are hard to predict and variable. Any asymmetry in the overhead would manifest itself in a distorted and variable duty cycle.

Instead, inside the core device, output timing is generated by the gateware and the CPU only programs switching commands with certain timestamps that the CPU computes.

This guarantees precise timing as long as the CPU can keep generating timestamps that are increasing fast enough. In case it fails to do that (and attempts to program an event with a timestamp smaller than the current RTIO clock timestamp), a *`RTIOUnderflow`* exception is raised. The kernel causing it may catch it (using a regular `try... except...` construct), or it will be propagated to the host.

Try reducing the period of the generated waveform until the CPU cannot keep up with the generation of switching events and the underflow exception is raised. Then try catching it:

```
from artiq.experiment import *


def print_underflow():
    print("RTIO underflow occured")
```

(continues on next page)

(continued from previous page)

```
class Tutorial(EnvExperiment):
    def build(self):
        self.setattr_device("core")
        self.setattr_device("ttl0")

    @kernel
    def run(self):
        self.core.reset()
        try:
            for i in range(1000000):
                self.ttl0.pulse(...)
                delay(...)
        except RTIOUnderflow:
            print_underflow()
```

6.4 Parallel and sequential blocks

It is often necessary that several pulses overlap one another. This can be expressed through the use of `with parallel` constructs, in which the events generated by the individual statements are executed at the same time. The duration of the `parallel` block is the duration of its longest statement.

Try the following code and observe the generated pulses on a 2-channel oscilloscope or logic analyzer:

```
for i in range(1000000):
    with parallel:
        self.ttl0.pulse(2*us)
        self.ttl1.pulse(4*us)
    delay(4*us)
```

ARTIQ can implement `with parallel` blocks without having to resort to any of the typical parallel processing approaches. It simply remembers the position on the timeline when entering the `parallel` block and then seeks back to that position after submitting the events generated by each statement. In other words, the statements in the `parallel` block are actually executed sequentially, only the RTIO events generated by them are scheduled to be executed in parallel. Note that if a statement takes a lot of CPU time to execute (this different from the events scheduled by a statement taking a long time), it may cause a subsequent statement to miss the deadline for timely submission of its events. This then causes a *RTIOUnderflow* exception to be raised.

Within a parallel block, some statements can be made sequential again using a `with sequential` construct. Observe the pulses generated by this code:

```
for i in range(1000000):
    with parallel:
        with sequential:
            self.ttl0.pulse(2*us)
            delay(1*us)
            self.ttl0.pulse(1*us)
        self.ttl1.pulse(4*us)
    delay(4*us)
```

Particular care needs to be taken when working with `parallel` blocks in cases where a large number of RTIO events are generated as it possible to create sequencing errors (*RTIO sequence error*). Sequence errors do not halt execution of the kernel for performance reasons and instead are reported in the core log. If the `aqctl_corelog` process has been

started with `artiq_ctlmgr`, then these errors will be posted to the master log. However, if an experiment is executed through `artiq_run`, these errors will not be visible outside of the core log.

A sequence error is caused when the scalable event dispatcher (SED) cannot queue an RTIO event due to its timestamp being the same as or earlier than another event in its queue. By default, the SED has 8 lanes which allows `parallel` events to work without sequence errors in most cases, however if many (>8) events are queued with conflicting timestamps this error can surface.

These errors can usually be overcome by reordering the generation of the events. Alternatively, the number of SED lanes can be increased in the gateware.

6.5 RTIO analyzer

The core device records the real-time I/O waveforms into a circular buffer. It is possible to dump any Python object so that it appears alongside the waveforms using the `rtio_log` function, which accepts a channel name (i.e. a log target) as the first argument:

```
from artiq.experiment import *


class Tutorial(EnvExperiment):
    def build(self):
        self.setattr_device("core")
        self.setattr_device("ttl0")

    @kernel
    def run(self):
        self.core.reset()
        for i in range(100):
            self.ttl0.pulse(...)
            rtio_log("ttl0", "i", i)
            delay(...)
```

Afterwards, the recorded data can be extracted and written to a VCD file using `artiq_coreanalyzer -w rtio.vcd` (see: *Core device RTIO analyzer tool*). VCD files can be viewed using third-party tools such as GtkWave.

6.6 Direct Memory Access (DMA)

DMA allows you to store fixed sequences of pulses in system memory, and have the DMA core in the FPGA play them back at high speed. Pulse sequences that are too fast for the CPU (i.e. would cause RTIO underflows) can still be generated using DMA. The only modification of the sequence that the DMA core supports is shifting it in time (so it can be played back at any position of the timeline), everything else is fixed at the time of recording the sequence.

Try this:

```
from artiq.experiment import *


class DMAPulses(EnvExperiment):
    def build(self):
        self.setattr_device("core")
        self.setattr_device("core_dma")
```

(continues on next page)

(continued from previous page)

```
        self.setattr_device("ttl0")

    @kernel
    def record(self):
        with self.core_dma.record("pulses"):
            # all RTIO operations now go to the "pulses"
            # DMA buffer, instead of being executed immediately.
            for i in range(50):
                self.ttl0.pulse(100*ns)
                delay(100*ns)

    @kernel
    def run(self):
        self.core.reset()
        self.record()
        # prefetch the address of the DMA buffer
        # for faster playback trigger
        pulses_handle = self.core_dma.get_handle("pulses")
        self.core.break_realtime()
        while True:
            # execute RTIO operations in the DMA buffer
            # each playback advances the timeline by 50*(100+100) ns
            self.core_dma.playback_handle(pulses_handle)
```

CHAPTER

SEVEN

COMPILER

The ARTIQ compiler transforms the Python code of the kernels into machine code executable on the core device. It is invoked automatically when calling a function that uses the `@kernel` decorator.

7.1 Supported Python features

A number of Python features can be used inside a kernel for compilation and execution on the core device. They include `for` and `while` loops, conditionals (`if`, `else`, `elif`), functions, exceptions, and statically typed variables of the following types:

- Booleans
- 32-bit signed integers (default size)
- 64-bit signed integers (use `numpy.int64` to convert)
- Double-precision floating point numbers
- Lists of any supported types
- String constants
- User-defined classes, with attributes of any supported types (attributes that are not used anywhere in the kernel are ignored)

For a demonstration of some of these features, see the `mandelbrot.py` example.

When several instances of a user-defined class are referenced from the same kernel, every attribute must have the same type in every instance of the class.

7.2 Remote procedure calls

Kernel code can call host functions without any additional ceremony. However, such functions are assumed to return *None*, and if a value other than *None* is returned, an exception is raised. To call a host function returning a value other than *None* its return type must be annotated using the standard Python syntax, e.g.:

```
def return_four() -> TInt32:
    return 4
```

The Python types correspond to ARTIQ type annotations as follows:

Python	ARTIQ
NoneType	TNone
bool	TBool
int	TInt32 or TInt64
float	TFloat
str	TStr
list of T	TList(T)
NumPy array	TArray(T, num_dims)
range	TRange32, TRange64
numpy.int32	TInt32
numpy.int64	TInt64
numpy.float64	TFloat

7.3 Pitfalls

The ARTIQ compiler accepts *nearly* a strict subset of Python 3. However, by necessity there is a number of differences that can lead to bugs.

Arbitrary-length integers are not supported at all on the core device; all integers are either 32-bit or 64-bit. This especially affects calculations that result in a 32-bit signed overflow; if the compiler detects a constant that doesn't fit into 32 bits, the entire expression will be upgraded to 64-bit arithmetics, however if all constants are small, 32-bit arithmetics will be used even if the result will overflow. Overflows are not detected.

The result of calling the builtin `round` function is different when used with the builtin `float` type and the `numpy.float64` type on the host interpreter; `round(1.0)` returns an integer value 1, whereas `round(numpy.float64(1.0))` returns a floating point value `numpy.float64(1.0)`. Since both `float` and `numpy.float64` are mapped to the builtin `float` type on the core device, this can lead to problems in functions marked `@portable`; the workaround is to explicitly cast the argument of `round` to `float`: `round(float(numpy.float64(1.0)))` returns an integer on the core device as well as on the host interpreter.

Empty lists do not have valid list element types, so they cannot be used in the kernel.

7.4 Asynchronous RPCs

If an RPC returns no value, it can be invoked in a way that does not block until the RPC finishes execution, but only until it is queued. (Submitting asynchronous RPCs too rapidly, as well as submitting asynchronous RPCs with arguments that are too large, can still block until completion.)

To define an asynchronous RPC, use the `@rpc` annotation with a flag:

```
@rpc(flags={"async"})
def record_result(x):
    self.results.append(x)
```

7.5 Additional optimizations

The ARTIQ compiler runs many optimizations, most of which perform well on code that has pristine Python semantics. It also contains more powerful, and more invasive, optimizations that require opt-in to activate.

7.5.1 Fast-math flags

The compiler does not normally perform algebraically equivalent transformations on floating-point expressions, because this can dramatically change the result. However, it can be instructed to do so if all of the following is true:

- Arguments and results will not be not-a-number or infinity values;
- The sign of a zero value is insignificant;
- Any algebraically equivalent transformations, such as reassociation or replacing division with multiplication by reciprocal, are legal to perform.

If this is the case for a given kernel, a `fast-math` flag can be specified to enable more aggressive optimization for this specific kernel:

```
@kernel(flags={"fast-math"})
def calculate(x, y, z):
    return x * z + y * z
```

This flag particularly benefits loops with I/O delays performed in fractional seconds rather than machine units, as well as updates to DDS phase and frequency.

7.5.2 Kernel invariants

The compiler attempts to remove or hoist out of loops any redundant memory load operations, as well as propagate known constants into function bodies, which can enable further optimization. However, it must make conservative assumptions about code that it is unable to observe, because such code can change the value of the attribute, making the optimization invalid.

When an attribute is known to never change while the kernel is running, it can be marked as a *kernel invariant* to enable more aggressive optimization for this specific attribute.

```
class Converter:
    kernel_invariants = {"ratio"}

    def __init__(self, ratio=1.0):
        self.ratio = ratio

    @kernel
    def convert(self, value):
        return value * self.ratio ** 2
```

In the synthetic example above, the compiler will be able to detect that the result of evaluating `self.ratio ** 2` never changes and replace it with a constant, removing an expensive floating-point operation.

```
class Worker:
    kernel_invariants = {"interval"}

    def __init__(self, interval=1.0*us):
```

(continues on next page)

(continued from previous page)

```
        self.interval = interval

    def work(self):
        # something useful

class Looper:
    def __init__(self, worker):
        self.worker = worker

    @kernel
    def loop(self):
        for _ in range(100):
            delay(self.worker.interval / 5.0)
            self.worker.work()
```

In the synthetic example above, the compiler will be able to detect that the result of evaluating `self.interval / 5.0` never changes, even though it neither knows the value of `self.worker.interval` beforehand nor can it see through the `self.worker.work()` function call, and hoist the expensive floating-point division out of the loop, transforming the code for `loop` into an equivalent of the following:

```
@kernel
def loop(self):
    precomputed_delay_mu = self.core.seconds_to_mu(self.worker.interval / 5.0)
    for _ in range(100):
        delay_mu(precomputed_delay_mu)
        self.worker.work()
```

CHAPTER

EIGHT

GETTING STARTED WITH THE MANAGEMENT SYSTEM

The management system is the high-level part of ARTIQ that schedules the experiments, distributes and stores the results, and manages devices and parameters.

The manipulations described in this tutorial can be carried out using a single computer, without any special hardware.

8.1 Starting your first experiment with the master

In the previous tutorial, we used the `artiq_run` utility to execute our experiments, which is a simple stand-alone tool that bypasses the ARTIQ management system. We will now see how to run an experiment using the master (the central program in the management system that schedules and executes experiments) and the dashboard (that connects to the master and controls it).

First, create a folder `~/artiq-master` and copy the file `device_db.py` (containing the device database) found in the `examples/master` directory from the ARTIQ sources. The master uses those files in the same way as `artiq_run`.

Then create a `~/artiq-master/repository` sub-folder to contain experiments. The master scans this `repository` folder to determine what experiments are available (the name of the folder can be changed using `-r`).

Create a very simple experiment in `~/artiq-master/repository` and save it as `mgmt_tutorial.py`:

```
from artiq.experiment import *


class MgmtTutorial(EnvExperiment):
    """Management tutorial"""
    def build(self):
        pass  # no devices used

    def run(self):
        print("Hello World")
```

Start the master with:

```
$ cd ~/artiq-master
$ artiq_master
```

This last command should not return, as the master keeps running.

Now, start the dashboard with the following commands in another terminal:

```
$ cd ~
$ artiq_dashboard
```

Note: The `artiq_dashboard` program uses a file called `artiq_dashboard.pyon` in the current directory to save and restore the GUI state (window/dock positions, last values entered by the user, etc.).

The dashboard should display the list of experiments from the repository folder in a dock called "Explorer". There should be only the experiment we created. Select it and click "Submit", then look at the "Log" dock for the output from this simple experiment.

Note: Multiple clients may be connected at the same time, possibly on different machines, and will be synchronized. See the `-s` option of `artiq_dashboard` and the `--bind` option of `artiq_master` to use the network. Both IPv4 and IPv6 are supported.

8.2 Adding an argument

Experiments may have arguments whose values can be set in the dashboard and used in the experiment's code. Modify the experiment as follows:

```
def build(self):
    self.setattr_argument("count", NumberValue(ndecimals=0, step=1))

def run(self):
    for i in range(self.count):
        print("Hello World", i)
```

`NumberValue` represents a floating point numeric argument. There are many other types, see *artiq.language.environment* and *artiq.language.scan*.

Use the command-line client to trigger a repository rescan:

```
artiq_client scan-repository
```

The dashboard should now display a spin box that allows you to set the value of the `count` argument. Try submitting the experiment as before.

8.3 Setting up Git integration

So far, we have used the bare filesystem for the experiment repository, without any version control. Using Git to host the experiment repository helps with the tracking of modifications to experiments and with the traceability of a result to a particular version of an experiment.

Note: The workflow we will describe in this tutorial corresponds to a situation where the ARTIQ master machine is also used as a Git server where multiple users may push and pull code. The Git setup can be customized according to your needs; the main point to remember is that when scanning or submitting, the ARTIQ master uses the internal Git data (*not* any working directory that may be present) to fetch the latest *fully completed commit* at the repository's head.

We will use the current `repository` folder as working directory for making local modifications to the experiments, move it away from the master data directory, and create a new `repository` folder that holds the Git data used by the master. Stop the master with Ctrl-C and enter the following commands:

```
$ cd ~/artiq-master
$ mv repository ~/artiq-work
$ mkdir repository
$ cd repository
$ git init --bare
```

Now, push data to into the bare repository. Initialize a regular (non-bare) Git repository into our working directory:

```
$ cd ~/artiq-work
$ git init
```

Then commit our experiment:

```
$ git add mgmt_tutorial.py
$ git commit -m "First version of the tutorial experiment"
```

and finally, push the commit into the master's bare repository:

```
$ git remote add origin ~/artiq-master/repository
$ git push -u origin master
```

Start the master again with the `-g` flag, telling it to treat the contents of the `repository` folder (not `artiq-work`) as a bare Git repository:

```
$ cd ~/artiq-master
$ artiq_master -g
```

Note: You need at least one commit in the repository before you can start the master.

There should be no errors displayed, and if you start the GUI again, you will find the experiment there.

To complete the master configuration, we must tell Git to make the master rescan the repository when new data is added to it. Create a file `~/artiq-master/repository/hooks/post-receive` with the following contents:

```
#!/bin/sh
artiq_client scan-repository --async
```

Then set the execution permission on it:

```
$ chmod 755 ~/artiq-master/repository/hooks/post-receive
```

Note: Remote machines may also push and pull into the master's bare repository using e.g. Git over SSH.

Let's now make a modification to the experiment. In the source present in the working directory, add an exclamation mark at the end of "Hello World". Before committing it, check that the experiment can still be executed correctly by running it directly from the filesystem using:

```
$ artiq_client submit ~/artiq-work/mgmt_tutorial.py
```

Note: You may also use the "Open file outside repository" feature of the GUI, by right-clicking on the explorer.

Note: Submitting an experiment from the repository using the `artiq_client` command-line tool is done using the `-R` flag.

Verify the log in the GUI. If you are happy with the result, commit the new version and push it into the master's repository:

```
$ cd ~/artiq-work
$ git commit -a -m "More enthusiasm"
$ git push
```

Note: Notice that commands other than `git push` are not needed anymore.

The master should now run the new version from its repository.

As an exercise, add another experiment to the repository, commit and push the result, and verify that it appears in the GUI.

8.4 Datasets

Modify the `run()` method of the experiment as follows:

```
def run(self):
    self.set_dataset("parabola", np.full(self.count, np.nan), broadcast=True)
    for i in range(self.count):
        self.mutate_dataset("parabola", i, i*i)
        time.sleep(0.5)
```

Note: You need to import the `time` module, and the `numpy` module as `np`.

Commit, push and submit the experiment as before. Go to the "Datasets" dock of the GUI and observe that a new dataset has been created. We will now create a new XY plot showing this new result.

Plotting in the ARTIQ dashboard is achieved by programs called "applets". Applets are independent programs that add simple GUI features and are run as separate processes (to achieve goals of modularity and resilience against poorly written applets). Users may write their own applets, or use those supplied with ARTIQ (in the `artiq.applets` module) that cover basic plotting.

Applets are configured through their command line to select parameters such as the names of the datasets to plot. The list of command-line options can be retrieved using the `-h` option; for example you can run `python3 -m artiq.applets.plot_xy -h` in a terminal.

In our case, create a new applet from the XY template by right-clicking on the applet list, and edit the applet command line so that it retrieves the `parabola` dataset (the command line should now be `${artiq_applet}plot_xy parabola`). Run the experiment again, and observe how the points are added one by one to the plot.

After the experiment has finished executing, the results are written to a HDF5 file that resides in `~/artiq-master/results/<date>/<hour>`. Open that file with HDFView or h5dump, and observe the data we just generated as well as the Git commit ID of the experiment (a hexadecimal hash such as `947acb1f90ae1b8862efb489a9cc29f7d4e0c645` that represents the data at a particular time in the Git repository). The list of Git commit IDs can be found using the `git log` command in `~/artiq-work`.

Note: HDFView and h5dump are third-party tools not supplied with ARTIQ.

CHAPTER

NINE

CORE DEVICE

The core device is a FPGA-based hardware component that contains a softcore CPU tightly coupled with the so-called RTIO core that provides precision timing. The CPU executes Python code that is statically compiled by the ARTIQ compiler, and communicates with the core device peripherals (TTL, DDS, etc.) over the RTIO core. This architecture provides high timing resolution, low latency, low jitter, high level programming capabilities, and good integration with the rest of the Python experiment code.

While it is possible to use all the other parts of ARTIQ (controllers, master, GUI, dataset management, etc.) without a core device, many experiments require it.

9.1 Flash storage

The core device contains some flash space that can be used to store configuration data.

This storage area is used to store the core device MAC address, IP address and even the idle kernel.

The flash storage area is one sector (typically 64 kB) large and is organized as a list of key-value records.

This flash storage space can be accessed by using `artiq_coremgmt` (see: *Core device management tool*).

9.2 FPGA board ports

All boards have a serial interface running at 115200bps 8-N-1 that can be used for debugging.

9.2.1 Kasli

Kasli is a versatile core device designed for ARTIQ as part of the Sinara family of boards. All variants support interfacing to various EEM daughterboards (TTL, DDS, ADC, DAC…) connected directly to it.

Standalone variants

Kasli is connected to the network using a 1000Base-X SFP module. No-name BiDi (1000Base-BX) modules have been used successfully. The SFP module for the network should be installed into the SFP0 cage. The other SFP cages are not used.

The RTIO clock frequency is 125MHz or 150MHz, which is generated by the Si5324.

DRTIO master variants

Kasli can be used as a DRTIO master that provides local RTIO channels and can additionally control one DRTIO satellite.

The RTIO clock frequency is 125MHz or 150MHz, which is generated by the Si5324. The DRTIO line rate is 2.5Gbps or 3Gbps.

As with the standalone configuration, the SFP module for the Ethernet network should be installed into the SFP0 cage. The DRTIO connections are on SFP1 and SFP2, and optionally on the SATA connector.

DRTIO satellite/repeater variants

Kasli can be used as a DRTIO satellite with a 125MHz or 150MHz RTIO clock and a 2.5Gbps or 3Gbps DRTIO line rate.

The DRTIO upstream connection is on SFP0 or optionally on the SATA connector, and the remaining SFPs are downstream ports.

9.2.2 KC705

An alternative target board for the ARTIQ core device is the KC705 development board from Xilinx. It supports the NIST CLOCK and QC2 hardware (FMC).

Common problems

- The SW13 switches on the board need to be set to 00001.
- When connected, the CLOCK adapter breaks the JTAG chain due to TDI not being connected to TDO on the FMC mezzanine.
- On some boards, the JTAG USB connector is not correctly soldered.

VADJ

With the NIST CLOCK and QC2 adapters, for safe operation of the DDS buses (to prevent damage to the IO banks of the FPGA), the FMC VADJ rail of the KC705 should be changed to 3.3V. Plug the Texas Instruments USB-TO-GPIO PMBus adapter into the PMBus connector in the corner of the KC705 and use the Fusion Digital Power Designer software to configure (requires Windows). Write to chip number U55 (address 52), channel 4, which is the VADJ rail, to make it 3.3V instead of 2.5V. Power cycle the KC705 board to check that the startup voltage on the VADJ rail is now 3.3V.

NIST CLOCK

With the CLOCK hardware, the TTL lines are mapped as follows:

RTIO channel	TTL line	Capability
3,7,11,15	TTL3,7,11,15	Input+Output
0-2,4-6,8-10,12-14	TTL0-2,4-6,8-10,12-14	Output
16	PMT0	Input
17	PMT1	Input
18	SMA_GPIO_N	Input+Output
19	LED	Output
20	AMS101_LDAC_B	Output
21	LA32_P	Clock

The board has RTIO SPI buses mapped as follows:

RTIO channel	CS_N	MOSI	MISO	CLK
22	AMS101_CS_N	AMS101_MOSI		AMS101_CLK
23	SPI0_CS_N	SPI0_MOSI	SPI0_MISO	SPI0_CLK
24	SPI1_CS_N	SPI1_MOSI	SPI1_MISO	SPI1_CLK
25	SPI2_CS_N	SPI2_MOSI	SPI2_MISO	SPI2_CLK
26	MMC_SPI_CS_N	MMC_SPI_MOSI	MMC_SPI_MISO	MMC_SPI_CLK

The DDS bus is on channel 27.

NIST QC2

With the QC2 hardware, the TTL lines are mapped as follows:

RTIO channel	TTL line	Capability
0-39	TTL0-39	Input+Output
40	SMA_GPIO_N	Input+Output
41	LED	Output
42	AMS101_LDAC_B	Output
43, 44	CLK0, CLK1	Clock

The board has RTIO SPI buses mapped as follows:

RTIO channel	CS_N	MOSI	MISO	CLK
45	AMS101_CS_N	AMS101_MOSI		AMS101_CLK
46	SPI0_CS_N	SPI0_MOSI	SPI0_MISO	SPI0_CLK
47	SPI1_CS_N	SPI1_MOSI	SPI1_MISO	SPI1_CLK
48	SPI2_CS_N	SPI2_MOSI	SPI2_MISO	SPI2_CLK
49	SPI3_CS_N	SPI3_MOSI	SPI3_MISO	SPI3_CLK

There are two DDS buses on channels 50 (LPC, DDS0-DDS11) and 51 (HPC, DDS12-DDS23).

The QC2 hardware uses TCA6424A I2C I/O expanders to define the directions of its TTL buffers. There is one such expander per FMC card, and they are selected using the PCA9548 on the KC705.

To avoid I/O contention, the startup kernel should first program the TCA6424A expanders and then call `output()` on all `TTLInOut` channels that should be configured as outputs.

See *artiq.coredevice.i2c* for more details.

Clocking

The KC705 in standalone variants supports an internal 125 MHz RTIO clock (based on its crystal oscillator, or external reference for PLL for DRTIO variants) and an external clock, that can be selected using the `rtio_clock` configuration entry. Valid values are:

- `int_125` - internal crystal oscillator, 125 MHz output (default),
- `ext0_bypass` - external clock.

KC705 in DRTIO variants and Kasli generates the RTIO clock using a PLL locked either to an internal crystal or to an external frequency reference. Valid values are:

- `int_125` - internal crystal oscillator using PLL, 125 MHz output (default),
- `int_100` - internal crystal oscillator using PLL, 100 MHz output,
- `int_150` - internal crystal oscillator using PLL, 150 MHz output,
- `ext0_synth0_10to125` - external 10 MHz reference using PLL, 125 MHz output,
- `ext0_synth0_100to125` - external 100 MHz reference using PLL, 125 MHz output,
- `ext0_synth0_125to125` - external 125 MHz reference using PLL, 125 MHz output,
- `ext0_bypass`, `ext0_bypass_125`, `ext0_bypass_100` - external clock - with explicit aliases available.

The selected option can be observed in the core device boot logs.

Options `rtio_clock=int_XXX` and `rtio_clock=ext0_synth0_XXXXX` generate the RTIO clock using a PLL locked either to an internal crystal or to an external frequency reference (depending on exact option). `rtio_clock=ext0_bypass` bypasses that PLL and the user must supply the RTIO clock (typically 125 MHz) at the Kasli front panel SMA input. Bypassing the PLL ensures the skews between input clock, Kasli downstream clock outputs, and RTIO clock are deterministic accross reboots of the system. This is useful when phase determinism is required in situtations where the reference clock fans out to other devices before reaching Kasli.

CHAPTER

TEN

MANAGEMENT SYSTEM

The management system described below is optional: experiments can be run one by one using `artiq_run`, and the controllers can run stand-alone (without a controller manager). For their very first steps with ARTIQ or in simple or particular cases, users do not need to deploy the management system.

10.1 Components

10.1.1 Master

The *master* is responsible for managing the parameter and device databases, the experiment repository, scheduling and running experiments, archiving results, and distributing real-time results.

The master is a headless component, and one or several clients (command-line or GUI) use the network to interact with it.

10.1.2 Controller manager

Controller managers (started using the `artiq_ctlmgr` command that is part of the `artiq-comtools` package) are responsible for running and stopping controllers on a machine. There is one controller manager per network node that runs controllers.

A controller manager connects to the master and uses the device database to determine what controllers need to be run. Changes in the device database are tracked by the manager and controllers are started and stopped accordingly.

Controller managers use the local network address of the connection to the master to filter the device database and run only those controllers that are allocated to the current node. Hostname resolution is supported.

Warning: With some network setups, the current machine's hostname without the domain name resolves to a localhost address (127.0.0.1 or even 127.0.1.1). If you wish to use controllers across a network, make sure that the hostname you provide resolves to an IP address visible on the network (e.g. try providing the full hostname including the domain name).

10.1.3 Command-line client

The *command-line client* connects to the master and permits modification and monitoring of the databases, monitoring the experiment schedule and log, and submitting experiments.

10.1.4 Dashboard

The *dashboard* connects to the master and is the main way of interacting with it. The main features of the dashboard are scheduling of experiments, setting of their arguments, examining the schedule, displaying real-time results, and debugging TTL and DDS channels in real time.

10.2 Experiment scheduling

10.2.1 Basics

To use hardware resources more efficiently, potentially compute-intensive pre-computation and analysis phases of other experiments are executed in parallel with the body of the current experiment that accesses the hardware.

See also:

These steps are implemented in `Experiment`. However, user-written experiments should usually derive from (subclass) `artiq.language.environment.EnvExperiment`.

Experiments are divided into three phases that are programmed by the user:

1. The **preparation** stage, that pre-fetches and pre-computes any data that necessary to run the experiment. Users may implement this stage by overloading the `prepare()` method. It is not permitted to access hardware in this stage, as doing so may conflict with other experiments using the same devices.
2. The **running** stage, that corresponds to the body of the experiment, and typically accesses hardware. Users must implement this stage and overload the `run()` method.
3. The **analysis** stage, where raw results collected in the running stage are post-processed and may lead to updates of the parameter database. This stage may be implemented by overloading the `analyze()` method.

Note: Only the `run()` method implementation is mandatory; if the experiment does not fit into the pipelined scheduling model, it can leave one or both of the other methods empty (which is the default).

The three phases of several experiments are then executed in a pipelined manner by the scheduler in the ARTIQ master: experiment A executes its preparation stage, then experiment A executes its running stage while experiment B executes its preparation stage, and so on.

Note: The next experiment (B) may start `run()`ing before all events placed into (core device) RTIO buffers by the previous experiment (A) have been executed. These events can then execute while experiment B is `run()`ing. Using `reset()` clears the RTIO buffers, discarding pending events, including those left over from A.

Interactions between events of different experiments can be avoided by preventing the `run()` method of experiment A from returning until all events have been executed. This is discussed in the section on RTIO *Synchronization*.

10.2.2 Priorities and timed runs

When determining what experiment to begin executing next (i.e. entering the preparation stage), the scheduling looks at the following factors, by decreasing order of precedence:

1. Experiments may be scheduled with a due date. If there is one and it is not reached yet, the experiment is not eligible for preparation.
2. The integer priority value specified by the user.
3. The due date itself. The earlier the due date, the earlier the experiment is scheduled.
4. The run identifier (RID), an integer that is incremented at each experiment submission. This ensures that, all other things being equal, experiments are scheduled in the same order as they are submitted.

10.2.3 Pauses

In the run stage, an experiment may yield to the scheduler by calling the `pause()` method of the scheduler. If there are other experiments with higher priority (e.g. a high-priority timed experiment has reached its due date), they are preemptively executed, and then `pause()` returns. Otherwise, `pause()` returns immediately. To check whether `pause()` would in fact *not* return immediately, use *artiq.master.scheduler.Scheduler.check_pause()*.

The experiment must place the hardware in a safe state and disconnect from the core device (typically, by calling `self.core.comm.close()` from the kernel, which is equivalent to `artiq.coredevice.core.Core.close()`) before calling `pause()`.

Accessing the `pause()` and *check_pause()* methods is done through a virtual device called `scheduler` that is accessible to all experiments. The scheduler virtual device is requested like regular devices using *get_device()* (`self.get_device()`) or *setattr_device()* (`self.setattr_device()`).

check_pause() can be called (via RPC) from a kernel, but `pause()` must not.

10.2.4 Multiple pipelines

Multiple pipelines can operate in parallel inside the same master. It is the responsibility of the user to ensure that experiments scheduled in one pipeline will never conflict with those of another pipeline over resources (e.g. same devices).

Pipelines are identified by their name, and are automatically created (when an experiment is scheduled with a pipeline name that does not exist) and destroyed (when they run empty).

10.3 Git integration

The master may use a Git repository for the storage of experiment source code. Using Git has many advantages. For example, each result file (HDF5) contains the commit ID corresponding to the exact source code that produced it, which helps reproducibility.

Even though the master also supports non-bare repositories, it is recommended to use a bare repository so that it can easily support push transactions from clients. Create it with e.g.:

```
$ mkdir experiments
$ cd experiments
$ git init --bare
```

You want Git to notify the master every time the repository is pushed to (updated), so that it is rescanned for experiments and e.g. the GUI controls and the experiment list are updated.

Create a file named `post-receive` in the `hooks` folder (this folder has been created by the `git` command), containing the following:

```
#!/bin/sh
artiq_client scan-repository
```

Then set the execution permission on it:

```
$ chmod 755 hooks/post-receive
```

You may now run the master with the Git support enabled:

```
$ artiq_master -g -r /path_to/experiments
```

Push commits containing experiments to the bare repository using e.g. Git over SSH, and the new experiments should automatically appear in the dashboard.

Note: If you plan to run the ARTIQ system entirely on a single machine, you may also consider using a non-bare repository and the `post-commit` hook to trigger repository scans every time you commit changes (locally). The ARTIQ master never uses the repository's working directory, but only what is committed. More precisely, when scanning the repository, it fetches the last (atomically) completed commit at that time of repository scan and checks it out in a temporary folder. This commit ID is used by default when subsequently submitting experiments. There is one temporary folder by commit ID currently referenced in the system, so concurrently running experiments from different repository revisions is fully supported by the master.

The dashboard always runs experiments from the repository. The command-line client, by default, runs experiment from the raw filesystem (which is useful for iterating rapidly without creating many disorganized commits). If you want to use the repository instead, simply pass the `-R` option.

10.4 Scheduler API reference

The scheduler is exposed to the experiments via a virtual device called `scheduler`. It can be requested like any regular device, and then the methods below can be called on the returned object.

The scheduler virtual device also contains the attributes `rid`, `pipeline_name`, `priority` and `expid` that contain the corresponding information about the current run.

class artiq.master.scheduler.**Scheduler**(*ridc*, *worker_handlers*, *experiment_db*, *log_submissions*)

check_pause(*rid*)

Returns `True` if there is a condition that could make `pause` not return immediately (termination requested or higher priority run).

The typical purpose of this function is to check from a kernel whether returning control to the host and pausing would have an effect, in order to avoid the cost of switching kernels in the common case where `pause` does nothing.

This function does not have side effects, and does not have to be followed by a call to `pause`.

check_termination(*rid*)

Returns `True` if termination is requested.

delete(*rid*)

Kills the run with the specified RID.

get_status()

Returns a dictionary containing information about the runs currently tracked by the scheduler.

Must not be modified.

request_termination(*rid*)

Requests graceful termination of the run with the specified RID.

submit(*pipeline_name*, *expid*, *priority=0*, *due_date=None*, *flush=False*)

Submits a new run.

When called through an experiment, the default values of `pipeline_name`, `expid` and `priority` correspond to those of the current run.

10.5 Client control broadcasts (CCBs)

Client control broadcasts are requests made by experiments for clients to perform some action. Experiments do so by requesting the `ccb` virtual device and calling its `issue` method. The first argument of the issue method is the name of the broadcast, and any further positional and keyword arguments are passed to the broadcast.

CCBs are used by experiments to configure applets in the dashboard, for example for plotting purposes.

class artiq.dashboard.applets_ccb.**AppletsCCBDock**(**args*, ***kwargs*)

ccb_create_applet(*name*, *command*, *group=None*, *code=None*)

Requests the creation of a new applet.

An applet is identified by its name and an optional list of groups that represent a path (nested groups). If `group` is a string, it corresponds to a single group. If `group` is `None` or an empty list, it corresponds to the root.

`command` gives the command line used to run the applet, as if it was started from a shell. The dashboard substitutes variables such as `$python` that gives the complete file name of the Python interpreter running the dashboard.

If the name already exists (after following any specified groups), the command or code of the existing applet with that name is replaced, and the applet is restarted and shown at its previous position. If not, a new applet entry is created and the applet is shown at any position on the screen.

If the group(s) do not exist, they are created.

If `code` is not `None`, it should be a string that contains the full source code of the applet. In this case, `command` is used to specify (optional) command-line arguments to the applet.

This function is called when a CCB `create_applet` is issued.

ccb_disable_applet(*name*, *group=None*)

Disables an applet.

The applet is identified by its name, after following any specified groups.

This function is called when a CCB `disable_applet` is issued.

`ccb_disable_applet_group`(*group*)

Disables all the applets in a group.

If the group is nested, `group` should be a list, with the names of the parents preceding the name of the group to disable.

This function is called when a CCB `disable_applet_group` is issued.

10.6 Front-end tool reference

10.6.1 artiq_master

ARTIQ master

```
usage: artiq_master
       [-h]
       [--version]
       [--device-db DEVICE_DB]
       [--dataset-db DATASET_DB]
       [-g]
       [-r REPOSITORY]
       [--experiment-subdir EXPERIMENT_SUBDIR]
       [-v]
       [-q]
       [--log-file LOG_FILE]
       [--log-backup-count LOG_BACKUP_COUNT]
       [--name NAME]
       [--log-submissions LOG_SUBMISSIONS]
```

Named Arguments

--version	print the ARTIQ version number
--name	friendly name, displayed in dashboards to identify master instead of server address
--log-submissions	set the filename to create the experiment subimission

databases

--device-db	device database file (default: “‘device_db.py”’) Default: “device_db.py”
--dataset-db	dataset file (default: “‘dataset_db.pyon”’) Default: “dataset_db.pyon”

repository

-g, --git use the Git repository backend

Default: False

-r, --repository path to the repository (default: ‘“repository”’)

Default: “repository”

--experiment-subdir path to the experiment folder from the repository root (default: ‘’)

Default: “”

logging

-v, --verbose increase logging level of the master process

Default: 0

-q, --quiet decrease logging level of the master process

Default: 0

--log-file store logs in rotated files; set the base filename

Default: “”

--log-backup-count number of old log files to keep, or 0 to keep all log files. ‘.<yyyy>-<mm>-<dd>’ is added to the base filename (default: 0)

Default: 0

10.6.2 artiq_client

ARTIQ CLI client

```
usage: artiq_client
       [-h]
       [-s SERVER]
       [--port PORT]
       [--version]
       {submit,delete,set-dataset,del-dataset,show,scan-devices,scan-repository,ls}
       ...
```

Positional Arguments

action Possible choices: submit, delete, set-dataset, del-dataset, show, scan-devices, scan-repository, ls

Named Arguments

-s, --server	hostname or IP of the master to connect to Default: "::1"
--port	TCP port to use to connect to the master
--version	print the ARTIQ version number

Sub-commands:

submit

submit an experiment

```
artiq_client submit
[-h]
[-p PIPELINE]
[-P PRIORITY]
[-t TIMED]
[-f]
[-R]
[-r REVISION]
[--content]
[-c CLASS_NAME]
FILE
[ARGUMENTS ...]
```

Positional Arguments

FILE	file containing the experiment to run
ARGUMENTS	run arguments

Named Arguments

-p, --pipeline	pipeline to run the experiment in (default: "main") Default: "main"
-P, --priority	priority (higher value means sooner scheduling, default: 0) Default: 0
-t, --timed	set a due date for the experiment
-f, --flush	flush the pipeline before preparing the experiment Default: False
-R, --repository	use the experiment repository Default: False
-r, --revision	use a specific repository revision (defaults to head, ignored without -R)

--content	submit by content Default: False
-c, --class-name	name of the class to run

delete

delete an experiment from the schedule

```
artiq_client delete
[-h]
[-g]
RID
```

Positional Arguments

RID	run identifier (RID)

Named Arguments

-g	request graceful termination Default: False

set-dataset

add or modify a dataset

```
artiq_client set-dataset
[-h]
[-p | -n]
NAME
VALUE
```

Positional Arguments

NAME	name of the dataset
VALUE	value in PYON format

Named Arguments

-p, --persist	make the dataset persistent Default: False
-n, --no-persist	make the dataset non-persistent Default: False

del-dataset

delete a dataset

```
artiq_client del-dataset
[-h]
name
```

Positional Arguments

name	name of the dataset

show

show schedule, log, devices or datasets

```
artiq_client show
[-h]
WHAT
```

Positional Arguments

WHAT	Possible choices: schedule, log, ccb, devices, datasets select object to show: [‘schedule’, ‘log’, ‘ccb’, ‘devices’, ‘datasets’]

scan-devices

trigger a device database (re)scan

```
artiq_client scan-devices
[-h]
```

scan-repository

trigger a repository (re)scan

```
artiq_client scan-repository
[-h]
[--async]
[REVISION]
```

Positional Arguments

REVISION use a specific repository revision (defaults to head)

Named Arguments

--async trigger scan and return immediately

Default: False

ls

list a directory on the master

```
artiq_client ls
[-h]
[directory]
```

Positional Arguments

directory Default: “”

10.6.3 artiq_dashboard

ARTIQ Dashboard

```
usage: artiq_dashboard
       [-h]
       [--version]
       [-s SERVER]
       [--port-notify PORT_NOTIFY]
       [--port-control PORT_CONTROL]
       [--port-broadcast PORT_BROADCAST]
       [--db-file DB_FILE]
       [-p PLUGIN_MODULES]
```

Named Arguments

--version	print the ARTIQ version number
-s, --server	hostname or IP of the master to connect to
	Default: "::1"
--port-notify	TCP port to connect to for notifications
	Default: 3250
--port-control	TCP port to connect to for control
	Default: 3251
--port-broadcast	TCP port to connect to for broadcasts
	Default: 1067
--db-file	database file for local GUI settings
-p, --load-plugin	Python module to load on startup

10.6.4 artiq_session

ARTIQ session manager. Automatically runs the master, dashboard and local controller manager on the current machine. The latter requires the artiq-comtools package to be installed.

```
usage: artiq_session
       [-h]
       [--version]
       [-m M]
       [-d D]
       [-c C]
```

Named Arguments

--version	print the ARTIQ version number
-m	add argument to the master command line
	Default: []
-d	add argument to the dashboard command line
	Default: []
-c	add argument to the controller manager command line
	Default: []

CHAPTER

ELEVEN

THE ENVIRONMENT

Experiments interact with an environment that consists of devices, arguments and datasets. Access to the environment is handled by the class *artiq.language.environment.EnvExperiment* that experiments should derive from.

11.1 The device database

The device database contains information about the devices available in a ARTIQ installation, what drivers to use, what controllers to use and on what machine, and where the devices are connected.

The master (or `artiq_run`) instantiates the device drivers (and the RPC clients in the case of controllers) for the experiments based on the contents of the device database.

The device database is stored in the memory of the master and is generated by a Python script typically called `device_db.py`. That script must define a global variable `device_db` with the contents of the database. The device database is a Python dictionary whose keys are the device names, and values can have several types.

11.1.1 Local devices

Local device entries are dictionaries that contain a `type` field set to `local`. They correspond to device drivers that are created locally on the master (as opposed to going through the controller mechanism). The fields `module` and `class` determine the location of the Python class that the driver consists of. The `arguments` field is another (possibly empty) dictionary that contains arguments to pass to the device driver constructor.

11.1.2 Controllers

Controller entries are dictionaries whose `type` field is set to `controller`. When an experiment requests such a device, a RPC client (see `sipyco.pc_rpc`) is created and connected to the appropriate controller. Controller entries are also used by controller managers to determine what controllers to run.

The `best_effort` field is a boolean that determines whether to use `sipyco.pc_rpc.Client` or `sipyco.pc_rpc.BestEffortClient`. The `host` and `port` fields configure the TCP connection. The `target` field contains the name of the RPC target to use (you may use `sipyco_rpctool` on a controller to list its targets). Controller managers run the `command` field in a shell to launch the controller, after replacing `{port}` and `{bind}` by respectively the TCP port the controller should listen to (matches the `port` field) and an appropriate bind address for the controller's listening socket.

11.1.3 Aliases

If an entry is a string, that string is used as a key for another lookup in the device database.

11.2 Arguments

Arguments are values that parameterize the behavior of an experiment and are set before the experiment is executed.

Requesting the values of arguments can only be done in the build phase of an experiment. The value requests are also used to define the GUI widgets shown in the explorer when the experiment is selected.

11.3 Datasets

Datasets are values (possibly arrays) that are read and written by experiments and live in a key-value store.

A dataset may be broadcasted, that is, distributed to all clients connected to the master. For example, the ARTIQ GUI may plot it while the experiment is in progress to give rapid feedback to the user. Broadcasted datasets live in a global key-value store; experiments should use distinctive real-time result names in order to avoid conflicts. Broadcasted datasets may be used to communicate values across experiments; for example, a periodic calibration experiment may update a dataset read by payload experiments. Broadcasted datasets are replaced when a new dataset with the same key (name) is produced.

Broadcasted datasets may be persistent: the master stores them in a file typically called `dataset_db.pyon` so they are saved across master restarts.

Datasets produced by an experiment run may be archived in the HDF5 output for that run.

CHAPTER

TWELVE

DISTRIBUTED REAL TIME INPUT/OUTPUT (DRTIO)

DRTIO is a time and data transfer system that allows ARTIQ RTIO channels to be distributed among several satellite devices synchronized and controlled by a central core device.

The link is a high speed duplex serial line operating at 1Gbps or more, over copper or optical fiber.

The main source of DRTIO traffic is the remote control of RTIO output and input channels. The protocol is optimized to maximize throughput and minimize latency, and handles flow control and error conditions (underflows, overflows, etc.)

The DRTIO protocol also supports auxiliary, low-priority and non-realtime traffic. The auxiliary channel supports overriding and monitoring TTL I/Os. Auxiliary traffic never interrupts or delays the main traffic, so that it cannot cause unexpected poor performance (e.g. RTIO underflows).

Time transfer and clock syntonization is typically done over the serial link alone. The DRTIO code is organized as much as possible to support porting to different types of transceivers (Xilinx MGTs, Altera MGTs, soft transceivers running off regular FPGA IOs, etc.) and different synchronization mechanisms.

The lower layers of DRTIO are similar to White Rabbit, with the following main differences:

- lower latency
- deterministic latency
- real-time/auxiliary channels
- higher bandwidth
- no Ethernet compatibility
- only star or tree topologies are supported

From ARTIQ kernels, DRTIO channels are used in the same way as local RTIO channels.

12.1 Using DRTIO

12.1.1 Terminology

In a system of interconnected DRTIO devices, each RTIO core (driving RTIO PHYs; for example a RTIO core would connect to a large bank of TTL signals) is assigned a number and is called a *destination*. One DRTIO device normally contains one RTIO core.

On one DRTIO device, the immediate path that a RTIO request must take is called a *hop*: the request can be sent to the local RTIO core, or to another device downstream. Each possible hop is assigned a number. Hop 0 is normally the local RTIO core, and hops 1 and above correspond to the respective downstream ports of the device.

DRTIO devices are arranged in a tree topology, with the core device at the root. For each device, its distance from the root (in number of devices that are crossed) is called its *rank*. The root has rank 0, the devices immediately connected to it have rank 1, and so on.

12.1.2 The routing table

The routing table defines, for each destination, the list of hops ("route") that must be taken from the root in order to reach it.

It is stored in a binary format that can be manipulated with the *artiq_route utility*. The binary file is then programmed into the flash storage of the core device under the `routing_table` key. It is automatically distributed to downstream devices when the connections are established. Modifying the routing table requires rebooting the core device for the new table to be taken into account.

All routes must end with the local RTIO core of the last device (0).

The local RTIO core of the core device is a destination like any other, and it needs to be explicitly part of the routing table for kernels to be able to access it.

If no routing table is programmed, the core device takes a default routing table for a star topology (i.e. with no devices of rank 2 or above), with destination 0 being the core device's local RTIO core and destinations 1 and above corresponding to devices on the respective downstream ports.

Here is an example of creating and programming a routing table for a chain of 3 devices:

```
# create an empty routing table
$ artiq_route rt.bin init

# set destination 0 to the local RTIO core
$ artiq_route rt.bin set 0 0

# for destination 1, first use hop 1 (the first downstream port)
# then use the local RTIO core of that second device.
$ artiq_route rt.bin set 1 1 0

# for destination 2, use hop 1 and reach the second device as
# before, then use hop 1 on that device to reach the third
# device, and finally use the local RTIO core (hop 0) of the
# third device.
$ artiq_route rt.bin set 2 1 1 0

$ artiq_route rt.bin show
  0:   0
  1:   1   0
  2:   1   1   0

$ artiq_coremgmt config write -f routing_table rt.bin
```

12.1.3 Addressing distributed RTIO cores from kernels

Remote RTIO channels are accessed in the same way as local ones. Bits 16-24 of the RTIO channel number define the destination. Bits 0-15 of the RTIO channel number select the channel within the destination.

12.1.4 Link establishment

After devices have booted, it takes several seconds for all links in a DRTIO system to become established (especially with the long locking times of low-bandwidth PLLs that are used for jitter reduction purposes). Kernels should not attempt to access destinations until all required links are up (when this happens, the `RTIODestinationUnreachable` exception is raised). ARTIQ provides the method *`get_rtio_destination_status()`* that determines whether a destination can be reached. We recommend calling it in a loop in your startup kernel for each important destination, to delay startup until they all can be reached.

12.1.5 Latency

Each hop increases the RTIO latency of a destination by a significant amount; that latency is however constant and can be compensated for in kernels. To limit latency in a system, fully utilize the downstream ports of devices to reduce the depth of the tree, instead of creating chains.

12.2 Internal details

12.2.1 Real-time and auxiliary packets

DRTIO is a packet-based protocol that uses two types of packets:

- real-time packets, which are transmitted at high priority at a high bandwidth and are used for the bulk of RTIO commands and data. In the ARTIQ DRTIO implementation, real-time packets are processed entirely in gateware.
- auxiliary packets, which are lower-bandwidth and are used for ancillary tasks such as housekeeping and monitoring/injection. Auxiliary packets are low-priority and their transmission has no impact on the timing of real-time packets (however, transmission of real-time packets slows down the transmission of auxiliary packets). In the ARTIQ DRTIO implementation, the contents of the auxiliary packets are read and written directly by the firmware, with the gateware simply handling the transmission of the raw data.

12.2.2 Link layer

The lower layer of the DRTIO protocol stack is the link layer, which is responsible for delimiting real-time and auxiliary packets, and assisting with the establishment of a fixed-latency high speed serial transceiver link.

DRTIO uses the IBM (Widmer and Franaszek) 8b/10b encoding. D characters (the encoded 8b symbols) always transmit real-time packet data, whereas K characters are used for idling and transmitting auxiliary packet data.

At every logic clock cycle, the high-speed transceiver hardware transmits some amount N of 8b/10b characters (typically, N is 2 or 4) and receives the same amount. With DRTIO, those characters must be all of the D type or all of the K type; mixing D and K characters in the same logic clock cycle is not allowed.

A real-time packet is defined by a series of D characters containing the packet's payload, delimited by at least one K character. Real-time packets must be padded to satisfy the requirement that only D or only K characters are transmitted during a logic clock cycle, by making their length a multiple of N.

K characters, which are transmitted whenever there is no real-time data to transmit and to delimit real-time packets, are chosen using a 3-bit K selection word. If this K character is the first character in the set of N characters processed by the transceiver in the logic clock cycle, the mapping between the K selection word and the 8b/10b K space contains commas. If the K character is any of the subsequent characters processed by the transceiver, a different mapping is used that does not contain any commas. This scheme allows the receiver to align its logic clock with that of the transmitter, simply by shifting its logic clock so that commas are received into the first character position.

Note: Due to the shoddy design of transceiver hardware, this simple process of clock and comma alignment is difficult to perform in practice. The paper "High-speed, fixed-latency serial links with Xilinx FPGAs" (by Xue LIU, Qing-xu DENG, Bo-ning HOU and Ze-ke WANG) discusses techniques that can be used. The ARTIQ implementation simply keeps resetting the receiver until the comma is aligned, since relatively long lock times are acceptable.

The series of K selection words is then used to form auxiliary packets and the idle pattern. When there is no auxiliary packet to transfer or to delimitate auxiliary packets, the K selection word `100` is used. To transfer data from an auxiliary packet, the K selection word `0ab` is used, with `ab` containing two bits of data from the packet. An auxiliary packet is delimited by at least one `100` K selection word.

Both real-time traffic and K selection words are scrambled in order to make the generated electromagnetic interference practically independent from the DRTIO traffic. A multiplicative scrambler is used and its state is shared between the real-time traffic and K selection words, so that real-time data can be descrambled immediately after the scrambler has been synchronized from the K characters. Another positive effect of the scrambling is that commas always appear regularly in the absence of any traffic (and in practice also appear regularly on a busy link). This makes a receiver always able to synchronize itself to an idling transmitter, which removes the need for relatively complex link initialization states.

Due to the use of K characters both as delimiters for real-time packets and as information carrier for auxiliary packets, auxiliary traffic is guaranteed a minimum bandwidth simply by having a maximum size limit on real-time packets.

12.2.3 Clocking

At the DRTIO satellite device, the recovered and aligned transceiver clock is used for clocking RTIO channels, after appropriate jitter filtering using devices such as the Si5324. The same clock is also used for clocking the DRTIO transmitter (loop timing), which simplifies clock domain transfers and allows for precise round-trip-time measurements to be done.

12.2.4 RTIO clock synchronization

As part of the DRTIO link initialization, a real-time packet is sent by the core device to each satellite device to make them load their respective timestamp counters with the timestamp values from their respective packets.

12.2.5 RTIO outputs

Controlling a remote RTIO output involves placing the RTIO event into the buffer of the destination. The core device maintains a cache of the buffer space available in each destination. If, according to the cache, there is space available, then a packet containing the event information (timestamp, address, channel, data) is sent immediately and the cached value is decremented by one. If, according to the cache, no space is available, then the core device sends a request for the space available in the destination and updates the cache. The process repeats until at least one remote buffer entry is available for the event, at which point a packet containing the event information is sent as before.

Detecting underflow conditions is the responsibility of the core device; should an underflow occur then no DRTIO packet is transmitted. Sequence errors are handled similarly.

12.2.6 RTIO inputs

The core device sends a request to the satellite for reading data from one of its channels. The request contains a timeout, which is the RTIO timestamp to wait for until an input event appears. The satellite then replies with either an input event (containing timestamp and data), a timeout, or an overflow error.

CHAPTER

THIRTEEN

CORE LANGUAGE REFERENCE

The most commonly used features from the ARTIQ language modules and from the core device modules are bundled together in `artiq.experiment` and can be imported with `from artiq.experiment import *`.

13.1 `artiq.language.core` module

Core ARTIQ extensions to the Python language.

exception artiq.language.core.**TerminationRequested**

Raised by `pause` when the user has requested termination.

artiq.language.core.**at_mu**(*time*)

Sets the RTIO time to the specified absolute value, in machine units.

artiq.language.core.**delay**(*duration*)

Increases the RTIO time by the given amount (in seconds).

artiq.language.core.**delay_mu**(*duration*)

Increases the RTIO time by the given amount (in machine units).

artiq.language.core.**host_only**(*function*)

This decorator marks a function so that it can only be executed in the host Python interpreter.

artiq.language.core.**kernel**(*arg=None*, *flags={}*)

This decorator marks an object's method for execution on the core device.

When a decorated method is called from the Python interpreter, the `core` attribute of the object is retrieved and used as core device driver. The core device driver will typically compile, transfer and run the method (kernel) on the device.

When kernels call another method:

- if the method is a kernel for the same core device, it is compiled and sent in the same binary. Calls between kernels happen entirely on the device.
- if the method is a regular Python method (not a kernel), it generates a remote procedure call (RPC) for execution on the host.

The decorator takes an optional parameter that defaults to :attr`core` and specifies the name of the attribute to use as core device driver.

This decorator must be present in the global namespace of all modules using it for the import cache to work properly.

artiq.language.core.**kernel_from_string**(*parameters*, *body_code*, *decorator=<function kernel>*)

Build a kernel function from the supplied source code in string form, similar to `exec()/eval()`.

Operating on pieces of source code as strings is a very brittle form of metaprogramming; kernels generated like this are hard to debug, and inconvenient to write. Nevertheless, this can sometimes be useful to work around restrictions in ARTIQ Python. In that instance, care should be taken to keep string-generated code to a minimum and cleanly separate it from surrounding code.

The resulting function declaration is also evaluated using `exec()` for use from host Python code. To encourage a modicum of code hygiene, no global symbols are available by default; any objects accessed by the function body must be passed in explicitly as parameters.

Parameters

- **parameters** – A list of parameter names the generated functions accepts. Each entry can either be a string or a tuple of two strings; if the latter, the second element specifies the type annotation.
- **body_code** – The code for the function body, in string form. `return` statements can be used to return values, as usual.
- **decorator** – One of `kernel` or `portable` (optionally with parameters) to specify how the function will be executed.

Returns The function generated from the arguments.

artiq.language.core.**now_mu**()

Retrieve the current RTIO timeline cursor, in machine units.

Note the conceptual difference between this and the current value of the hardware RTIO counter; see e.g. *artiq.coredevice.core.Core.get_rtio_counter_mu()* for the latter.

artiq.language.core.**portable**(*arg=None*, *flags={}*)

This decorator marks a function for execution on the same device as its caller.

In other words, a decorated function called from the interpreter on the host will be executed on the host (no compilation and execution on the core device). A decorated function called from a kernel will be executed on the core device (no RPC).

This decorator must be present in the global namespace of all modules using it for the import cache to work properly.

artiq.language.core.**rpc**(*arg=None*, *flags={}*)

This decorator marks a function for execution on the host interpreter. This is also the default behavior of ARTIQ; however, this decorator allows specifying additional flags.

artiq.language.core.**set_time_manager**(*time_manager*)

Set the time manager used for simulating kernels by running them directly inside the Python interpreter. The time manager responds to the entering and leaving of interleave/parallel/sequential blocks, delays, etc. and provides a time-stamped logging facility for events.

artiq.language.core.**syscall**(*arg=None*, *flags={}*)

This decorator marks a function as a system call. When executed on a core device, a C function with the provided name (or the same name as the Python function, if not provided) will be called. When executed on host, the Python function will be called as usual.

Every argument and the return value must be annotated with ARTIQ types.

Only drivers should normally define syscalls.

13.2 artiq.language.environment module

class artiq.language.environment.**BooleanValue**(*default=<class 'artiq.language.environment.NoDefault'>*)

A boolean argument.

exception artiq.language.environment.**DefaultMissing**

Raised by the `default` method of argument processors when no default value is available.

class artiq.language.environment.**EnumerationValue**(*choices*, *default=<class 'artiq.language.environment.NoDefault'>*)

An argument that can take a string value among a predefined set of values.

Parameters **choices** – A list of string representing the possible values of the argument.

class artiq.language.environment.**EnvExperiment**(*managers_or_parent*, **args*, ***kwargs*)

Base class for top-level experiments that use the *HasEnvironment* environment manager.

Most experiments should derive from this class.

prepare()

This default prepare method calls *prepare()* for all children, in the order of registration, if the child has a *prepare()* method.

class artiq.language.environment.**Experiment**

Base class for top-level experiments.

Deriving from this class enables automatic experiment discovery in Python modules.

analyze()

Entry point for analyzing the results of the experiment.

This method may be overloaded by the user to implement the analysis phase of the experiment, for example fitting curves.

Splitting this phase from *run()* enables tweaking the analysis algorithm on pre-existing data, and CPU-bound analyses to be run overlapped with the next experiment in a pipelined manner.

This method must not interact with the hardware.

prepare()

Entry point for pre-computing data necessary for running the experiment.

Doing such computations outside of *run()* enables more efficient scheduling of multiple experiments that need to access the shared hardware during part of their execution.

This method must not interact with the hardware.

run()

The main entry point of the experiment.

This method must be overloaded by the user to implement the main control flow of the experiment.

This method may interact with the hardware.

The experiment may call the scheduler's `pause()` method while in *run()*.

class artiq.language.environment.**HasEnvironment**(*managers_or_parent*, **args*, ***kwargs*)

Provides methods to manage the environment of an experiment (arguments, devices, datasets).

append_to_dataset(*key*, *value*)

Append a value to a dataset.

The target dataset must be a list (i.e. support `append()`), and must have previously been set from this experiment.

The broadcast/persist/archive mode of the given key remains unchanged from when the dataset was last set. Appended values are transmitted efficiently as incremental modifications in broadcast mode.

build()

Should be implemented by the user to request arguments.

Other initialization steps such as requesting devices may also be performed here.

There are two situations where the requested devices are replaced by `DummyDevice()` and arguments are set to their defaults (or `None`) instead: when the repository is scanned to build the list of available experiments and when the dataset browser `artiq_browser` is used to open or run the analysis stage of an experiment. Do not rely on being able to operate on devices or arguments in *build()*.

Datasets are read-only in this method.

Leftover positional and keyword arguments from the constructor are forwarded to this method. This is intended for experiments that are only meant to be executed programmatically (not from the GUI).

call_child_method(*method*, **args*, ***kwargs*)

Calls the named method for each child, if it exists for that child, in the order of registration.

Parameters

- **method** (*str*) – Name of the method to call
- **args** – Tuple of positional arguments to pass to all children
- **kwargs** – Dict of keyword arguments to pass to all children

get_argument(*key*, *processor*, *group=None*, *tooltip=None*)

Retrieves and returns the value of an argument.

This function should only be called from `build`.

Parameters

- **key** – Name of the argument.
- **processor** – A description of how to process the argument, such as instances of `BooleanValue` and `NumberValue`.
- **group** – An optional string that defines what group the argument belongs to, for user interface purposes.
- **tooltip** – An optional string to describe the argument in more detail, applied as a tooltip to the argument name in the user interface.

get_dataset(*key*, *default=<class 'artiq.language.environment.NoDefault'>*, *archive=True*)

Returns the contents of a dataset.

The local storage is searched first, followed by the master storage (which contains the broadcasted datasets from all experiments) if the key was not found initially.

If the dataset does not exist, returns the default value. If no default is provided, raises `KeyError`.

By default, datasets obtained by this method are archived into the output HDF5 file of the experiment. If an archived dataset is requested more than one time or is modified, only the value at the time of the first call is archived. This may impact reproducibility of experiments.

Parameters **archive** – Set to `False` to prevent archival together with the run's results. Default is `True`.

`get_device`(*key*)

Creates and returns a device driver.

`get_device_db`()

Returns the full contents of the device database.

`mutate_dataset`(*key*, *index*, *value*)

Mutate an existing dataset at the given index (e.g. set a value at a given position in a NumPy array)

If the dataset was created in broadcast mode, the modification is immediately transmitted.

If the index is a tuple of integers, it is interpreted as `slice(*index)`. If the index is a tuple of tuples, each sub-tuple is interpreted as `slice(*sub_tuple)` (multi-dimensional slicing).

`set_dataset`(*key*, *value*, *broadcast=False*, *persist=False*, *archive=True*)

Sets the contents and handling modes of a dataset.

Datasets must be scalars (`bool`, `int`, `float` or NumPy scalar) or NumPy arrays.

Parameters

- **broadcast** – the data is sent in real-time to the master, which dispatches it.
- **persist** – the master should store the data on-disk. Implies broadcast.
- **archive** – the data is saved into the local storage of the current run (archived as a HDF5 file).

`set_default_scheduling`(*priority=None*, *pipeline_name=None*, *flush=None*)

Sets the default scheduling options.

This function should only be called from `build`.

`setattr_argument`(*key*, *processor=None*, *group=None*, *tooltip=None*)

Sets an argument as attribute. The names of the argument and of the attribute are the same.

The key is added to the instance's kernel invariants.

`setattr_dataset`(*key*, *default=<class 'artiq.language.environment.NoDefault'>*, *archive=True*)

Sets the contents of a dataset as attribute. The names of the dataset and of the attribute are the same.

`setattr_device`(*key*)

Sets a device driver as attribute. The names of the device driver and of the attribute are the same.

The key is added to the instance's kernel invariants.

`class artiq.language.environment.NoDefault`

Represents the absence of a default value.

`class artiq.language.environment.NumberValue`(*default=<class 'artiq.language.environment.NoDefault'>*, *unit=''*, *scale=None*, *step=None*, *min=None*, *max=None*, *ndecimals=2*, *type='auto'*)

An argument that can take a numerical value.

If `type=="auto"`, the result will be a `float` unless ndecimals = 0, scale = 1 and step is an integer. Setting `type` to `int` will also result in an error unless these conditions are met.

When `scale` is not specified, and the unit is a common one (i.e. defined in `artiq.language.units`), then the scale is obtained from the unit using a simple string match. For example, milliseconds (`"ms"`) units set the scale to 0.001. No unit (default) corresponds to a scale of 1.0.

For arguments with uncommon or complex units, use both the unit parameter (a string for display) and the scale parameter (a numerical scale for experiments). For example, `NumberValue(1, unit="xyz", scale=0.001)` will display as 1 xyz in the GUI window because of the unit setting, and appear as the numerical value 0.001 in the code because of the scale setting.

Parameters

- **unit** – A string representing the unit of the value.
- **scale** – A numerical scaling factor by which the displayed value is multiplied when referenced in the experiment.
- **step** – The step with which the value should be modified by up/down buttons in a UI. The default is the scale divided by 10.
- **min** – The minimum value of the argument.
- **max** – The maximum value of the argument.
- **ndecimals** – The number of decimals a UI should use.
- **type** – Type of this number. Accepts `"float"`, `"int"` or `"auto"`. Defaults to `"auto"`.

class artiq.language.environment.**PYONValue**(*default=<class 'artiq.language.environment.NoDefault'>*)

An argument that can be any PYON-serializable value.

class artiq.language.environment.**StringValue**(*default=<class 'artiq.language.environment.NoDefault'>*)

A string argument.

13.3 artiq.language.scan module

Implementation and management of scan objects.

A scan object (e.g. *artiq.language.scan.RangeScan*) represents a one-dimensional sweep of a numerical range. Multi-dimensional scans are constructed by combining several scan objects, for example using *artiq.language.scan.MultiScanManager*.

Iterate on a scan object to scan it, e.g.

```
for variable in self.scan:
    do_something(variable)
```

Iterating multiple times on the same scan object is possible, with the scan yielding the same values each time. Iterating concurrently on the same scan object (e.g. via nested loops) is also supported, and the iterators are independent from each other.

class artiq.language.scan.**CenterScan**(*center*, *span*, *step*, *randomize=False*, *seed=None*)

A scan object that yields evenly spaced values within a span around a center. If `step` is finite, then `center` is always included. Values outside `span` around center are never included. If `randomize` is True the points are randomly ordered.

class artiq.language.scan.**ExplicitScan**(*sequence*)

A scan object that yields values from an explicitly defined sequence.

class artiq.language.scan.**MultiScanManager**(**args*)

Makes an iterator that returns elements from the first scan object until it is exhausted, then proceeds to the next iterable, until all of the scan objects are exhausted. Used for treating consecutive scans as a single scan.

Scan objects must be passed as a list of tuples (name, scan_object). Íteration produces scan points that have attributes that correspond to the names of the scan objects, and have the last value yielded by that scan object.

class artiq.language.scan.**NoScan**(*value*, *repetitions=1*)

A scan object that yields a single value for a specified number of repetitions.

class artiq.language.scan.**RangeScan**(*start*, *stop*, *npoints*, *randomize=False*, *seed=None*)

A scan object that yields a fixed number of evenly spaced values in a range. If `randomize` is True the points are randomly ordered.

class artiq.language.scan.**Scannable**(*default=<class 'artiq.language.environment.NoDefault'>*, *unit=''*, *scale=None*, *global_step=None*, *global_min=None*, *global_max=None*, *ndecimals=2*)

An argument (as defined in *artiq.language.environment*) that takes a scan object.

When `scale` is not specified, and the unit is a common one (i.e. defined in `artiq.language.units`), then the scale is obtained from the unit using a simple string match. For example, milliseconds (`"ms"`) units set the scale to 0.001. No unit (default) corresponds to a scale of 1.0.

For arguments with uncommon or complex units, use both the unit parameter (a string for display) and the scale parameter (a numerical scale for experiments). For example, a scan shown between 1 xyz and 10 xyz in the GUI with `scale=0.001` and `unit="xyz"` results in values between 0.001 and 0.01 being scanned.

Parameters

- **default** – The default scan object. This parameter can be a list of scan objects, in which case the first one is used as default and the others are used to configure the default values of scan types that are not initially selected in the GUI.
- **global_min** – The minimum value taken by the scanned variable, common to all scan modes. The user interface takes this value to set the range of its input widgets.
- **global_max** – Same as global_min, but for the maximum value.
- **global_step** – The step with which the value should be modified by up/down buttons in a user interface. The default is the scale divided by 10.
- **unit** – A string representing the unit of the scanned variable.
- **scale** – A numerical scaling factor by which the displayed values are multiplied when referenced in the experiment.
- **ndecimals** – The number of decimals a UI should use.

13.4 artiq.language.units module

This module contains floating point constants that correspond to common physical units (ns, MHz, ...). They are provided for convenience (e.g write `MHz` instead of `1000000.0`) and code clarity purposes.

CHAPTER

FOURTEEN

CORE DRIVERS REFERENCE

These drivers are for the core device and the peripherals closely integrated into it, which do not use the controller mechanism.

14.1 System drivers

14.1.1 artiq.coredevice.core module

exception artiq.coredevice.core.**CompileError**(*diagnostic*)

class artiq.coredevice.core.**Core**(*dmgr*, *host*, *ref_period*, *ref_multiplier=8*, *target='rv32g'*)

Core device driver.

Parameters

- **host** – hostname or IP address of the core device.
- **ref_period** – period of the reference clock for the RTIO subsystem. On platforms that use clock multiplication and SERDES-based PHYs, this is the period after multiplication. For example, with a RTIO core clocked at 125MHz and a SERDES multiplication factor of 8, the reference period is 1ns. The time machine unit is equal to this period.
- **ref_multiplier** – ratio between the RTIO fine timestamp frequency and the RTIO coarse timestamp frequency (e.g. SERDES multiplication factor).

break_realtime()

Set the time cursor after the current value of the hardware RTIO counter plus a margin of 125000 machine units.

If the time cursor is already after that position, this function does nothing.

get_rtio_counter_mu()

Retrieve the current value of the hardware RTIO timeline counter.

As the timing of kernel code executed on the CPU is inherently non-deterministic, the return value is by necessity only a lower bound for the actual value of the hardware register at the instant when execution resumes in the caller.

For a more detailed description of these concepts, see *ARTIQ Real-Time I/O Concepts*.

get_rtio_destination_status(*destination*)

Returns whether the specified RTIO destination is up. This is particularly useful in startup kernels to delay startup until certain DRTIO destinations are up.

mu_to_seconds(*mu*)

Convert machine units (RTIO cycles) to seconds.

Parameters **mu** – cycle count to convert.

precompile(*function*, **args*, ***kwargs*)

Precompile a kernel and return a callable that executes it on the core device at a later time.

Arguments to the kernel are set at compilation time and passed to this function, as additional positional and keyword arguments. The returned callable accepts no arguments.

Precompiled kernels may use RPCs.

Object attributes at the beginning of a precompiled kernel execution have the values they had at precompilation time. If up-to-date values are required, use RPC to read them. Similarly, modified values are not written back, and explicit RPC should be used to modify host objects. Carefully review the source code of drivers calls used in precompiled kernels, as they may rely on host object attributes being transfered between kernel calls. Examples include code used to control DDS phase, and Urukul RF switch control via the CPLD register.

The return value of the callable is the return value of the kernel, if any.

The callable may be called several times.

reset()

Clear RTIO FIFOs, release RTIO PHY reset, and set the time cursor at the current value of the hardware RTIO counter plus a margin of 125000 machine units.

seconds_to_mu(*seconds*)

Convert seconds to the corresponding number of machine units (RTIO cycles).

Parameters **seconds** – time (in seconds) to convert.

wait_until_mu(*cursor_mu*)

Block execution until the hardware RTIO counter reaches the given value (see *get_rtio_counter_mu()*).

If the hardware counter has already passed the given time, the function returns immediately.

14.1.2 artiq.coredevice.exceptions module

exception artiq.coredevice.exceptions.**CacheError**

Raised when putting a value into a cache row would violate memory safety.

exception artiq.coredevice.exceptions.**ClockFailure**

Raised when RTIO PLL has lost lock.

class artiq.coredevice.exceptions.**CoreException**(*exceptions*, *exception_info*, *traceback*, *stack_pointers*)

Information about an exception raised or passed through the core device.

exception artiq.coredevice.exceptions.**DMAError**

Raised when performing an invalid DMA operation.

exception artiq.coredevice.exceptions.**I2CError**

Raised when a I2C transaction fails.

exception artiq.coredevice.exceptions.**InternalError**

Raised when the runtime encounters an internal error condition.

exception artiq.coredevice.exceptions.**RTIODestinationUnreachable**

Raised with a RTIO operation could not be completed due to a DRTIO link being down.

exception artiq.coredevice.exceptions.**RTIOOverflow**

Raised when at least one event could not be registered into the RTIO input FIFO because it was full (CPU not reading fast enough).

This does not interrupt operations further than cancelling the current read attempt and discarding some events. Reading can be reattempted after the exception is caught, and events will be partially retrieved.

exception artiq.coredevice.exceptions.**RTIOUnderflow**

Raised when the CPU or DMA core fails to submit a RTIO event early enough (with respect to the event's timestamp).

The offending event is discarded and the RTIO core keeps operating.

exception artiq.coredevice.exceptions.**SPIError**

Raised when a SPI transaction fails.

14.1.3 artiq.coredevice.dma module

Direct Memory Access (DMA) extension.

This feature allows storing pre-defined sequences of output RTIO events into the core device's SDRAM, and playing them back at higher speeds than the CPU alone could achieve.

class artiq.coredevice.dma.**CoreDMA**(*dmgr*, *core_device='core'*)

Core device Direct Memory Access (DMA) driver.

Gives access to the DMA functionality of the core device.

erase(*name*)

Removes the DMA trace with the given name from storage.

get_handle(*name*)

Returns a handle to a previously recorded DMA trace. The returned handle is only valid until the next call to *record()* or *erase()*.

playback(*name*)

Replays a previously recorded DMA trace. This function blocks until the entire trace is submitted to the RTIO FIFOs.

playback_handle(*handle*)

Replays a handle obtained with *get_handle()*. Using this function is much faster than *playback()* for replaying a set of traces repeatedly, but incurs the overhead of managing the handles onto the programmer.

record(*name*)

Returns a context manager that will record a DMA trace called `name`. Any previously recorded trace with the same name is overwritten. The trace will persist across kernel switches.

class artiq.coredevice.dma.**DMARecordContextManager**

Context manager returned by *CoreDMA.record()*.

Upon entering, starts recording a DMA trace. All RTIO operations are redirected to a newly created DMA buffer after this call, and `now` is reset to zero.

Upon leaving, stops recording a DMA trace. All recorded RTIO operations are stored in a newly created trace, and `now` is restored to the value it had before the context manager was entered.

14.1.4 artiq.coredevice.cache module

class artiq.coredevice.cache.**CoreCache**(*dmgr*, *core_device='core'*)

Core device cache access

get(*key*)

Extract a value from the core device cache. After a value is extracted, it cannot be replaced with another value using *put()* until all kernel functions finish executing; attempting to replace it will result in a *artiq.coredevice.exceptions.CacheError*.

If the cache does not contain any value associated with `key`, an empty list is returned.

The value is not copied, so mutating it will change what's stored in the cache.

Parameters **key** (*str*) – cache key

Returns a list of 32-bit integers

put(*key*, *value*)

Put a value into the core device cache. The value will persist until reboot.

To remove a value from the cache, call *put()* with an empty list.

Parameters

- **key** (*str*) – cache key
- **value** (*list*) – a list of 32-bit integers

14.2 Digital I/O drivers

14.2.1 artiq.coredevice.ttl module

Drivers for TTL signals on RTIO.

TTL channels (including the clock generator) all support output event replacement. For example, pulses of "zero" length (e.g. *TTLInOut.on()* immediately followed by *TTLInOut.off()*, without a delay) are suppressed.

class artiq.coredevice.ttl.**TTLClockGen**(*dmgr*, *channel*, *acc_width=24*, *core_device='core'*)

RTIO TTL clock generator driver.

This should be used with TTL channels that have a clock generator built into the gateware (not compatible with regular TTL channels).

The time cursor is not modified by any function in this class.

Parameters

- **channel** – channel number
- **acc_width** – accumulator width in bits

frequency_to_ftw(*frequency*)

Returns the frequency tuning word corresponding to the given frequency.

ftw_to_frequency(*ftw*)

Returns the frequency corresponding to the given frequency tuning word.

set(*frequency*)

Like *set_mu()*, but using Hz.

set_mu(*frequency*)

Set the frequency of the clock, in machine units, at the current position of the time cursor.

This also sets the phase, as the time of the first generated rising edge corresponds to the time of the call.

The clock generator contains a 24-bit phase accumulator operating on the RTIO clock. At each RTIO clock tick, the frequency tuning word is added to the phase accumulator. The most significant bit of the phase accumulator is connected to the TTL line. Setting the frequency tuning word has the additional effect of setting the phase accumulator to 0x800000.

Due to the way the clock generator operates, frequency tuning words that are not powers of two cause jitter of one RTIO clock cycle at the output.

stop()

Stop the toggling of the clock and set the output level to 0.

class artiq.coredevice.ttl.**TTLInOut**(*dmgr*, *channel*, *gate_latency_mu=None*, *core_device='core'*)

RTIO TTL input/output driver.

In output mode, provides functions to set the logic level on the signal.

In input mode, provides functions to analyze the incoming signal, with real-time gating to prevent overflows.

RTIO TTLs supports zero-length transition suppression. For example, if two pulses are emitted back-to-back with no delay between them, they will be merged into a single pulse with a duration equal to the sum of the durations of the original pulses.

This should be used with bidirectional channels.

Note that the channel is in input mode by default. If you need to drive a signal, you must call *output()*. If the channel is in output mode most of the time in your setup, it is a good idea to call *output()* in the startup kernel.

There are three input APIs: gating, sampling and watching. When one API is active (e.g. the gate is open, or the input events have not been fully read out), another API must not be used simultaneously.

Parameters **channel** – channel number

count(*up_to_timestamp_mu*)

Consume RTIO input events until the hardware timestamp counter has reached the specified timestamp and return the number of observed events.

This function does not interact with the timeline cursor.

See the gate_*() family of methods to select the input transitions that generate events, and *timestamp_mu()* to obtain the timestamp of the first event rather than an accumulated count.

Parameters **up_to_timestamp_mu** – The timestamp up to which execution is blocked, that is, up to which input events are guaranteed to be taken into account. (Events with later timestamps might still be registered if they are already available.)

Returns The number of events before the timeout elapsed (0 if none observed).

Examples

To count events on channel `ttl_input`, up to the current timeline position:

```
ttl_input.count(now_mu())
```

If other events are scheduled between the end of the input gate period and when the number of events is counted, using `now_mu()` as timeout consumes an unnecessary amount of timeline slack. In such cases, it can be beneficial to pass a more precise timestamp, for example:

```
gate_end_mu = ttl_input.gate_rising(100 * us)

# Schedule a long pulse sequence, represented here by a delay.
delay(10 * ms)

# Get number of rising edges. This will block until the end of
# the gate window, but does not wait for the long pulse sequence
# afterwards, thus (likely) completing with a large amount of
# slack left.
num_rising_edges = ttl_input.count(gate_end_mu)
```

The `gate_*()` family of methods return the cursor at the end of the window, allowing this to be expressed in a compact fashion:

```
ttl_input.count(ttl_input.gate_rising(100 * us))
```

gate_both(*duration*)

Register both rising and falling edge events for the specified duration (in seconds).

The time cursor is advanced by the specified duration.

Returns The timeline cursor at the end of the gate window, for convenience when used with *count()*/*timestamp_mu()*.

gate_both_mu(*duration*)

Register both rising and falling edge events for the specified duration (in machine units).

The time cursor is advanced by the specified duration.

Returns The timeline cursor at the end of the gate window, for convenience when used with *count()*/*timestamp_mu()*.

gate_falling(*duration*)

Register falling edge events for the specified duration (in seconds).

The time cursor is advanced by the specified duration.

Returns The timeline cursor at the end of the gate window, for convenience when used with *count()*/*timestamp_mu()*.

gate_falling_mu(*duration*)

Register falling edge events for the specified duration (in machine units).

The time cursor is advanced by the specified duration.

Returns The timeline cursor at the end of the gate window, for convenience when used with *count()*/*timestamp_mu()*.

gate_rising(*duration*)

Register rising edge events for the specified duration (in seconds).

The time cursor is advanced by the specified duration.

Returns The timeline cursor at the end of the gate window, for convenience when used with *count()*/*timestamp_mu()*.

gate_rising_mu(*duration*)

Register rising edge events for the specified duration (in machine units).

The time cursor is advanced by the specified duration.

Returns The timeline cursor at the end of the gate window, for convenience when used with *count()*/*timestamp_mu()*.

input()

Set the direction to input at the current position of the time cursor.

There must be a delay of at least one RTIO clock cycle before any other command can be issued.

This method only configures the direction at the FPGA. When using buffered I/O interfaces, such as the Sinara TTL cards, the buffer direction must be configured separately in the hardware.

off()

Set the output to a logic low state at the current position of the time cursor.

The channel must be in output mode.

The time cursor is not modified by this function.

on()

Set the output to a logic high state at the current position of the time cursor.

The channel must be in output mode.

The time cursor is not modified by this function.

output()

Set the direction to output at the current position of the time cursor.

There must be a delay of at least one RTIO clock cycle before any other command can be issued.

This method only configures the direction at the FPGA. When using buffered I/O interfaces, such as the Sinara TTL cards, the buffer direction must be configured separately in the hardware.

pulse(*duration*)

Pulse the output high for the specified duration (in seconds).

The time cursor is advanced by the specified duration.

pulse_mu(*duration*)

Pulse the output high for the specified duration (in machine units).

The time cursor is advanced by the specified duration.

sample_get()

Returns the value of a sample previously obtained with *sample_input()*.

Multiple samples may be queued (using multiple calls to *sample_input()*) into the RTIO FIFOs and subsequently read out using multiple calls to this function.

This function does not interact with the time cursor.

sample_get_nonrt()

Convenience function that obtains the value of a sample at the position of the time cursor, breaks realtime, and returns the sample value.

sample_input()

Instructs the RTIO core to read the value of the TTL input at the position of the time cursor.

The time cursor is not modified by this function.

timestamp_mu(*up_to_timestamp_mu*)

Return the timestamp of the next RTIO input event, or -1 if the hardware timestamp counter reaches the given value before an event is received.

This function does not interact with the timeline cursor.

See the `gate_*()` family of methods to select the input transitions that generate events, and *count()* for usage examples.

Parameters **up_to_timestamp_mu** – The timestamp up to which execution is blocked, that is, up to which input events are guaranteed to be taken into account. (Events with later timestamps might still be registered if they are already available.)

Returns The timestamp (in machine units) of the first event received; -1 on timeout.

watch_done()

Stop watching the input at the position of the time cursor.

Returns `True` if the input has not changed state while it was being watched.

The time cursor is not modified by this function. This function always makes the slack negative.

watch_stay_off()

Like *watch_stay_on()*, but for low levels.

watch_stay_on()

Checks that the input is at a high level at the position of the time cursor and keep checking until *watch_done()* is called.

Returns `True` if the input is high. A call to this function must always be followed by an eventual call to *watch_done()* (use e.g. a try/finally construct to ensure this).

The time cursor is not modified by this function.

class artiq.coredevice.ttl.**TTLOut**(*dmgr*, *channel*, *core_device='core'*)

RTIO TTL output driver.

This should be used with output-only channels.

Parameters **channel** – channel number

off()

Set the output to a logic low state at the current position of the time cursor.

The time cursor is not modified by this function.

on()

Set the output to a logic high state at the current position of the time cursor.

The time cursor is not modified by this function.

pulse(*duration*)

Pulse the output high for the specified duration (in seconds).

The time cursor is advanced by the specified duration.

pulse_mu(*duration*)

Pulse the output high for the specified duration (in machine units).

The time cursor is advanced by the specified duration.

14.2.2 artiq.coredevice.edge_counter module

Driver for RTIO-enabled TTL edge counter.

Like for the TTL input PHYs, sensitivity can be configured over RTIO (`gate_rising()`, etc.). In contrast to the former, however, the count is accumulated in gateware, and only a single input event is generated at the end of each gate period:

```
with parallel:
    doppler_cool()
    self.pmt_counter.gate_rising(1 * ms)

with parallel:
    readout()
    self.pmt_counter.gate_rising(100 * us)

print("Doppler cooling counts:", self.pmt_counter.fetch_count())
print("Readout counts:", self.pmt_counter.fetch_count())
```

For applications where the timestamps of the individual input events are not required, this has two advantages over `TTLInOut.count()` beyond raw throughput. First, it is easy to count events during multiple separate periods without blocking to read back counts in between, as illustrated in the above example. Secondly, as each count total only takes up a single input event, it is much easier to acquire counts on several channels in parallel without risking input FIFO overflows:

```
# Using the TTLInOut driver, pmt_1 input events are only processed
# after pmt_0 is done counting. To avoid RTIOOverflows, a round-robin
# scheme would have to be implemented manually.

with parallel:
    self.pmt_0.gate_rising(10 * ms)
    self.pmt_1.gate_rising(10 * ms)

counts_0 = self.pmt_0.count(now_mu()) # blocks
counts_1 = self.pmt_1.count(now_mu())

#

# Using gateware counters, only a single input event each is
# generated, greatly reducing the load on the input FIFOs:

with parallel:
    self.pmt_0_counter.gate_rising(10 * ms)
    self.pmt_1_counter.gate_rising(10 * ms)
```

(continues on next page)

(continued from previous page)

```
counts_0 = self.pmt_0_counter.fetch_count() # blocks
counts_1 = self.pmt_1_counter.fetch_count()
```

See `artiq.gateware.rtio.phy.edge_counter` and `artiq.gateware.eem.DIO.add_std()` for the gateware components.

exception artiq.coredevice.edge_counter.**CounterOverflow**

Raised when an edge counter value is read which indicates that the counter might have overflowed.

class artiq.coredevice.edge_counter.**EdgeCounter**(*dmgr*, *channel*, *gateware_width=31*, *core_device='core'*)

RTIO TTL edge counter driver driver.

Like for regular TTL inputs, timeline periods where the counter is sensitive to a chosen set of input transitions can be specified. Unlike the former, however, the specified edges do not create individual input events; rather, the total count can be requested as a single input event from the core (typically at the end of the gate window).

Parameters

- **channel** – The RTIO channel of the gateware phy.
- **gateware_width** – The width of the gateware counter register, in bits. This is only used for overflow handling; to change the size, the gateware needs to be rebuilt.

fetch_count() → numpy.int32

Wait for and return count total from previously requested input event.

It is valid to trigger multiple gate periods without immediately reading back the count total; the results will be returned in order on subsequent fetch calls.

This function blocks until a result becomes available.

fetch_timestamped_count(*timeout_mu=<Mock name='mock.int64()' id='140737283821088'>*) -> (*numpy.int64*, *numpy.int32*)

Wait for and return the timestamp and count total of a previously requested input event.

It is valid to trigger multiple gate periods without immediately reading back the count total; the results will be returned in order on subsequent fetch calls.

This function blocks until a result becomes available or the given timeout elapses.

Returns A tuple of timestamp (-1 if timeout elapsed) and counter value. (The timestamp is that of the requested input event – typically the gate closing time – and not that of any input edges.)

gate_both(*duration*)

Count both rising and falling edges for the given duration, and request the total at the end.

The counter is reset at the beginning of the gate period. Use *set_config()* directly for more detailed control.

Parameters **duration** – The duration for which the gate is to stay open.

Returns The timestamp at the end of the gate period, in machine units.

gate_both_mu(*duration_mu*)

See *gate_both_mu()*.

gate_falling(*duration*)

Count falling edges for the given duration and request the total at the end.

The counter is reset at the beginning of the gate period. Use *set_config()* directly for more detailed control.

Parameters **duration** – The duration for which the gate is to stay open.

Returns The timestamp at the end of the gate period, in machine units.

gate_falling_mu(*duration_mu*)

See *gate_falling()*.

gate_rising(*duration*)

Count rising edges for the given duration and request the total at the end.

The counter is reset at the beginning of the gate period. Use *set_config()* directly for more detailed control.

Parameters **duration** – The duration for which the gate is to stay open.

Returns The timestamp at the end of the gate period, in machine units.

gate_rising_mu(*duration_mu*)

See *gate_rising()*.

set_config(*count_rising: bool*, *count_falling: bool*, *send_count_event: bool*, *reset_to_zero: bool*)

Emit an RTIO event at the current timeline position to set the gateware configuration.

For most use cases, the *gate_** wrappers will be more convenient.

Parameters

- **count_rising** – Whether to count rising signal edges.
- **count_falling** – Whether to count falling signal edges.
- **send_count_event** – If *True*, an input event with the current counter value is generated on the next clock cycle (once).
- **reset_to_zero** – If *True*, the counter value is reset to zero on the next clock cycle (once).

14.2.3 artiq.coredevice.shiftreg module

class artiq.coredevice.shiftreg.**ShiftReg**(*dmgr*, *clk*, *ser*, *latch*, *n=32*, *dt=9.999999999999999e-06*, *ser_in=None*)

Driver for shift registers/latch combos connected to TTLs

set(*data*)

Sets the values of the latch outputs. This does not advance the timeline and the waveform is generated before *now*.

14.2.4 artiq.coredevice.spi2 module

Driver for generic SPI on RTIO.

This ARTIQ coredevice driver corresponds to the "new" MiSoC SPI core (v2).

Output event replacement is not supported and issuing commands at the same time is an error.

class artiq.coredevice.spi2.**NRTSPIMaster**(*dmgr*, *busno=0*, *core_device='core'*)

Core device non-realtime Serial Peripheral Interface (SPI) bus master. Owns one non-realtime SPI bus.

With this driver, SPI transactions and are performed by the CPU without involving RTIO.

Realtime and non-realtime buses are separate and defined at bitstream compilation time.

See *SPIMaster* for a description of the methods.

set_config_mu(*flags=0*, *length=8*, *div=6*, *cs=1*)

Set the `config` register.

Note that the non-realtime SPI cores are usually clocked by the system clock and not the RTIO clock. In many cases, the SPI configuration is already set by the firmware and you do not need to call this method.

class artiq.coredevice.spi2.**SPIMaster**(*dmgr*, *channel*, *div=0*, *length=0*, *core_device='core'*)

Core device Serial Peripheral Interface (SPI) bus master.

Owns one SPI bus.

This ARTIQ coredevice driver corresponds to the "new" MiSoC SPI core (v2).

Transfer Sequence:

- If necessary, set the `config` register (*set_config()* and *set_config_mu()*) to activate and configure the core and to set various transfer parameters like transfer length, clock divider, and chip selects.
- *write()* to the `data` register. Writing starts the transfer.
- If the transfer included submitting the SPI input data as an RTIO input event (`SPI_INPUT` set), then *read()* the `data`.
- If `SPI_END` was not set, repeat the transfer sequence.

A **transaction** consists of one or more **transfers**. The chip select pattern is asserted for the entire length of the transaction. All but the last transfer are submitted with `SPI_END` cleared in the configuration register.

Parameters

- **channel** – RTIO channel number of the SPI bus to control.
- **div** – Initial CLK divider, see also: *update_xfer_duration_mu()*
- **length** – Initial transfer length, see also: *update_xfer_duration_mu()*
- **core_device** – Core device name

frequency_to_div(*f*)

Convert a SPI clock frequency to the closest SPI clock divider.

read()

Read SPI data submitted by the SPI core.

For bit alignment and bit ordering see *set_config()*.

This method does not alter the timeline.

Returns SPI input data.

`set_config`(*flags*, *length*, *freq*, *cs*)

Set the configuration register.

- If `SPI_CS_POLARITY` is cleared (`cs` active low, the default), "`cs` all deasserted" means "all `cs_n` bits high".
- `cs_n` is not mandatory in the pads supplied to the gateware core. Framing and chip selection can also be handled independently through other means, e.g. `TTLOut`.
- If there is a `miso` wire in the pads supplied in the gateware, input and output may be two signals ("4-wire SPI"), otherwise `mosi` must be used for both output and input ("3-wire SPI") and `SPI_HALF_DUPLEX` must to be set when reading data or when the slave drives the `mosi` signal at any point.
- The first bit output on `mosi` is always the MSB/LSB (depending on `SPI_LSB_FIRST`) of the `data` written, independent of the `length` of the transfer. The last bit input from `miso` always ends up in the LSB/MSB (respectively) of the `data` read, independent of the `length` of the transfer.
- `cs` is asserted at the beginning and deasserted at the end of the transaction.
- `cs` handling is agnostic to whether it is one-hot or decoded somewhere downstream. If it is decoded, "`cs` all deasserted" should be handled accordingly (no slave selected). If it is one-hot, asserting multiple slaves should only be attempted if `miso` is either not connected between slaves, or open collector, or correctly multiplexed externally.
- Changes to the configuration register take effect on the start of the next transfer with the exception of `SPI_OFFLINE` which takes effect immediately.
- The SPI core can only be written to when it is idle or waiting for the next transfer data. Writing (*set_config()*, *set_config_mu()* or *write()*) when the core is busy will result in an RTIO busy error being logged.

This method advances the timeline by one coarse RTIO clock cycle.

Configuration flags:

- `SPI_OFFLINE`: all pins high-z (reset=1)
- `SPI_END`: transfer in progress (reset=1)
- `SPI_INPUT`: submit SPI read data as RTIO input event when transfer is complete (reset=0)
- `SPI_CS_POLARITY`: active level of `cs_n` (reset=0)
- `SPI_CLK_POLARITY`: idle level of `clk` (reset=0)
- `SPI_CLK_PHASE`: first edge after `cs` assertion to sample data on (reset=0). In Motorola/Freescale SPI language (`SPI_CLK_POLARITY`, `SPI_CLK_PHASE`) == (CPOL, CPHA):
  - (0, 0): idle low, output on falling, input on rising
  - (0, 1): idle low, output on rising, input on falling
  - (1, 0): idle high, output on rising, input on falling
  - (1, 1): idle high, output on falling, input on rising
- `SPI_LSB_FIRST`: LSB is the first bit on the wire (reset=0)
- `SPI_HALF_DUPLEX`: 3-wire SPI, in/out on `mosi` (reset=0)

Parameters

- **`flags`** – A bit map of *SPI_** flags.
- **`length`** – Number of bits to write during the next transfer. (reset=1)

- **freq** – Desired SPI clock frequency. (reset=f_rtio/2)
- **cs** – Bit pattern of chip selects to assert. Or number of the chip select to assert if `cs` is decoded downstream. (reset=0)

set_config_mu(*flags*, *length*, *div*, *cs*)

Set the `config` register (in SPI bus machine units).

See also:

set_config()

Parameters

- **flags** – A bit map of *SPI_** flags.
- **length** – Number of bits to write during the next transfer. (reset=1)
- **div** – Counter load value to divide the RTIO clock by to generate the SPI clock. (minimum=2, reset=2) `f_rtio_clk/f_spi == div`. If `div` is odd, the setup phase of the SPI clock is one coarse RTIO clock cycle longer than the hold phase.
- **cs** – Bit pattern of chip selects to assert. Or number of the chip select to assert if `cs` is decoded downstream. (reset=0)

update_xfer_duration_mu(*div*, *length*)

Calculate and set the transfer duration.

This method updates the SPI transfer duration which is used in *write()* to advance the timeline.

Use this method (and avoid having to call *set_config_mu()*) when the divider and transfer length have been configured (using *set_config()* or *set_config_mu()*) by previous experiments and are known.

This method is portable and can also be called from e.g. `__init__()`.

Warning: If this method is called while recording a DMA sequence, the playback of the sequence will not update the driver state. When required, update the driver state manually (by calling this method) after playing back a DMA sequence.

Parameters

- **div** – SPI clock divider (see: *set_config_mu()*)
- **length** – SPI transfer length (see: *set_config_mu()*)

write(*data*)

Write SPI data to shift register register and start transfer.

- The `data` register and the shift register are 32 bits wide.
- Data writes take one `ref_period` cycle.
- A transaction consisting of a single transfer (`SPI_END`) takes `xfer_duration_mu =(n + 1)*div` cycles RTIO time where `n` is the number of bits and `div` is the SPI clock divider.
- Transfers in a multi-transfer transaction take up to one SPI clock cycle less time depending on multiple parameters. Advanced users may rewind the timeline appropriately to achieve faster multi-transfer transactions.
- The SPI core will be busy for the duration of the SPI transfer.

- For bit alignment and bit ordering see *set_config()*.
- The SPI core can only be written to when it is idle or waiting for the next transfer data. Writing (*set_config()*, *set_config_mu()* or *write()*) when the core is busy will result in an RTIO busy error being logged.

This method advances the timeline by the duration of one single-transfer SPI transaction (`xfer_duration_mu`).

Parameters `data` – SPI output data to be written.

14.2.5 `artiq.coredevice.i2c` module

Non-realtime drivers for I2C chips on the core device.

`class artiq.coredevice.i2c.I2CSwitch(dmgr, busno=0, address=232, core_device='core')`

Driver for the I2C bus switch.

PCA954X (or other) type detection is done by the CPU during I2C init.

I2C transactions not real-time, and are performed by the CPU without involving RTIO.

On the KC705, this chip is used for selecting the I2C buses on the two FMC connectors. HPC=1, LPC=2.

`set(channel)`

Enable one channel.

Parameters `channel` – channel number (0-7)

`unset()`

Disable output of the I2C switch.

`class artiq.coredevice.i2c.PCF8574A(dmgr, busno=0, address=124, core_device='core')`

Driver for the PCF8574 I2C remote 8-bit I/O expander.

I2C transactions not real-time, and are performed by the CPU without involving RTIO.

`get()`

Retrieve quasi-bidirectional pin input data.

Returns Pin data

`set(data)`

Drive data on the quasi-bidirectional pins.

Parameters `data` – Pin data. High bits are weakly driven high (and thus inputs), low bits are strongly driven low.

`class artiq.coredevice.i2c.TCA6424A(dmgr, busno=0, address=68, core_device='core')`

Driver for the TCA6424A I2C I/O expander.

I2C transactions not real-time, and are performed by the CPU without involving RTIO.

On the NIST QC2 hardware, this chip is used for switching the directions of TTL buffers.

`set(outputs)`

Drive all pins of the chip to the levels given by the specified 24-bit word.

On the QC2 hardware, the LSB of the word determines the direction of TTL0 (on a given FMC card) and the MSB that of TTL23.

A bit set to 1 means the TTL is an output.

artiq.coredevice.i2c.**i2c_poll**(*busno*, *busaddr*)

Poll I2C device at address.

Parameters

- **busno** – I2C bus number
- **busaddr** – 8 bit I2C device address (LSB=0)

Returns True if the poll was ACKed

artiq.coredevice.i2c.**i2c_read_byte**(*busno*, *busaddr*)

Read one byte from a device.

Parameters

- **busno** – I2C bus number
- **busaddr** – 8 bit I2C device address (LSB=0)

Returns Byte read

artiq.coredevice.i2c.**i2c_read_many**(*busno*, *busaddr*, *addr*, *data*)

Transfer multiple bytes from a device.

Parameters

- **busno** – I2c bus number
- **busaddr** – 8 bit I2C device address (LSB=0)
- **addr** – 8 bit data address
- **data** – List of integers to be filled with the data read. One entry ber byte.

artiq.coredevice.i2c.**i2c_write_byte**(*busno*, *busaddr*, *data*, *ack=True*)

Write one byte to a device.

Parameters

- **busno** – I2C bus number
- **busaddr** – 8 bit I2C device address (LSB=0)
- **data** – Data byte to be written
- **nack** – Allow NACK

artiq.coredevice.i2c.**i2c_write_many**(*busno*, *busaddr*, *addr*, *data*, *ack_last=True*)

Transfer multiple bytes to a device.

Parameters

- **busno** – I2c bus number
- **busaddr** – 8 bit I2C device address (LSB=0)
- **addr** – 8 bit data address
- **data** – Data bytes to be written
- **ack_last** – Expect I2C ACK of the last byte written. If *False*, the last byte may be NACKed (e.g. EEPROM full page writes).

14.3 RF generation drivers

14.3.1 artiq.coredevice.urukul module

class artiq.coredevice.urukul.**CPLD**(*dmgr*, *spi_device*, *io_update_device=None*, *dds_reset_device=None*, *sync_device=None*, *sync_sel=0*, *clk_sel=0*, *clk_div=0*, *rf_sw=0*, *refclk=125000000.0*, *att=0*, *sync_div=None*, *core_device='core'*)

Urukul CPLD SPI router and configuration interface.

Parameters

- **spi_device** – SPI bus device name
- **io_update_device** – IO update RTIO TTLOut channel name
- **dds_reset_device** – DDS reset RTIO TTLOut channel name
- **sync_device** – AD9910 SYNC_IN RTIO TTLClockGen channel name
- **refclk** – Reference clock (SMA, MMCX or on-board 100 MHz oscillator) frequency in Hz
- **clk_sel** – Reference clock selection. For hardware revision >= 1.3 valid options are: 0 - internal 100MHz XO; 1 - front-panel SMA; 2 internal MMCX. For hardware revision <= v1.2 valid options are: 0 - either XO or MMCX dependent on component population; 1 SMA. Unsupported clocking options are silently ignored.
- **clk_div** – Reference clock divider. Valid options are 0: variant dependent default (divide-by-4 for AD9910 and divide-by-1 for AD9912); 1: divide-by-1; 2: divide-by-2; 3: divide-by-4. On Urukul boards with CPLD gateware before v1.3.1 only the default (0, i.e. variant dependent divider) is valid.
- **sync_sel** – SYNC (multi-chip synchronisation) signal source selection. 0 corresponds to SYNC_IN being supplied by the FPGA via the EEM connector. 1 corresponds to SYNC_OUT from DDS0 being distributed to the other chips.
- **rf_sw** – Initial CPLD RF switch register setting (default: 0x0). Knowledge of this state is not transferred between experiments.
- **att** – Initial attenuator setting shift register (default: 0x00000000). See also *get_att_mu()* which retrieves the hardware state without side effects. Knowledge of this state is not transferred between experiments.
- **sync_div** – SYNC_IN generator divider. The ratio between the coarse RTIO frequency and the SYNC_IN generator frequency (default: 2 if *sync_device* was specified).
- **core_device** – Core device name

If the clocking is incorrect (for example, setting `clk_sel` to the front panel SMA with no clock connected), then the `init()` method of the DDS channels can fail with the error message `PLL lock timeout`.

att_to_mu(*att: float*) → numpy.int32

Convert an attenuation setting in dB to machine units.

Parameters **att** – Attenuation setting in dB.

Returns Digital attenuation setting.

cfg_sw(*channel: numpy.int32*, *on: bool*)

Configure the RF switches through the configuration register.

These values are logically OR-ed with the LVDS lines on EEM1.

Parameters

- **channel** – Channel index (0-3)
- **on** – Switch value

cfg_switches(*state: numpy.int32*)

Configure all four RF switches through the configuration register.

Parameters **state** – RF switch state as a 4 bit integer.

cfg_write(*cfg: numpy.int32*)

Write to the configuration register.

See *urukul_cfg()* for possible flags.

Parameters **cfg** – 24 bit data to be written. Will be stored at `cfg_reg`.

get_att_mu() → numpy.int32

Return the digital step attenuator settings in machine units.

The result is stored and will be used in future calls of *set_att_mu()* and *set_att()*.

See also:

get_channel_att_mu()

Returns 32 bit attenuator settings

get_channel_att(*channel: numpy.int32*) → float

Get digital step attenuator value for a channel in SI units.

See also:

get_channel_att_mu()

Parameters **channel** – Attenuator channel (0-3).

Returns Attenuation setting in dB. Higher value is more attenuation. Minimum attenuation is 0*dB, maximum attenuation is 31.5*dB.

get_channel_att_mu(*channel: numpy.int32*) → numpy.int32

Get digital step attenuator value for a channel in machine units.

The result is stored and will be used in future calls of *set_att_mu()* and *set_att()*.

See also:

get_att_mu()

Parameters **channel** – Attenuator channel (0-3).

Returns 8-bit digital attenuation setting: 255 minimum attenuation, 0 maximum attenuation (31.5 dB)

init(*blind: bool = False*)

Initialize and detect Urukul.

Resets the DDS I/O interface and verifies correct CPLD gateware version. Does not pulse the DDS MASTER_RESET as that confuses the AD9910.

Parameters **blind** – Do not attempt to verify presence and compatibility.

io_rst()

Pulse IO_RST

mu_to_att(*att_mu: numpy.int32*) → float

Convert a digital attenuation setting to dB.

Parameters **att_mu** – Digital attenuation setting.

Returns Attenuation setting in dB.

set_all_att_mu(*att_reg: numpy.int32*)

Set all four digital step attenuators (in machine units).

See also:

set_att_mu()

Parameters **att_reg** – Attenuator setting string (32 bit)

set_att(*channel: numpy.int32*, *att: float*)

Set digital step attenuator in SI units.

This method will write the attenuator settings of all four channels.

See also:

set_att_mu()

Parameters

- **channel** – Attenuator channel (0-3).
- **att** – Attenuation setting in dB. Higher value is more attenuation. Minimum attenuation is 0*dB, maximum attenuation is 31.5*dB.

set_att_mu(*channel: numpy.int32*, *att: numpy.int32*)

Set digital step attenuator in machine units.

This method will also write the attenuator settings of the three other channels. Use *get_att_mu()* to retrieve the hardware state set in previous experiments.

Parameters

- **channel** – Attenuator channel (0-3).
- **att** – 8-bit digital attenuation setting: 255 minimum attenuation, 0 maximum attenuation (31.5 dB)

set_profile(*profile: numpy.int32*)

Set the PROFILE pins.

The PROFILE pins are common to all four DDS channels.

Parameters **profile** – PROFILE pins in numeric representation (0-7).

set_sync_div(*div: numpy.int32*)

Set the SYNC_IN AD9910 pulse generator frequency and align it to the current RTIO timestamp.

The SYNC_IN signal is derived from the coarse RTIO clock and the divider must be a power of two. Configure `sync_sel == 0`.

Parameters **div** – SYNC_IN frequency divider. Must be a power of two. Minimum division ratio is 2. Maximum division ratio is 16.

sta_read() → numpy.int32

Read the status register.

Use any of the following functions to extract values:

- *urukul_sta_rf_sw()*
- *urukul_sta_smp_err()*
- *urukul_sta_pll_lock()*
- *urukul_sta_ifc_mode()*
- *urukul_sta_proto_rev()*

Returns The status register value.

artiq.coredevice.urukul.**urukul_cfg**(*rf_sw*, *led*, *profile*, *io_update*, *mask_nu*, *clk_sel*, *sync_sel*, *rst*, *io_rst*, *clk_div*)

Build Urukul CPLD configuration register

artiq.coredevice.urukul.**urukul_sta_ifc_mode**(*sta*)

Return the IFC_MODE status from Urukul status register value.

artiq.coredevice.urukul.**urukul_sta_pll_lock**(*sta*)

Return the PLL_LOCK status from Urukul status register value.

artiq.coredevice.urukul.**urukul_sta_proto_rev**(*sta*)

Return the PROTO_REV value from Urukul status register value.

artiq.coredevice.urukul.**urukul_sta_rf_sw**(*sta*)

Return the RF switch status from Urukul status register value.

artiq.coredevice.urukul.**urukul_sta_smp_err**(*sta*)

Return the SMP_ERR status from Urukul status register value.

14.3.2 artiq.coredevice.ad9910 module

class artiq.coredevice.ad9910.**AD9910**(*dmgr*, *chip_select*, *cpld_device*, *sw_device=None*, *pll_n=40*, *pll_cp=7*, *pll_vco=5*, *sync_delay_seed=- 1*, *io_update_delay=0*, *pll_en=1*)

AD9910 DDS channel on Urukul.

This class supports a single DDS channel and exposes the DDS, the digital step attenuator, and the RF switch.

Parameters

- **chip_select** – Chip select configuration. On Urukul this is an encoded chip select and not "one-hot": 3 to address multiple chips (as configured through CFG_MASK_NU), 4-7 for individual channels.
- **cpld_device** – Name of the Urukul CPLD this device is on.
- **sw_device** – Name of the RF switch device. The RF switch is a TTLOut channel available as the `sw` attribute of this instance.
- **pll_n** – DDS PLL multiplier. The DDS sample clock is f_ref/clk_div*pll_n where f_ref is the reference frequency and clk_div is the reference clock divider (both set in the parent Urukul CPLD instance).

- **pll_en** – PLL enable bit, set to 0 to bypass PLL (default: 1). Note that when bypassing the PLL the red front panel LED may remain on.
- **pll_cp** – DDS PLL charge pump setting.
- **pll_vco** – DDS PLL VCO range selection.
- **sync_delay_seed** – SYNC_IN delay tuning starting value. To stabilize the SYNC_IN delay tuning, run *tune_sync_delay()* once and set this to the delay tap number returned (default: -1 to signal no synchronization and no tuning during *init()*). Can be a string of the form "eeprom_device:byte_offset" to read the value from a I2C EEPROM; in which case, *io_update_delay* must be set to the same string value.
- **io_update_delay** – IO_UPDATE pulse alignment delay. To align IO_UPDATE to SYNC_CLK, run *tune_io_update_delay()* and set this to the delay tap number returned. Can be a string of the form "eeprom_device:byte_offset" to read the value from a I2C EEPROM; in which case, *sync_delay_seed* must be set to the same string value.

amplitude_to_asf(*amplitude: float*) → numpy.int32

Return 14-bit amplitude scale factor corresponding to given fractional amplitude.

amplitude_to_ram(*amplitude: list(elt=float)*, *ram: list(elt=numpy.int32)*)

Convert amplitude values to RAM profile data.

To be used with `RAM_DEST_ASF`.

Parameters

- **amplitude** – List of amplitude values in units of full scale.
- **ram** – List to write RAM data into. Suitable for *write_ram()*.

asf_to_amplitude(*asf: numpy.int32*) → float

Return amplitude as a fraction of full scale corresponding to given amplitude scale factor.

cfg_sw(*state: bool*)

Set CPLD CFG RF switch state. The RF switch is controlled by the logical or of the CPLD configuration shift register RF switch bit and the SW TTL line (if used).

Parameters **state** – CPLD CFG RF switch bit

clear_smp_err()

Clear the SMP_ERR flag and enables SMP_ERR validity monitoring.

Violations of the SYNC_IN sample and hold margins will result in SMP_ERR being asserted. This then also activates the red LED on the respective Urukul channel.

Also modifies CFR2.

frequency_to_ftw(*frequency: float*) → numpy.int32

Return the 32-bit frequency tuning word corresponding to the given frequency.

frequency_to_ram(*frequency: list(elt=float)*, *ram: list(elt=numpy.int32)*)

Convert frequency values to RAM profile data.

To be used with `RAM_DEST_FTW`.

Parameters

- **frequency** – List of frequency values in Hz.
- **ram** – List to write RAM data into. Suitable for *write_ram()*.

ftw_to_frequency(*ftw: numpy.int32*) → float

Return the frequency corresponding to the given frequency tuning word.

get(*profile: numpy.int32 = 7*)

Get the frequency, phase, and amplitude.

See also:

get_mu()

Parameters **profile** – Profile number to get (0-7, default: 7)

Returns A tuple `(frequency, phase, amplitude)`

get_amplitude() → float

Get the value stored to the AD9910's amplitude scale factor (ASF) register.

Returns amplitude in units of full scale.

get_asf() → numpy.int32

Get the value stored to the AD9910's amplitude scale factor (ASF) register.

Returns Amplitude scale factor

get_att() → float

Get digital step attenuator value in SI units.

See also:

artiq.coredevice.urukul.CPLD.get_channel_att()

Returns Attenuation in dB.

get_att_mu() → numpy.int32

Get digital step attenuator value in machine units.

See also:

artiq.coredevice.urukul.CPLD.get_channel_att_mu()

Returns Attenuation setting, 8 bit digital.

get_frequency() → float

Get the value stored to the AD9910's frequency tuning word (FTW) register.

Returns frequency in Hz.

get_ftw() → numpy.int32

Get the value stored to the AD9910's frequency tuning word (FTW) register.

Returns Frequency tuning word

get_mu(*profile: numpy.int32 = 7*)

Get the frequency tuning word, phase offset word, and amplitude scale factor.

See also:

get()

Parameters **profile** – Profile number to get (0-7, default: 7)

Returns A tuple `(ftw, pow, asf)`

`get_phase()` → float

Get the value stored to the AD9910's phase offset word (POW) register.

Returns phase offset in turns.

`get_pow()` → numpy.int32

Get the value stored to the AD9910's phase offset word (POW) register.

Returns Phase offset word

`init`(*blind: bool = False*)

Initialize and configure the DDS.

Sets up SPI mode, confirms chip presence, powers down unused blocks, configures the PLL, waits for PLL lock. Uses the IO_UPDATE signal multiple times.

Parameters `blind` – Do not read back DDS identity and do not wait for lock.

`measure_io_update_alignment`(*delay_start: numpy.int64*, *delay_stop: numpy.int64*) → numpy.int32

Use the digital ramp generator to locate the alignment between IO_UPDATE and SYNC_CLK.

The ramp generator is set up to a linear frequency ramp (dFTW/t_SYNC_CLK=1) and started at a coarse RTIO time stamp plus *delay_start* and stopped at a coarse RTIO time stamp plus *delay_stop*.

Parameters

- `delay_start` – Start IO_UPDATE delay in machine units.
- `delay_stop` – Stop IO_UPDATE delay in machine units.

Returns Odd/even SYNC_CLK cycle indicator.

`pow_to_turns`(*pow_: numpy.int32*) → float

Return the phase in turns corresponding to a given phase offset word.

`power_down`(*bits: numpy.int32 = 15*)

Power down DDS.

Parameters `bits` – Power down bits, see datasheet

`read16`(*addr: numpy.int32*) → numpy.int32

Read from 16 bit register.

Parameters `addr` – Register address

`read32`(*addr: numpy.int32*) → numpy.int32

Read from 32 bit register.

Parameters `addr` – Register address

`read64`(*addr: numpy.int32*) → numpy.int64

Read from 64 bit register.

Parameters `addr` – Register address

Returns 64 bit integer register value

`read_ram`(*data: list(elt=numpy.int32)*)

Read data from RAM.

The profile to read from and the step, start, and end address need to be configured before and separately using `set_profile_ram()` and the parent CPLD *set_profile*.

Parameters **data** – List to be filled with data read from RAM.

set(*frequency: float = 0.0*, *phase: float = 0.0*, *amplitude: float = 1.0*, *phase_mode: numpy.int32 = -1*, *ref_time_mu: numpy.int64 = <Mock name='mock.int64()' id='140737283821088'>*, *profile: numpy.int32 = 7*, *ram_destination: numpy.int32 = -1*) → float

Set DDS data in SI units.

See also:

set_mu()

Parameters

- **frequency** – Frequency in Hz
- **phase** – Phase tuning word in turns
- **amplitude** – Amplitude in units of full scale
- **phase_mode** – Phase mode constant
- **ref_time_mu** – Fiducial time stamp in machine units
- **profile** – Single tone profile to affect.
- **ram_destination** – RAM destination.

Returns Resulting phase offset in turns

set_amplitude(*amplitude: float*)

Set the value stored to the AD9910's amplitude scale factor (ASF) register.

Parameters **amplitude** – amplitude to be stored, in units of full scale.

set_asf(*asf: numpy.int32*)

Set the value stored to the AD9910's amplitude scale factor (ASF) register.

Parameters **asf** – Amplitude scale factor to be stored, range: 0 to 0x3fff.

set_att(*att: float*)

Set digital step attenuator in SI units.

This method will write the attenuator settings of all four channels.

See also:

artiq.coredevice.urukul.CPLD.set_att()

Parameters **att** – Attenuation in dB.

set_att_mu(*att: numpy.int32*)

Set digital step attenuator in machine units.

This method will write the attenuator settings of all four channels.

See also:

artiq.coredevice.urukul.CPLD.set_att_mu()

Parameters **att** – Attenuation setting, 8 bit digital.

set_cfr1(*power_down: numpy.int32 = 0*, *phase_autoclear: numpy.int32 = 0*, *drg_load_lrr: numpy.int32 = 0*, *drg_autoclear: numpy.int32 = 0*, *phase_clear: numpy.int32 = 0*, *internal_profile: numpy.int32 = 0*, *ram_destination: numpy.int32 = 0*, *ram_enable: numpy.int32 = 0*, *manual_osk_external: numpy.int32 = 0*, *osk_enable: numpy.int32 = 0*, *select_auto_osk: numpy.int32 = 0*)

Set CFR1. See the AD9910 datasheet for parameter meanings.

This method does not pulse IO_UPDATE.

Parameters

- **power_down** – Power down bits.
- **phase_autoclear** – Autoclear phase accumulator.
- **phase_clear** – Asynchronous, static reset of the phase accumulator.
- **drg_load_lrr** – Load digital ramp generator LRR.
- **drg_autoclear** – Autoclear digital ramp generator.
- **internal_profile** – Internal profile control.
- **ram_destination** – RAM destination (RAM_DEST_FTW, RAM_DEST_POW, RAM_DEST_ASF, RAM_DEST_POWASF).
- **ram_enable** – RAM mode enable.
- **manual_osk_external** – Enable OSK pin control in manual OSK mode.
- **osk_enable** – Enable OSK mode.
- **select_auto_osk** – Select manual or automatic OSK mode.

set_cfr2(*asf_profile_enable: numpy.int32 = 1*, *drg_enable: numpy.int32 = 0*, *effective_ftw: numpy.int32 = 1*, *sync_validation_disable: numpy.int32 = 0*, *matched_latency_enable: numpy.int32 = 0*)

Set CFR2. See the AD9910 datasheet for parameter meanings.

This method does not pulse IO_UPDATE.

Parameters

- **asf_profile_enable** – Enable amplitude scale from single tone profiles.
- **drg_enable** – Digital ramp enable.
- **effective_ftw** – Read effective FTW.
- **sync_validation_disable** – Disable the SYNC_SMP_ERR pin indicating (active high) detection of a synchronization pulse sampling error.
- **matched_latency_enable** – Simultaneous application of amplitude, phase, and frequency changes to the DDS arrive at the output
  - matched_latency_enable = 0: in the order listed
  - matched_latency_enable = 1: simultaneously.

set_frequency(*frequency: float*)

Set the value stored to the AD9910's frequency tuning word (FTW) register.

Parameters **frequency** – frequency to be stored, in Hz.

set_ftw(*ftw: numpy.int32*)

Set the value stored to the AD9910's frequency tuning word (FTW) register.

Parameters **ftw** – Frequency tuning word to be stored, range: 0 to 0xffffffff.

set_mu(*ftw: numpy.int32 = 0*, *pow_: numpy.int32 = 0*, *asf: numpy.int32 = 16383*, *phase_mode: numpy.int32 = -1*, *ref_time_mu: numpy.int64 = <Mock name='mock.int64()' id='140737283821088'>*, *profile: numpy.int32 = 7*, *ram_destination: numpy.int32 = -1*) → numpy.int32

Set DDS data in machine units.

This uses machine units (FTW, POW, ASF). The frequency tuning word width is 32, the phase offset word width is 16, and the amplitude scale factor width is 14.

After the SPI transfer, the shared IO update pin is pulsed to activate the data.

Parameters

- **ftw** – Frequency tuning word: 32 bit.
- **pow** – Phase tuning word: 16 bit unsigned.
- **asf** – Amplitude scale factor: 14 bit unsigned.
- **phase_mode** – If specified, overrides the default phase mode set by *set_phase_mode()* for this call.
- **ref_time_mu** – Fiducial time used to compute absolute or tracking phase updates. In machine units as obtained by *now_mu()*.
- **profile** – Single tone profile number to set (0-7, default: 7). Ineffective if *ram_destination* is specified.
- **ram_destination** – RAM destination (RAM_DEST_FTW, RAM_DEST_POW, RAM_DEST_ASF, RAM_DEST_POWASF). If specified, write free DDS parameters to the ASF/FTW/POW registers instead of to the single tone profile register (default behaviour, see *profile*).

Returns Resulting phase offset word after application of phase tracking offset. When using PHASE_MODE_CONTINUOUS in subsequent calls, use this value as the "current" phase.

set_phase(*turns: float*)

Set the value stored to the AD9910's phase offset word (POW) register.

Parameters **turns** – phase offset to be stored, in turns.

set_phase_mode(*phase_mode: numpy.int32*)

Set the default phase mode.

for future calls to *set()* and *set_mu()*. Supported phase modes are:

- PHASE_MODE_CONTINUOUS: the phase accumulator is unchanged when changing frequency or phase. The DDS phase is the sum of the phase accumulator and the phase offset. The only discontinuous changes in the DDS output phase come from changes to the phase offset. This mode is also knows as "relative phase mode". $\phi(t) = q(t') + p + (t - t')f$
- PHASE_MODE_ABSOLUTE: the phase accumulator is reset when changing frequency or phase. Thus, the phase of the DDS at the time of the change is equal to the specified phase offset. $\phi(t) = p + (t - t')f$
- PHASE_MODE_TRACKING: when changing frequency or phase, the phase accumulator is cleared and the phase offset is offset by the value the phase accumulator would have if the DDS had been running at the specified frequency since a given fiducial time stamp. This is functionally equivalent to PHASE_MODE_ABSOLUTE. The only difference is the fiducial time stamp. This mode is also known as "coherent phase mode". The default fiducial time stamp is 0. $\phi(t) = p + (t - T)f$

Where:

- $\phi(t)$: the DDS output phase
- $q(t) = \phi(t) - p$: DDS internal phase accumulator

- p: phase offset
- f: frequency
- t': time stamp of setting p, f
- T: fiducial time stamp
- t: running time

Warning: This setting may become inconsistent when used as part of a DMA recording. When using DMA, it is recommended to specify the phase mode explicitly when calling `set()` or `set_mu()`.

`set_pow`(*pow_: numpy.int32*)

Set the value stored to the AD9910's phase offset word (POW) register.

Parameters **pow** – Phase offset word to be stored, range: 0 to 0xffff.

`set_profile_ram`(*start: numpy.int32*, *end: numpy.int32*, *step: numpy.int32 = 1*, *profile: numpy.int32 = 0*, *nodwell_high: numpy.int32 = 0*, *zero_crossing: numpy.int32 = 0*, *mode: numpy.int32 = 1*)

Set the RAM profile settings.

Parameters

- **start** – Profile start address in RAM.
- **end** – Profile end address in RAM (last address).
- **step** – Profile time step in units of t_DDS, typically 4 ns (default: 1).
- **profile** – Profile index (0 to 7) (default: 0).
- **nodwell_high** – No-dwell high bit (default: 0, see AD9910 documentation).
- **zero_crossing** – Zero crossing bit (default: 0, see AD9910 documentation).
- **mode** – Profile RAM mode (`RAM_MODE_DIRECTSWITCH`, `RAM_MODE_RAMPUP`, `RAM_MODE_BIDIR_RAMP`, `RAM_MODE_CONT_BIDIR_RAMP`, or `RAM_MODE_CONT_RAMPUP`, default: `RAM_MODE_RAMPUP`)

`set_sync`(*in_delay: numpy.int32*, *window: numpy.int32*, *en_sync_gen: numpy.int32 = 0*)

Set the relevant parameters in the multi device synchronization register. See the AD9910 datasheet for details. The SYNC clock generator preset value is set to zero, and the SYNC_OUT generator is disabled by default.

Parameters

- **in_delay** – SYNC_IN delay tap (0-31) in steps of ~75ps
- **window** – Symmetric SYNC_IN validation window (0-15) in steps of ~75ps for both hold and setup margin.
- **en_sync_gen** – Whether to enable the DDS-internal sync generator (SYNC_OUT, cf. sync_sel == 1). Should be left off for the normal use case, where the SYNC clock is supplied by the core device.

`tune_io_update_delay`() → numpy.int32

Find a stable IO_UPDATE delay alignment.

Scan through increasing IO_UPDATE delays until a delay is found that lets IO_UPDATE be registered in the next SYNC_CLK cycle. Return a IO_UPDATE delay that is as far away from that SYNC_CLK edge as possible.

This method assumes that the IO_UPDATE TTLOut device has one machine unit resolution (SERDES).

This method and *tune_sync_delay()* can be run in any order.

Returns Stable IO_UPDATE delay to be passed to the constructor *AD9910* via the device database.

tune_sync_delay(*search_seed: numpy.int32 = 15*)

Find a stable SYNC_IN delay.

This method first locates a valid SYNC_IN delay at zero validation window size (setup/hold margin) by scanning around *search_seed*. It then looks for similar valid delays at successively larger validation window sizes until none can be found. It then decreases the validation window a bit to provide some slack and stability and returns the optimal values.

This method and *tune_io_update_delay()* can be run in any order.

Parameters **search_seed** – Start value for valid SYNC_IN delay search. Defaults to 15 (half range).

Returns Tuple of optimal delay and window size.

turns_amplitude_to_ram(*turns: list(elt=float)*, *amplitude: list(elt=float)*, *ram: list(elt=numpy.int32)*)

Convert phase and amplitude values to RAM profile data.

To be used with `RAM_DEST_POWASF`.

Parameters

- **turns** – List of phase values in turns.
- **amplitude** – List of amplitude values in units of full scale.
- **ram** – List to write RAM data into. Suitable for *write_ram()*.

turns_to_pow(*turns: float*) → numpy.int32

Return the 16-bit phase offset word corresponding to the given phase in turns.

turns_to_ram(*turns: list(elt=float)*, *ram: list(elt=numpy.int32)*)

Convert phase values to RAM profile data.

To be used with `RAM_DEST_POW`.

Parameters

- **turns** – List of phase values in turns.
- **ram** – List to write RAM data into. Suitable for *write_ram()*.

write16(*addr: numpy.int32*, *data: numpy.int32*)

Write to 16 bit register.

Parameters

- **addr** – Register address
- **data** – Data to be written

write32(*addr: numpy.int32*, *data: numpy.int32*)

Write to 32 bit register.

Parameters

- **addr** – Register address
- **data** – Data to be written

write64(*addr: numpy.int32*, *data_high: numpy.int32*, *data_low: numpy.int32*)

Write to 64 bit register.

Parameters

- **addr** – Register address
- **data_high** – High (MSB) 32 bits of the data
- **data_low** – Low (LSB) 32 data bits

write_ram(*data: list(elt=numpy.int32)*)

Write data to RAM.

The profile to write to and the step, start, and end address need to be configured before and separately using *set_profile_ram()* and the parent CPLD *set_profile*.

Parameters **data** – Data to be written to RAM.

14.3.3 artiq.coredevice.ad9912 module

class artiq.coredevice.ad9912.**AD9912**(*dmgr*, *chip_select*, *cpld_device*, *sw_device=None*, *pll_n=10*)

AD9912 DDS channel on Urukul

This class supports a single DDS channel and exposes the DDS, the digital step attenuator, and the RF switch.

Parameters

- **chip_select** – Chip select configuration. On Urukul this is an encoded chip select and not "one-hot".
- **cpld_device** – Name of the Urukul CPLD this device is on.
- **sw_device** – Name of the RF switch device. The RF switch is a TTLOut channel available as the `sw` attribute of this instance.
- **pll_n** – DDS PLL multiplier. The DDS sample clock is f_ref/clk_div*pll_n where f_ref is the reference frequency and clk_div is the reference clock divider (both set in the parent Urukul CPLD instance).

cfg_sw(*state: bool*)

Set CPLD CFG RF switch state. The RF switch is controlled by the logical or of the CPLD configuration shift register RF switch bit and the SW TTL line (if used).

Parameters **state** – CPLD CFG RF switch bit

frequency_to_ftw(*frequency: float*) → numpy.int64

Returns the 48-bit frequency tuning word corresponding to the given frequency.

ftw_to_frequency(*ftw: numpy.int64*) → float

Returns the frequency corresponding to the given frequency tuning word.

`get()`

Get the frequency and phase.

See also:

get_mu()

Returns A tuple `(frequency, phase)`.

`get_att()` → float

Get digital step attenuator value in SI units.

See also:

artiq.coredevice.urukul.CPLD.get_channel_att()

Returns Attenuation in dB.

`get_att_mu()` → numpy.int32

Get digital step attenuator value in machine units.

See also:

artiq.coredevice.urukul.CPLD.get_channel_att_mu()

Returns Attenuation setting, 8 bit digital.

`get_mu()`

Get the frequency tuning word and phase offset word.

See also:

get()

Returns A tuple `(ftw, pow)`.

`init()`

Initialize and configure the DDS.

Sets up SPI mode, confirms chip presence, powers down unused blocks, and configures the PLL. Does not wait for PLL lock. Uses the IO_UPDATE signal multiple times.

`pow_to_turns`(*pow_: numpy.int32*) → float

Return the phase in turns corresponding to a given phase offset word.

Parameters `pow` – Phase offset word.

Returns Phase in turns.

`read`(*addr: numpy.int32*, *length: numpy.int32*) → numpy.int32

Variable length read from a register. Up to 4 bytes.

Parameters

- `addr` – Register address
- `length` – Length in bytes (1-4)

Returns Data read

set(*frequency: float*, *phase: float = 0.0*)

Set profile 0 data in SI units.

See also:

set_mu()

Parameters

- **frequency** – Frequency in Hz
- **phase** – Phase tuning word in turns

set_att(*att: float*)

Set digital step attenuator in SI units.

This method will write the attenuator settings of all four channels.

See also:

artiq.coredevice.urukul.CPLD.set_att()

Parameters **att** – Attenuation in dB. Higher values mean more attenuation.

set_att_mu(*att: numpy.int32*)

Set digital step attenuator in machine units.

This method will write the attenuator settings of all four channels.

See also:

artiq.coredevice.urukul.CPLD.set_att_mu()

Parameters **att** – Attenuation setting, 8 bit digital.

set_mu(*ftw: numpy.int64*, *pow_: numpy.int32 = 0*)

Set profile 0 data in machine units.

After the SPI transfer, the shared IO update pin is pulsed to activate the data.

Parameters

- **ftw** – Frequency tuning word: 48 bit unsigned.
- **pow** – Phase tuning word: 16 bit unsigned.

turns_to_pow(*phase: float*) → numpy.int32

Returns the 16-bit phase offset word corresponding to the given phase.

write(*addr: numpy.int32*, *data: numpy.int32*, *length: numpy.int32*)

Variable length write to a register. Up to 4 bytes.

Parameters

- **addr** – Register address
- **data** – Data to be written: int32
- **length** – Length in bytes (1-4)

14.3.4 artiq.coredevice.ad9914 module

Driver for the AD9914 DDS (with parallel bus) on RTIO.

class artiq.coredevice.ad9914.**AD9914**(*dmgr*, *sysclk*, *bus_channel*, *channel*, *core_device='core'*)

Driver for one AD9914 DDS channel.

The time cursor is not modified by any function in this class.

Output event replacement is not supported and issuing commands at the same time is an error.

Parameters

- **sysclk** – DDS system frequency. The DDS system clock must be a phase-locked multiple of the RTIO clock.
- **bus_channel** – RTIO channel number of the DDS bus.
- **channel** – channel number (on the bus) of the DDS device to control.

amplitude_to_asf(*amplitude*)

Returns 12-bit amplitude scale factor corresponding to given amplitude.

asf_to_amplitude(*asf*)

Returns the amplitude corresponding to the given amplitude scale factor.

exit_x()

Exits extended-resolution mode.

frequency_to_ftw(*frequency*)

Returns the 32-bit frequency tuning word corresponding to the given frequency.

frequency_to_xftw(*frequency*)

Returns the 63-bit frequency tuning word corresponding to the given frequency (extended resolution mode).

ftw_to_frequency(*ftw*)

Returns the frequency corresponding to the given frequency tuning word.

init()

Resets and initializes the DDS channel.

This needs to be done for each DDS channel before it can be used, and it is recommended to use the startup kernel for this purpose.

init_sync(*sync_delay*)

Resets and initializes the DDS channel as well as configures the AD9914 DDS for synchronisation. The synchronisation procedure follows the steps outlined in the AN-1254 application note.

This needs to be done for each DDS channel before it can be used, and it is recommended to use the startup kernel for this.

This function cannot be used in a batch; the correct way of initializing multiple DDS channels is to call this function sequentially with a delay between the calls. 10ms provides a good timing margin.

Parameters **sync_delay** – integer from 0 to 0x3f that sets the value of SYNC_OUT (bits 3-5) and SYNC_IN (bits 0-2) delay ADJ bits.

pow_to_turns(*pow*)

Returns the phase in turns corresponding to the given phase offset word.

set(*frequency*, *phase=0.0*, *phase_mode=- 1*, *amplitude=1.0*)

Like *set_mu()*, but uses Hz and turns.

set_mu(*ftw*, *pow=0*, *phase_mode=- 1*, *asf=4095*, *ref_time_mu=- 1*)

Sets the DDS channel to the specified frequency and phase.

This uses machine units (FTW and POW). The frequency tuning word width is 32, the phase offset word width is 16, and the amplitude scale factor width is 12.

The "frequency update" pulse is sent to the DDS with a fixed latency with respect to the current position of the time cursor.

Parameters

- **ftw** – frequency to generate.
- **pow** – adds an offset to the phase.
- **phase_mode** – if specified, overrides the default phase mode set by *set_phase_mode()* for this call.
- **ref_time_mu** – reference time used to compute phase. Specifying this makes it easier to have a well-defined phase relationship between DDSes on the same bus that are updated at a similar time.

Returns Resulting phase offset word after application of phase tracking offset. When using `PHASE_MODE_CONTINUOUS` in subsequent calls, use this value as the "current" phase.

set_phase_mode(*phase_mode*)

Sets the phase mode of the DDS channel. Supported phase modes are:

- `PHASE_MODE_CONTINUOUS`: the phase accumulator is unchanged when switching frequencies. The DDS phase is the sum of the phase accumulator and the phase offset. The only discrete jumps in the DDS output phase come from changes to the phase offset.
- `PHASE_MODE_ABSOLUTE`: the phase accumulator is reset when switching frequencies. Thus, the phase of the DDS at the time of the frequency change is equal to the phase offset.
- `PHASE_MODE_TRACKING`: when switching frequencies, the phase accumulator is set to the value it would have if the DDS had been running at the specified frequency since the start of the experiment.

Warning: This setting may become inconsistent when used as part of a DMA recording. When using DMA, it is recommended to specify the phase mode explicitly when calling *set()* or *set_mu()*.

set_x(*frequency*, *amplitude=1.0*)

Like *set_x_mu()*, but uses Hz and turns.

Note that the precision of `float` is less than the precision of the extended frequency tuning word.

set_x_mu(*xftw*, *amplitude=4095*)

Set the DDS frequency and amplitude with an extended-resolution (63-bit) frequency tuning word.

Phase control is not implemented in this mode; the phase offset can assume any value.

After this function has been called, exit extended-resolution mode before calling functions that use standard-resolution mode.

turns_to_pow(*turns*)

Returns the 16-bit phase offset word corresponding to the given phase in turns.

xftw_to_frequency(*xftw*)

Returns the frequency corresponding to the given frequency tuning word (extended resolution mode).

14.3.5 artiq.coredevice.mirny module

RTIO driver for Mirny (4 channel GHz PLLs)

class artiq.coredevice.mirny.**Almazny**(*High frequency mezzanine board for Mirny*)

Parameters **host_mirny** – Mirny device Almazny is connected to

att_to_mu(*att*)

Convert an attenuator setting in dB to machine units.

Parameters **att** – attenuator setting in dB [0-31.5]

Returns attenuator setting in machine units

mu_to_att(*att_mu*)

Convert a digital attenuator setting to dB.

Parameters **att_mu** – attenuator setting in machine units

Returns attenuator setting in dB

output_toggle(*oe*)

Toggles output on all shift registers on or off.

Parameters **oe** – toggle output enable (bool)

set_att(*channel*, *att*, *rf_switch=True*)

Sets attenuators on chosen shift register (channel).

Parameters

- **channel** – index of the register [0-3]
- **att** – attenuation setting in dBm [0-31.5]
- **rf_switch** – rf switch (bool)

set_att_mu(*channel*, *att_mu*, *rf_switch=True*)

Sets attenuators on chosen shift register (channel).

Parameters

- **channel** – index of the register [0-3]
- **att_mu** – attenuation setting in machine units [0-63]
- **rf_switch** – rf switch (bool)

class artiq.coredevice.mirny.**Mirny**(*dmgr*, *spi_device*, *refclk=100000000.0*, *clk_sel='XO'*, *core_device='core'*)

Mirny PLL-based RF generator.

Parameters

- **spi_device** – SPI bus device
- **refclk** – Reference clock (SMA, MMCX or on-board 100 MHz oscillator) frequency in Hz

- **clk_sel** – Reference clock selection. Valid options are: "XO" - onboard crystal oscillator; "SMA" - front-panel SMA connector; "MMCX" - internal MMCX connector. Passing an integer writes it as `clk_sel` in the CPLD's register 1. The effect depends on the hardware revision.
- **core_device** – Core device name (default: "core")

att_to_mu(*att*)

Convert an attenuation setting in dB to machine units.

Parameters **att** – Attenuation setting in dB.

Returns Digital attenuation setting.

init(*blind=False*)

Initialize and detect Mirny.

Select the clock source based the board's hardware revision. Raise ValueError if the board's hardware revision is not supported.

Parameters **blind** – Verify presence and protocol compatibility. Raise ValueError on failure.

read_reg(*addr*)

Read a register

set_att(*channel*, *att*)

Set digital step attenuator in SI units.

This method will write the attenuator settings of the selected channel.

See also:

set_att_mu()

Parameters

- **channel** – Attenuator channel (0-3).
- **att** – Attenuation setting in dB. Higher value is more attenuation. Minimum attenuation is 0*dB, maximum attenuation is 31.5*dB.

set_att_mu(*channel*, *att*)

Set digital step attenuator in machine units.

Parameters **att** – Attenuation setting, 8 bit digital.

write_ext(*addr*, *length*, *data*, *ext_div=4*)

Perform SPI write to a prefixed address

write_reg(*addr*, *data*)

Write a register

14.3.6 artiq.coredevice.adf5356 module

RTIO driver for the Analog Devices ADF[45]35[56] family of GHz PLLs on Mirny-style prefixed SPI buses.

class artiq.coredevice.adf5356.**ADF5356**(*dmgr*, *cpld_device*, *sw_device*, *channel*, *ref_doubler=False*, *ref_divider=False*, *core='core'*)

Analog Devices AD[45]35[56] family of GHz PLLs.

Parameters

- **cpld_device** – Mirny CPLD device name
- **sw_device** – Mirny RF switch device name
- **channel** – Mirny RF channel index
- **ref_doubler** – enable/disable reference clock doubler
- **ref_divider** – enable/disable reference clock divide-by-2
- **core_device** – Core device name (default: "core")

disable_output()

Disable output A of the PLL chip.

enable_output()

Enable output A of the PLL chip. This is the default after init.

f_pfd() → numpy.int64

Return the PFD frequency for the cached set of registers.

f_vco() → numpy.int64

Return the VCO frequency for the cached set of registers.

info()

Return a summary of high-level parameters as a dict.

init(*blind=False*)

Initialize and configure the PLL.

Parameters **blind** – Do not attempt to verify presence.

output_divider() → numpy.int32

Return the value of the output A divider.

output_power_mu()

Return the power level at output A of the PLL chip in machine units.

pll_frac1() → numpy.int32

Return the main fractional value (FRAC1) for the cached set of registers.

pll_frac2() → numpy.int32

Return the auxiliary fractional value (FRAC2) for the cached set of registers.

pll_mod2() → numpy.int32

Return the auxiliary modulus value (MOD2) for the cached set of registers.

pll_n() → numpy.int32

Return the PLL integer value (INT) for the cached set of registers.

read_muxout()

Read the state of the MUXOUT line.

By default, this is configured to be the digital lock detection.

ref_counter() → numpy.int32

Return the reference counter value (R) for the cached set of registers.

set_att(*att*)

Set digital step attenuator in SI units.

This method will write the attenuator settings of the channel.

See also:

artiq.coredevice.mirny.Mirny.set_att()

Parameters **att** – Attenuation in dB.

set_att_mu(*att*)

Set digital step attenuator in machine units.

Parameters **att** – Attenuation setting, 8 bit digital.

set_frequency(*f*)

Output given frequency on output A.

Parameters **f** – 53.125 MHz <= f <= 6800 MHz

set_output_power_mu(*n*)

Set the power level at output A of the PLL chip in machine units.

This driver defaults to $n = 3$ at init.

Parameters **n** – output power setting, 0, 1, 2, or 3 (see ADF5356 datasheet, fig. 44).

sync()

Write all registers to the device. Attempts to lock the PLL.

artiq.coredevice.adf5356.**calculate_pll**(*f_vco: numpy.int64*, *f_pfd: numpy.int64*)

Calculate fractional-N PLL parameters such that

`f_vco = f_pfd * (n + (frac1 + frac2/mod2) / mod1)`

where `mod1 = 2**24` and `mod2 <= 2**28`

Parameters

- **f_vco** – target VCO frequency
- **f_pfd** – PFD frequency

Returns `(n, frac1, (frac2_msb, frac2_lsb), (mod2_msb, mod2_lsb))`

14.3.7 artiq.coredevice.spline module

class artiq.coredevice.spline.**Spline**(*width*, *time_width*, *channel*, *core_device*, *scale=1.0*)

Spline interpolating RTIO channel.

One knot of a polynomial basis spline (B-spline) $u(t)$ is defined by the coefficients u_n up to order $n = k$. If the coefficients are evaluated starting at time t_0, the output $u(t)$ for $t > t_0, t_0$ is:

$$\begin{aligned} u(t) &= \sum_{n=0}^{k} \frac{u_n}{n!}(t - t_0)^n \\ &= u_0 + u_1(t - t_0) + \frac{u_2}{2}(t - t_0)^2 + \dots \end{aligned}$$

This class contains multiple methods to convert spline knot data from SI to machine units and multiple methods that set the current spline coefficient data. None of these advance the timeline. The *smooth()* method is the only method that advances the timeline.

Parameters

- **width** – Width in bits of the quantity that this spline controls
- **time_width** – Width in bits of the time counter of this spline
- **channel** – RTIO channel number
- **core_device** – Core device that this spline is attached to
- **scale** – Scale for conversion between machine units and physical units; to be given as the "full scale physical value".

coeff_as_packed(*coeff*)

Convert floating point spline coefficients into 32 bit integer packed data.

This is a host-only method that can be used to generate packed spline coefficient data to be frozen into kernels at compile time.

coeff_as_packed_mu(*coeff64*)

Pack 64 bit integer machine units coefficients into 32 bit integer RTIO data list.

This is a host-only method that can be used to generate packed spline coefficient data to be frozen into kernels at compile time.

coeff_to_mu(*coeff*, *coeff64*)

Convert a floating point list of coefficients into a 64 bit integer (preallocated).

Parameters

- **coeff** – TList(TFloat) list of coefficients in physical units.
- **coeff64** – TList(TInt64) preallocated list of coefficients in machine units.

from_mu(*value: numpy.int32*) → float

Convert 32 bit integer `value` from machine units to floating point physical units.

pack_coeff_mu(*coeff*, *packed*)

Pack coefficients into RTIO data

Parameters

- **coeff** – TList(TInt64) list of machine units spline coefficients. Lowest (zeroth) order first. The coefficient list is zero-extended by the RTIO gateware.

- **packed** – TList(TInt32) list for packed RTIO data. Must be pre-allocated. Length in bits is `n*width + (n - 1)*n//2*time_width`

set(*value: float*)

Set spline value.

Parameters **value** – Spline value relative to full-scale.

set_coeff(*coeff*)

Set spline coefficients.

Missing coefficients (high order) are zero-extended byt the RTIO gateware.

If more coefficients are supplied than the gateware supports the extra coefficients are ignored.

Parameters **value** – List of floating point spline coefficients, lowest order (constant) coefficient first. Units are the unit of this spline's value times increasing powers of 1/s.

set_coeff_mu(*value*)

Set spline raw values.

Parameters **value** – Spline packed raw values.

set_mu(*value: numpy.int32*)

Set spline value (machine units).

Parameters **value** – Spline value in integer machine units.

smooth(*start: float*, *stop: float*, *duration: float*, *order: numpy.int32*)

Initiate an interpolated value change.

For zeroth order (step) interpolation, the step is at `start + duration/2`.

First order interpolation corresponds to a linear value ramp from `start` to `stop` over `duration`.

The third order interpolation is constrained to have zero first order derivative at both *start* and *stop*.

For first order and third order interpolation (linear and cubic) the interpolator needs to be stopped explicitly at the stop time (e.g. by setting spline coefficient data or starting a new *smooth()* interpolation).

This method advances the timeline by `duration`.

Parameters

- **start** – Initial value of the change. In physical units.
- **stop** – Final value of the change. In physical units.
- **duration** – Duration of the interpolation. In physical units.
- **order** – Order of the interpolation. Only 0, 1, and 3 are valid: step, linear, cubic.

to_mu(*value: float*) → numpy.int32

Convert floating point `value` from physical units to 32 bit integer machine units.

to_mu64(*value: float*) → numpy.int64

Convert floating point `value` from physical units to 64 bit integer machine units.

14.3.8 artiq.coredevice.sawg module

Driver for the Smart Arbitrary Waveform Generator (SAWG) on RTIO.

The SAWG is an "improved DDS" built in gateware and interfacing to high-speed DACs.

Output event replacement is supported except on the configuration channel.

class artiq.coredevice.sawg.**Config**(*channel*, *core*, *cordic_gain=1.0*)

SAWG configuration.

Exposes the configurable quantities of a single SAWG channel.

Access to the configuration registers for a SAWG channel can not be concurrent. There must be at least `_rtio_interval` machine units of delay between accesses. Replacement is not supported and will be lead to an `RTIOCollision` as this is likely a programming error. All methods therefore advance the timeline by the duration of one configuration register transfer.

Parameters

- **channel** – RTIO channel number of the channel.
- **core** – Core device.

set_clr(*clr0: numpy.int32*, *clr1: numpy.int32*, *clr2: numpy.int32*)

Set the accumulator clear mode for the three phase accumulators.

When the `clr` bit for a given DDS/DUC phase accumulator is set, that phase accumulator will be cleared with every phase offset RTIO command and the output phase of the DDS/DUC will be exactly the phase RTIO value ("absolute phase update mode").

$$q'(t) = p' + (t - t')f'$$

In turn, when the bit is cleared, the phase RTIO channels determine a phase offset to the current (carrier-) value of the DDS/DUC phase accumulator. This "relative phase update mode" is sometimes also called "continuous phase mode".

$$q'(t) = q(t') + (p' - p) + (t - t')f'$$

Where:

- q, q': old/new phase accumulator
- p, p': old/new phase offset
- f': new frequency
- t': timestamp of setting new p, f
- t: running time

Parameters

- **clr0** – Auto-clear phase accumulator of the `phase0`/ `frequency0` DUC. Default: `True`
- **clr1** – Auto-clear phase accumulator of the `phase1`/ `frequency1` DDS. Default: `True`
- **clr2** – Auto-clear phase accumulator of the `phase2`/ `frequency2` DDS. Default: `True`

set_div(*div: numpy.int32*, *n: numpy.int32 = 0*)

Set the spline evolution divider and current counter value.

The divider and the spline evolution are synchronized across all spline channels within a SAWG channel. The DDS/DUC phase accumulators always evolves at full speed.

Note: The spline evolution divider has not been tested extensively and is currently considered a technological preview only.

Parameters

- **div** – Spline evolution divider, such that `t_sawg_spline/t_rtio_coarse = div + 1`. Default: `0`.
- **n** – Current value of the counter. Default: `0`.

set_duc_max(*limit: float*)

Set the digital up-converter (DUC) I and Q data summing junctions upper limit.

Each of the three summing junctions has a saturating adder with configurable upper and lower limits. The three summing junctions are:

- At the in-phase input to the `phase0/frequency0` fast DUC, after the anti-aliasing FIR filter.
- At the quadrature input to the `phase0/frequency0` fast DUC, after the anti-aliasing FIR filter. The in-phase and quadrature data paths both use the same limits.
- Before the DAC, where the following three data streams are added together:
  - the output of the `offset` spline,
  - (optionally, depending on `i_enable`) the in-phase output of the `phase0/frequency0` fast DUC, and
  - (optionally, depending on `q_enable`) the quadrature output of the `phase0/frequency0` fast DUC of the buddy channel.

Refer to the documentation of *SAWG* for a mathematical description of the summing junctions.

Parameters **limit** – Limit value `[-1, 1]`. The output of the limiter will never exceed this limit. The default limits are the full range `[-1, 1]`.

See also:

- *set_duc_max()*: Upper limit of the in-phase and quadrature inputs to the DUC.
- *set_duc_min()*: Lower limit of the in-phase and quadrature inputs to the DUC.
- *set_out_max()*: Upper limit of the DAC output.
- *set_out_min()*: Lower limit of the DAC output.

set_duc_max_mu(*limit: numpy.int32*)

Set the digital up-converter (DUC) I and Q data summing junctions upper limit. In machine units.

The default limits are chosen to reach maximum and minimum DAC output amplitude.

For a description of the limiter functions in normalized units see:

See also:

set_duc_max()

set_duc_min(*limit: float*)

See also:

set_duc_max()

set_duc_min_mu(*limit: numpy.int32*)

See also:

set_duc_max_mu()

set_iq_en(*i_enable: numpy.int32*, *q_enable: numpy.int32*)

Enable I/Q data on this DAC channel.

Every pair of SAWG channels forms a buddy pair. The `iq_en` configuration controls which DDS data is emitted to the DACs.

Refer to the documentation of *SAWG* for a mathematical description of `i_enable` and `q_enable`.

Note: Quadrature data from the buddy channel is currently a technological preview only. The data is ignored in the SAWG gateware and not added to the DAC output. This is equivalent to the `q_enable` switch always being `0`.

Parameters

- **i_enable** – Controls adding the in-phase DUC-DDS data of *this* SAWG channel to *this* DAC channel. Default: `1`.
- **q_enable** – controls adding the quadrature DUC-DDS data of this SAWG's *buddy* channel to *this* DAC channel. Default: `0`.

set_out_max(*limit: float*)

See also:

set_duc_max()

set_out_max_mu(*limit: numpy.int32*)

See also:

set_duc_max_mu()

set_out_min(*limit: float*)

See also:

set_duc_max()

set_out_min_mu(*limit: numpy.int32*)

See also:

set_duc_max_mu()

class artiq.coredevice.sawg.**SAWG**(*dmgr*, *channel_base*, *parallelism*, *core_device='core'*)

Smart arbitrary waveform generator channel. The channel is parametrized as:

```
oscillators = exp(2j*pi*(frequency0*t + phase0))*(
    amplitude1*exp(2j*pi*(frequency1*t + phase1)) +
    amplitude2*exp(2j*pi*(frequency2*t + phase2)))

output = (offset +
    i_enable*Re(oscillators) +
    q_enable*Im(buddy_oscillators))
```

This parametrization can be viewed as two complex (quadrature) oscillators (`frequency1`/`phase1` and `frequency2`/`phase2`) that are executing and sampling at the coarse RTIO frequency. They can represent frequencies within the first Nyquist zone from `-f_rtio_coarse/2` to `f_rtio_coarse/2`.

Note: The coarse RTIO frequency `f_rtio_coarse` is the inverse of `ref_period*multiplier`. Both are arguments of the `Core` device, specified in the device database `device_db.py`.

The sum of their outputs is then interpolated by a factor of `parallelism` (2, 4, 8 depending on the bitstream) using a finite-impulse-response (FIR) anti-aliasing filter (more accurately a half-band filter).

The filter is followed by a configurable saturating limiter.

After the limiter, the data is shifted in frequency using a complex digital up-converter (DUC, `frequency0`/`phase0`) running at `parallelism` times the coarse RTIO frequency. The first Nyquist zone of the DUC extends from `-f_rtio_coarse*parallelism/2` to `f_rtio_coarse*parallelism/2`. Other Nyquist zones are usable depending on the interpolation/modulation options configured in the DAC.

The real/in-phase data after digital up-conversion can be offset using another spline interpolator `offset`.

The `i_enable`/`q_enable` switches enable emission of quadrature signals for later analog quadrature mixing distinguishing upper and lower sidebands and thus doubling the bandwidth. They can also be used to emit four-tone signals.

Note: Quadrature data from the buddy channel is currently ignored in the SAWG gateware and not added to the DAC output. This is equivalent to the `q_enable` switch always being `0`.

The configuration channel and the nine *artiq.coredevice.spline.Spline* interpolators are accessible as attributes:

- `config`: *Config*
- `offset`, `amplitude1`, `amplitude2`: in units of full scale
- `phase0`, `phase1`, `phase2`: in units of turns
- `frequency0`, `frequency1`, `frequency2`: in units of Hz

Note: The latencies (pipeline depths) of the nine data channels (i.e. all except `config`) are matched. Equivalent channels (e.g. `phase1` and `phase2`) are exactly matched. Channels of different type or functionality (e.g. `offset` vs `amplitude1`, DDS vs DUC, `phase0` vs `phase1`) are only matched to within one coarse RTIO cycle.

Parameters

- **channel_base** – RTIO channel number of the first channel (amplitude). The configuration channel and frequency/phase/amplitude channels are then assumed to be successive channels.

- **parallelism** – Number of output samples per coarse RTIO clock cycle.
- **core_device** – Name of the core device that this SAWG is on.

reset()

Re-establish initial conditions.

This clears all spline interpolators, accumulators and configuration settings.

This method advances the timeline by the time required to perform all 7 writes to the configuration channel, plus 9 coarse RTIO cycles.

14.3.9 artiq.coredevice.basemod_att module

14.3.10 artiq.coredevice.phaser module

class artiq.coredevice.phaser.**Phaser**(*dmgr*, *channel_base*, *miso_delay=1*, *tune_fifo_offset=True*, *clk_sel=0*, *sync_dly=0*, *dac=None*, *trf0=None*, *trf1=None*, *core_device='core'*)

Phaser 4-channel, 16-bit, 1 GS/s DAC coredevice driver.

Phaser contains a 4 channel, 1 GS/s DAC chip with integrated upconversion, quadrature modulation compensation and interpolation features.

The coredevice produces 2 IQ (in-phase and quadrature) data streams with 25 MS/s and 14 bit per quadrature. Each data stream supports 5 independent numerically controlled IQ oscillators (NCOs, DDSs with 32 bit frequency, 16 bit phase, 15 bit amplitude, and phase accumulator clear functionality) added together. See *PhaserChannel* and *PhaserOscillator*.

Together with a data clock, framing marker, a checksum and metadata for register access the streams are sent in groups of 8 samples over 1.5 Gb/s FastLink via a single EEM connector from coredevice to Phaser.

On Phaser in the FPGA the data streams are buffered and interpolated from 25 MS/s to 500 MS/s 16 bit followed by a 500 MS/s digital upconverter with adjustable frequency and phase. The interpolation passband is 20 MHz wide, passband ripple is less than 1e-3 amplitude, stopband attenuation is better than 75 dB at offsets > 15 MHz and better than 90 dB at offsets > 30 MHz.

The four 16 bit 500 MS/s DAC data streams are sent via a 32 bit parallel LVDS bus operating at 1 Gb/s per pin pair and processed in the DAC (Texas Instruments DAC34H84). On the DAC 2x interpolation, sinx/x compensation, quadrature modulator compensation, fine and coarse mixing as well as group delay capabilities are available. If desired, these features my be configured via the *dac* dictionary.

The latency/group delay from the RTIO events setting *PhaserOscillator* or *PhaserChannel* DUC parameters all the way to the DAC outputs is deterministic. This enables deterministic absolute phase with respect to other RTIO input and output events (see *get_next_frame_mu()*).

The four analog DAC outputs are passed through anti-aliasing filters.

In the baseband variant, the even/in-phase DAC channels feed 31.5 dB range attenuators and are available on the front panel. The odd outputs are available at MMCX connectors on board.

In the upconverter variant, each IQ output pair feeds one quadrature upconverter (Texas Instruments TRF372017) with integrated PLL/VCO. This digitally configured analog quadrature upconverter supports offset tuning for carrier and sideband suppression. The output from the upconverter passes through the 31.5 dB range step attenuator and is available at the front panel.

The DAC, the analog quadrature upconverters and the attenuators are configured through a shared SPI bus that is accessed and controlled via FPGA registers.

Note: Various register settings of the DAC and the quadrature upconverters are available to be modified through the *dac*, *trf0*, *trf1* dictionaries. These can be set through the device database (*device_db.py*). The settings are frozen during instantiation of the class and applied during *init()*. See the `DAC34H84` and `TRF372017` source for details.

Note: To establish deterministic latency between RTIO time base and DAC output, the DAC FIFO read pointer value (*fifo_offset*) must be fixed. If *tune_fifo_offset=True* (the default) a value with maximum margin is determined automatically by *dac_tune_fifo_offset* each time *init()* is called. This value should be used for the *fifo_offset* key of the *dac* settings of Phaser in *device_db.py* and automatic tuning should be disabled by *tune_fifo_offset=False*.

Parameters

- **channel** – Base RTIO channel number
- **core_device** – Core device name (default: "core")
- **miso_delay** – Fastlink MISO signal delay to account for cable and buffer round trip. Tuning this might be automated later.
- **tune_fifo_offset** – Tune the DAC FIFO read pointer offset (default=True)
- **clk_sel** – Select the external SMA clock input (1 or 0)
- **sync_dly** – SYNC delay with respect to ISTR.
- **dac** – DAC34H84 DAC settings as a dictionary.
- **trf0** – Channel 0 TRF372017 quadrature upconverter settings as a dictionary.
- **trf1** – Channel 1 TRF372017 quadrature upconverter settings as a dictionary.

Attributes:

- **channel: List of two *PhaserChannel*** To access oscillators, digital upconverters, PLL/VCO analog quadrature upconverters and attenuators.

`clear_dac_alarms`()

Clear DAC alarm flags.

`dac_iotest`(*pattern*) → numpy.int32

Performs a DAC IO test according to the datasheet.

Parameters **pattern** – List of four int32 containing the pattern

Returns Bit error mask (16 bits)

`dac_read`(*addr*, *div=34*) → numpy.int32

Read from a DAC register.

Parameters

- **addr** – Register address to read from
- **div** – SPI clock divider. Needs to be at least 250 (1 µs SPI clock) to read the temperature register.

dac_sync()

Trigger DAC synchronisation for both output channels.

The DAC sif_sync is de-asserts, then asserted. The synchronisation is triggered on assertion.

By default, the fine-mixer (NCO) and QMC are synchronised. This includes applying the latest register settings.

The synchronisation sources may be configured through the *syncsel_x* fields in the *dac* configuration dictionary (see *__init__()*).

Note: Synchronising the NCO clears the phase-accumulator

dac_tune_fifo_offset()

Scan through *fifo_offset* and configure midpoint setting.

Returns Optimal *fifo_offset* setting with maximum margin to write pointer.

dac_write(*addr*, *data*)

Write 16 bit to a DAC register.

Parameters

- **addr** – Register address
- **data** – Register data to write

duc_stb()

Strobe the DUC configuration register update.

Transfer staging to active registers. This affects both DUC channels.

get_crc_err()

Get the frame CRC error counter.

Returns The number of frames with CRC mismatches sind the reset of the device. Overflows at 256.

get_dac_alarms()

Read the DAC alarm flags.

Returns DAC alarm flags (see datasheet for bit meaning)

get_dac_temperature() → numpy.int32

Read the DAC die temperature.

Returns DAC temperature in degree Celsius

get_next_frame_mu()

Return the timestamp of the frame strictly after *now_mu()*.

Register updates (DUC, DAC, TRF, etc.) scheduled at this timestamp and multiples of *self.t_frame* later will have deterministic latency to output.

get_sta()

Get the status register value.

Bit flags are:

- PHASER_STA_DAC_ALARM: DAC alarm pin
- PHASER_STA_TRF0_LD: Quadrature upconverter 0 lock detect

- PHASER_STA_TRF1_LD: Quadrature upconverter 1 lock detect
- PHASER_STA_TERM0: ADC channel 0 termination indicator
- PHASER_STA_TERM1: ADC channel 1 termination indicator
- PHASER_STA_SPI_IDLE: SPI machine is idle and data registers can be read/written

Returns Status register

init(*debug=False*)

Initialize the board.

Verifies board and chip presence, resets components, performs communication and configuration tests and establishes initial conditions.

measure_frame_timestamp()

Measure the timestamp of an arbitrary frame and store it in *self.frame_tstamp*.

To be used as reference for aligning updates to the FastLink frames. See *get_next_frame_mu()*.

read32(*addr*) → numpy.int32

Read 32 bit from a sequence of FPGA registers.

read8(*addr*) → numpy.int32

Read from FPGA register.

Parameters **addr** – Address to read from (7 bit)

Returns Data read (8 bit)

set_cfg(*clk_sel=0*, *dac_resetb=1*, *dac_sleep=0*, *dac_txena=1*, *trf0_ps=0*, *trf1_ps=0*, *att0_rstn=1*, *att1_rstn=1*)

Set the configuration register.

Each flag is a single bit (0 or 1).

Parameters

- **clk_sel** – Select the external SMA clock input
- **dac_resetb** – Active low DAC reset pin
- **dac_sleep** – DAC sleep pin
- **dac_txena** – Enable DAC transmission pin
- **trf0_ps** – Quadrature upconverter 0 power save
- **trf1_ps** – Quadrature upconverter 1 power save
- **att0_rstn** – Active low attenuator 0 reset
- **att1_rstn** – Active low attenuator 1 reset

set_dac_cmix(*fs_8_step*)

Set the DAC coarse mixer frequency for both channels

Use of the coarse mixer requires the DAC mixer to be enabled. The mixer can be configured via the *dac* configuration dictionary (see *__init__()*).

The selected coarse mixer frequency becomes active without explicit synchronisation.

Parameters **fs_8_step** – coarse mixer frequency shift in 125 MHz steps. This should be an integer between -3 and 4 (inclusive).

`set_fan`(*duty*)

Set the fan duty cycle.

Parameters `duty` – Duty cycle (0. to 1.)

`set_fan_mu`(*pwm*)

Set the fan duty cycle.

Parameters `pwm` – Duty cycle in machine units (8 bit)

`set_leds`(*leds*)

Set the front panel LEDs.

Parameters `leds` – LED settings (6 bit)

`set_sync_dly`(*dly*)

Set SYNC delay.

Parameters `dly` – DAC SYNC delay setting (0 to 7)

`spi_cfg`(*select*, *div*, *end*, *clk_phase=0*, *clk_polarity=0*, *half_duplex=0*, *lsb_first=0*, *offline=0*, *length=8*)

Set the SPI machine configuration

Parameters

- `select` – Chip selects to assert (DAC, TRF0, TRF1, ATT0, ATT1)
- `div` – SPI clock divider relative to 250 MHz fabric clock
- `end` – Whether to end the SPI transaction and deassert chip select
- `clk_phase` – SPI clock phase (sample on first or second edge)
- `clk_polarity` – SPI clock polarity (idle low or high)
- `half_duplex` – Read MISO data from MOSI wire
- `lsb_first` – Transfer the least significant bit first
- `offline` – Put the SPI interfaces offline and don't drive voltages
- `length` – SPI transfer length (1 to 8 bits)

`spi_read`()

Read from the SPI input data register.

`spi_write`(*data*)

Write 8 bits into the SPI data register and start/continue the transaction.

`write32`(*addr*, *data: numpy.int32*)

Write 32 bit to a sequence of FPGA registers.

`write8`(*addr*, *data*)

Write data to FPGA register.

Parameters

- `addr` – Address to write to (7 bit)
- `data` – Data to write (8 bit)

class artiq.coredevice.phaser.**PhaserChannel**(*phaser*, *index*, *trf*)

Phaser channel IQ pair.

A Phaser channel contains:

- multiple oscillators (in the coredevice phy),
- an interpolation chain and digital upconverter (DUC) on Phaser,
- several channel-specific settings in the DAC:
  - quadrature modulation compensation QMC
  - numerically controlled oscillator NCO or coarse mixer CMIX,
- the analog quadrature upconverter (in the Phaser-Upconverter hardware variant), and
- a digitally controlled step attenuator.

Attributes:

- `oscillator`: List of five *PhaserOscillator*.

Note: The amplitude sum of the oscillators must be less than one to avoid clipping or overflow. If any of the DDS or DUC frequencies are non-zero, it is not sufficient to ensure that the sum in each quadrature is within range.

Note: The interpolation filter on Phaser has an intrinsic sinc-like overshoot in its step response. That overshoot is a direct consequence of its near-brick-wall frequency response. For large and wide-band changes in oscillator parameters, the overshoot can lead to clipping or overflow after the interpolation. Either band-limit any changes in the oscillator parameters or back off the amplitude sufficiently.

---

cal_trf_vco()

Start calibration of the upconverter (hardware variant) VCO.

TRF outputs should be disabled during VCO calibration.

en_trf_out(*rf=1*, *lo=0*)

Enable the rf/lo outputs of the upconverter (hardware variant).

Parameters

- **rf** – 1 to enable RF output, 0 to disable
- **lo** – 1 to enable LO output, 0 to disable

get_att_mu() → numpy.int32

Read current attenuation.

The current attenuation value is read without side effects.

Returns Current attenuation in machine units

get_dac_data() → numpy.int32

Get a sample of the current DAC data.

The data is split accross multiple registers and thus the data is only valid if constant.

Returns DAC data as 32 bit IQ. I/DACA/DACC in the 16 LSB, Q/DACB/DACD in the 16 MSB

`set_att`(*att*)

Set channel attenuation in SI units.

Parameters `att` – Attenuation in dB

`set_att_mu`(*data*)

Set channel attenuation.

Parameters `data` – Attenuator data in machine units (8 bit)

`set_dac_test`(*data: numpy.int32*)

Set the DAC test data.

Parameters `data` – 32 bit IQ test data, I/DACA/DACC in the 16 LSB, Q/DACB/DACD in the 16 MSB

`set_duc_cfg`(*clr=0*, *clr_once=0*, *select=0*)

Set the digital upconverter (DUC) and interpolator configuration.

Parameters

- `clr` – Keep the phase accumulator cleared (persistent)
- `clr_once` – Clear the phase accumulator for one cycle
- `select` – Select the data to send to the DAC (0: DUC data, 1: test data, other values: reserved)

`set_duc_frequency`(*frequency*)

Set the DUC frequency in SI units.

Parameters `frequency` – DUC frequency in Hz (passband from -200 MHz to 200 MHz, wrapping around at +- 250 MHz)

`set_duc_frequency_mu`(*ftw*)

Set the DUC frequency.

Parameters `ftw` – DUC frequency tuning word (32 bit)

`set_duc_phase`(*phase*)

Set the DUC phase in SI units.

Parameters `phase` – DUC phase in turns

`set_duc_phase_mu`(*pow*)

Set the DUC phase offset.

Parameters `pow` – DUC phase offset word (16 bit)

`set_nco_frequency`(*frequency*)

Set the NCO frequency in SI units.

This method stages the new NCO frequency, but does not apply it.

Use of the DAC-NCO requires the DAC mixer and NCO to be enabled. These can be configured via the *dac* configuration dictionary (see *__init__()*).

Parameters `frequency` – NCO frequency in Hz (passband from -400 MHz to 400 MHz, wrapping around at +- 500 MHz)

`set_nco_frequency_mu`(*ftw*)

Set the NCO frequency.

This method stages the new NCO frequency, but does not apply it.

Use of the DAC-NCO requires the DAC mixer and NCO to be enabled. These can be configured via the *dac* configuration dictionary (see *__init__()*).

Parameters **ftw** – NCO frequency tuning word (32 bit)

`set_nco_phase`(*phase*)

Set the NCO phase in SI units.

By default, the new NCO phase applies on completion of the SPI transfer. This also causes a staged NCO frequency to be applied. Different triggers for applying NCO settings may be configured through the *sync_sel_mixerxx* fields in the *dac* configuration dictionary (see *__init__()*).

Use of the DAC-NCO requires the DAC mixer and NCO to be enabled. These can be configured via the *dac* configuration dictionary (see *__init__()*).

Parameters **phase** – NCO phase in turns

`set_nco_phase_mu`(*pow*)

Set the NCO phase offset.

By default, the new NCO phase applies on completion of the SPI transfer. This also causes a staged NCO frequency to be applied. Different triggers for applying NCO settings may be configured through the *sync_sel_mixerxx* fields in the *dac* configuration dictionary (see *__init__()*).

Use of the DAC-NCO requires the DAC mixer and NCO to be enabled. These can be configured via the *dac* configuration dictionary (see *__init__()*).

Parameters **pow** – NCO phase offset word (16 bit)

`trf_read`(*addr*, *cnt_mux_sel=0*) → numpy.int32

Quadrature upconverter register read.

Parameters

- **addr** – Register address to read (0 to 7)
- **cnt_mux_sel** – Report VCO counter min or max frequency

Returns Register data (32 bit)

`trf_write`(*data*, *readback=False*)

Write 32 bits to quadrature upconverter register.

Parameters

- **data** – Register data (32 bit) containing encoded address
- **readback** – Whether to return the read back MISO data

`class artiq.coredevice.phaser.PhaserOscillator`(*channel*, *index*)

Phaser IQ channel oscillator (NCO/DDS).

Note: Latencies between oscillators within a channel and between oscillator parameters (amplitude and phase/frequency) are deterministic (with respect to the 25 MS/s sample clock) but not matched.

`set_amplitude_phase`(*amplitude*, *phase=0.0*, *clr=0*)

Set Phaser MultiDDS amplitude and phase.

Parameters

- **amplitude** – Amplitude in units of full scale
- **phase** – Phase in turns
- **clr** – Clear the phase accumulator (persistent)

`set_amplitude_phase_mu`(*asf=32767*, *pow=0*, *clr=0*)

Set Phaser MultiDDS amplitude, phase offset and accumulator clear.

Parameters

- **asf** – Amplitude (15 bit)
- **pow** – Phase offset word (16 bit)
- **clr** – Clear the phase accumulator (persistent)

`set_frequency`(*frequency*)

Set Phaser MultiDDS frequency.

Parameters **frequency** – Frequency in Hz (passband from -10 MHz to 10 MHz, wrapping around at +- 12.5 MHz)

`set_frequency_mu`(*ftw*)

Set Phaser MultiDDS frequency tuning word.

Parameters **ftw** – Frequency tuning word (32 bit)

14.4 DAC/ADC drivers

14.4.1 artiq.coredevice.ad53xx module

"RTIO driver for the Analog Devices AD53[67][0123] family of multi-channel Digital to Analog Converters.

Output event replacement is not supported and issuing commands at the same time is an error.

`class artiq.coredevice.ad53xx.AD53xx`(*dmgr*, *spi_device*, *ldac_device=None*, *clr_device=None*, *chip_select=1*, *div_write=4*, *div_read=16*, *vref=5.0*, *offset_dacs=8192*, *core='core'*)

Analog devices AD53[67][0123] family of multi-channel Digital to Analog Converters.

Parameters

- **spi_device** – SPI bus device name
- **ldac_device** – LDAC RTIO TTLOut channel name (optional)
- **clr_device** – CLR RTIO TTLOut channel name (optional)
- **chip_select** – Value to drive on SPI chip select lines during transactions (default: 1)
- **div_write** – SPI clock divider for write operations (default: 4, 50MHz max SPI clock with {t_high, t_low} >=8ns)
- **div_read** – SPI clock divider for read operations (default: 16, not optimized for speed; datasheet says t22: 25ns min SCLK edge to SDO valid, and suggests the SPI speed for reads should be <=20 MHz)

- **vref** – DAC reference voltage (default: 5.)
- **offset_dacs** – Initial register value for the two offset DACs, device dependent and must be set correctly for correct voltage to mu conversions. Knowledge of his state is not transferred between experiments. (default: 8192)
- **core_device** – Core device name (default: "core")

calibrate(*channel*, *vzs*, *vfs*)

Two-point calibration of a DAC channel.

Programs the offset and gain register to trim out DAC errors. Does not take effect until LDAC is pulsed (see *load()*).

Calibration consists of measuring the DAC output voltage for a channel with the DAC set to zero-scale (0x0000) and full-scale (0xffff).

Note that only negative offsets and full-scale errors (DAC gain too high) can be calibrated in this fashion.

Parameters **channel** – The number of the calibrated channel

Params vzs Measured voltage with the DAC set to zero-scale (0x0000)

Params vfs Measured voltage with the DAC set to full-scale (0xffff)

init(*blind=False*)

Configures the SPI bus, drives LDAC and CLR high, programmes the offset DACs, and enables overtemperature shutdown.

This method must be called before any other method at start-up or if the SPI bus has been accessed by another device.

Parameters **blind** – If `True`, do not attempt to read back control register or check for overtemperature.

load()

Pulse the LDAC line.

Note that there is a <= 1.5us "BUSY" period (t10) after writing to a DAC input/gain/offset register. All DAC registers may be programmed normally during the busy period, however LDACs during the busy period cause the DAC output to change *after* the BUSY period has completed, instead of the usual immediate update on LDAC behaviour.

This method advances the timeline by two RTIO clock periods.

read_reg(*channel=0*, *op=1024*)

Read a DAC register.

This method advances the timeline by the duration of two SPI transfers plus two RTIO coarse cycles plus 270 ns and consumes all slack.

Parameters

- **channel** – Channel number to read from (default: 0)
- **op** – Operation to perform, one of `AD53XX_READ_X1A`, `AD53XX_READ_X1B`, `AD53XX_READ_OFFSET`, `AD53XX_READ_GAIN` etc. (default: `AD53XX_READ_X1A`).

Returns The 16 bit register value

set_dac(*voltages*, *channels=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]*)

Program multiple DAC channels and pulse LDAC to update the DAC outputs.

This method does not advance the timeline; write events are scheduled in the past. The DACs will synchronously start changing their output levels *now*.

If no LDAC device was defined, the LDAC pulse is skipped.

Parameters

- **voltages** – list of voltages to program the DAC channels to
- **channels** – list of DAC channels to program. If not specified, we program the DAC channels sequentially, starting at 0.

set_dac_mu(*values*, *channels=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]*)

Program multiple DAC channels and pulse LDAC to update the DAC outputs.

This method does not advance the timeline; write events are scheduled in the past. The DACs will synchronously start changing their output levels *now*.

If no LDAC device was defined, the LDAC pulse is skipped.

See *load()*.

Parameters

- **values** – list of DAC values to program
- **channels** – list of DAC channels to program. If not specified, we program the DAC channels sequentially, starting at 0.

voltage_to_mu(*voltage*)

Returns the 16-bit DAC register value required to produce a given output voltage, assuming offset and gain errors have been trimmed out.

The 16-bit register value may also be used with 14-bit DACs. The additional bits are disregarded by 14-bit DACs.

Parameters **voltage** – Voltage in SI units. Valid voltages are: [-2*vref, + 2*vref - 1 LSB] + voltage offset.

Returns The 16-bit DAC register value

write_dac(*channel*, *voltage*)

Program the DAC output voltage for a channel.

The DAC output is not updated until LDAC is pulsed (see *load()*). This method advances the timeline by the duration of one SPI transfer.

write_dac_mu(*channel*, *value*)

Program the DAC input register for a channel.

The DAC output is not updated until LDAC is pulsed (see *load()*). This method advances the timeline by the duration of one SPI transfer.

write_gain_mu(*channel*, *gain=65535*)

Program the gain register for a DAC channel.

The DAC output is not updated until LDAC is pulsed (see *load()*). This method advances the timeline by the duration of one SPI transfer.

Parameters **gain** – 16-bit gain register value (default: 0xffff)

write_offset(*channel*, *voltage*)

Program the DAC offset voltage for a channel.

An offset of +V can be used to trim out a DAC offset error of -V. The DAC output is not updated until LDAC is pulsed (see *load()*). This method advances the timeline by the duration of one SPI transfer.

Parameters **voltage** – the offset voltage

write_offset_dacs_mu(*value*)

Program the OFS0 and OFS1 offset DAC registers.

Writes to the offset DACs take effect immediately without requiring a LDAC. This method advances the timeline by the duration of two SPI transfers.

Parameters **value** – Value to set both offset DAC registers to

write_offset_mu(*channel*, *offset=32768*)

Program the offset register for a DAC channel.

The DAC output is not updated until LDAC is pulsed (see *load()*). This method advances the timeline by the duration of one SPI transfer.

Parameters **offset** – 16-bit offset register value (default: 0x8000)

artiq.coredevice.ad53xx.**ad53xx_cmd_read_ch**(*channel*, *op*)

Returns the word that must be written to the DAC to read a given DAC channel register.

Parameters

- **channel** – DAC channel to read (8 bits)
- **op** – The channel register to read, one of AD53XX_READ_X1A, AD53XX_READ_X1B, AD53XX_READ_OFFSET, AD53XX_READ_GAIN etc.

Returns The 24-bit word to be written to the DAC to initiate read

artiq.coredevice.ad53xx.**ad53xx_cmd_write_ch**(*channel*, *value*, *op*)

Returns the word that must be written to the DAC to set a DAC channel register to a given value.

Parameters

- **channel** – DAC channel to write to (8 bits)
- **value** – 16-bit value to write to the register
- **op** – The channel register to write to, one of AD53XX_CMD_DATA, AD53XX_CMD_OFFSET or AD53XX_CMD_GAIN.

Returns The 24-bit word to be written to the DAC

artiq.coredevice.ad53xx.**voltage_to_mu**(*voltage*, *offset_dacs=8192*, *vref=5.0*)

Returns the 16-bit DAC register value required to produce a given output voltage, assuming offset and gain errors have been trimmed out.

The 16-bit register value may also be used with 14-bit DACs. The additional bits are disregarded by 14-bit DACs.

Also used to return offset register value required to produce a given voltage when the DAC register is set to mid-scale. An offset of V can be used to trim out a DAC offset error of -V.

Parameters

- **voltage** – Voltage in SI units. Valid voltages are: [-2*vref, + 2*vref - 1 LSB] + voltage offset.
- **offset_dacs** – Register value for the two offset DACs (default: 0x2000)

- **vref** – DAC reference voltage (default: 5.)

Returns The 16-bit DAC register value

14.4.2 artiq.coredevice.zotino module

RTIO driver for the Zotino 32-channel, 16-bit 1MSPS DAC.

Output event replacement is not supported and issuing commands at the same time is an error.

class artiq.coredevice.zotino.**Zotino**(*dmgr*, *spi_device*, *ldac_device=None*, *clr_device=None*, *div_write=4*, *div_read=16*, *vref=5.0*, *core='core'*)

Zotino 32-channel, 16-bit 1MSPS DAC.

Controls the AD5372 DAC and the 8 user LEDs via a shared SPI interface.

Parameters

- **spi_device** – SPI bus device name
- **ldac_device** – LDAC RTIO TTLOut channel name.
- **clr_device** – CLR RTIO TTLOut channel name.
- **div_write** – SPI clock divider for write operations (default: 4, 50MHz max SPI clock)
- **div_read** – SPI clock divider for read operations (default: 16, not optimized for speed; datasheet says t22: 25ns min SCLK edge to SDO valid, and suggests the SPI speed for reads should be <=20 MHz)
- **vref** – DAC reference voltage (default: 5.)
- **core_device** – Core device name (default: "core")

set_leds(*leds*)

Sets the states of the 8 user LEDs.

Parameters **leds** – 8-bit word with LED state

14.4.3 artiq.coredevice.sampler module

class artiq.coredevice.sampler.**Sampler**(*dmgr*, *spi_adc_device*, *spi_pgia_device*, *cnv_device*, *div=8*, *gains=0*, *core_device='core'*)

Sampler ADC.

Controls the LTC2320-16 8 channel 16 bit ADC with SPI interface and the switchable gain instrumentation amplifiers.

Parameters

- **spi_adc_device** – ADC SPI bus device name
- **spi_pgia_device** – PGIA SPI bus device name
- **cnv_device** – CNV RTIO TTLOut channel name
- **div** – SPI clock divider (default: 8)
- **gains** – Initial value for PGIA gains shift register (default: 0x0000). Knowledge of this state is not transferred between experiments.
- **core_device** – Core device name

get_gains_mu()

Read the PGIA gain settings of all channels.

Returns The PGIA gain settings in machine units.

init()

Initialize the device.

Sets up SPI channels.

sample(*data*)

Acquire a set of samples.

See also:

sample_mu()

Parameters **data** – List of floating point data samples to fill.

sample_mu(*data*)

Acquire a set of samples.

Perform a conversion and transfer the samples.

This assumes that the input FIFO of the ADC SPI RTIO channel is deep enough to buffer the samples (half the length of *data* deep). If it is not, there will be RTIO input overflows.

Parameters **data** – List of data samples to fill. Must have even length. Samples are always read from the last channel (channel 7) down. The *data* list will always be filled with the last item holding to the sample from channel 7.

set_gain_mu(*channel*, *gain*)

Set instrumentation amplifier gain of a channel.

The four gain settings (0, 1, 2, 3) corresponds to gains of (1, 10, 100, 1000) respectively.

Parameters

- **channel** – Channel index
- **gain** – Gain setting

artiq.coredevice.sampler.**adc_mu_to_volt**(*data*, *gain=0*)

Convert ADC data in machine units to Volts.

Parameters

- **data** – 16 bit signed ADC word
- **gain** – PGIA gain setting (0: 1, ..., 3: 1000)

Returns Voltage in Volts

14.4.4 artiq.coredevice.novogorny module

class artiq.coredevice.novogorny.**Novogorny**(*dmgr*, *spi_device*, *cnv_device*, *div=8*, *gains=0*, *core_device='core'*)

Novogorny ADC.

Controls the LTC2335-16 8 channel ADC with SPI interface and the switchable gain instrumentation amplifiers using a shift register.

Parameters

- **spi_device** – SPI bus device name
- **cnv_device** – CNV RTIO TTLOut channel name
- **div** – SPI clock divider (default: 8)
- **gains** – Initial value for PGIA gains shift register (default: 0x0000). Knowledge of this state is not transferred between experiments.
- **core_device** – Core device name

burst_mu(*data*, *dt_mu*, *ctrl=0*)

Acquire a burst of samples.

If the burst is too long and the sample rate too high, there will be RTIO input overflows.

High sample rates lead to gain errors since the impedance between the instrumentation amplifier and the ADC is high.

Parameters

- **data** – List of data values to write result packets into. In machine units.
- **dt** – Sample interval in machine units.
- **ctrl** – ADC control word to write during each result packet transfer.

configure(*data*)

Set up the ADC sequencer.

Parameters **data** – List of 8 bit control words to write into the sequencer table.

sample(*next_ctrl=0*)

Acquire a sample

See also:

sample_mu()

Parameters **next_ctrl** – ADC control word for the next sample

Returns The ADC result packet (Volt)

sample_mu(*next_ctrl=0*)

Acquire a sample:

Perform a conversion and transfer the sample.

Parameters **next_ctrl** – ADC control word for the next sample

Returns The ADC result packet (machine units)

set_gain_mu(*channel*, *gain*)

Set instrumentation amplifier gain of a channel.

The four gain settings (0, 1, 2, 3) corresponds to gains of (1, 10, 100, 1000) respectively.

Parameters

- **channel** – Channel index
- **gain** – Gain setting

artiq.coredevice.novogorny.**adc_channel**(*data*)

Return the channel index from a result packet

artiq.coredevice.novogorny.**adc_ctrl**(*channel=1*, *softspan=7*, *valid=1*)

Build a LTC2335-16 control word

artiq.coredevice.novogorny.**adc_data**(*data*)

Return the ADC value from a result packet

artiq.coredevice.novogorny.**adc_softspan**(*data*)

Return the softspan configuration index from a result packet

artiq.coredevice.novogorny.**adc_value**(*data*, *v_ref=5.0*)

Convert a ADC result packet to SI units (Volt)

14.4.5 artiq.coredevice.fastino module

RTIO driver for the Fastino 32channel, 16 bit, 2.5 MS/s per channel, streaming DAC.

class artiq.coredevice.fastino.**Fastino**(*dmgr*, *channel*, *core_device='core'*, *log2_width=0*)

Fastino 32-channel, 16-bit, 2.5 MS/s per channel streaming DAC

The RTIO PHY supports staging DAC data before transmitting them by writing to the DAC RTIO addresses, if a channel is not "held" by setting its bit using *set_hold()*, the next frame will contain the update. For the DACs held, the update is triggered explicitly by setting the corresponding bit using `set_update()`. Update is self-clearing. This enables atomic DAC updates synchronized to a frame edge.

The *log2_width=0* RTIO layout uses one DAC channel per RTIO address and a dense RTIO address space. The RTIO words are narrow. (32 bit) and few-channel updates are efficient. There is the least amount of DAC state tracking in kernels, at the cost of more DMA and RTIO data. The setting here and in the RTIO PHY (gateware) must match.

Other *log2_width* (up to *log2_width=5*) settings pack multiple (in powers of two) DAC channels into one group and into one RTIO write. The RTIO data width increases accordingly. The *log2_width* LSBs of the RTIO address for a DAC channel write must be zero and the address space is sparse. For *log2_width=5* the RTIO data is 512 bit wide.

If *log2_width* is zero, the *set_dac()*/*set_dac_mu()* interface must be used. If non-zero, the *set_group()*/*set_group_mu()* interface must be used.

Parameters

- **channel** – RTIO channel number
- **core_device** – Core device name (default: "core")
- **log2_width** – Width of DAC channel group (logarithm base 2). Value must match the corresponding value in the RTIO PHY (gateware).

apply_cic(*channel_mask*)

Apply the staged interpolator configuration on the specified channels.

Each Fastino channel starting with gateware v0.2 includes a fourth order (cubic) CIC interpolator with variable rate change and variable output gain compensation (see *stage_cic()*).

Fastino gateware before v0.2 does not include the interpolators and the methods affecting the CICs should not be used.

Channels using non-unity interpolation rate should have continous DAC updates enabled (see *set_continuous()*) unless their output is supposed to be constant.

This method resets and settles the affected interpolators. There will be no output updates for the next *order* = *3* input samples. Affected channels will only accept one input sample per input sample period. This method synchronizes the input sample period to the current frame on the affected channels.

If application of new interpolator settings results in a change of the overall gain, there will be a corresponding output step.

init()

Initialize the device.

- disables RESET, DAC_CLR, enables AFE_PWR
- clears error counters, enables error counting
- turns LEDs off
- clears *hold* and *continuous* on all channels
- clear and resets interpolators to unit rate change on all channels

It does not change set channel voltages and does not reset the PLLs or clock domains.

Note: On Fastino gateware before v0.2 this may lead to 0 voltage being emitted transiently.

read(*addr*)

Read from Fastino register.

TODO: untested

Parameters **addr** – Address to read from.

Returns The data read.

set_cfg(*reset=0*, *afe_power_down=0*, *dac_clr=0*, *clr_err=0*)

Set configuration bits.

Parameters

- **reset** – Reset SPI PLL and SPI clock domain.
- **afe_power_down** – Disable AFE power.
- **dac_clr** – Assert all 32 DAC_CLR signals setting all DACs to mid-scale (0 V).
- **clr_err** – Clear error counters and PLL reset indicator. This clears the sticky red error LED. Must be cleared to enable error counting.

set_continuous(*channel_mask*)

Enable continuous DAC updates on channels regardless of new data being submitted.

`set_dac`(*dac*, *voltage*)

Set DAC data to given voltage.

Parameters

- **dac** – DAC channel (0-31).
- **voltage** – Desired output voltage.

`set_dac_mu`(*dac*, *data*)

Write DAC data in machine units.

Parameters

- **dac** – DAC channel to write to (0-31).
- **data** – DAC word to write, 16 bit unsigned integer, in machine units.

`set_group`(*dac*, *voltage*)

Set DAC group data to given voltage.

Parameters

- **dac** – DAC channel (0-31).
- **voltage** – Desired output voltage.

`set_group_mu`(*dac: numpy.int32*, *data: list(elt=numpy.int32)*)

Write a group of DAC channels in machine units.

Parameters

- **dac** – First channel in DAC channel group (0-31). The *log2_width* LSBs must be zero.
- **data** – List of DAC data pairs (2x16 bit unsigned) to write, in machine units. Data exceeding group size is ignored. If the list length is less than group size, the remaining DAC channels within the group are cleared to 0 (machine units).

`set_hold`(*hold*)

Set channels to manual update.

Parameters **hold** – Bit mask of channels to hold (32 bit).

`set_leds`(*leds*)

Set the green user-defined LEDs

Parameters **leds** – LED status, 8 bit integer each bit corresponding to one green LED.

`stage_cic`(*rate*) → numpy.int32

Compute and stage interpolator configuration.

This method approximates the desired interpolation rate using a 10 bit floating point representation (6 bit mantissa, 4 bit exponent) and then determines an optimal interpolation gain compensation exponent to avoid clipping. Gains for rates that are powers of two are accurately compensated. Other rates lead to overall less than unity gain (but more than 0.5 gain).

The overall gain including gain compensation is *actual_rate**order/2**ceil(log2(actual_rate**order))* where *order* = *3*.

Returns the actual interpolation rate.

`stage_cic_mu`(*rate_mantissa*, *rate_exponent*, *gain_exponent*)

Stage machine unit CIC interpolator configuration.

update(*update*)

Schedule channels for update.

Parameters **update** – Bit mask of channels to update (32 bit).

voltage_group_to_mu(*voltage*, *data*)

Convert SI Volts to packed DAC channel group machine units.

Parameters

- **voltage** – List of SI Volt voltages.
- **data** – List of DAC channel data pairs to write to. Half the length of *voltage*.

voltage_to_mu(*voltage*)

Convert SI Volts to DAC machine units.

Parameters **voltage** – Voltage in SI Volts.

Returns DAC data word in machine units, 16 bit integer.

write(*addr*, *data*)

Write data to a Fastino register.

Parameters

- **addr** – Address to write to.
- **data** – Data to write.

14.5 Miscellaneous

14.5.1 artiq.coredevice.suservo module

class artiq.coredevice.suservo.**Channel**(*dmgr*, *channel*, *servo_device*)

Sampler-Urukul Servo channel

Parameters

- **channel** – RTIO channel number
- **servo_device** – Name of the parent SUServo device

dds_offset_to_mu(*offset*)

Convert IIR offset (negative setpoint) from units of full scale to machine units (see *set_dds_mu()*, *set_dds_offset_mu()*).

For positive ADC voltages as setpoints, this should be negative. Due to rounding and representation as two's complement, `offset=1` can not be represented while `offset=-1` can.

get_profile_mu(*profile*, *data*)

Retrieve profile data.

Profile data is returned in the `data` argument in machine units packed as: `[ftw >> 16, b1, pow, adc | (delay << 8), offset, a1, ftw & 0xffff, b0]`.

See also:

The individual fields are described in *set_iir_mu()* and *set_dds_mu()*.

This method advances the timeline by 32 µs and consumes all slack.

Parameters

- **profile** – Profile number (0-31)
- **data** – List of 8 integers to write the profile data into

get_y(*profile*)

Get a profile's IIR state (filter output, Y0).

The IIR state is also know as the "integrator", or the DDS amplitude scale factor. It is 17 bits wide and unsigned.

This method does not advance the timeline but consumes all slack.

If reading servo state through this method collides with the servo writing that same data, the data can become invalid. To ensure consistent and valid data, stop the servo before using this method.

Parameters **profile** – Profile number (0-31)

Returns IIR filter output in Y0 units of full scale

get_y_mu(*profile*)

Get a profile's IIR state (filter output, Y0) in machine units.

The IIR state is also know as the "integrator", or the DDS amplitude scale factor. It is 17 bits wide and unsigned.

This method does not advance the timeline but consumes all slack.

If reading servo state through this method collides with the servo writing that same data, the data can become invalid. To ensure consistent and valid data, stop the servo before using this method.

Parameters **profile** – Profile number (0-31)

Returns 17 bit unsigned Y0

set(*en_out*, *en_iir=0*, *profile=0*)

Operate channel.

This method does not advance the timeline. Output RF switch setting takes effect immediately and is independent of any other activity (profile settings, other channels). The RF switch behaves like *artiq.coredevice.ttl.TTLOut*. RTIO event replacement is supported. IIR updates take place once the RF switch has been enabled for the configured delay and the profile setting has been stable. Profile changes take between one and two servo cycles to reach the DDS.

Parameters

- **en_out** – RF switch enable
- **en_iir** – IIR updates enable
- **profile** – Active profile (0-31)

set_dds(*profile*, *frequency*, *offset*, *phase=0.0*)

Set profile DDS coefficients.

This method advances the timeline by four servo memory accesses. Profile parameter changes are not synchronized. Activate a different profile or stop the servo to ensure synchronous changes.

Parameters

- **profile** – Profile number (0-31)
- **frequency** – DDS frequency in Hz
- **offset** – IIR offset (negative setpoint) in units of full scale, see *dds_offset_to_mu()*

- **phase** – DDS phase in turns

set_dds_mu(*profile*, *ftw*, *offs*, *pow_=0*)

Set profile DDS coefficients in machine units.

See also:

`set_amplitude()`

Parameters

- **profile** – Profile number (0-31)
- **ftw** – Frequency tuning word (32 bit unsigned)
- **offs** – IIR offset (17 bit signed)
- **pow** – Phase offset word (16 bit)

set_dds_offset(*profile*, *offset*)

Set only IIR offset in DDS coefficient profile.

See *set_dds()* for setting the complete DDS profile.

Parameters

- **profile** – Profile number (0-31)
- **offset** – IIR offset (negative setpoint) in units of full scale

set_dds_offset_mu(*profile*, *offs*)

Set only IIR offset in DDS coefficient profile.

See *set_dds_mu()* for setting the complete DDS profile.

Parameters

- **profile** – Profile number (0-31)
- **offs** – IIR offset (17 bit signed)

set_iir(*profile*, *adc*, *kp*, *ki=0.0*, *g=0.0*, *delay=0.0*)

Set profile IIR coefficients.

This method advances the timeline by four servo memory accesses. Profile parameter changes are not synchronized. Activate a different profile or stop the servo to ensure synchronous changes.

Gains are given in units of output full per scale per input full scale.

The transfer function is (up to time discretization and coefficient quantization errors):

$$H(s) = k_p + \frac{k_i}{s + \frac{k_i}{g}}$$

Where:

- $s = \sigma + i\omega$ is the complex frequency
- k_p is the proportional gain
- k_i is the integrator gain
- g is the integrator gain limit

Parameters

- **profile** – Profile number (0-31)
- **adc** – ADC channel to take IIR input from (0-7)
- **kp** – Proportional gain (1). This is usually negative (closed loop, positive ADC voltage, positive setpoint). When 0, this implements a pure I controller.
- **ki** – Integrator gain (rad/s). When 0 (the default) this implements a pure P controller. Same sign as `kp`.
- **g** – Integrator gain limit (1). When 0 (the default) the integrator gain limit is infinite. Same sign as `ki`.
- **delay** – Delay (in seconds, 0-300 µs) before allowing IIR updates after invoking *set()*. This is rounded to the nearest number of servo cycles (~1.2 µs). Since the RF switch (*set()*) can be opened at any time relative to the servo cycle, the first DDS update that carries updated IIR data will occur approximately between `delay + 1 cycle` and `delay + 2 cycles` after *set()*.

set_iir_mu(*profile*, *adc*, *a1*, *b0*, *b1*, *dly=0*)

Set profile IIR coefficients in machine units.

The recurrence relation is (all data signed and MSB aligned):

$$a_0 y_n = a_1 y_{n-1} + b_0 (x_n + o)/2 + b_1 (x_{n-1} + o)/2$$

Where:

- y_n and y_{n-1} are the current and previous filter outputs, clipped to $[0, 1[$.
- x_n and x_{n-1} are the current and previous filter inputs in $[-1, 1[$.
- o is the offset
- a_0 is the normalization factor 2^{11}
- a_1 is the feedback gain
- b_0 and b_1 are the feedforward gains for the two delays

See also:

set_iir()

Parameters

- **profile** – Profile number (0-31)
- **adc** – ADC channel to take IIR input from (0-7)
- **a1** – 18 bit signed A1 coefficient (Y1 coefficient, feedback, integrator gain)
- **b0** – 18 bit signed B0 coefficient (recent, X0 coefficient, feed forward, proportional gain)
- **b1** – 18 bit signed B1 coefficient (old, X1 coefficient, feed forward, proportional gain)
- **dly** – IIR update suppression time. In units of IIR cycles (~1.2 µs, 0-255).

set_y(*profile*, *y*)

Set a profile's IIR state (filter output, Y0).

The IIR state is also know as the "integrator", or the DDS amplitude scale factor. It is 17 bits wide and unsigned.

This method must not be used when the servo could be writing to the same location. Either deactivate the profile, or deactivate IIR updates, or disable servo iterations.

This method advances the timeline by one servo memory access.

Parameters

- **profile** – Profile number (0-31)
- **y** – IIR state in units of full scale

set_y_mu(*profile*, *y*)

Set a profile's IIR state (filter output, Y0) in machine units.

The IIR state is also know as the "integrator", or the DDS amplitude scale factor. It is 17 bits wide and unsigned.

This method must not be used when the servo could be writing to the same location. Either deactivate the profile, or deactivate IIR updates, or disable servo iterations.

This method advances the timeline by one servo memory access.

Parameters

- **profile** – Profile number (0-31)
- **y** – 17 bit unsigned Y0

class artiq.coredevice.suservo.**SUServo**(*dmgr*, *channel*, *pgia_device*, *cpld_devices*, *dds_devices*, *gains=0*, *core_device='core'*)

Sampler-Urukul Servo parent and configuration device.

Sampler-Urukul Servo is a integrated device controlling one 8-channel ADC (Sampler) and two 4-channel DDS (Urukuls) with a DSP engine connecting the ADC data and the DDS output amplitudes to enable feedback. SU Servo can for example be used to implement intensity stabilization of laser beams with an amplifier and AOM driven by Urukul and a photodetector connected to Sampler.

Additionally SU Servo supports multiple preconfigured profiles per channel and features like automatic integrator hold.

Notes

- See the SU Servo variant of the Kasli target for an example of how to connect the gateware and the devices. Sampler and each Urukul need two EEM connections.
- Ensure that both Urukuls are AD9910 variants and have the on-board dip switches set to 1100 (first two on, last two off).
- Refer to the Sampler and Urukul documentation and the SU Servo example device database for runtime configuration of the devices (PLLs, gains, clock routing etc.)

Parameters

- **channel** – RTIO channel number
- **pgia_device** – Name of the Sampler PGIA gain setting SPI bus
- **cpld_devices** – Names of the Urukul CPLD SPI buses
- **dds_devices** – Names of the AD9910 devices
- **gains** – Initial value for PGIA gains shift register (default: 0x0000). Knowledge of this state is not transferred between experiments.

- **core_device** – Core device name

get_adc(*channel*)

Get the latest ADC reading (IIR filter input X0).

This method does not advance the timeline but consumes all slack.

If reading servo state through this method collides with the servo writing that same data, the data can become invalid. To ensure consistent and valid data, stop the servo before using this method.

The PGIA gain setting must be known prior to using this method, either by setting the gain (*set_pgia_mu()*) or by supplying it (`gains` or via the constructor/device database).

Parameters **adc** – ADC channel number (0-7)

Returns ADC voltage

get_adc_mu(*adc*)

Get the latest ADC reading (IIR filter input X0) in machine units.

This method does not advance the timeline but consumes all slack.

If reading servo state through this method collides with the servo writing that same data, the data can become invalid. To ensure consistent and valid data, stop the servo before using this method.

Parameters **adc** – ADC channel number (0-7)

Returns 17 bit signed X0

get_status()

Get current SU Servo status.

This method does not advance the timeline but consumes all slack.

The `done` bit indicates that a SU Servo cycle has completed. It is pulsed for one RTIO cycle every SU Servo cycle and asserted continuously when the servo is not `enabled` and the pipeline has drained (the last DDS update is done).

This method returns and clears the clip indicator for all channels. An asserted clip indicator corresponds to the servo having encountered an input signal on an active channel that would have resulted in the IIR state exceeding the output range.

Returns Status. Bit 0: enabled, bit 1: done, bits 8-15: channel clip indicators.

init()

Initialize the servo, Sampler and both Urukuls.

Leaves the servo disabled (see *set_config()*), resets and configures all DDS.

Urukul initialization is performed blindly as there is no readback from the DDS or the CPLDs.

This method does not alter the profile configuration memory or the channel controls.

read(*addr*)

Read from servo memory.

This method does not advance the timeline but consumes all slack.

Parameters **addr** – Memory location address.

set_config(*enable*)

Set SU Servo configuration.

This method advances the timeline by one servo memory access. It does not support RTIO event replacement.

Parameters **(int)** (*enable*) – Enable servo operation. Enabling starts servo iterations beginning with the ADC sampling stage. The first DDS update will happen about two servo cycles (~2.3 µs) after enabling the servo. The delay is deterministic. This also provides a mean for synchronization of servo updates to other RTIO activity. Disabling takes up to two servo cycles (~2.3 µs) to clear the processing pipeline.

set_pgia_mu(*channel*, *gain*)

Set instrumentation amplifier gain of a ADC channel.

The four gain settings (0, 1, 2, 3) corresponds to gains of (1, 10, 100, 1000) respectively.

Parameters

- **channel** – Channel index
- **gain** – Gain setting

write(*addr*, *value*)

Write to servo memory.

This method advances the timeline by one coarse RTIO cycle.

Parameters

- **addr** – Memory location address.
- **value** – Data to be written.

artiq.coredevice.suservo.**adc_mu_to_volts**(*x*, *gain*)

Convert servo ADC data from machine units to Volt.

artiq.coredevice.suservo.**y_mu_to_full_scale**(*y*)

Convert servo Y data from machine units to units of full scale.

14.5.2 artiq.coredevice.grabber module

class artiq.coredevice.grabber.**Grabber**(*dmgr*, *channel_base*, *res_width=12*, *count_shift=0*, *core_device='core'*)

Driver for the Grabber camera interface.

gate_roi(*mask*)

Defines which ROI engines produce input events.

At the end of each video frame, the output from each ROI engine that has been enabled by the mask is enqueued into the RTIO input FIFO.

This function sets the mask at the current position of the RTIO time cursor.

Setting the mask using this function is atomic; in other words, if the system is in the middle of processing a frame and the mask is changed, the processing will complete using the value of the mask that it started with.

Parameters **mask** – bitmask enabling or disabling each ROI engine.

gate_roi_pulse(*mask*, *dt*)

Sets a temporary mask for the specified duration (in seconds), before disabling all ROI engines.

input_mu(*data*)

Retrieves the accumulated values for one frame from the ROI engines. Blocks until values are available.

The input list must be a list of integers of the same length as there are enabled ROI engines. This method replaces the elements of the input list with the outputs of the enabled ROI engines, sorted by number.

If the number of elements in the list does not match the number of ROI engines that produced output, an exception will be raised during this call or the next.

setup_roi(*n*, *x0*, *y0*, *x1*, *y1*)

Defines the coordinates of a ROI.

The coordinates are set around the current position of the RTIO time cursor.

The user must keep the ROI engine disabled for the duration of more than one video frame after calling this function, as the output generated for that video frame is undefined.

Advances the timeline by 4 coarse RTIO cycles.

exception artiq.coredevice.grabber.**OutOfSyncException**

Raised when an incorrect number of ROI engine outputs has been retrieved from the RTIO input FIFO.

CHAPTER

FIFTEEN

LIST OF AVAILABLE NDSPS

The following network device support packages are available for ARTIQ. If you would like to add yours to this list, just send us an email or a pull request.

Equipment	Nix package	Conda package	Documentation	URL
PDQ2	Not available	Not available	HTML	https://github.com/m-labs/pdq
Lab Brick Digital Attenuator	`lda`	`lda`	HTML	https://github.com/m-labs/lda
Novatech 409B	`novatech409b`	`novatech409b`	HTML	https://github.com/m-labs/novatech409b
Thorlabs T-Cubes	`thorlabs_tcube`	`thorlabs_tcube`	HTML	https://github.com/m-labs/thorlabs_tcube
Korad KA3005P	`korad_ka3005p`	`korad_ka3005p`	HTML	https://github.com/m-labs/korad_ka3005p
Newfocus 8742	`newfocus8742`	`newfocus8742`	HTML	https://github.com/quartiq/newfocus8742
Princeton Instruments PICam	Not available	Not available	Not available	https://github.com/quartiq/picam
Anel HUT2 power distribution	`hut2`	`hut2`	HTML	https://github.com/quartiq/hut2
TOPTICA lasers	`toptica-lasersdk-artiq`	`toptica-lasersdk-artiq`	Not available	https://github.com/quartiq/lasersdk-artiq
HighFinesse wavemeters	`highfinesse-net`	`highfinesse-net`	HTML	https://github.com/quartiq/highfinesse-net
InfluxDB database	Not available	Not available	HTML	https://gitlab.com/charlesbaynham/artiq_influx_generic

CHAPTER

SIXTEEN

DEVELOPING A NETWORK DEVICE SUPPORT PACKAGE (NDSP)

Most ARTIQ devices are interfaced through "controllers" that expose RPC interfaces to the network (based on SiPyCo). The master never does direct I/O to the devices, but issues RPCs to the controllers when needed. As opposed to running everything on the master, this architecture has those main advantages:

- Each driver can be run on a different machine, which alleviates cabling issues and OS compatibility problems.
- Reduces the impact of driver crashes.
- Reduces the impact of driver memory leaks.

This mechanism is for "slow" devices that are directly controlled by a PC, typically over a non-realtime channel such as USB.

Certain devices (such as the PDQ2) may still perform real-time operations by having certain controls physically connected to the core device (for example, the trigger and frame selection signals on the PDQ2). For handling such cases, parts of the NDSPs may be kernels executed on the core device.

A network device support package (NDSP) is composed of several parts:

1. The *driver*, which contains the Python API functions to be called over the network, and performs the I/O to the device. The top-level module of the driver is called `artiq.devices.XXX.driver`.
2. The *controller*, which instantiates, initializes and terminates the driver, and sets up the RPC server. The controller is a front-end command-line tool to the user and is called `artiq.frontend.aqctl_XXX`. A `setup.py` entry must also be created to install it.
3. An optional *client*, which connects to the controller and exposes the functions of the driver as a command-line interface. Clients are front-end tools (called `artiq.frontend.aqcli_XXX`) that have `setup.py` entries. In most cases, a custom client is not needed and the generic `sipyco_rpctool` utility can be used instead. Custom clients are only required when large amounts of data must be transferred over the network API, that would be unwieldy to pass as `sipyco_rpctool` command-line parameters.
4. An optional *mediator*, which is code executed on the client that supplements the network API. A mediator may contain kernels that control real-time signals such as TTL lines connected to the device. Simple devices use the network API directly and do not have a mediator. Mediator modules are called `artiq.devices.XXX.mediator` and their public classes are exported at the `artiq.devices.XXX` level (via `__init__.py`) for direct import and use by the experiments.

16.1 The driver and controller

A controller is a piece of software that receives commands from a client over the network (or the `localhost` interface), drives a device, and returns information about the device to the client. The mechanism used is remote procedure calls (RPCs) using `sipyco.pc_rpc`, which makes the network layers transparent for the driver's user.

The controller we will develop is for a "device" that is very easy to work with: the console from which the controller is run. The operation that the driver will implement is writing a message to that console.

For using RPC, the functions that a driver provides must be the methods of a single object. We will thus define a class that provides our message-printing method:

```
class Hello:
    def message(self, msg):
        print("message: " + msg)
```

For a more complex driver, you would put this class definition into a separate Python module called `driver`.

To turn it into a server, we use `sipyco.pc_rpc`. Import the function we will use:

```
from sipyco.pc_rpc import simple_server_loop
```

and add a `main` function that is run when the program is executed:

```
def main():
    simple_server_loop({"hello": Hello()}, "::1", 3249)

if __name__ == "__main__":
    main()
```

tip Defining the `main` function instead of putting its code directly in the `if __name__ == "__main__"` body enables the controller to be used as well as a setuptools entry point.

The parameters `::1` and `3249` are respectively the address to bind the server to (IPv6 localhost) and the TCP port to use. Then add a line:

```
#!/usr/bin/env python3
```

at the beginning of the file, save it to `aqctl_hello.py` and set its execution permissions:

```
$ chmod 755 aqctl_hello.py
```

Run it as:

```
$ ./aqctl_hello.py
```

and verify that you can connect to the TCP port:

```
$ telnet ::1 3249
Trying ::1...
Connected to ::1.
Escape character is '^]'.
```

tip Use the key combination Ctrl-AltGr-9 to get the `telnet>` prompt, and enter `close` to quit Telnet. Quit the controller with Ctrl-C.

Also verify that a target (service) named "hello" (as passed in the first argument to `simple_server_loop`) exists using the `sipyco_rpctool` program from the ARTIQ front-end tools:

```
$ sipyco_rpctool ::1 3249 list-targets
Target(s):   hello
```

16.2 The client

Clients are small command-line utilities that expose certain functionalities of the drivers. The `sipyco_rpctool` utility contains a generic client that can be used in most cases, and developing a custom client is not required. Try these commands

```
$ sipyco_rpctool ::1 3249 list-methods
$ sipyco_rpctool ::1 3249 call message test
```

In case you are developing a NDSP that is complex enough to need a custom client, we will see how to develop one. Create a `aqcli_hello.py` file with the following contents:

```
#!/usr/bin/env python3

from sipyco.pc_rpc import Client


def main():
    remote = Client("::1", 3249, "hello")
    try:
        remote.message("Hello World!")
    finally:
        remote.close_rpc()

if __name__ == "__main__":
    main()
```

Run it as before, while the controller is running. You should see the message appearing on the controller's terminal:

```
$ ./aqctl_hello.py
message: Hello World!
```

When using the driver in an experiment, the `Client` instance can be returned by the environment mechanism (via the `get_device` and `attr_device` methods of *artiq.language.environment.HasEnvironment*) and used normally as a device.

> **warning** RPC servers operate on copies of objects provided by the client, and modifications to mutable types are not written back. For example, if the client passes a list as a parameter of an RPC method, and that method `append()`s an element to the list, the element is not appended to the client's list.

16.3 Command-line arguments

Use the Python `argparse` module to make the bind address(es) and port configurable on the controller, and the server address, port and message configurable on the client.

We suggest naming the controller parameters `--bind` (which adds a bind address in addition to a default binding to localhost), `--no-bind-localhost` (which disables the default binding to localhost), and `--port`, so that those parameters stay consistent across controllers. Use `-s/--server` and `--port` on the client. The `sipyco.common_args.simple_network_args` library function adds such arguments for the controller, and the `sipyco.common_args.bind_address_from_args` function processes them.

The controller's code would contain something similar to this:

```
from sipyco.common_args import simple_network_args

def get_argparser():
    parser = argparse.ArgumentParser(description="Hello world controller")
    simple_network_args(parser, 3249)  # 3249 is the default TCP port
    return parser

def main():
    args = get_argparser().parse_args()
    simple_server_loop(Hello(), bind_address_from_args(args), args.port)
```

We suggest that you define a function `get_argparser` that returns the argument parser, so that it can be used to document the command line parameters using sphinx-argparse.

16.4 Logging

For the debug, information and warning messages, use the `logging` Python module and print the log on the standard error output (the default setting). The logging level is by default "WARNING", meaning that only warning messages and more critical messages will get printed (and no debug nor information messages). By calling `sipyco.common_args.verbosity_args` with the parser as argument, you add support for the `--verbose` (`-v`) and `--quiet` (`-q`) arguments in the parser. Each occurrence of `-v` (resp. `-q`) in the arguments will increase (resp. decrease) the log level of the logging module. For instance, if only one `-v` is present in the arguments, then more messages (info, warning and above) will get printed. If only one `-q` is present in the arguments, then only errors and critical messages will get printed. If `-qq` is present in the arguments, then only critical messages will get printed, but no debug/info/warning/error.

The program below exemplifies how to use logging:

```
import argparse
import logging

from sipyco.common_args import verbosity_args, init_logger_from_args


# get a logger that prints the module name
logger = logging.getLogger(__name__)


def get_argparser():
    parser = argparse.ArgumentParser(description="Logging example")
    parser.add_argument("--someargument",
```

(continues on next page)

(continued from previous page)

```
                        help="some argument")
    # [...]
    add_verbosity_args(parser) # This adds the -q and -v handling
    return parser


def main():
    args = get_argparser().parse_args()
    init_logger_from_args(args) # This initializes logging system log level according to␣
↪-v/-q args

    logger.debug("this is a debug message")
    logger.info("this is an info message")
    logger.warning("this is a warning message")
    logger.error("this is an error message")
    logger.critical("this is a critical message")

if __name__ == "__main__":
    main()
```

16.5 Integration with ARTIQ experiments

To integrate the controller into an experiment, simply add an entry into the `device_db.py` following the example shown below for the `aqctl_hello.py` from above:

```
device_db.update({
"hello": {
        "type": "controller",
        "host": "::1",
        "port": 3249,
        "command": "python /abs/path/to/aqctl_hello.py -p {port}"
},
})
```

From the experiment this can now be added using `self.setattr_device("hello")` in the `build()` phase of the experiment, and methods accessed via:

```
self.hello.message("Hello world!")
```

To automatically initiate controllers the `artiq_ctlmgr` utility from `artiq-comtools` can be used. By default, this initiates all controllers hosted on the local address of master connection.

16.6 Remote execution support

If you wish to support remote execution in your controller, you may do so by simply replacing `simple_server_loop` with `sipyco.remote_exec.simple_rexec_server_loop`.

16.7 General guidelines

- Do not use `__del__` to implement the cleanup code of your driver. Instead, define a `close` method, and call it using a `try...finally...` block in the controller.
- Format your source code according to PEP8. We suggest using `flake8` to check for compliance.
- Use new-style formatting (`str.format`) except for logging where it is not well supported, and double quotes for strings.
- The device identification (e.g. serial number, or entry in `/dev`) to attach to must be passed as a command-line parameter to the controller. We suggest using `-d` and `--device` as parameter name.
- Controllers must be able to operate in "simulation" mode, where they behave properly even if the associated hardware is not connected. For example, they can print the data to the console instead of sending it to the device, or dump it into a file.
- The simulation mode is entered whenever the `--simulation` option is specified.
- Keep command line parameters consistent across clients/controllers. When adding new command line options, look for a client/controller that does a similar thing and follow its use of `argparse`. If the original client/controller could use `argparse` in a better way, improve it.
- Use docstrings for all public methods of the driver (note that those will be retrieved by `sipyco_rpctool`).
- Choose a free default TCP port and add it to the default port list in this manual.

16.8 Hosting your code

We suggest that you create a Git repository for your code, and publish it on https://git.m-labs.hk/, GitLab, GitHub, or a similar website of your choosing. Then send us a message or pull request for your NDSP to be added to the list in this manual.

CHAPTER

SEVENTEEN

UTILITIES

17.1 Local running tool

Local experiment running tool

```
usage: artiq_run
       [-h]
       [--version]
       [--device-db DEVICE_DB]
       [--dataset-db DATASET_DB]
       [-c CLASS_NAME]
       [-o HDF5]
       FILE
       [ARGUMENTS ...]
```

17.1.1 Positional Arguments

FILE	file containing the experiment to run
ARGUMENTS	run arguments

17.1.2 Named Arguments

--version	print the ARTIQ version number
--device-db	device database file (default: “‘device_db.py”’)
	Default: “device_db.py”
--dataset-db	dataset file (default: “‘dataset_db.pyon”’)
	Default: “dataset_db.pyon”
-c, --class-name	name of the class to run
-o, --hdf5	write results to specified HDF5 file (default: print them)

17.2 Static compiler

This tool compiles an experiment into a ELF file. It is primarily used to prepare binaries for the default experiment loaded in non-volatile storage of the core device. Experiments compiled with this tool are not allowed to use RPCs, and their `run` entry point must be a kernel.

ARTIQ static compiler

```
usage: artiq_compile
       [-h]
       [--version]
       [--device-db DEVICE_DB]
       [--dataset-db DATASET_DB]
       [-c CLASS_NAME]
       [-o OUTPUT]
       FILE
       [ARGUMENTS ...]
```

17.2.1 Positional Arguments

FILE	file containing the experiment to compile
ARGUMENTS	run arguments

17.2.2 Named Arguments

--version	print the ARTIQ version number
--device-db	device database file (default: '"device_db.py"')
	Default: "device_db.py"
--dataset-db	dataset file (default: '"dataset_db.pyon"')
	Default: "dataset_db.pyon"
-c, --class-name	name of the class to compile
-o, --output	output file

17.3 Flash storage image generator

This tool compiles key/value pairs into a binary image suitable for flashing into the flash storage space of the core device.

ARTIQ flash storage image generator

```
usage: artiq_mkfs
       [-h]
       [-s KEY STRING]
```

(continues on next page)

(continued from previous page)

```
       [-f KEY FILENAME]
       output
```

17.3.1 Positional Arguments

output output file

17.3.2 Named Arguments

-s add string

Default: []

-f add file contents

Default: []

17.4 Flashing/Loading tool

ARTIQ flashing/deployment tool

```
usage: artiq_flash
       [-h]
       [--version]
       [-n]
       [-H HOSTNAME]
       [-J JUMP]
       [-t TARGET]
       [-I PREINIT_COMMAND]
       [-f STORAGE]
       [-d DIR]
       [--srcbuild]
       [--no-rtm-jtag]
       [ACTION ...]
```

17.4.1 Positional Arguments

ACTION actions to perform, default: flash everything

Default: []

17.4.2 Named Arguments

--version	print the ARTIQ version number
-n, --dry-run	only show the openocd script that would be run Default: False
-H, --host	SSH host where the board is located
-J, --jump	SSH host to jump through
-t, --target	target board, default: "kasli", one of: kasli sayma metlino kc705 Default: "kasli"
-I, --preinit-command	add a pre-initialization OpenOCD command. Useful for selecting a board when several are connected. Default: []
-f, --storage	write file to storage area
-d, --dir	look for board binaries in this directory
--srcbuild	board binaries directory is laid out as a source build tree Default: False
--no-rtm-jtag	do not attempt JTAG to the RTM Default: False

Valid actions:

- gateware: write main gateware bitstream to flash
- rtm_gateware: write RTM gateware bitstream to flash
- bootloader: write bootloader to flash
- storage: write storage image to flash
- firmware: write firmware to flash
- load: load main gateware bitstream into device (volatile but fast)
- rtm_load: load RTM gateware bitstream into device
- erase: erase flash memory
- start: trigger the target to (re)load its gateware bitstream from flash

Prerequisites:

- Connect the board through its/a JTAG adapter.
- Have OpenOCD installed and in your $PATH.
- Have access to the JTAG adapter's devices. Udev rules from OpenOCD: 'sudo cp openocd/contrib/99-openocd.rules /etc/udev/rules.d' and replug the device. Ensure you are member of the plugdev group: 'sudo adduser $USER plugdev' and re-login.

17.5 Core device management tool

The artiq_coremgmt utility gives remote access to the core device logs, the *Flash storage*, and other management functions.

To use this tool, you need to specify a `device_db.py` device database file which contains a `comm` device (an example is provided in `examples/master/device_db.py`). This tells the tool how to connect to the core device and with which parameters (e.g. IP address, TCP port). When not specified, the artiq_coremgmt utility will assume that there is a file named `device_db.py` in the current directory.

To read core device logs:

```
$ artiq_coremgmt log
```

To set core device log level and UART log level (possible levels are `TRACE`, `DEBUG`, `INFO`, `WARN` and `ERROR`):

```
$ artiq_coremgmt log set_level LEVEL
$ artiq_coremgmt log set_uart_level LEVEL
```

Note that enabling the `TRACE` log level results in small core device slowdown, and printing large amounts of log messages to the UART results in significant core device slowdown.

To read the record whose key is `mac`:

```
$ artiq_coremgmt config read mac
```

To write the value `test_value` in the key `my_key`:

```
$ artiq_coremgmt config write -s my_key test_value
$ artiq_coremgmt config read my_key
b'test_value'
```

You can also write entire files in a record using the `-f` parameter. This is useful for instance to write the startup and idle kernels in the flash storage:

```
$ artiq_coremgmt config write -f idle_kernel idle.elf
$ artiq_coremgmt config read idle_kernel | head -c9
b'\x7fELF
```

You can write several records at once:

```
$ artiq_coremgmt config write -s key1 value1 -f key2 filename -s key3 value3
```

To remove the previously written key `my_key`:

```
$ artiq_coremgmt config remove my_key
```

You can remove several keys at once:

```
$ artiq_coremgmt config remove key1 key2
```

To erase the entire flash storage area:

```
$ artiq_coremgmt config erase
```

You do not need to remove a record in order to change its value, just overwrite it:

```
$ artiq_coremgmt config write -s my_key some_value
$ artiq_coremgmt config write -s my_key some_other_value
$ artiq_coremgmt config read my_key
b'some_other_value'
```

ARTIQ core device management tool

```
usage: artiq_coremgmt
       [-h]
       [--version]
       [--device-db DEVICE_DB]
       [-D DEVICE]
       {log,config,reboot,debug}
       ...
```

17.5.1 Positional Arguments

tool Possible choices: log, config, reboot, debug

17.5.2 Named Arguments

--version print the ARTIQ version number

--device-db device database file (default: ‘“device_db.py”’)

Default: “device_db.py”

-D, --device use specified core device address instead of reading device database

17.5.3 Sub-commands:

log

read logs and change log levels

```
artiq_coremgmt log
[-h]
{clear,set_level,set_uart_level}
...
```

Positional Arguments

action Possible choices: clear, set_level, set_uart_level

Sub-commands:

clear

clear log buffer

```
artiq_coremgmt log clear
[-h]
```

set_level

set minimum level for messages to be logged

```
artiq_coremgmt log set_level
[-h]
LEVEL
```

Positional Arguments

LEVEL log level (one of: OFF ERROR WARN INFO DEBUG TRACE)

set_uart_level

set minimum level for messages to be logged to UART

```
artiq_coremgmt log set_uart_level
[-h]
LEVEL
```

Positional Arguments

LEVEL log level (one of: OFF ERROR WARN INFO DEBUG TRACE)

config

read and change core device configuration

```
artiq_coremgmt config
[-h]
{read,write,remove,erase}
...
```

Positional Arguments

action Possible choices: read, write, remove, erase

Sub-commands:

read

read key from core device config

```
artiq_coremgmt config read
[-h]
KEY
```

Positional Arguments

KEY key to be read from core device config

write

write key-value records to core device config

```
artiq_coremgmt config write
[-h]
[-s KEY STRING]
[-f KEY FILENAME]
```

Named Arguments

-s, --string key-value records to be written to core device config

Default: []

-f, --file key and file whose content to be written to core device config

Default: []

remove

remove key from core device config

```
artiq_coremgmt config remove
[-h]
...
```

Positional Arguments

KEY key to be removed from core device config

Default: []

erase

fully erase core device config

```
artiq_coremgmt config erase
[-h]
```

reboot

reboot the running system

```
artiq_coremgmt reboot
[-h]
```

debug

specialized debug functions

```
artiq_coremgmt debug
[-h]
{allocator}
...
```

Positional Arguments

action Possible choices: allocator

Sub-commands:

allocator

show heap layout

```
artiq_coremgmt debug allocator
[-h]
```

17.6 Core device logging controller

ARTIQ controller for core device logs

```
usage: aqctl_corelog
       [-h]
       [--simulation]
       CORE_ADDR
```

17.6.1 Positional Arguments

CORE_ADDR hostname or IP address of the core device

17.6.2 Named Arguments

--simulation Simulation - does not connect to device

Default: False

17.7 Moninj proxy

ARTIQ moninj proxy

```
usage: aqctl_moninj_proxy
       [-h]
       CORE_ADDR
```

17.7.1 Positional Arguments

CORE_ADDR hostname or IP address of the core device

17.8 Core device RTIO analyzer tool

`artiq_coreanalyzer` is a tool to convert core device RTIO logs to VCD waveform files that are readable by third-party tools such as GtkWave. This tool extracts pre-recorded data from an ARTIQ core device buffer (or from a file with the `-r` option), and converts it to a standard VCD file format. See *RTIO analyzer* for an example, or `artiq.test.coredevice.test_analyzer` for a relevant unit test.

ARTIQ core device RTIO analysis tool

```
usage: artiq_coreanalyzer
       [-h]
       [--device-db DEVICE_DB]
       [-r READ_DUMP]
       [-p]
       [-w WRITE_VCD]
       [-d WRITE_DUMP]
       [-u]
```

17.8.1 Named Arguments

--device-db device database file (default: ‘“device_db.py”’)

Default: “device_db.py”

-r, --read-dump read raw dump file instead of accessing device

-p, --print-decoded print raw decoded messages

Default: False

-w, --write-vcd format and write contents to VCD file

-d, --write-dump write raw dump file

-u, --vcd-uniform-interval emit uniform time intervals between timed VCD events and show RTIO event interval (in SI seconds) and timestamp (in machine units) as separate VCD channels

Default: False

Note: The RTIO analyzer does not support SAWG.

17.9 DRTIO routing table manipulation tool

ARTIQ DRTIO routing table manipulation tool

```
usage: artiq_route
       [-h]
       FILE
       {init,show,set}
       ...
```

17.9.1 Positional Arguments

FILE	target file
action	Possible choices: init, show, set

17.9.2 Sub-commands:

init

create a new empty routing table

```
artiq_route init
[-h]
```

show

show contents of routing table

```
artiq_route show
[-h]
```

set

set routing table entry

```
artiq_route set
[-h]
DESTINATION
[HOP ...]
```

Positional Arguments

DESTINATION	destination to operate on
HOP	hop(s) to the destination

CHAPTER

EIGHTEEN

DEFAULT NETWORK PORTS

Component	Default port
Core device (management)	1380
Core device (main)	1381
Core device (analyzer)	1382
Moninj (core device or proxy)	1383
Moninj (proxy control)	1384
Master (logging input)	1066
Master (broadcasts)	1067
Core device logging controller	1068
InfluxDB bridge	3248
Controller manager	3249
Master (notifications)	3250
Master (control)	3251
PDQ2 (out-of-tree)	3252
LDA (out-of-tree)	3253
Novatech 409B (out-of-tree)	3254
Thorlabs T-Cube (out-of-tree)	3255
Korad KA3005P (out-of-tree)	3256
Newfocus 8742 (out-of-tree)	3257
PICam (out-of-tree)	3258
PTB Drivers (out-of-tree)	3259-3270
HUT2 (out-of-tree)	3271
TOPTICA Laser SDK (out-of-tree)	3272
HighFinesse (out-of-tree)	3273
InfluxDB schedule bridge	3275
InfluxDB driver (out-of-tree)	3276

CHAPTER

NINETEEN

FAQ

19.1 How do I . . .

19.1.1 find ARTIQ examples?

The examples are installed in the `examples` folder of the ARTIQ package. You can find where the ARTIQ package is installed on your machine with:

```
python3 -c "import artiq; print(artiq.__path__[0])"
```

Copy the `examples` folder from that path into your home/user directory, and start experimenting!

19.1.2 prevent my first RTIO command from causing an underflow?

The first RTIO event is programmed with a small timestamp above the value of the timecounter when the core device is reset. If the kernel needs more time than this timestamp to produce the event, an underflow will occur. You can prevent it by calling `break_realtime` just before programming the first event, or by adding a sufficient delay.

If you are not resetting the core device, the time cursor stays where the previous experiment left it.

19.1.3 organize datasets in folders?

Use the dot (".") in dataset names to separate folders. The GUI will automatically create and delete folders in the dataset tree display.

19.1.4 write a generator feeding a kernel feeding an analyze function?

Like this:

```
def run(self):
    self.parse(self.pipe(iter(range(10))))

def pipe(self, gen):
    for i in gen:
        r = self.do(i)
        yield r

def parse(self, gen):
```

(continues on next page)

(continued from previous page)

```
    for i in gen:
        pass

@kernel
def do(self, i):
    return i
```

19.1.5 create and use variable lengths arrays in kernels?

Don't. Preallocate everything. Or chunk it and e.g. read 100 events per function call, push them upstream and retry until the gate time closes.

19.1.6 execute multiple slow controller RPCs in parallel without losing time?

Use `threading.Thread`: portable, fast, simple for one-shot calls.

19.1.7 write part of my experiment as a coroutine/asyncio task/generator?

You can not change the API that your experiment exposes: `build()`, `prepare()`, `run()` and `analyze()` need to be regular functions, not generators or asyncio coroutines. That would make reusing your own code in sub-experiments difficult and fragile. You can however wrap your own generators/coroutines/tasks in regular functions that you then expose as part of the API.

19.1.8 determine the pyserial URL to attach to a device by its serial number?

You can list your system's serial devices and print their vendor/product id and serial number by running:

```
$ python3 -m serial.tools.list_ports -v
```

It will give you the `/dev/ttyUSBxx` (or the `COMxx` for Windows) device names. The `hwid:` field gives you the string you can pass via the `hwgrep://` feature of pyserial serial_for_url() in order to open a serial device.

The preferred way to specify a serial device is to make use of the `hwgrep://` URL: it allows to select the serial device by its USB vendor ID, product ID and/or serial number. Those never change, unlike the device file name.

For instance, if you want to specify the Vendor/Product ID and the USB Serial Number, you can do:

`-d "hwgrep://<VID>:<PID> SNR=<serial_number>"`. for example:

`-d "hwgrep://0403:faf0 SNR=83852734"`

19.1.9 run unit tests?

The unit tests assume that the Python environment has been set up in such a way that `import artiq` will import the code being tested, and that this is still true for any subprocess created. This is not the way setuptools operates as it adds the path to ARTIQ to `sys.path` which is not passed to subprocesses; as a result, running the tests via `setup.py` is not supported. The user must first install the package or set `PYTHONPATH`, and then run the tests with e.g. `python3 -m unittest discover` in the `artiq/test` folder and `lit .` in the `artiq/test/lit` folder.

For the hardware-in-the-loop unit tests, set the `ARTIQ_ROOT` environment variable to the path to a device database containing the relevant devices.

The core device tests require the following TTL devices and connections:

- `ttl_out`: any output-only TTL.
- `ttl_out_serdes`: any output-only TTL that uses a SERDES (i.e. has a fine timestamp). Can be aliased to `ttl_out`.
- `loop_out`: any output-only TTL. Must be physically connected to `loop_in`. Can be aliased to `ttl_out`.
- `loop_in`: any input-capable TTL. Must be physically connected to `loop_out`.
- `loop_clock_out`: a clock generator TTL. Must be physically connected to `loop_clock_in`.
- `loop_clock_in`: any input-capable TTL. Must be physically connected to `loop_clock_out`.

If TTL devices are missing, the corresponding tests are skipped.

19.1.10 find the dashboard and browser configuration files are stored?

```
python -c "from artiq.tools import get_user_config_dir; print(get_user_config_dir())"
```

PYTHON MODULE INDEX

a

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z