
ARTIQ Documentation

Release 8.9043+e573d36

M-Labs and contributors

Mar 25, 2026

OVERVIEW

1	Introduction	1
2	ARTIQ Releases	3
2.1	Release notes	3
3	Installing ARTIQ	17
3.1	Installing via Nix (Linux)	17
3.2	Installing via MSYS2 (Windows)	21
3.3	Installing via Conda [DEPRECATED]	21
3.4	Upgrading ARTIQ	22
4	(Re)flashing your core device	25
4.1	Obtaining board binaries	25
4.2	Installing and configuring OpenOCD	25
4.3	Writing the flash	27
4.4	Connecting to the UART log	27
5	Networking and configuration	29
5.1	Setting up core device networking	29
5.2	Configuring the core device	30
6	Building and developing ARTIQ	33
6.1	Installing Vivado	33
6.2	System description file	34
6.3	Nix development environment	35
6.4	Building ARTIQ	37
7	ARTIQ Real-Time I/O concepts	41
7.1	Timeline and terminology	41
7.2	Output errors and exceptions	43
7.3	Input channels and events	45
7.4	Event spreading	46
7.5	Seamless handover	46
7.6	Synchronization	47
8	Getting started with the core device	49
8.1	Host/core device interaction (RPC)	50
8.2	Real-time Input/Output (RTIO)	51
8.3	Parallel and sequential blocks	52
8.4	RTIO analyzer	53
8.5	Direct Memory Access (DMA)	54

9	Using the management system	57
9.1	Running your first experiment with the master	57
9.2	Running the dashboard and controller manager	59
9.3	Adding a new experiment	59
9.4	Adding arguments	60
9.5	Interactive arguments	61
9.6	Setting up Git integration	61
9.7	The ARTIQ session	63
10	Data and user interfaces	65
10.1	Datasets and results	65
10.2	Non-RTIO devices and the controller manager	67
10.3	Using MonInj	69
10.4	Waveform	70
10.5	Shortcuts	71
11	Using DRTIO and subkernels	73
11.1	Using DRTIO	73
11.2	Distributed Direct Memory Access (DDMA)	75
11.3	Subkernels	75
12	Environment	79
12.1	The device database	79
12.2	Arguments	81
12.3	Datasets	81
13	Compiler	83
13.1	ARTIQ Python code	83
13.2	Pitfalls	87
13.3	Flags and optimizations	88
14	Management system	91
14.1	Components	91
14.2	Git integration	93
14.3	Experiment scheduling	94
14.4	Internal details	96
15	DRTIO system	99
15.1	Terminology	99
15.2	The routing table	99
15.3	Internal details	100
16	Core device	103
16.1	Configuration storage	103
16.2	Common configuration commands	104
16.3	Clocking	105
16.4	Board details	105
16.5	Variant details	106
17	Developing a Network Device Support Package (NDSP)	109
17.1	Components of a NDSP	109
17.2	The driver and controller	110
17.3	The client	111
17.4	Integration with ARTIQ experiments	112
17.5	Remote execution support	112

17.6	Command-line arguments	112
17.7	Logging	113
17.8	Additional guidelines	114
17.9	Hosting your code	114
18	List of available NDSPs	115
19	Default network ports	117
20	Main front-end tools	119
20.1	artiq.frontend.artiq_run	119
20.2	artiq.frontend.artiq_master	119
20.3	artiq.frontend.artiq_client	121
20.4	artiq.frontend.artiq_dashboard	124
20.5	artiq.frontend.artiq_browser	125
20.6	artiq.frontend.artiq_session	125
20.7	artiq.comtools.artiq_ctlmgr	126
21	Core language and environment	127
21.1	artiq.language.core module	127
21.2	artiq.language.environment module	129
21.3	artiq.language.scan module	133
21.4	artiq.language.units module	134
22	Core real-time drivers	135
22.1	System drivers	135
22.2	Digital I/O drivers	139
22.3	RF generation drivers	151
22.4	DAC/ADC drivers	188
22.5	Miscellaneous	205
23	Management system interface	213
23.1	In experiments	213
23.2	In applets	214
24	Utilities	217
24.1	ARTIQ Firmware Service (AFWS) client	217
24.2	Static compiler	218
24.3	Flash storage image generator	219
24.4	Flashing/Loading tool	219
24.5	Core device management tool	220
24.6	Device database template generator	223
24.7	RTIO channel name map tool	224
24.8	Core device RTIO analyzer tool	224
24.9	DRTIO routing table manipulation tool	225
24.10	ARTIQ RTIO monitor	225
24.11	MonInj proxy	226
24.12	Core device RTIO analyzer proxy	226
24.13	Core device logging controller	227
25	NixOS for ARTIQ (preinstall handbook)	229
25.1	Getting started	229
25.2	Nix and declarative configuration	230
25.3	Customizing your configuration	232
25.4	Nix and NixOS tips	234

26	FAQ (How do I...)	237
26.1	use this documentation?	237
26.2	build this documentation?	237
26.3	roll back to older versions of ARTIQ, or obtain it through other installation methods?	238
26.4	troubleshoot networking problems?	238
26.5	fix ‘no startup kernel found’ / ‘no idle kernel found’ in the core log?	239
26.6	fix ‘Mismatch between gateway and software versions’?	239
26.7	change configuration settings of satellite devices?	239
26.8	fix unreliable DRTIO master-satellite links?	239
26.9	add or remove EEM peripherals or DRTIO satellites?	240
26.10	see command-line help?	240
26.11	find ARTIQ examples?	240
26.12	fix failed to connect to moninj in the dashboard?	240
26.13	fix address already in use when running ARTIQ commands?	241
26.14	diagnose and fix sequence errors?	242
26.15	understand applet commands?	243
26.16	organize datasets in folders?	243
26.17	organize applets in groups?	243
26.18	organize experiment windows in the dashboard?	243
26.19	fix errors when restarting management system after a crash?	243
26.20	create and use variable-length arrays in kernels?	243
26.21	understand how best to send data between kernel and host?	243
26.22	write part of my experiment as a coroutine/asyncio task/generator?	244
26.23	determine the pyserial URL to connect to a device by its serial number?	244
26.24	run unit tests?	245
26.25	find the dashboard and browser configuration files?	245
27	Additional Resources	247
27.1	Other related documentation	247
27.2	“Help, I’ve done my best and I can’t get any further!”	247
	Python Module Index	249
	Index	251

INTRODUCTION

ARTIQ (Advanced Real-Time Infrastructure for Quantum physics) is a leading-edge control and data acquisition 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. Many laboratories around the world have adopted ARTIQ as their control system and some 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 [FPGA evaluation kit](#) and of a [Zynq 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 MSYS2 packages (for Windows). See [the manual](#) for installation instructions. 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 often [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.

Copyright (C) 2014-2025 M-Labs Limited. Licensed under GNU LGPL version 3+.

ARTIQ RELEASES

ARTIQ follows a rolling release model, with beta, stable, and legacy channels. Different releases are saved as different branches on the M-Labs [ARTIQ repository](#). The `master` branch represents the beta version, where at any time the next stable release of ARTIQ is currently in development. This branch is unstable and does not yet guarantee reliability or consistency, but may also already offer new features and improvements; see the [beta release notes](#) for the most up-to-date information. The `release-[number]` branches represent stable releases, of which the most recent is considered the current stable version, and the second-most recent the current legacy version.

To install the current stable version of ARTIQ, consult the *current* [Installing ARTIQ](#) page. To install beta or legacy versions, consult the same page in their respective manuals. Instructions given in pre-legacy versions of the manual may or may not install their corresponding ARTIQ systems, and may or may not currently be supported (e.g. M-Labs does not host older ARTIQ versions for Conda, and Conda support will probably eventually be removed entirely). Regardless, all out-of-date versions remain available as complete source code on the repository.

The beta manual is hosted [here](#). The current manual is hosted [here](#). The legacy manual is hosted [here](#). Older versions of the manual can be rebuilt from the source files in `doc/manual`, retrieved from the respective branch.

2.1 Release notes

2.1.1 ARTIQ-8

Highlights:

- **New hardware support:**
 - Support for Shuttler, a 16-channel 125MSPS DAC card intended for ion transport. Waveform generator and user API are similar to the NIST PDQ.
 - Implemented Phaser-servo. This requires recent gateway on Phaser.
 - Almazny v1.2 with finer RF switch control.
 - Metlino and Sayma support has been dropped due to complications with synchronous RTIO clocking.
 - More user LEDs are exposed to RTIO on Kasli.
 - Implemented Phaser-MIQRO support. This requires the proprietary Phaser MIQRO gateway variant from QUARTIQ.
 - Sampler: fixed ADC MU to Volt conversion factor for Sampler v2.2+. For earlier hardware versions, specify the hardware version in the device database file (e.g. `"hw_rev": "v2.1"`) to use the correct conversion factor.
- Support for distributed DMA, where DMA is run directly on satellites for corresponding RTIO events, increasing bandwidth in scenarios with heavy satellite usage.
- Support for subkernels, where kernels are run on satellite device CPUs to offload some of the processing and RTIO operations.

- CPU (on softcore platforms) and AXI bus (on Zynq) are now clocked synchronously with the RTIO clock, to facilitate implementation of local processing on DRTIO satellites, and to slightly reduce RTIO latency.
- Support for DRTIO-over-EEM, used with Shuttler.
- Support for WRPLL low-noise clock recovery.
- Enabled event spreading on DRTIO satellites, using high watermark for lane switching.
- Added channel names to RTIO error messages.
- The RTIO analyzer is now proxied by `aqctl_coreanalyzer_proxy` typically running on the master machine, similarly to `aqctl_moninj_proxy`.
- **GUI:**
 - Integrated waveform analyzer, removing the need for external VCD viewers such as GtkWave.
 - Implemented Applet Request Interfaces which allow applets to modify datasets and set the current values of widgets in the dashboard’s experiment windows.
 - Implemented a new `EntryArea` widget which allows argument entry widgets to be used in applets.
 - The “Close all applets” command (shortcut: `Ctrl-Alt-W`) now ignores docked applets, making it a convenient way to clean up after exploratory work without destroying a carefully arranged default workspace.
 - **Hotkeys now organize experiment windows in the order they were last interacted with:**
 - * `CTRL+SHIFT+T` tiles experiment windows
 - * `CTRL+SHIFT+C` cascades experiment windows
 - By enabling the `quickstyle` option, `EnumerationValue` entry widgets can now alternatively display its choices as buttons that submit the experiment on click.
- Datasets can now be associated with units and scale factors, and displayed accordingly in the dashboard including applets, like widgets such as `NumberValue` already did in earlier ARTIQ versions.
- Experiments can now request arguments interactively from the user at any time.
- Persistent datasets are now stored in a LMDB database for improved performance.
- Python’s built-in types (such as `float`, or `List[...]`) can now be used in type annotations on kernel functions.
- MSYS2 packaging for Windows, which replaces Conda. Conda packages are still available to support legacy installations, but may be removed in a future release.
- Experiments can now be submitted with revisions set to a branch / tag name instead of only git hashes.
- Grabber image input now has an optional timeout.
- On NAR3-supported devices (Kasli-SoC, ZC706), when a Rust panic occurs, a minimal environment is started where the network and `artiq_coremgmt` can be used. This allows the user to inspect logs, change configuration options, update the firmware, and reboot the device.
- Full Python 3.11 support.

Breaking changes:

- `SimpleApplet` now calls widget constructors with an additional `ctl` parameter for control operations, which includes dataset operations. It can be ignored if not needed. For an example usage, refer to the `big_number.py` applet.
- `SimpleApplet` and `TitleApplet` now call `data_changed` with additional parameters. Derived applets should change the function signature as below:

```
# SimpleApplet
def data_changed(self, value, metadata, persist, mods)
# SimpleApplet (old version)
def data_changed(self, data, mods)
# TitleApplet
def data_changed(self, value, metadata, persist, mods, title)
# TitleApplet (old version)
def data_changed(self, data, mods, title)
```

Accesses to the data argument should be replaced as below:

```
data[key][0] ==> persist[key]
data[key][1] ==> value[key]
```

- The `ndecimals` parameter in `NumberValue` and `Scannable` has been renamed to `precision`. Parameters after and including `scale` in both constructors are now keyword-only. Refer to the updated `no_hardware/arguments_demo.py` example for current usage.
- `Almazny v1.2` is incompatible with the legacy versions and is the default. To use legacy versions, specify `almazny_hw_rev` in the JSON description.
- `kasli_generic.py` has been merged into `kasli.py`, and the demonstration designs without JSON descriptions have been removed. The base classes remain present in `kasli.py` to support third-party flows without JSON descriptions.
- Legacy PYON databases should be converted to LMDB with the script below:

```
from sipyco import pyon
import lmbd

old = pyon.load_file("dataset_db.pyon")
new = lmbd.open("dataset_db.mdb", subdir=False, map_size=2**30)
with new.begin(write=True) as txn:
    for key, value in old.items():
        txn.put(key.encode(), pyon.encode((value, {})).encode())
new.close()
```

- `artiq.wavesynth` has been removed.

2.1.2 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.

2.1.3 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 `AssertionErrors` 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/pull/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`).

2.1.4 ARTIQ-5

Highlights:

- **Performance improvements:**
 - Faster RTIO event submission (1.5x improvement in pulse rate test) See: <https://git.m-labs.hk/M-Labs/artiq/issues/615>
 - Faster compilation times (3 seconds saved on kernel compilation time on a typical medium-size experiment) See: <https://git.m-labs.hk/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://git.m-labs.hk/M-Labs/artiq/issues/1171>).
 - firmware, gateway 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 gateway-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.gateway.rtio.phy.edge_counter/ artiq.gateway.eem.DIO.add_std` for the gateway components.
 - With DRTIO, Siphaser uses a better calibration mechanism. See: <https://git.m-labs.hk/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 gateway 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_rpc_tool` has been renamed to `sipyco_rpc_tool`.

2.1.5 ARTIQ-4

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.

2.1.6 ARTIQ-3

3.7

No further notes.

3.6

No further notes.

3.5

No further notes.

3.4

No further notes.

3.3

No further notes.

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.

3.1

No further notes.

3.0

- The `--embed` option of applets is replaced with the environment variable `ARTIQ_APPLET_EMBED`. The GUI sets this environment 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://git.m-labs.hk/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.

2.1.7 ARTIQ-2

2.5

No further notes.

2.4

No further notes.

2.3

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

2.2

No further notes.

2.1

No further notes.

2.0

No further notes.

2.0rc2

No further notes.

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\\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.

2.1.8 ARTIQ-1

1.3

No further notes.

1.2

No further notes.

1.1

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

1.0

No further notes.

1.0rc4

- `setattr_argument` and `setattr_device` add their key to `kernel_invariants`.

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.

1.0rc2

- The CPU speed in the pipistrello gateway 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).

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.
- `RTIOWarningError` has been renamed to `RTIOWarning`
- 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).

INSTALLING ARTIQ

M-Labs recommends installing ARTIQ through Nix (on Linux) or MSYS2 (on Windows). It is also possible to use Conda (on either platform), but this is not preferred, and likely to become unsupported in the near future.

3.1 Installing via Nix (Linux)

First, install the Nix package manager. Some distributions provide a package for it. Otherwise, use the official install script, as described on the [Nix website](#), e.g.:

```
$ sh <(curl -L https://nixos.org/nix/install) --no-daemon
```

Prefer the single-user installation for simplicity. Enable [Nix flakes](#), for example by running:

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

3.1.1 User environment installation

There are few options for accessing ARTIQ through Nix. The easiest way is to install it into the user environment:

```
$ nix profile install git+https://git.m-labs.hk/M-Labs/artiq.git?ref=release-8
```

Answer “Yes” to the questions about setting Nix configuration options (for more details see [Installation details](#) below.) You should now have a minimal installation of ARTIQ, where the usual front-end commands ([artiq_run](#), [artiq_master](#), [artiq_dashboard](#), etc.) are all available to you.

This installation is however relatively limited. Without further instructions, Nix takes its cues from the main ARTIQ flake (the `flake.nix` file at the root of the repository linked in the command) and creates a dedicated Python environment for the ARTIQ commands alone. This means that other useful Python packages, which are not necessary to run ARTIQ but which you might want to use in experiments (`pandas`, `matplotlib`...), are not available.

3.1.2 Flake custom environments

Modifying the environment and making additional packages visible to the ARTIQ commands requires using the Nix language and writing your own flake. Create an empty directory with a file `flake.nix` containing the following:

```
{
  inputs.extrapkg.url = "git+https://git.m-labs.hk/M-Labs/artiq-extrapkg.git?ref=release-8";
  outputs = { self, extrapkg }:
    let
      pkgs = extrapkg.pkgs;
```

(continues on next page)

(continued from previous page)

```

artiq = extrapkg.packages.x86_64-linux;
in {
  # This section defines the new environment
  packages.x86_64-linux.default = pkgs.buildEnv {
    name = "artiq-env";
    paths = [
      # =====
      # ADD PACKAGES BELOW
      # =====
      (pkgs.python3.withPackages(ps : [
        # List desired Python packages here.
        artiq.artiq
        #ps.paramiko # needed if and only if flashing boards remotely (artiq_flash -
↪H)
        #artiq.flake8-artiq
        #artiq.dax
        #artiq.dax-applets

        # The NixOS package collection contains many other packages that you may find
        # interesting. Here are some examples:
        #ps.pandas
        #ps.numba
        #ps.matplotlib
        # or if you need Qt (will recompile):
        #(ps.matplotlib.override { enableQt = true; })
        #ps.bokeh
        #ps.cirq
        #ps.qiskit
        # Note that NixOS also provides packages ps.numpy and ps.scipy, but it is
        # not necessary to explicitly add these, since they are dependencies of
        # ARTIQ and incorporated with an ARTIQ install anyway.
      ]))
      # List desired non-Python packages here
      #artiq.openocd-bscanspi # needed if flashing board with artiq_flash
      # Additional NDSPs can be included:
      #artiq.korad_ka3005p
      #artiq.novatech409b
      # Other potentially interesting non-Python packages from the NixOS package_
↪collection:
      #pkgs.gtkwave
      #pkgs.spyder
      #pkgs.R
      #pkgs.julia
      # =====
      # ADD PACKAGES ABOVE
      # =====
    ];
  };
};
# This section configures additional settings to be able to use M-Labs binary caches
nixConfig = { # work around https://github.com/NixOS/nix/issues/6771
  extra-trusted-public-keys = "nixbld.m-labs.hk-

```

(continues on next page)

(continued from previous page)

```

↪1:5aSRVA5b320xbNvu30tqxVPXpld73bhtOeH6uAjRyHc=";
  extra-substituters = "https://nixbld.m-labs.hk";
};
}

```

To spawn a shell in this environment, navigate to the directory containing the `flake.nix` and run:

```
$ nix shell
```

The resulting shell will have access to ARTIQ as well as any additional packages you may have added. You can exit this shell at any time with CTRL+D or with the command `exit`. Note that a first execution of `nix shell` on a given flake may take some time; repetitions of the same command will use stored versions of packages and run much more quickly.

You might be interested in creating multiple directories containing separate `flake.nix` files to represent different sets of packages for different purposes. If you are familiar with Conda, using Nix in this way is similar to having multiple Conda environments.

To find more packages you can browse the [Nix package search](#) website. If your favorite package is not available with Nix, contact M-Labs using the `helpdesk@` email.

Note

If you find you prefer using flakes to your original `nix profile` installation, you can remove it from your system by running:

```
$ nix profile list
```

finding the entry with its `Original flake URL` listed as the Gitea ARTIQ repository, noting its index number (in a fresh Nix system it will normally be the only entry, at index 0), and running:

```
$ nix profile remove [index]
```

While using flakes, ARTIQ is not strictly ‘installed’ in a permanent way. However, Nix will keep collected packages in `/nix/store` for each flake, which over time or with many different flakes and versions can take up large amounts of storage space. To clear this cache, run `nix-collect-garbage`. (After a garbage collection, `nix shell` will require some time again when first used).

3.1.3 Installation details

“Do you want to allow configuration setting... (y/N)?”

When installing and initializing ARTIQ using commands like `nix shell`, `nix develop`, or `nix profile install`, you may encounter prompts to modify certain configuration settings. These settings correspond to the `nixConfig` section in the ARTIQ flake:

```

do you want to allow configuration setting 'extra-sandbox-paths' to be set to '/opt' (y/
↪N)?
do you want to allow configuration setting 'extra-substituters' to be set to 'https://
↪nixbld.m-labs.hk' (y/N)?
do you want to allow configuration setting 'extra-trusted-public-keys' to be set to
↪'nixbld.m-labs.hk-1:5aSRVA5b320xbNvu30tqxVPXpld73bhtOeH6uAjRyHc=' (y/N)?

```

Note

The first is necessary in order to be able to use Vivado to build ARTIQ gateway (e.g. *Building and developing ARTIQ*). The latter two are necessary in order to use the M-Labs nixbld server as a binary cache; refusing these will result in Nix attempting to build these binaries from source, which is possible to do, but requires a considerable amount of time (on the order of hours) on most machines.

It is recommended to accept all three settings by responding with `y`. If asked to permanently mark these values as trusted, choose `y` again. This action saves the configuration to `~/.local/share/nix/trusted-settings.json`, allowing future prompts to be bypassed.

Alternatively, you can also use the option `accept-flake-config` on a per-command basis by appending `--accept-flake-config`, for example:

```
nix shell --accept-flake-config
```

Or add the option to `~/.config/nix/nix.conf` to make the setting apply to all commands by default:

```
extra-experimental-features = flakes
accept-flake-config = true
```

Note

Should you wish to revert to the default settings, you can do so at any time by editing the appropriate options in the aforementioned configuration files.

“Ignoring untrusted substituter, you are not a trusted user”

If the following message displays when running `nix shell` or `nix develop`

```
warning: ignoring untrusted substituter 'https://nixbld.m-labs.hk', you are not a
↳ trusted user.
Run `man nix.conf` for more information on the `substituters` configuration option.
```

and Nix tries to build some packages from source, this means that you are using `multi-user mode` in Nix, which may be the case for example when Nix is installed via `pacman` in Arch Linux. By default, users accessing Nix in multi-user mode are “unprivileged” and cannot use untrusted substituters. To change this, edit `/etc/nix/nix.conf` and add the following line (or append to the key if the key already exists):

```
trusted-substituters = https://nixbld.m-labs.hk
```

This will add the substituter as a trusted substituter for all users using Nix.

Alternatively, add the following line:

```
trusted-users = <username> # Replace <username> with your username
```

This will set your user as a trusted user, allowing you to specify untrusted substituters.

Warning

Setting users as trusted users will effectively grant root access to those users. See the [Nix documentation](#) for more information.

3.2 Installing via MSYS2 (Windows)

We recommend using our [offline installer](#), which contains all the necessary packages and requires no additional configuration. After installation, simply launch MSYS2 with ARTIQ from the Windows Start menu.

Alternatively, you may install [MSYS2](#), then edit `C:\msys64\etc\pacman.conf` and add at the end:

```
[artiq]
SigLevel = Optional TrustAll
Server = https://msys2.m-labs.hk/artiq
```

Launch MSYS2 CLANG64 from the Windows Start menu to open the MSYS2 shell, and enter the following commands:

```
$ pacman -Syy
$ pacman -S mingw-w64-clang-x86_64-artiq
```

As above in the Nix section, you may find yourself wanting to add other useful packages (pandas, matplotlib, etc.). MSYS2 uses a port of ArchLinux's pacman to manage (add, remove, and update) packages. To add a specific package, you can simply use a command of the form:

```
$ pacman -S <package name>
```

For more see the [MSYS2 documentation](#) on package management. If your favorite package is not available with MSYS2, contact M-Labs using the [helpdesk@](#) email.

3.3 Installing via Conda [DEPRECATED]

Warning

Installing ARTIQ via Conda is not recommended. Instead, Linux users should install it via Nix and Windows users should install it via MSYS2. Conda support may be removed in future ARTIQ releases and M-Labs can only provide very limited technical support for Conda.

First, install [Anaconda](#) or the more minimalistic [Miniconda](#). After installing either Anaconda or Miniconda, open a new terminal and verify that the following command works:

```
$ conda
```

Executing just `conda` should print the help of the `conda` command. If your shell cannot find the `conda` command, make sure that the Conda binaries are in your `$PATH`. If `$ echo $PATH` does not show the Conda directories, add them: execute e.g. `$ 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

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 re-running 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.

3.4 Upgrading ARTIQ

Note

When you upgrade ARTIQ, as well as updating the software on your host machine, it may also be necessary to reflash the gateway and firmware of your core device to keep them compatible. New numbered release versions in particular incorporate breaking changes and are not generally compatible. See [\(Re\)flashing your core device](#) for instructions.

3.4.1 Upgrading with Nix

Run `$ nix profile upgrade` if you installed ARTIQ into your user profile. If you use 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 the environment with `$ nix shell`. If you use multiple flakes, each has its own `flake.lock` and can be updated or rolled back separately.

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

3.4.2 Upgrading with MSYS2

Run `pacman -Syu` to update all MSYS2 packages, including ARTIQ. If you get a message telling you that the shell session must be restarted after a partial update, open the shell again after the partial update and repeat the command. See the [MSYS2](#) and [Pacman](#) manuals for more information, including how to update individual packages if required.

3.4.3 Upgrading 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. As a rule, keep previous environments around unless you are certain that they are no longer needed and the new environment is working correctly.

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

Switching between Conda environments using commands such as `$ conda deactivate artiq-7` and `$ conda activate artiq-8` is the recommended way to roll back to previous versions of ARTIQ.

You can list the environments you have created using:

```
$ conda env list
```


(RE)FLASHING YOUR CORE DEVICE

Note

If you have purchased a pre-assembled system from M-Labs or QUARTIQ, the gateway and firmware of your devices will already be flashed to the newest version of ARTIQ. Flashing your device is only necessary if you obtained your hardware in a different way, or if you want to change your system configuration or upgrade your ARTIQ version after the original purchase. Otherwise, skip straight to *Networking and configuration*.

4.1 Obtaining board binaries

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

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. For more information about *afws_client* see also the corresponding entry on the *Utilities* page.

For certain configurations (KC705 or ZC706 only) it is also possible to source firmware from the [M-Labs Hydra server](#) (in *main* and *zynq* respectively).

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

4.2 Installing and configuring OpenOCD

Warning

These instructions are not applicable to Zynq devices (Kasli-SoC or ZC706), which do not use the utility *artiq_flash*. If your core device is a Zynq device, skip straight to *Writing the flash*.

ARTIQ supplies the utility *artiq_flash*, which uses OpenOCD to write the binary images into an FPGA board's flash memory. With MSYS2, OpenOCD is included with the installation by default. For Nix, make sure to include

the package `artiq.openocd-bscanspi` in your flake (in the *example custom flake*, you can simply uncomment the relevant line). Alternatively, you can use the ARTIQ main flake's board development shell. Nix profile installations do not include OpenOCD.

Note that this is **not** `pkgs.openocd`; the latter is OpenOCD from the Nix package collection, which does not support ARTIQ/Sinara boards.

Tip

The board development shell is an alternative, non-minimal ARTIQ environment which includes additional tools for working with ARTIQ, including OpenOCD. You can enter it with:

```
$ nix develop git+https://git.m-labs.hk/M-Labs/artiq?ref=release-8#boards
```

However, unless you want more of these additional tools, it's usually preferable to use a lighter custom flake, such as the example in *Installing ARTIQ*.

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

4.2.1 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 configure OpenOCD through `/etc/nixos/configuration.nix` instead.

4.2.2 Linux using Conda

Note

With Conda, install OpenOCD as follows:

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

If you are using a 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
```

4.2.3 On Windows

A third-party tool, [Zadig](#), is necessary. It is also included with the MSYS2 offline installer and available from the Start Menu as `Zadig Driver Installer`. 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 it has not previously been plugged into, even on the same system.

4.3 Writing the flash

First ensure the board is connected to your computer. In the case of Kasli, the JTAG adapter is integrated into the Kasli board; for flashing (and debugging) you can simply connect your computer to the micro-USB connector on the Kasli front panel. For Kasli-SoC, which uses `artiq_coremgmt` to flash over network, an Ethernet connection and an IP address, supplied either with the `-D` option or in your *device database*, are sufficient.

For Kasli-SoC or ZC706:

```
$ artiq_coremgmt [-D IP_address] config write -f boot <afws_directory>/boot.bin
$ artiq_coremgmt reboot
```

If the device is not reachable due to corrupted firmware or networking problems, extract the SD card and copy `boot.bin` onto it manually.

For Kasli:

```
$ artiq_flash -d <afws_directory>
```

For KC705:

```
$ artiq_flash -t kc705 -d <afws_directory>
```

The SW13 switches need to be set to 00001.

Flashing over network is also possible for Kasli and KC705, assuming IP networking has already been set up. In this case, the `-H HOSTNAME` option is used; see the entry for `artiq_flash` in the *Utilities* reference.

4.4 Connecting to the UART log

A UART is a peripheral device for asynchronous serial communication; in the case of core device boards, it allows the reading of the UART log, which is used for debugging, especially when problems with booting or networking disallow checking core logs with `artiq_coremgmt log`. If you had no issues flashing your board you can proceed directly to *Networking and configuration*.

Otherwise, ensure your core device is connected to your PC with a data micro-USB cable, as above, and wait at least fifteen seconds after startup to try to connect. To help find the correct port to connect to, you can list your system's serial devices by running:

```
$ python -m serial.tools.list_ports -v
```

This will give you the list of `/dev/ttyUSBx` or `COMx` device names (on Linux and Windows respectively). Most commonly, the correct option is the third, i.e. index number 2, but it can vary.

On Linux:

Run the commands:

```
stty 115200 < /dev/ttyUSBx  
cat /dev/ttyUSBx
```

When you restart or reflash the core device you should see the startup logs in the terminal. If you encounter issues, try other `ttyUSBx` names, and make certain that your user is part of the `dialout` group (run `groups` in a terminal to check).

On Windows:

Use a program such as PuTTY to connect to the COM port. Connect to every available COM port at first, restart the core device, see which port produces meaningful output, and close the others. It may be necessary to install the [FTDI drivers](#) first.

Note that the correct parameters for the serial port are 115200bps 8-N-1 for every core device.

NETWORKING AND CONFIGURATION

5.1 Setting up core device 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. Kasli-SoC already directly features RJ45 10/100/1000 Ethernet.

5.1.1 IP address and ping

If you purchased a Kasli or Kasli-SoC device from M-Labs, it will arrive with an IP address already set, normally the address requested in the web shop at time of purchase. If you did not specify an address at purchase, the default IP M-Labs uses is 192.168.1.75. If you did not obtain your hardware from M-Labs, or if you have just reflashed your core device, see *Tips and troubleshooting* below.

Once you know the IP, check that you can ping your device:

```
$ ping <IP_address>
```

If ping fails, check that the Ethernet LED is ON; on Kasli, it is the LED next to the SFP0 connector. As a next step, try connecting to the serial port to read the UART log. See *Connecting to the UART log*.

5.1.2 Core management tool

The tool used to configure the core device is the command-line utility *artiq_coremgmt*. In order for it to connect to your core device, it is necessary to supply it somehow with the correct IP address for your core device. This can be done directly through use of the `-D` option, for example in:

```
$ artiq_coremgmt -D <IP_address> log
```

Note

This command reads and displays the core log. If you have recently rebooted or reflashed your core device, you should see the startup logs in your terminal.

Normally, however, the core device IP is supplied through the *device database* for your system, which comes in the form of a Python script called `device_db.py` (see also *The device database*). If you purchased a system from M-Labs,

the `device_db.py` for your system will have been provided for you, either on the USB stick, inside `~/artiq` on your NUC, or sent by email.

Make sure the field `core_addr` at the top of the file is set to your core device's correct IP address, and always execute `artiq_coremgmt` from the same directory the device database is placed in.

Once you can reach your core device, the IP can be changed at any time by running:

```
$ artiq_coremgmt [-D old_IP] config write -s ip <new_IP>
```

and then rebooting the device:

```
$ artiq_coremgmt [-D old_IP] reboot
```

Make sure to correspondingly edit your `device_db.py` after rebooting.

5.1.3 Tips and troubleshooting

For Kasli-SoC:

If the `ip config` is not set, Kasli-SoC firmware defaults to using the IP address `192.168.1.56`.

For ZC706:

If the `ip config` is not set, ZC706 firmware defaults to using the IP address `192.168.1.52`.

For Kasli or KC705:

If the `ip config` field is not set or set to `use_dhcp`, the device will attempt to obtain an IP address and default gateway using DHCP. The chosen IP address will be in log output, which can be accessed via the *UART log*.

If a static IP address is preferred, it can be flashed directly (OpenOCD must be installed and configured, as in *(Re)flashing your core device*), along with, as necessary, default gateway, IPv6, and/or MAC address:

```
$ artiq_mkfs flash_storage.img [-s mac xx:xx:xx:xx:xx:xx] [-s ip xx.xx.xx.xx/xx] [-  
→s ipv4_default_route xx.xx.xx.xx] [-s ip6 xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx/  
→xx] [-s ipv6_default_route xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx]  
$ artiq_flash -t [board] -V [variant] -f flash_storage.img storage start
```

On Kasli or Kasli-SoC devices, specifying the MAC address is unnecessary, as they can obtain it from their EEPROM. If you only want to access the core device from the same subnet, default gateway and IPv4 prefix length may also be omitted. On any board, once a device can be reached by `artiq_coremgmt`, these values can be set and edited at any time, following the procedure for IP above.

Regarding IPv6, note that the device also has a link-local address that corresponds to its EUI-64, which can be used simultaneously to the (potentially unrelated) IPv6 address defined by using the `ip6` configuration key.

If problems persist, see the *network troubleshooting* section of the FAQ.

5.2 Configuring the core device

Note

The following steps are optional, and you only need to execute them if they are necessary for your specific system. To learn more about how ARTIQ works and how to use it first, you might skip to the first tutorial page, *ARTIQ Real-Time I/O concepts*. For all configuration options, the core device generally must be restarted for changes to take effect.

5.2.1 Flash idle and/or startup kernel

The *idle kernel* is the kernel (that is, a piece of code running on the core device; see *ARTIQ Real-Time I/O concepts* for further explanation) which the core device runs in between experiments and whenever not connected to the host. It is saved directly to the core device's flash storage in compiled form. Potential uses include cleanup of the environment between experiments, state maintenance for certain hardware, or anything else that should run continuously whenever the system is not otherwise occupied.

To flash an idle kernel, first write an idle experiment. Note that since the idle kernel runs regardless of whether the core device is connected to the host, remote procedure calls or RPCs (functions called by a kernel to run on the host) are forbidden and the `run()` method must be a kernel marked with `@kernel`. Once written, you can compile and flash your idle experiment:

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

The *startup kernel* is a kernel executed once and only once immediately whenever the core device powers on. Uses include initializing DDSes and setting TTL directions. For DRTIO systems, the startup kernel should wait until the desired destinations, including local RTIO, are up, using `self.core.get_rtio_destination_status` (see `get_rtio_destination_status()`).

To flash a startup kernel, proceed as with the idle kernel, but using the `startup_kernel` key in the `artiq_coremgmt` command.

Note

Subkernels (see *Using DRTIO and subkernels*) are allowed in idle (and startup) experiments without any additional ceremony. `artiq_compile` will produce a `.tar` rather than a `.elf`; simply substitute `idle.tar` for `idle.elf` in the `artiq_coremgmt config write` command.

5.2.2 Select the RTIO clock source

The core device may use any of an external clock signal, its internal clock with external frequency reference, or its internal clock with internal crystal reference. Clock source and timing are set at power-up. To find out what clock signal you are using, check the startup logs with `artiq_coremgmt log`.

The default is to use an internal 125MHz clock. To select a source, use a command of the form:

```
$ artiq_coremgmt config write -s rtio_clock int_125 # internal 125MHz clock (default)
$ artiq_coremgmt config write -s rtio_clock ext0_synth0_10to125 # external 10MHz_
↪reference used to synthesize internal 125MHz
```

See *Clocking* for availability of specific options.

5.2.3 Set up resolving RTIO channels to their names

This feature allows you to print the channels' respective names alongside with their numbers in RTIO error messages. To enable it, run the `artiq_rtiomap` tool and write its result into the device config at the `device_map` key:

```
$ artiq_rtiomap dev_map.bin
$ artiq_coremgmt config write -f device_map dev_map.bin
```

More information on the `artiq_rtiomap` utility can be found on the *Utilities* page.

5.2.4 Enable event spreading

This feature changes the logic used for queueing RTIO output events in the core device for a more efficient use of FPGA resources, at the cost of introducing nondeterminism and potential unpredictability in certain timing errors (specifically gateway *sequence errors*). It can be enabled with the config key `sed_spread_enable`. See *Event spreading*.

5.2.5 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, you will first need to prepare and load a different routing table. See *Using DRTIO*.

BUILDING AND 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 usually provide board binaries for you; you can use the AFWS client to get updated versions if necessary, as described in *Obtaining board binaries*. It is not necessary to build them yourself.

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 obtain everything from source (look into the `flake.nix` file to see what resources are used, and run the commands manually, adapting to your system) - but Nix makes the process a lot easier.

6.1 Installing Vivado

It is necessary to independently install AMD's [Vivado](#), which requires a login for download and can't be automatically obtained by package managers. The "appropriate" Vivado version to use for building gateware and firmware can vary. Some versions contain bugs that lead to hidden or visible failures, others work fine. Refer to the `flake.nix` file from the ARTIQ repository in order to determine which version is used at M-Labs.

Tip

Text-search `flake.nix` for a mention of `/opt/Xilinx/Vivado`. Given e.g. the line

```
profile = "set -e; source /opt/Xilinx/Vivado/2022.2/settings64.sh"
```

the intended Vivado version is 2022.2.

Download and run the official installer. If using NixOS, note that this will require a FHS chroot environment; the ARTIQ flake provides such an environment, which you can enter with the command `vivado-env` from the development environment (i.e. after `nix develop`). Other tips:

- Be aware that Vivado is notoriously not a lightweight piece of software and you will likely need **at least 70GB+** of free space to install it.
- If you do not want to write to `/opt`, you can install into a folder of your home directory.

- During the Vivado installation, uncheck `Install cable drivers` (they are not required, as we use better open source alternatives).
- 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/`.
- Vivado installation issues are not uncommon. Searching for similar problems on [the M-Labs forum](#) or [Vivado's own support forums](#) might be helpful when looking for solutions.

6.2 System description file

ARTIQ gateway and firmware binaries are dependent on the system configuration. In other words, a specific set of ARTIQ binaries is bound to the exact arrangement of real-time hardware it was generated for: the core device itself, its role in a DRTIO context (master, satellite, or standalone), the (real-time) peripherals in use, the physical EEM ports they will be connected to, and various other basic specifications. This information is normally provided to the software in the form of a JSON file called the system description or system configuration file.

Warning

System configuration files are only used with Kasli and Kasli-SoC boards. KC705 and ZC706 ARTIQ configurations, due to their relative rarity and specialization, are handled on a case-by-case basis and selected through a variant name such as `nist_clock`, with no system description file necessary. See below in [Building ARTIQ](#) for where to find the list of supported variants. Writing new KC705 or ZC706 variants is not a trivial task, and not particularly recommended, unless you are an FPGA developer and know what you're doing.

If you already have your system configuration file on hand, you can edit it to reflect any changes in configuration. If you purchased your original system from M-Labs, or recently purchased new hardware to add to it, you can obtain your up-to-date system configuration file through AFWS at any time using the command `$ afws_client get_json` (see [AFWS client](#)). If you are starting from scratch, a close reading of `coredevice_generic.schema.json` in `artiq/coredevice` will be helpful.

System descriptions do not need to be very complex. At its most basic, a system description looks something like:

```
{
  "target": "kasli",
  "variant": "example",
  "hw_rev": "v2.0",
  "base": "master",
  "peripherals": [
    {
      "type": "grabber",
      "ports": [0]
    }
  ]
}
```

Only these five fields are required, and the `peripherals` list can in principle be empty. A limited number of more extensive examples can currently be found in [the ARTIQ-Zynq repository](#), as well as in the main repository under `artiq/examples/kasli_shuttler`. Once your system description file is complete, you can use `artiq_ddb_template` (see also [Utilities](#)) to test it and to generate a template for the corresponding *device database*.

6.2.1 DRTIO descriptions

Note that in DRTIO systems it is necessary to create one description file *per core device*. Satellites and their connected peripherals must be described separately. Satellites also need to be reflashed separately, albeit only if their personal system descriptions have changed. (The layout of satellites relative to the master is configurable on the fly and will be established much later, in the routing table; see [Configuring the routing table](#). It is not necessary to rebuild or reflash if only changing the DRTIO routing table).

In contrast, only one device database should be generated even for a DRTIO system. Use a command of the form:

```
$ artiq_ddb_template -s 1 <satellite1>.json -s 2 <satellite2>.json <master>.json
```

The numbers designate the respective satellite's destination number, which must correspond to the destination numbers used when generating the routing table later.

6.2.2 Common system description changes

To add or remove peripherals from the system, add or remove their entries from the `peripherals` field. When replacing hardware with upgraded versions, update the corresponding `hw_rev` (hardware revision) field. Other fields to consider include:

- `enable_wrpll` (a simple boolean, see [Clocking](#))
- `sed_lanes` (increasing the number of SED lanes can reduce sequence errors, but correspondingly consumes more FPGA resources, see [Sequence errors](#))
- various defaults (e.g. `core_addr` defines a default IP address, which can be freely reconfigured later).

6.3 Nix development environment

- Install [Nix](#) if you haven't already. Prefer a single-user installation for simplicity.
- Configure Nix to support building ARTIQ:
 - Enable flakes, for example by adding `experimental-features = nix-command flakes` to `nix.conf`. See also the [NixOS Wiki on flakes](#).
 - Add `/opt` (or your Vivado location) as an Nix sandbox, for example by adding `extra-sandbox-paths = /opt` to `nix.conf`.
 - Make sure that you have accepted and marked as permanent the additional settings described in [Installation details](#). You can check on this manually by ensuring the file `trusted-settings.json` in `~/.local/share/nix/` exists and contains the following:

```
{
  "extra-sandbox-paths": {
    "/opt": true
  },
  "extra-substituters": {
    "https://nixbld.m-labs.hk": true
  },
  "extra-trusted-public-keys": {
    "nixbld.m-labs.hk-1:5aSRVA5b320xbNvu30tqxVPXpld73bhtOeH6uAjRyHc=": true
  }
}
```

- If using NixOS, make the equivalent changes to your `configuration.nix` instead.

- Clone the [ARTIQ Git repository](#), or the [ARTIQ-Zynq repository](#) for Zynq devices (Kasli-SoC or ZC706). By default, you are working with the `master` branch, which represents the beta version and is not stable (see [ARTIQ Releases](#)). Checkout the most recent release (`git checkout release-[number]`) for a stable version.
- If your Vivado installation is not in its default location `/opt`, open `flake.nix` and edit it accordingly (note that the edits must be made in the main ARTIQ flake, even if you are working with Zynq, see also tip below).
- Run `nix develop` at the root of the repository, where `flake.nix` is.

Note

You can also target legacy versions of ARTIQ; use Git to checkout older release branches. Note however that older releases of ARTIQ required different processes for developing and building, which you are broadly more likely to figure out by (also) consulting the corresponding older versions of the manual.

Once you have run `nix develop` you are in the ARTIQ development environment. All ARTIQ commands and utilities – `artiq_run`, `artiq_master`, etc. – should be available, as well as all the packages necessary to build or run ARTIQ itself. You can exit the environment at any time using Control+D or the `exit` command and re-enter it by re-running `nix develop` again in the same location.

Tip

If you are developing for Zynq, you will have noted that the ARTIQ-Zynq repository consists largely of firmware. The firmware for Zynq (NAR3) is more modern than that used for current mainline ARTIQ, and is intended to eventually replace it; for now it constitutes most of the difference between the two ARTIQ variants. The gateway for Zynq, on the other hand, is largely imported from mainline ARTIQ.

If you intend to modify the source housed in the original ARTIQ repository, but build and test the results on a Zynq device, clone both repositories and set your `PYTHONPATH` after entering the ARTIQ-Zynq development shell:

```
$ export PYTHONPATH=/absolute/path/to/your/artiq:$PYTHONPATH
```

Note that this only applies for incremental builds. If you want to use `nix build`, or make changes to the dependencies, look into changing the inputs of the `flake.nix` instead. You can do this by replacing the URL of the Gitea ARTIQ repository with `path:/absolute/path/to/your/artiq`; remember that Nix pins dependencies, so to incorporate new changes you will need to exit the development shell, update the environment with `nix flake update`, and re-run `nix develop`.

6.3.1 Building only standard binaries

If you are working with original ARTIQ, and you only want to build a set of standard binaries (i.e. without changing the source code), you can also enter the `boards` shell without cloning the repository, using `nix develop` as follows:

```
$ nix develop git+https://git.m-labs.hk/M-Labs/artiq.git\?ref=release-[number]#boards
```

Leave off `\?ref=release-[number]` to prefer the current beta version instead of a numbered release.

Note

With this command you are making use of the ARTIQ flake's provided `artiq-boards-shell`, a lighter environment optimized for building firmware and flashing boards, which can also be accessed by running `nix develop .#boards` if you have already cloned the repository. Developers should be aware that in this shell the current copy of the ARTIQ sources is not added to your `PYTHONPATH`, which is why the shell can be entered without a local repository. Run `nix flake show` and read `flake.nix` carefully to understand the different available shells.

The parallel command does exist for ARTIQ-Zynq:

```
$ nix develop git+https://git.m-labs.hk/M-Labs/artiq-zynq?ref=release-[number]
```

but if you are building ARTIQ-Zynq without intention to change the source, it is not actually necessary to enter the development environment at all; Nix is capable of accessing the official flake directly to set up a build, eliminating the requirement for any particular environment. For original ARTIQ, the development environment (specifically the `#boards shell`) is still the easiest way to access the necessary tools for flashing a board.

6.4 Building ARTIQ

For general troubleshooting and debugging, especially with a ‘fresh’ board, see also [Connecting to the UART log](#).

6.4.1 Kasli or KC705 (ARTIQ original)

For Kasli, if you have your system description file on-hand, you can at this point build both firmware and gateway with a command of the form:

```
$ python -m artiq.gateway.targets.kasli <description>.json
```

With KC705, use:

```
$ python -m artiq.gateway.targets.kc705 -V <variant>
```

This will create a directory `artiq_kasli` or `artiq_kc705` containing the binaries in a subdirectory named after your description file or variant. Flash the board as described in [Writing the flash](#), adding the option `--srcbuild`, e.g., assuming your board is already connected by JTAG USB:

```
$ artiq_flash --srcbuild [-t kc705] -d artiq_<board>/<variant>
```

Note

To see supported KC705 variants, run:

```
$ python -m artiq.gateway.targets.kc705 --help
```

Look for the option `-V VARIANT`, `--variant VARIANT`.

6.4.2 Kasli-SoC or ZC706 (ARTIQ on Zynq)

The building process for Zynq devices is a little more complex. The easiest method is to leverage `nix build` and the `makeArtiqZynqPackage` utility provided by the official flake. The ensuing command is rather long, because it uses a multi-clause expression in the Nix language to describe the desired result; it can be executed piece-by-piece using the [Nix REPL](#), but `nix build` provides a lot of useful conveniences.

For Kasli-SoC, run:

```
$ nix build --print-build-logs --impure --expr 'let fl = builtins.getFlake "git+https://
↳ git.m-labs.hk/m-labs/artiq-zynq?ref=release-[number]"; in (fl.makeArtiqZynqPackage
↳ {target="kasli_soc"; variant="<variant>"; json=<path/to/description.json>});).kasli_soc-
↳ <variant>-sd'
```

Replace `<variant>` with `master`, `satellite`, or `standalone`, depending on your targeted DRTIO role. Remove `?ref=release-[number]` to use the current beta version rather than a numbered release. If you have cloned the

repository and prefer to use your local copy of the flake, replace the corresponding clause with `builtins.getFlake "/absolute/path/to/your/artiq-zynq"`.

For ZC706, you can use a command of the same form:

```
$ nix build --print-build-logs --impure --expr 'let fl = builtins.getFlake "git+https://
↳git.m-labs.hk/m-labs/artiq-zynq?ref=release-[number]"; in (fl.makeArtiqZynqPackage
↳{target="zc706"; variant="<variant>";}).zc706-<variant>-sd'
```

or you can use the more direct version:

```
$ nix build --print-build-logs git+https://git.m-labs.hk/m-labs/artiq-zynq?ref=release-
↳[number]#zc706-<variant>-sd
```

(which is possible for ZC706 because there is no need to be able to specify a system description file in the arguments.)

Note

To see supported ZC706 variants, you can run the following at the root of the repository:

```
$ src/gateway/zc706.py --help
```

Look for the option `-V VARIANT`, `--variant VARIANT`. If you have not cloned the repository or are not in the development environment, try:

```
$ nix flake show git+https://git.m-labs.hk/m-labs/artiq-zynq?ref=release-[number] |
↳grep "package 'zc706.*sd"
```

to see the list of suitable build targets directly.

Any of these commands should produce a directory `result` which contains a file `boot.bin`. As described in *Writing the flash*, if your core device is currently accessible over the network, it can be flashed with *artiq_coremgmt*. If it is not connected to the network:

1. Power off the board, extract the SD card and load `boot.bin` onto it manually.
2. Insert the SD card back into the board.
3. Ensure that the DIP switches (labeled BOOT MODE) are set correctly, to SD.
4. Power the board back on.

Optionally, the SD card may also be loaded at the same time with an additional file `config.txt`, which can contain preset configuration values in the format `key=value`, one per line. The keys are those used with *artiq_coremgmt*. This allows e.g. presetting an IP address and any other configuration information.

After a successful boot, the “FPGA DONE” light should be illuminated and the board should respond to ping when plugged into Ethernet.

Booting over JTAG/Ethernet

It is also possible to boot Zynq devices over USB and Ethernet. Flip the DIP switches to JTAG. The scripts `remote_run.sh` and `local_run.sh` in the ARTIQ-Zynq repository, intended for use with a remote JTAG server or a local connection to the core device respectively, are used at M-Labs to accomplish this. Both make use of the netboot tool *artiq_netboot*, see also its source [here](#), which is included in the ARTIQ-Zynq development environment. Adapt the relevant script to your system or read it closely to understand the options and the commands being run; note for example that `remote_run.sh` as written only supports ZC706.

You will need to generate the gateway, firmware and bootloader first, either through `nix build` or incrementally as below. After an incremental build add the option `-i` when running either of the scripts. If using `nix build`, note that target names of the form `<board>-<variant>-jtag` (run `nix flake show` to see all targets) will output the three necessary files without combining them into `boot.bin`.

Warning

A known Xilinx hardware bug on Zynq prevents repeatedly loading the SZL bootloader over JTAG (i.e. repeated calls of the `*_run.sh` scripts) without a POR reset. On Kasli-SoC, you can physically set a jumper on the PS_POR_B pins of your board and use the M-Labs [POR reset script](#).

6.4.3 Zynq incremental build

The `boot.bin` file used in a Zynq SD card boot is in practice the combination of several files, normally `top.bit` (the gateway), `runtime` or `satman` (the firmware) and `szl.elf` (an open-source bootloader for Zynq [written by M-Labs](#), used in ARTIQ in place of Xilinx's FSBL). In some circumstances, especially if you are developing ARTIQ, you may prefer to construct these components separately. Be sure that you have cloned the repository and entered the development environment as described above.

To compile the gateway and firmware, enter the `src` directory and run two commands as follows:

For Kasli-SoC:

```
$ gateway/kasli_soc.py -g ../build/gateway <description.json>
$ make TARGET=kasli_soc GWARGS="path/to/description.json" <fw-type>
```

For ZC706:

```
$ gateway/zc706.py -g ../build/gateway -V <variant>
$ make TARGET=zc706 GWARGS="-V <variant>" <fw-type>
```

where `fw-type` is `runtime` for standalone or DRTIO master builds and `satman` for DRTIO satellites. Both the gateway and the firmware will generate into the `../build` destination directory. At this stage you can *boot from JTAG*; either of the `*_run.sh` scripts will expect the gateway and firmware files at their default locations, and the `szl.elf` bootloader is retrieved automatically.

If you prefer to boot from SD card, you will need to construct your own `boot.bin`. Build `szl.elf` from source by running a command of the form:

```
$ nix build git+https://git.m-labs.hk/m-labs/zynq-rs#<board>-szl
```

For easiest access run this command in the build directory. The `szl.elf` file will be in the subdirectory `result`. To combine all three files into the boot image, create a file called `boot.bif` in `build` with the following contents:

```
the_ROM_image:
{
  [bootloader]result/szl.elf
  gateway/top.bit
  firmware/armv7-none-eabihf/release/<fw-type>
}
EOF
```

Save this file. Now use `mkbootimage` to create `boot.bin`.

```
$ mkbootimage boot.bif boot.bin
```

Boot from SD card as above.

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 *gateway*). 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 gateway 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 gateway at the other end guarantees the *all or nothing* level of excellent timing precision.

A FIFO for an output channel holds 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 gateway and are waiting to be read out by the CPU on the other end.

7.1 Timeline and terminology

The set of all input and output events on all channels constitutes the *timeline*. A high-resolution wall clock (`rtio_counter_mu`) counts clock cycles and manages the precise timing of the events. Output events are executed when their timestamp matches the current clock value. Input events are recorded when they reach the gateway and stamped with the current clock value accordingly.

The kernel runtime environment maintains a timeline cursor (called `now_mu`) used as the timestamp when output events are submitted to the FIFOs. Both `now_mu` and `rtio_counter_mu` are counted in integer *machine units*, or *mu*, rather than SI units. The machine unit represents the maximum resolution of RTIO timing in an ARTIQ system. The duration of a machine unit is the *reference period* of the system, and may be changed by the user, but normally corresponds to a duration of one nanosecond.

The timeline cursor `now_mu` can be moved forward or backward on the timeline using `artiq.language.core.delay()` and `artiq.language.core.delay_mu()` (for delays given in SI units or machine units respectively). The absolute value of `now_mu` on the timeline can be retrieved using `artiq.language.core.now_mu()` and it can be set using `artiq.language.core.at_mu()`. The difference between the cursor and the wall clock is referred to as *slack*. A system is considered in a situation of *positive slack* when the cursor is ahead of the wall clock, i.e., in the future; respectively, it is in *negative slack* if the cursor is behind the wall clock, i.e. in the past.

RTIO timestamps, the timeline cursor, and the `rtio_counter_mu` wall clock are all counted 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 typical one-nanosecond machine unit, this covers 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 `mu_to_seconds()`). When accumulating time, do it in machine units and not in SI units, so that rounding errors do not accumulate.

Note

Absolute timestamps are also referred to as *RTIO fine timestamps*, because they run on a significantly finer resolution than the timestamps of the so-called *coarse RTIO clock*, the actual clocking signal provided to or generated by the core device. The frequency of the coarse RTIO clock is set by the core device *clocking settings* but is most commonly 125MHz, which corresponds to eight one-nanosecond machine units per coarse RTIO cycle.

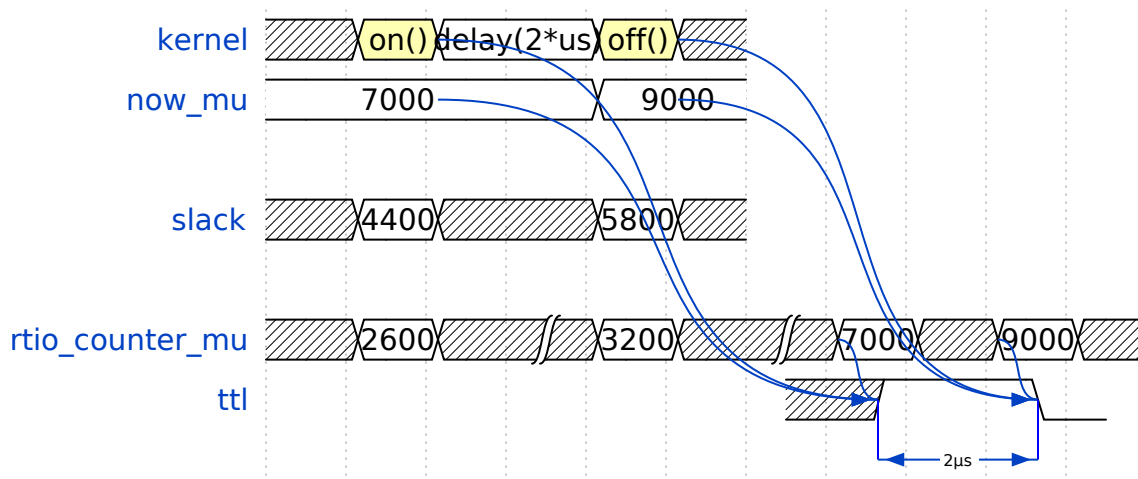
The *coarse timestamp* of an event is its timestamp as according to the lower resolution of the coarse clock. It is in practice a truncated version of the fine timestamp. In general, ARTIQ offers *precision* on the fine level, but *operates* at the coarse level; this is rarely relevant to the user, but understanding it may clarify the behavior of some RTIO issues (e.g. sequence errors).

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

```
t1.on()
delay(2*us)
t1.off()
```

The device `t1` represents a single digital output channel (`artiq.coredevice.t1.TTLOut`). The `artiq.coredevice.t1.TTLOut.on()` method places an rising edge on the timeline at the current cursor position (`now_mu`). Then the cursor is moved forward 2 μ s and a falling edge is placed at the new cursor position. Later, when the wall clock reaches the respective timestamps, the RTIO gateway executes the two events.

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



This sequence is exactly equivalent to:

```
t1.pulse(2*us)
```

This method `artiq.coredevice.t1.TTLOut.pulse()` advances the timeline cursor (using `delay()` internally) by exactly the amount given. Other methods such as `on()`, `off()`, `set()` do not modify the timeline cursor. The latter

are called *zero-duration* methods.

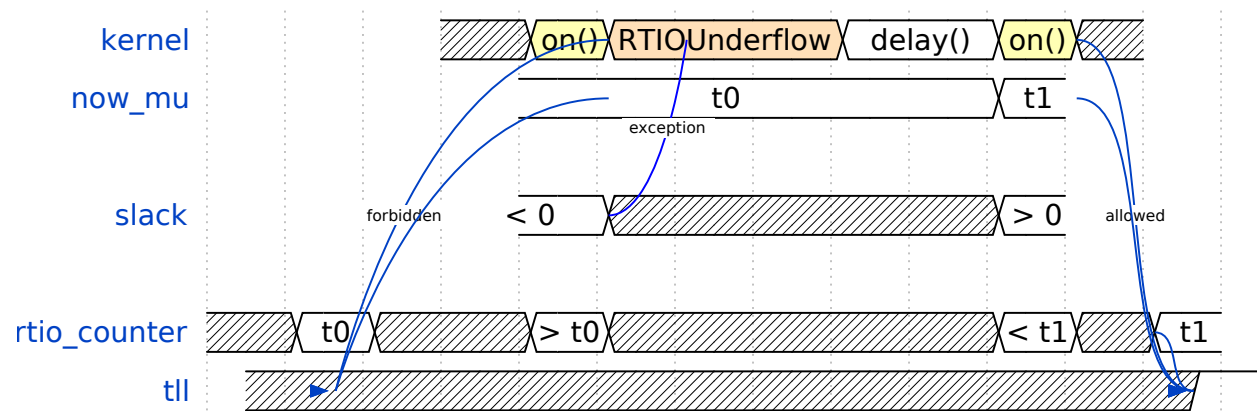
7.2 Output errors and exceptions

7.2.1 Underflows

A RTIO output event must always be programmed with a timestamp in the future. In other words, the timeline cursor `now_mu` must be in advance of the current wall clock `rtio_counter_mu`: the past cannot be altered. The following example tries to place a 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()
```

Once the timeline cursor has overtaken the wall clock, the exception does not reoccur and the event can be scheduled successfully. This can also be thought of as adding positive slack to the system.



To track down `RTIOUnderflow` exceptions 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. This makes it possible to visually find where and how an experiment has ‘run out’ of positive slack.

7.2.2 Sequence errors

A sequence error occurs when a sequence of coarse timestamps cannot be transferred to the gateway. Internally, the gateway stores output events in an array of FIFO buffers (the ‘lanes’). Within each particular lane, the coarse timestamps of events must be strictly increasing.

If an event with a timestamp coarsely equal to or lesser than the previous timestamp is submitted, *or* if the current lane is nearly full, the scaleable event dispatcher (SED) selects the next lane, wrapping around once the final lane is reached. If this lane also contains an event with a timestamp equal to or beyond the one being submitted, the placement fails and a sequence error occurs.

Note

For performance reasons, unlike *RTIOUnderflow*, most gateway errors do not halt execution of the kernel, because the kernel cannot wait for potential error reports before continuing. As a result, sequence errors are not raised as exceptions and cannot be caught. Instead, the offending event – in this case, the event that could not be queued – is discarded, the experiment continues, and the error is reported in the core log. To check the core log, use the command `artiq_coremgmt log`.

By default, the ARTIQ SED has eight lanes, which normally suffices to avoid sequence errors, but problems may still occur if many (>8) events are issued to the gateway with interleaving timestamps. Due to the strict timing limitations imposed on RTIO gateway, it is not possible for the SED to rearrange events in a lane once submitted, nor to anticipate future events when making lane choices. This makes sequence errors fairly ‘unintelligent’, but also generally fairly easy to eliminate by manually rearranging the generation of events (*not* rearranging the timing of the events themselves, which is rarely necessary.)

It is also possible to increase the number of SED lanes in the gateway, which will reduce the frequency of sequencing issues, but will correspondingly put more stress on FPGA resources and timing.

Other notes:

- Strictly increasing (coarse) timestamps never cause sequence errors.
- Strictly increasing *fine* timestamps within the same coarse cycle may still cause sequence errors.
- The number of lanes is a hard limit on the number of RTIO output events that may be emitted within one coarse cycle.
- Zero-duration methods (such as `artiq.coredevice.ttl.TTLOut.on()`) do not advance the timeline and so will always consume additional lanes if they are scheduled simultaneously. Adding a delay of at least one coarse RTIO cycle will prevent this (e.g. `delay_mu(np.int64(self.core.ref_multiplier))`).
- 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 sequence will not affect the result.

Note

To change the number of SED lanes, it is necessary to recompile the gateway and reflash your core device. Use the `sed_lanes` field in your system description file to set the value, then follow the instructions in *Building and developing ARTIQ*. Alternatively, if you have an active firmware subscription with M-Labs, contact `helpdesk@` for edited binaries.

7.2.3 Collisions

A collision occurs when events are submitted to a given RTIO output channel at a resolution the channel is not equipped to handle. Some channels implement ‘replacement behavior’, meaning that RTIO events submitted to the same timestamp will override each other (for example, if a `ttl.off()` and `ttl.on()` are scheduled to the same timestamp, the latter automatically overrides the former and only `ttl.on()` will be submitted to the channel). On the other hand, if replacement behavior is absent or disabled, or if the two events have the same coarse timestamp with differing fine timestamps, a collision error will be reported.

Like sequence errors, collisions originate in gateway and do not stop the execution of the kernel. The offending event is discarded and the problem is reported asynchronously via the core log.

7.2.4 Busy errors

A busy error occurs when at least one output event could not be executed because the output channel was already busy executing an event. This differs from a collision error in that a collision is triggered when a sequence of events overwhelms *communication* with a channel, and a busy error is triggered when *execution* is overwhelmed. Busy errors are only possible in the context of single events with execution times longer than a coarse RTIO clock cycle; the exact parameters will depend on the nature of the output channel (e.g. the specific peripheral device).

Offending event(s) are discarded and the problem is reported asynchronously via the core log.

7.3 Input channels and events

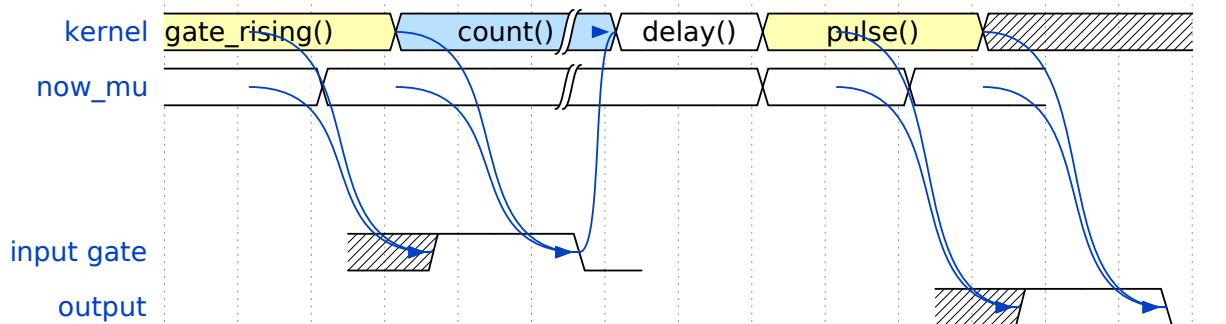
Input channels detect events, timestamp them, and place them in a buffer for the experiment to read out. The following example counts the rising edges occurring during a precisely timed 500 ns interval. If more than 20 rising edges are received, it outputs a pulse:

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

Note that many input methods will necessarily involve the wall clock catching up to the timeline cursor or advancing before it. This is to be expected: managing output events means working to plan the future, but managing input events means working to react to the past. For input channels, it is the past that is under discussion.

In this case, the `gate_rising()` waits for the duration of the 500ns interval (or *gate window*) and records an event for each rising edge. At the end of the interval it exits, leaving the timeline cursor at the end of the interval (`now_mu = rtio_counter_mu`). `count()` unloads these events from the input buffers and counts the number of events recorded, during which the wall clock necessarily advances (`rtio_counter_mu > now_mu`). Accordingly, before we place any further output events, a `delay()` is necessary to re-establish positive slack.

Similar situations arise with methods such as `TTLInOut.sample_get` and `TTLInOut.watch_done`.



7.3.1 Overflow exceptions

The RTIO input channels buffer input events received while an input gate is open, or when using the sampling API (`TTLInOut.sample_input`) at certain points in time. The events are kept in a FIFO until the CPU reads them out via e.g. `count()`, `timestamp_mu()` or `sample_get()`. The size of these FIFOs is finite and specified in gateware; in practice, it is limited by the resources available to the FPGA, and therefore differs depending on the specific core device being used. If a FIFO is full and another event comes in, this causes an overflow condition. The condition is converted into an `RTIOoverflow` exception that is raised on a subsequent invocation of one of the readout methods.

Overflow exceptions are generally best dealt with simply by reading out from the input buffers more frequently. In odd or particular cases, users may consider modifying the length of individual buffers in gateway.

Note

It is not possible to provoke an *RTIOOverflow* on a RTIO output channel. While output buffers are also of finite size, and can be filled up, the CPU will simply stall the submission of further events until it is once again possible to buffer them. Among other things, this means that padding the timeline cursor with large amounts of positive slack is not always a valid strategy to avoid *RTIOUnderflow* exceptions when generating fast event sequences. In practice only a fixed number of events can be generated in advance, and the rest of the processing will be carried out when the wall clock is much closer to `now_mu`.

For larger numbers of events which run up against this restriction, the correct method is to use *Direct Memory Access (DMA)*. In edge cases, enabling event spreading (see below) may fix the problem.

7.4 Event spreading

By default, the SED only ever switches lanes for timestamp sequence reasons, as described above in *Sequence errors*. If only output events of strictly increasing coarse timestamps are queued, the SED fills up a single lane and stalls when it is full, regardless of the state of other lanes. This is preserved to avoid nondeterminism in sequence errors and corresponding unpredictable failures (since the timing of ‘fullness’ depends on the timing of when events are *queued*, which can vary slightly based on CPU execution jitter).

For better utilization of resources and to maximize buffering capacity, *event spreading* may be enabled, which allows the SED to switch lanes immediately when they reach a certain high watermark of ‘fullness’, increasing the number of events that can be queued before stalls ensue. To enable event spreading, use the `sed_spread_enable` config key and set it to 1:

```
$ artiq_coregmt config write -s sed_spread_enable 1
```

This will change where and when sequence errors occur in your kernels, and might cause them to vary from execution to execution of the same experiment. It will generally reduce or eliminate *RTIOUnderflow* exceptions caused by queuing stalls and significantly increase the threshold on sequence length before *DMA* becomes necessary.

Note that event spreading can be particularly helpful in DRTIO satellites, as it is the space remaining in the *fullest* FIFO that is used as a metric for when the satellite can receive more data from the master. The setting is not system-wide and can and must be set independently for each core device in a system. In other words, to enable or disable event spreading in satellites, flash the satellite core configuration directly; this will have no effect on any other satellites or the master.

7.5 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()

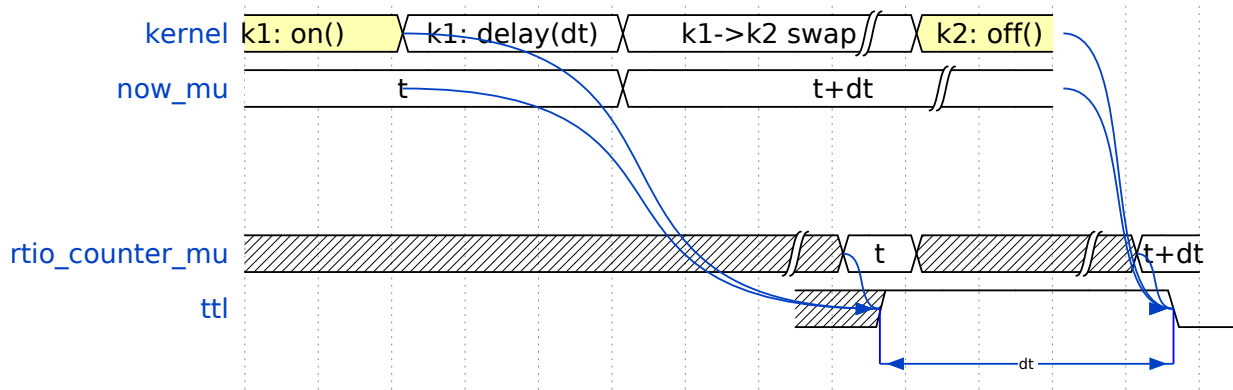
@kernel
def k1():
    ttl.on()
    delay(1*s)
```

(continues on next page)

(continued from previous page)

```
@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.



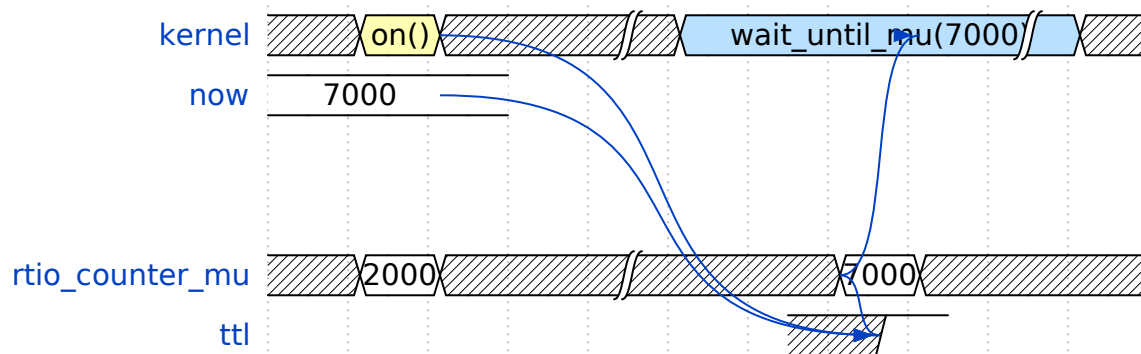
7.6 Synchronization

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. Generally, this is preferable: it frees up resources to the next kernel and allows work to be carried on from kernel to kernel without interruptions.

However, as a result, no guarantees are made about the state of the system when a new kernel enters. Slack may be positive, negative, or zero; input channels may be filled to overflowing, or empty; output channels may contain events currently being executed, contain events scheduled for the far future, or contain no events at all. Unexpected negative slack can cause RTIO underflows. Unexpected large positive slack may cause a system to appear to ‘lock’, as all its events are scheduled for a distant future and the CPU must wait for the output buffers to empty to continue.

As a result, when beginning a new experiment in an uncertain context, we often want to clear the RTIO FIFOs and initialize the timeline cursor to a reasonable point in the near future. The method `artiq.coredevice.core.Core.reset()` (`self.core.reset()`) is provided for this purpose. The example idle kernel implements this mechanism.

On the other hand, if a kernel exits while some of its events are still waiting to be executed, there is no guarantee made that the events in question ever *will* be executed (as opposed to being flushed out by a subsequent core reset). If a kernel should wait until all its events have been executed, use the method `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())
```

GETTING STARTED WITH 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("led0")

    @kernel
    def run(self):
        self.core.reset()
        self.led0.on()
```

The central part of our code is our LED class, which derives from *EnvExperiment*. Almost all experiments should derive from this class, which provides access to the environment as well as including the necessary experiment framework from the base-level *Experiment*. It will call our *build()* at the right time and provides the *setattr_device()* we use to gain access to our devices `core` and `led`. The *kernel()* decorator (`@kernel`) tells the system that the *run()* method is a kernel and must be compiled for and executed on the core device (instead of being interpreted and executed as regular Python code on the host).

Before you can run the example experiment, you will need to supply ARTIQ with the device database for your system, just as you did when configuring the core device. Make sure `device_db.py` is in the same directory as `led.py`. Check once again that the field `core_addr`, placed at the top of the file, matches the current IP address of your core device.

If you don't have a `device_db.py` for your system, consult *The device database* to find out how to construct one. You can also find example device databases in the `examples` folder of ARTIQ, sorted into corresponding subfolders by core device, which you can edit to match your system.

Note

To access the examples, 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 one 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.

8.1 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 automatically handle the communication between the host and the device, conveying between them what function to call, what parameters to call it with, and the resulting value, once returned.

Modify `led.py` 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("led0")

    @kernel
    def run(self):
        self.core.reset()
        s = input_led_state()
        self.core.break_realtime()
        if s:
            self.led0.on()
        else:
            self.led0.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 that 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 function is called on the core device, it sends a request to the host, which executes it. The core device waits until the host returns, and then continues the kernel; in this case, the host displays the prompt, collects user input, and the core device sets the LED state accordingly.

The return type of all RPC functions must be known in advance. If the return value is not `None`, the compiler requires a type annotation, like `-> TBool` in the example above. See also *ARTIQ types*.

Without the `break_realtime()` call, the RTIO events emitted by `self.led0.on()` or `self.led0.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 arbitrarily long amount of time, and therefore the deadline for the submission of RTIO events would have long passed when `self.led0.on()` or `self.led0.off()` are called (that is, the `rtio_counter_mu` wall clock will have advanced far ahead of the timeline cursor `now_mu`, and an *RTIOUnderflow* would result; see *ARTIQ Real-Time I/O concepts* for the full explanation of wall clock vs. timeline.) The `break_realtime()` call is necessary to waive the real-time requirements of the LED state change. Rather than delaying by any particular time interval, it reads `rtio_counter_mu` and moves up the `now_mu` cursor far enough to ensure it's once again safely ahead of the wall clock.

8.2 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 μ s long pulses separated by 2 μ s each. Connect an oscilloscope or logic analyzer to TTL0 and run `artiq_run rtio.py`. Notice that the generated signal’s period is precisely 4 μ s, and that it has a duty cycle of precisely 50%. This is not what one 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 gateway 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 the case that it fails to do so (and attempts to program an event with a timestamp smaller than the current RTIO clock timestamp), `RTIOUnderflow` is raised. The kernel causing it may catch it (using a regular `try... except...` construct), or allow it to propagate 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 occurred")
```

(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()

```

8.3 Parallel and sequential blocks

It is often necessary for several pulses to overlap one another. This can be expressed through the use of the `with parallel` construct, in which the events generated by individual statements are scheduled to execute at the same time, rather than sequentially. 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:

```

from artiq.experiment import *

class Tutorial(EnvExperiment):
    def build(self):
        self.setattr_device("core")
        self.setattr_device("ttl0")
        self.setattr_device("ttl1")

    @kernel
    def run(self):
        self.core.reset()
        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 its position on the timeline (`now_mu`) when entering the `parallel` block and resets to that position after each individual statement. At the end of the block, the cursor is advanced to the furthest position it reached during the block. In other words, the statements in a `parallel` block are actually executed sequentially. Only the RTIO events generated by the statements are *scheduled* in parallel.

Remember that while `now_mu` resets at the beginning of each statement in a `parallel` block, the wall clock advances regardless. If a particular statement takes a long time to execute (which is different from – and unrelated to! – the events *scheduled* by the statement taking a long time), the wall clock may advance past the reset value, putting any subsequent statements inside the block into a situation of negative slack (i.e., resulting in *RTIOUnderflow*). Sometimes underflows may be avoided simply by reordering statements within the parallel block. This especially applies to input methods, which generally necessarily block CPU progress until the wall clock has caught up to or overtaken the cursor.

Within a parallel block, some statements can be scheduled sequentially again using a `with sequential` block. 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)
```

Warning

`with parallel` specifically ‘parallelizes’ the *top-level* statements inside a block. Consider as an example:

```
1  for i in range(1000000):
2      with parallel:
3          self.ttl0.pulse(2*us)
4          if True:
5              self.ttl1.pulse(2*us)
6              self.ttl2.pulse(2*us)
7      delay(4*us)
```

This code will not schedule the three pulses to `ttl0`, `ttl1`, and `ttl2` in parallel. Rather, the pulse to `ttl1` is ‘parallelized’ *with the if statement*. The timeline cursor resets once, at the beginning of line #4; it will not repeat the reset at the deeper indentation level for #5 or #6.

In practice, the pulses to `ttl0` and `ttl1` will execute simultaneously, and the pulse to `ttl2` will execute after the pulse to `ttl1`, bringing the total duration of the `parallel` block to 4 us. Internally, lines #5 and #6, contained within the top-level if statement, are considered an atomic sequence and executed within an implicit `with sequential`. To schedule #5 and #6 in parallel, it is necessary to place them inside a second, nested `parallel` block within the if statement.

Particular care needs to be taken when working with `parallel` blocks which generate large numbers of RTIO events, as it is possible to cause sequencing issues in the gateway; see also [Sequence errors](#).

8.4 RTIO analyzer

The core device records all real-time I/O waveforms, as well as the variation of RTIO slack, into a circular buffer, the contents of which can be extracted using [artiq_coreanalyzer](#). Try for example:

```
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(5):
```

(continues on next page)

(continued from previous page)

```
self.ttl0.pulse(0.1 * ms)
delay(0.1 * ms)
```

When using `artiq_run`, the recorded buffer data can be extracted directly into the terminal, using a command in the form of:

```
$ artiq_coreanalyzer -p
```

Note

The first time this command is run, it will retrieve the entire contents of the analyzer buffer, which may include every experiment you have run so far. For a more manageable introduction, run the analyzer once to clear the buffer, run the experiment, and then run the analyzer a second time, so that only the data from this single experiment is displayed.

This will produce a list of the exact output events submitted to RTIO, printed in chronological order, along with the state of both `now_mu` and `rtio_counter_mu`. While useful in diagnosing some specific gateway errors (in particular, *sequencing issues*), it isn't the most readable of formats. An alternate is to export to VCD, which can be viewed using third-party tools such as GTKWave. Run the experiment again, and use a command in the form of:

```
$ artiq_coreanalyzer -w <file_name>.vcd
```

The `<file_name>.vcd` file should be immediately created and written. Check the directory the command was run in to find it.

Tip

Tutorials on GTKWave options (or other third-party tools) and how best to view VCD files can be found online. By default, the data in a trace like `rtio_slack` will probably be presented in a raw form. To see a stepped wave as in the ARTIQ dashboard, look for options to interpret the data as a real number, then as an analog signal.

Pay attention to the timescale of the waveform dock in your chosen viewer; if you have set your signals to display but nothing is visible, it is likely zoomed in or out much too far.

The easiest way to view recorded analyzer data, however, is directly in the ARTIQ dashboard, a feature which will be presented later in *Waveform*.

8.5 Direct Memory Access (DMA)

DMA allows for storing fixed sequences of RTIO events in system memory and having the DMA core in the FPGA play them back at high speed. Provided that the specifications of a desired event sequence are known far enough in advance, and no other RTIO issues (collisions, sequence errors) are provoked, even extremely fast and detailed event sequences can always be generated and executed. RTIO underflows occur when events cannot be generated *as fast as* they need to be executed, resulting in an exception when the wall clock 'catches up' to `now_mu`. The solution is to record these sequences to the DMA core. Once recorded, event sequences are fixed and cannot be modified, but can be safely replayed very quickly at any position in the timeline, potentially repeatedly.

Try this:

```

from artiq.experiment import *

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

    @kernel
    def record(self):
        with self.core_dma.record("pulses"):
            # all RTIO operations now_mu 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)

```

Note

Only output events are redirected to the DMA core. Input methods inside a `with dma` block will be called as they would be outside of the block, in the current real-time context, and input events will be buffered normally, not to DMA.

For more documentation on the methods used, see the [artiq.coredevice.dma](#) reference.

USING THE MANAGEMENT SYSTEM

In practice, rather than managing experiments by executing `artiq_run` over and over, most use cases are better served by using the ARTIQ *management system*. This is the high-level application part of ARTIQ, which can be used to schedule experiments, manage devices and parameters, and distribute and store results. It also allows for distributed use of ARTIQ, with a single master coordinating demands on the system issued over the network by multiple clients. Using this system, multiple users on different machines can schedule experiments or analyze results on the same ARTIQ system, potentially simultaneously, without interfering with each other.

The management system consists of at least two parts:

- a. the **ARTIQ master**, which runs on a single machine, facilitates communication with the core device and peripherals, and is responsible for most of the actual duties of the system,
- b. one or more **ARTIQ clients**, which may be local or remote and which communicate only with the master. Both a GUI (the **dashboard**) and a straightforward **command line client** are provided, with many of the same capabilities.

as well as, optionally,

- c. one or more **controller managers**, which coordinate the operation of certain (generally, non-realtime) classes of device and provide certain services to the clients,
- d. and one or more instances of the **ARTIQ browser**, a GUI application designed to facilitate the analysis of experiment results and datasets.

In this tutorial, we will explore the basic operation of the management system. Because the various components of the management system run wholly on the host machine, and not on the core device (in other words, they do not inherently involve any kernel functions), it is not necessary to have a core device or any specialized hardware set up to use it. The examples in this tutorial can all be carried out using only your host computer.

9.1 Running your first experiment with the master

Until now, we have executed experiments using `artiq_run`, which is a simple standalone tool that bypasses the management system. We will now see how to run an experiment using a master and a client. In this arrangement, the master is responsible for communicating with the core device, scheduling and keeping track of experiments, and carrying out RPCs the core device may call for. Clients submit experiments to the master to be scheduled and, if necessary, query the master about the state of experiments and their results.

First, create a folder called `~/artiq-master`. Copy into it the `device_db.py` for your system (your device database, exactly as in *Getting started with the core device*); the master uses the device database in the same way as `artiq_run` when communicating with the core device.

Tip

Since no devices are actually used in these examples, you can also use a device database in the model of the `device_db.py` from `examples/no_hardware`, which uses resources from `artiq/sim` instead of referencing or requiring any real local hardware.

Secondly, create a subfolder `~/artiq-master/repository` to contain experiments. By default, the master scans for a folder of this name to determine what experiments are available; if you'd prefer to use a different name, this can be changed by running `artiq_master -r [folder name]` instead of `artiq_master` below. Experiments don't have to be in the repository to be submitted to the master, but the repository contains those experiments the master is automatically aware of.

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 command should display `ARTIQ master is now ready` and not return, as the master keeps running. In another terminal, use the client to request this experiment:

```
$ artiq_client submit repository/mgmt_tutorial.py
```

This command should print a message in the format `RID: 0`, telling you the scheduling ID assigned to the experiment by the master, and exit. Note that it doesn't matter *where* the client is run; the client does not require direct access to `device_db.py` or the repository folder, and only directly communicates with the master. Relatedly, the path to an experiment a client submits is given relative to the location of the *master*, not the client.

Return to the terminal where the master is running. You should see an output similar to:

```
INFO:worker(0,mgmt_tutorial.py):print:Hello World
```

In other words, a worker created by the master has executed the experiment and carried out the print instruction. Congratulations!

Tip

In order to run the master and the clients on different PCs, start the master with a `--bind` flag:

```
$ artiq_master --bind [hostname or IP to bind to]
```

and then use the option `--server` or `-s` for clients, as in:

```
$ artiq_client -s [hostname or IP of the master]
$ artiq_dashboard -s [hostname or IP of the master]
```

Both IPv4 and IPv6 are supported. See also the individual references *artiq_master*, *artiq_dashboard*, and *artiq_client* for more details.

You may also notice that the master has created some other organizational files in its home directory, notably a folder `results`, where a HDF5 record is preserved of every experiment that is submitted and run. The files in `results` will be discussed in greater detail in *Data and user interfaces*.

9.2 Running the dashboard and controller manager

Submitting experiments with *artiq_client* has some interesting qualities: for instance, experiments can be requested simultaneously by different clients and be relied upon to execute cleanly in sequence, which is useful in a distributed context. On the other hand, on a local level, it doesn't necessarily carry many practical advantages over using *artiq_run*. The real convenience of the management system lies in its GUI, the dashboard. We will now try submitting an experiment using the dashboard.

First, start the controller manager:

```
$ artiq_ctlmgr
```

Like the master, this command should not return, as the controller manager keeps running. Note that the controller manager requires access to the device database, but not in the local directory – it gets that access automatically by connecting to the master.

Note

We will not be using controllers in this part of the tutorial. Nonetheless, the dashboard will expect to be able to contact certain controllers given in the device database, and print error messages if this isn't the case (e.g. `Is aqctl_moninj_proxy running?`). It is equally possible to check your device database and start the requisite controllers manually, or to temporarily delete their entries from `device_db.py`, but it's normally quite convenient to let the controller manager handle things. The role and use of controller managers will be covered in more detail in *Data and user interfaces*.

In a third terminal, start the dashboard:

```
$ artiq_dashboard
```

Like *artiq_client*, the dashboard requires no direct access to the device database or the repository. It communicates with the master to receive information about available experiments and the state of the system.

You should see the list of experiments from the repository in the dock called 'Explorer'. In our case, this will only be the single experiment we created, listed by the name we gave it in the docstring inside the triple quotes, "Management tutorial". Select it, and in the window that opens, click 'Submit'.

This time you will find the output displayed directly in the dock called 'Log'. The dashboard log combines the master's console output, the dashboard's own logs, and the device logs of the core device itself (if there is one in use); normally, this is the only log it's necessary to check.

9.3 Adding a new experiment

Create a new file in your repository folder, called `timed_tutorial.py`:

```

from artiq.experiment import *
import time

class TimedTutorial(EnvExperiment):
    """Timed tutorial"""
    def build(self):
        pass # no devices used

    def run(self):
        print("Hello World")
        time.sleep(10)
        print("Goodnight World")

```

Save it. You will notice that it does not immediately appear in the ‘Explorer’ dock. For stability reasons, the master operates with a cached idea of the repository, and changes in the file system will often not be reflected until a *repository rescan* is triggered.

You can ask it to do this through the command-line client:

```
$ artiq_client scan-repository
```

or you can right-click in the Explorer and select ‘Scan repository HEAD’. Now you should be able to select and submit the new experiment.

If you switch the ‘Log’ dock to its ‘Schedule’ tab while the experiment is still running, you will see the experiment appear, displaying its RID, status, priority, and other information. Click ‘Submit’ again while the first experiment is in progress, and a second iteration of the experiment will appear in the Schedule, queued up to execute next in line.

Note

You may have noted that experiments can be submitted with a due date, a priority level, a pipeline identifier, and other specific settings. Some of these are self-explanatory. Many are scheduling-related. For more information on experiment scheduling, see *Experiment scheduling*.

In the meantime, you can try out submitting either of the two experiments with different priority levels and take a look at the queues that ensue. If you are interested, you can try submitting experiments through the command line client at the same time, or even open a second dashboard in a different terminal. Observe that no matter the source, all submitted experiments will be accounted for and handled by the scheduler in an orderly way.

9.4 Adding arguments

Experiments may have arguments, values which can be set in the dashboard on submission and used in the experiment’s code. Create a new experiment called `argument_tutorial.py`, and give it the following `build()` and `run()` functions:

```

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

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

```

The method `setattr_argument()` acts to set the argument and make its value accessible, similar to the effect of `setattr_device()`. The second input sets the type of the argument; here, `NumberValue` represents a floating

point numerical value. To learn what other types are supported, see [artiq.language.environment](#) and [artiq.language.scan](#).

Rescan the repository as before. Open the new experiment in the dashboard. Above the submission options, you should now see a spin box that allows you to set the value of `count`. Try setting it and submitting it.

9.5 Interactive arguments

With standard arguments, it is only possible to use `setattr_argument()` in `build()`; these arguments are always requested at submission time. However, it is also possible to use *interactive* arguments, which can be requested and supplied inside `run()`, while the experiment is being executed. Modify the experiment as follows (and push the result):

```
def build(self):
    pass

def run(self):
    repeat = True
    while repeat:
        print("Hello World")
        with self.interactive(title="Repeat?") as interactive:
            interactive.setattr_argument("repeat", BooleanValue(True))
        repeat = interactive.repeat
```

Close and reopen the submission window, or click on the button labeled ‘Recompute all arguments’, in order to update the submission parameters. Submit again. It should print once, then wait; you may notice in ‘Schedule’ that the experiment does not exit, but hangs at status ‘running’.

Now, in the same dock as ‘Explorer’, navigate to the tab ‘Interactive Args’. You can now choose and submit a value for ‘repeat’. Every time an interactive argument is requested, the experiment pauses until an input is supplied.

Note

If you choose to ‘Cancel’ instead, an `CancelledArgsError` will be raised (which an experiment can catch, instead of halting).

In order to request and supply multiple interactive arguments at once, simply place them in the same `with` block; see also the example `interactive.py` in `examples/no_hardware`.

9.6 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 tracking modifications to experiments and with the traceability to a particular version of an experiment.

Note

The workflow we will describe in this tutorial is designed for a situation where the computer running the ARTIQ master is also used as a Git server to which multiple users may contribute code. This is not the only way Git integration can be useful, and the 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 checked-out files that may be present, to fetch the latest *fully completed commit* at the repository’s head. See also [Git integration](#), especially if you are unfamiliar with Git.

We will use our current `repository` folder as the working directory for making local modifications to the experiments, move it away from the master's data directory, and replace it with a new `repository` folder, which will hold only 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 initialize a regular (non-bare) Git repository in our working directory:

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

Then add and commit our experiments:

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

and finally, connect the two repositories and push the commit upstream to the master's repository:

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

Tip

If you are not familiar with command-line Git and would like to understand these commands in more detail, look for tutorials on the basic use of Git; there are many available online.

Start the master again with the `-g` flag, which tells it to treat its `repository` folder as a bare Git repository:

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

Note

Note that you need at least one commit in the repository before the master can be started.

Now you should be able to restart the dashboard and see your experiments there.

To make things more convenient, we will make Git tell the master to rescan the repository whenever new data is pushed from downstream. Create a file `~/artiq-master/repository/hooks/post-receive` with the following contents:

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

Then set its execution permissions:

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

Note

Remote client machines may also push and pull into the master repository, using e.g. Git over SSH.

Let's now make a modification to the experiments. In the working directory `artiq-work`, open `mgmt_tutorial.py` again and add an exclamation mark to the end of "Hello World". Before committing it, check that the experiment can still be executed correctly by submitting it directly from the working directory, using the command-line client:

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

Note

The `--content` flag submits by content, that is, by sending a raw file rather than selecting an experiment from the master's local environment. Since you are likely running the client and the master on the same machine, it is probably not strictly necessary here. In a distributed setup across multiple machines, the master will not have access to the client's filesystem, and the `--content` flag is the only way to run experiments directly from a client file. See also *Submission from the raw filesystem*.

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
```

Notice that commands other than `git commit` and `git push` are no longer necessary. The Git hook should cause a repository rescan automatically, and submitting the experiment in the dashboard should run the new version, with enthusiasm included.

9.7 The ARTIQ session

Often, you will want to run an instance of the controller manager and dashboard along with the ARTIQ master, whether or not you also intend to allow other clients to connect remotely. For convenience, all three can be started simultaneously with a single command:

```
$ artiq_session
```

Arguments to the individual tools (including `-s` and `--bind`) can still be specified using the `-m`, `-d` and `-c` options for master, dashboard and manager respectively. Use an equals sign to avoid confusion in parsing, for example:

```
$ artiq_session -m=-g
```

to start the session with Git integration. See also *artiq_session*.

DATA AND USER INTERFACES

Beyond running single experiments, or basic use of master, client, and dashboard, ARTIQ supports a much broader range of interactions between its different components. These integrations allow for system control which is extensive, flexible, and dynamic, with space for different workflows and methodologies depending on the needs of particular sets of experiments. We will now explore some of these further tools and systems.

Note

This page follows up directly on *Using the management system*, but discusses a broader range of features, most of which continue to rest on the foundation of the management system. Some sections (datasets, the browser) are still possible to try out using your PC alone; others (MonInj, the RTIO analyzer) are only meaningful in relation to real-time hardware.

10.1 Datasets and results

ARTIQ uses the concept of *datasets* to manage the data exchanged with experiments, both supplied *to* experiments (generally, from other experiments) and saved *from* experiments (i.e. results or records). We will now see how to view and manipulate datasets, both in experiments or through the management system. Create a new experiment as follows:

```
from artiq.experiment import *
import numpy as np
import time

class Datasets(EnvExperiment):
    """Dataset tutorial"""
    def build(self):
        pass # no devices used

    def run(self):
        self.set_dataset("parabola", np.full(10, np.nan), broadcast=True)
        for i in range(10):
            self.mutate_dataset("parabola", i, i*i)
            time.sleep(0.5)
```

Save it as `dataset_tutorial.py`. Commit and push your changes, or rescan the repository, whichever is appropriate for your management system configuration. Submit the experiment. In the same window as 'Explorer', navigate to the 'Dataset' tab and observe that a new dataset has been created under the name `parabola`. As the experiment runs, the values are progressively calculated and entered.

Note

By default, datasets are primarily attributes of the experiments that run them, and are not shared with the master or the dashboard. The `broadcast=True` argument specifies that a dataset should be shared in real-time with the master, which is then responsible for storing it in the dataset database `dataset_db.mdb` and dispatching it to any clients. For more about dataset options see the [Environment](#) page.

As long as `archive=False` is not explicitly set, datasets are among the information preserved by the master in the `results` folder. The files in `results` are organized in subdirectories based on the time they were executed, as `results/<date>/<hour>/`; their individual filenames are a combination of the RID assigned to the execution and the name of the experiment module itself. As such, results are stored in a unique and identifiable location for each iteration of an experiment, even if a dataset is overwritten in the master.

You can open the result file for this experiment with HDFView, `h5dump`, or any similar third-party tool. Observe that it contains the dataset we just generated, as well as other useful information such as RID, run time, start time, and the Git commit ID of the repository at the time of the experiment (a hexadecimal hash such as `947acb1f90ae1b8862efb489a9cc29f7d4e0c645`).

Tip

If you are not familiar with Git, try running `git log` in either of your connected Git repositories to see a history of commits in the repository which includes their respective hashes. As long as this history remains intact, you can use a hash of this kind of to uniquely identify, and even retrieve, the state of the files in the repository at the time this experiment was run. In other words, when running experiments from a Git repository, it's always possible to retrieve the code that led to a particular set of results.

A last interesting feature of the result files is that, for experiments with arguments, they also store the values of the arguments used for that iteration of the experiment. Again, this is for reproducibility: if it's ever necessary to find what arguments produced certain results, that information is preserved in the HDF5 file. To repeat an experiment with the exact same arguments as in a previous run, the 'Load HDF5' button in the submission window can be used to take them directly from a result file.

10.1.1 Applets

Most of the time, rather than the HDF dump, we would like to see our result datasets in a readable graphical form, preferably without opening any third-party applications. In the ARTIQ dashboard, this is achieved by programs called "applets". Applets provide simple, modular GUI features, and are run independently from the dashboard as separate processes for modularity and resilience. ARTIQ supplies several applets for basic plotting in the `artiq.applets` module, and provides interfaces so users can write their own.

See also

Resources for writing your own applets are detailed on the [Management system reference](#) page.

For our `parabola` dataset, we will create an XY plot using the provided `artiq.applets.plot_xy`. Applets are configured with simple command line options. To figure out what configurations are accepted, use the `-h` flag, as in:

```
$ python3 -m artiq.applets.plot_xy -h
```

In our case, we only need to supply our dataset to the applet to be plotted. Navigate to the "Applet" dock in the dashboard. Right-click in the empty list and select "New applet from template" and "XY". This will generate a version

of the applet command which shows all the configuration options; edit the line so that it retrieves the `parabola` dataset and erase the unused options. It should look like:

```
{artiq_applet}plot_xy parabola
```

Run the experiment again, and observe how the points are added to the plot in the applet window as they are generated.

Tip

Datasets and applets can both be arranged in groups for organizational purposes. (In fact, so can arguments; see the reference of `setattr_argument()`). For datasets, use a dot (`.`) in names to separate folders. For applets, left-click in the applet list to see the option ‘Create Group’. You can drag and drop to move applets in and out of groups, or select a particular group with a click to create new applets in that group. Deselect applets or groups with `CTRL+click`.

Tip

You can close all open, undocked applets with the shortcut `CTRL+ALT+W`. Docked applets will remain where they are. This is a convenient way to clean up after exploratory work without destroying a carefully arranged workspace.

10.1.2 The ARTIQ browser

ARTIQ also possesses a second GUI, specifically targeted for the manipulation and analysis of datasets, called the ARTIQ browser. It is standalone, and does not require either a running master or a core device to operate; a connection to the master is only necessary if you want to upload edited datasets back to the main management system. Open results in the browser by running:

```
$ cd ~/artiq-master
$ artiq_browser ./results
```

Navigate to the entry containing your `parabola` datasets in the file explorers on the left. To bring the dataset into the browser, click on the HDF5 file.

To open an experiment, click on ‘Experiment’ at the top left. Observe that instead of ‘Submit’, the option given is ‘Analyze’. Where `artiq_run` and `artiq_master` ultimately call `prepare()`, `run()`, and `analyze()`, the browser limits itself to `analyze()`. Nonetheless, it still accepts arguments.

As described later in *Experiment scheduling*, only `run()` is obligatory for experiments to implement, and only `run()` is permitted to access hardware; the preparation and analysis stages occur before and after, and are limited to the host machine. The browser allows for re-running the post-experiment `analyze()`, potentially with different arguments or an edited algorithm, while accessing the datasets from opened `results` files.

Notably, the browser does not merely act as an HDF5 viewer, but also allows the use of ARTIQ applets to plot and view the data. For this, see the lower left dock; applets can be opened, closed, and managed just as they are in the dashboard, once again accessing datasets from `results`.

10.2 Non-RTIO devices and the controller manager

As described in *ARTIQ Real-Time I/O concepts*, there are two classes of equipment a laboratory typically finds itself needing to operate. So far, we have largely discussed ARTIQ in terms of one only: specialized hardware which requires the very high-resolution timing control ARTIQ provides. The other class comprises the broad range of regular, “slow” laboratory devices, which do *not* require nanosecond precision and can generally be operated perfectly well from a regular PC over a non-realtime channel such as USB.

To handle these “slow” devices, ARTIQ uses *controllers*, intermediate pieces of software which are responsible for the direct I/O to these devices and offer RPC interfaces to the network. By convention, ARTIQ controllers are named with the prefix `aqctl_`. Controllers can be started and run standalone, but are generally handled through the *controller manager*, `artiq_ctlmgr`. The controller manager in turn communicates with the ARTIQ master, and through it with clients or the GUI.

Like clients, controllers do not need to be run on the same machine as the master. Various controllers in a device database may in fact be distributed across multiple machines, in whatever way is most convenient for the devices in question, alleviating cabling issues and OS compatibility problems. Each machine running controllers must run its own controller manager. Communication with the master happens over the network. Use the `-s` flag of `artiq_ctlmgr` to set the IP address or hostname of a master to bind to.

Tip

The controller manager is made available through the `artiq-comtools` package, maintained separately from the main ARTIQ repository. It is considered a dependency of ARTIQ, and is normally included in any ARTIQ installation, but can also be installed independently. This is especially useful when controllers are widely distributed; instead of installing ARTIQ on every machine that runs controllers, only `artiq-comtools` and its much lighter set of dependencies are necessary. See the source repository [here](#).

We have already used the controller manager in the previous part of the tutorial. To run it, the only command necessary is:

```
$ artiq_ctlmgr
```

Note however that in order for the controller manager to be able to start a controller, the controller in question must first exist and be properly installed on the machine the manager is running on. For laboratory devices, this normally means it must be part of a complete Network Device Support Package, or NDSP. *Some NDSPs are already available*. If your device is not on this list, the protocol is designed to make it relatively simple to write your own; for more information and a tutorial, see the [Developing a Network Device Support Package \(NDSP\)](#) page.

Once a device is correctly listed in `device_db.py`, it can be added to an experiment using `self.setattr_device([device_name])` and the methods its API offers called straightforwardly as `self.[device_name].[method_name]`. As long as the requisite controllers are running and available, the experiment can then be executed with `artiq_run` or through the management system. To understand how to add controllers to the device database, see also [The device database](#).

10.2.1 ARTIQ built-in controllers

Certain built-in controllers are also included in a standard ARTIQ installation, and can be run directly in your ARTIQ shell. They are listed at the end of the [Utilities](#) reference (the commands prefixed with `aqctl_` rather than `artiq_`) and included by default in device databases generated with `artiq_ddb_template`.

Broadly speaking, these controllers are edge cases, serving as proxies for interactions between clients and the core device, which otherwise do not make direct contact with each other. Features like dashboard MonInj and the RTIO analyzer’s Waveform tab, both discussed in more depth below, depend upon a respective proxy controller to function. A proxy controller is also used to communicate the core log to dashboards.

Although they are listed in the references for completeness’ sake, there is normally no reason to run the built-in controllers independently. A controller manager run alongside the master (or anywhere else, provided the given addresses are edited accordingly; proxy controllers communicate with the core device by network just as the master does) is more than sufficient.

10.3 Using MonInj

One of ARTIQ’s most convenient features is the Monitor/Injector, commonly known as MonInj. This feature allows for checking (monitoring) the state of various peripherals and setting (injecting) values for their parameters, directly and without any need to explicitly run an experiment for either. MonInj is integrated into ARTIQ on a gateway level, and (except in the case of injection on certain peripherals) can be used in parallel to running experiments, without interrupting them.

In order to use dashboard MonInj, `aqctl_moninj_proxy` or a local controller manager must be running. Given this, navigate to the dashboard’s MonInj tab. Mouse over the second button at the top of the dock, which is labeled ‘Add channels’. Clicking on it will open a small pop-up, which allows you to select RTIO channels from those currently available in your system.

Note

Multiple channels can be selected and added simultaneously. The button with a folder icon allows opening independent pop-up MonInj docks, into which channels can also be added. Configurations of docks and channels will be remembered between dashboard restarts.

Warning

Not all ARTIQ/Sinara real-time peripherals support both monitoring *and* injection, and some do not yet support either. Which peripherals belong to which categories has varied somewhat over the history of ARTIQ versions. Depending on the complexity of the peripheral, incorporating monitor or injection support represents a nontrivial engineering effort, which has generally only been undertaken when commissioned by particular research groups or users. The pop-up menu will display only channels that are valid targets for one or the other functionality.

For DDS/Urukul in particular, injection is supported by a slightly different implementation, which involves automatic submission of a miniature kernel which will override and terminate any other experiments currently executing. Accordingly, Urukul injection should be used carefully.

MonInj can always be tested using the user LEDs, which you can find the folder `ttl` in the pop-up menu. Channels are listed according to the types and names given in `device_db.py`. Add your LED channels to the main dock; their monitored values will be displayed automatically. Try running any experiment that has an effect on LED state to see the monitored values change.

Mouse over one of the LED channel fields to see the two buttons `OVR`, for override, and `LVL`, for level. Clicking ‘Override’ will cause MonInj to take direct control of the channel, overriding any experiments that may be running. Once the channel is overridden, its level can be changed directly from the dashboard, by clicking ‘Level’ to flip it back and forth.

10.3.1 Command-line monitor

For those peripherals which support monitoring, the command-line `artiq_rtiomon` utility can be used to see monitor output directly in the terminal. The command-line monitor does not require or interact with the management system or even the device database. Instead, it takes the core device IP address and a channel number as parameters and communicates with the core device directly.

Tip

To remember which channel numbers were assigned to which peripherals, check your device database, specifically the `channel` field in local entries.

10.4 Waveform

The RTIO analyzer was briefly presented in *RTIO analyzer*. Like MonInj, it is directly accessible to the dashboard through its own proxy controller, *aqctl_coreanalyzer_proxy*. To see it in action with the management system, navigate to the ‘Waveform’ tab of the dashboard. The dock should display several buttons and a currently empty list of waveforms, distinguishable only by the timeline along the top of the field. Use the ‘Add channels’ button, similar to that used by MonInj, to add waveforms to the list, for example `rtio_slack` and the `led0` user LED.

The circular arrow ‘Fetch analyzer data’ button has the same basic effect as using the command-line *artiq_coreanalyzer*: it extracts the full contents of the circular analyzer buffer. In order to start from a clean slate, click the fetch button a few times, until the analyzer dump is empty aside from stop message warning appears. Try running a simple experiment, for example this one, which underflows:

```
from artiq.experiment import *

class BlinkToUnderflow(EnvExperiment):
    def build(self):
        self.setattr_device("core")
        self.setattr_device("led0")

    @kernel
    def run(self):
        self.core.reset()
        for i in range(1000):
            self.led0.pulse(.2*us)
            delay(.2*us)
```

Now fetch the analyzer data again (only once)! Visible waveforms should appear in their respective fields. If nothing is visible to you, the timescale is likely zoomed too far out; adjust by zooming with CTRL+scroll and moving along the timeline by dragging it with your mouse. On a clean slate, `BlinkToUnderflow` should represent the first RTIO events on the record, and the waveforms accordingly will be displayed at the very beginning of the timeline.

Eventually, you should be able to see the up-and-down ‘square wave’ pattern of the blinking LED, coupled with a steadily descending line in the RTIO slack, representing the progressive wearing away of the slack gained using `self.core.reset()`. This kind of analysis can be especially useful in diagnosing underflows; with some practice, the waveform can be used to ascertain which parts of an experiment are consuming the greatest amounts of slack, thereby causing underflows down the line.

Tip

File options in the top left allow for saving and exporting RTIO traces and channel lists (including to VCD), as well as opening them from saved files.

10.4.1 RTIO logging

It is possible to dump any Python object so that it appears alongside the waveforms, using the built-in `rtio_log()` function, which accepts a log name as its first parameter and an arbitrary number of objects along with it. Try adding it to the `BlinkToUnderflow` experiment:

```
@kernel
def run(self):
    self.core.reset()
    for i in range(1000):
        self.led0.pulse(.2*us)
```

(continues on next page)

(continued from previous page)

```
rtio_log("test_trace", "i", i)
delay(.2*us)
```

Run this edited experiment. Fetch the analyzer data. Open the ‘Add channels’ pop-up again; `test_trace` should appear as an option now that the experiment has been run. Observe that every `i` is printed as a single-point event in a new waveform timeline.

10.5 Shortcuts

The last notable tab of the dashboard is called ‘Shortcuts’. To demonstrate its use, navigate to the ‘Explorer’ tab, left-click on an experiment, and select ‘Set shortcut’. Binding an experiment to one of the available keys will cause it to be automatically submitted any time the key is pressed. The ‘Shortcuts’ tab simply displays the current set of bound experiments, and provides controls for opening a submission window or deleting the shortcut.

Note

Experiments submitted by shortcut will always use the argument currently entered into the submission window, if one is open. If no window is currently open, it will simply use the value *last* entered into a submission window. This is true even if that value was never used to submit an experiment.

It is also possible to bring up a “Quick Open” dialogue where experiments can be called up by searching part of their name. This is bound to the hotkey `CTRL+P`. To immediately start the experiment with its default arguments, hit `CTRL+ENTER`.

USING DRTIO AND SUBKERNELS

In larger or more spread-out systems, a single core device might not be suited to managing all the RTIO operations or channels necessary. For these situations ARTIQ supplies Distributed Real-Time IO, or DRTIO. This allows systems to be configured with some or all of their RTIO channels distributed to one or several *satellite* core devices, which are linked to the *master* core device. These remote channels are then accessible in kernels on the master device exactly like local channels.

While the components of a system, as well as the distribution of peripherals among satellites, are necessarily fixed in the system configuration, the specific topology of master and satellite links is flexible and can be changed whenever necessary. It is supplied to the core device by means of a routing table (see below). Kasli and Kasli-SoC devices use SFP ports for DRTIO connections. Links should be high-speed duplex serial lines operating 1Gbps or more.

Certain peripheral cards with onboard FPGAs of their own (e.g. Shuttler) can be configured as satellites in a DRTIO setting, allowing them to run their own subkernels and make use of DDMA. In these cases, the EEM connection to the core device is used for DRTIO communication (DRTIO-over-EEM).

Note

As with other configuration changes (e.g. adding new hardware), if you are in possession of a non-distributed ARTIQ system and you'd like to expand it into a DRTIO setup, it's easily possible to do so, but you need to be sure that both master and satellite are (re)flashed with this in mind. As usual, if you obtained your hardware from M-Labs, you will normally be supplied with all the binaries you need, through `afws_client` or otherwise.

Warning

Do not confuse the DRTIO *master device* (used to mean the central controlling core device of a distributed system) with the *ARTIQ master* (the central piece of software of ARTIQ's management system, which interacts with `artiq_client` and the dashboard.) `artiq_run` can be used to run experiments on DRTIO systems just as easily as non-distributed ones, and the ARTIQ master interacts with the central core device regardless of whether it's configured as a DRTIO master or standalone.

11.1 Using DRTIO

11.1.1 Configuring the routing table

By default, DRTIO assumes a routing table for a star topology (i.e. all satellites directly connected to the master), with destination 0 being the master device's local RTIO core and destinations 1 and above corresponding to devices on the master's respective downstream ports. To use any other topology, it is necessary to supply a corresponding routing table in the form of a binary file, written to flash storage under the key `routing_table`. The binary file is easily generated in the correct format using `artiq_route`. This example is for a chain of 3 devices:

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

# set destination 0 to the master's 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
```

Destination numbers must correspond to the ones used in the *device database*, listed in the `satellite_cpu_targets` field. If unsure which destination number corresponds to which physical satellite device, check the channel numbers of the peripherals associated with that device; in DRTIO systems bits 16-24 of the RTIO channel number correspond to the destination number of the core device they are bound to. See also the *DRTIO system* page.

All routes must end with the local RTIO core of the destination device. Incorrect routing tables will cause `RTIODestinationUnreachable` exceptions. The local RTIO core of the master device is considered a destination like any other; it must be explicitly listed in the routing table to be accessible to kernels.

As with other configuration changes, the core device should be restarted (`artiq_coremgmt reboot`, power cycle, etc.) for changes to take effect.

11.1.2 Using the core language with DRTIO

Remote channels are accessed just as local channels are (e.g., most commonly, by calling `self.setattr_device()` and then referencing the device by name.)

11.1.3 Link establishment

After devices have booted, it takes several seconds for all links in a DRTIO system to become established. Kernels should not attempt to access destinations until all required links are up (trying to do so will raise `RTIODestinationUnreachable` exceptions). ARTIQ provides the method `get_rtio_destination_status()` which determines whether a destination can be reached. We recommend calling it in a loop in your startup kernel for each important destination in order to delay startup until they all can be reached.

11.1.4 Latency

Each hop (link traversed) increases the RTIO latency of a destination by a significant amount; however, this latency is 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. In some situations, the use of subkernels (see below) may also bypass potential latency issues.

11.2 Distributed Direct Memory Access (DDMA)

By default on DRTIO systems, all events recorded by the master's DMA core are kept and played back on the master. With distributed DMA, RTIO events that should be played back on remote destinations are distributed to the corresponding satellites. In some cases (typically, large buffers on several satellites with high event throughput), it allows for better performance and higher bandwidth, as the RTIO events do not have to be sent over the DRTIO link(s) during playback.

To enable distributed DMA for the master, simply provide an `enable_ddma=True` argument for the `record()` method - taking a snippet from the non-distributed example in the *core language tutorial*:

```
@kernel
def record(self):
    with self.core_dma.record("pulses", enable_ddma=True):
        # 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)
```

In standalone systems, as well as in subkernels (see below), this argument is ignored; in standalone systems it is meaningless and in subkernels it must always be enabled for structural reasons.

Enabling DDMA on a purely local sequence on a DRTIO system introduces an overhead during trace recording which comes from additional processing done on the record, so careful use is advised. Due to the extra time that communicating with relevant satellites takes, an additional delay before playback may be necessary to prevent a *RTIOUnderflow* when playing back a DDMA-enabled sequence.

11.3 Subkernels

Rather than only offloading the RTIO channels to satellites and limiting all processing to the master core device, it is also possible to run kernels directly on satellite devices. These are referred to as *subkernels*. Using subkernels to process and control remote RTIO channels can free up resources on the core device.

Subkernels behave for the most part like regular kernels; they accept arguments, can return values, and are marked by the decorator `@subkernel(destination=i)`, where `i` is the satellite's destination number as used in the routing table. To call a subkernel, call it like any other function. There are however a few caveats:

- subkernels do not support RPCs,
- subkernels do not support (recursive) DRTIO (but they can call other subkernels and send messages to each other, see below),
- they support DMA, for which DDMA is considered always enabled,
- they can raise exceptions, which they may catch locally or propagate to the calling kernel,
- their return values must be fully annotated with an ARTIQ type,
- their arguments should be annotated, and only basic ARTIQ types are supported,
- while `self` is allowed as an argument, it is retrieved at compile time and exists as a purely local object afterwards. Any changes made by other kernels will not be visible, and changes made locally will not be applied anywhere else.

11.3.1 Subkernels in practice

Subkernels begin execution as soon as possible when called. By default, they are not awaited, but awaiting is necessary to receive results or exceptions. The await function `subkernel_await(function, [timeout])` takes as argument the subkernel to be awaited and, optionally, a timeout value in milliseconds. If the timeout is reached without response from the subkernel, a `SubkernelError` is raised. If no timeout value is supplied the function waits indefinitely for the return. Negative timeout values are ignored.

For example, a subkernel performing integer addition:

```
from artiq.experiment import *

@subkernel(destination=1)
def subkernel_add(a: TInt32, b: TInt32) -> TInt32:
    return a + b

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

    @kernel
    def run(self):
        subkernel_add(2, 2)
        result = subkernel_await(subkernel_add)
        assert result == 4
```

Subkernels are compiled after the main kernel and immediately sent to the designated satellite. When they are called, the master simply instructs the subkernel to load and run the corresponding kernel. When `self` is used in subkernels, it is embedded into the compiled and uploaded data; this is the reason why changes made do not propagate between kernels.

If a subkernel is called on a satellite where a kernel is already running, the newer kernel overrides silently, and the previous kernel will not be completed.

Warning

Be careful with use of `self.core.reset()` around subkernels. Since `self` in subkernels is purely local, calling `self.core.reset()` in a subkernel will only affect that specific satellite and its own FIFOs. On the other hand, calling `self.core.reset()` in the master kernel will clear FIFOs in all satellites, regardless of whether a subkernel is running, but will not stop the subkernel. As a result, any event currently in a FIFO queue will be cleared, but the subkernels may continue to queue events. This is likely to result in odd behavior; it's best to avoid using `self.core.reset()` during the lifetime of any subkernels.

If a subkernel is complex and its binary relatively large, the delay between the call and actually running the subkernel may be substantial. If it's necessary to minimize this delay, `subkernel_preload(function)` should be used before the call.

Subkernels receive the value of the timeline cursor `now_mu` from the caller at the moment of the call. As there is a delay between calling the subkernel and its actual start, there will be a difference in `now_mu` that can be compensated with a delay in the subkernel. Additionally, preloading the subkernel would decrease the difference, as the subkernel does not have to be loaded before running.

While a subkernel is running, the satellite is disconnected from the RTIO interface of the master. As a result, regardless of what devices the subkernel itself uses, none of the RTIO devices on that satellite will be available to the master, nor will messages be passed on to any further satellites downstream. This applies both to regular RTIO operations and

DDMA. While a subkernel is running, a satellite may use its own local DMA, but an attempt by any other device to run DDMA through the satellite will fail. Control is returned to the master when no subkernel is running – to be sure that a device will be accessible, await before performing any RTIO operations on the affected satellite.

Note

Subkernels do not exit automatically if a master kernel exits, and are seamlessly carried over between experiments. Much like RTIO events left in FIFO queues, the nature of seamless transition means subkernels left running after the end of an experiment cannot be guaranteed to complete (as they may be overridden by newer subkernels in the next experiment). Following experiments must also be aware of the risk of attempting to reach RTIO devices currently ‘blocked’ by an active subkernel left over from a previous experiment. This can be avoided simply by having each experiment await all of its subkernels at some point before exiting. Alternatively, if necessary, a system can be sanitized by calling trivial kernels in each satellite – any leftover subkernels will be overridden and automatically cancelled.

11.3.2 Calling other kernels

Subkernels can call other kernels and subkernels. For a more complex example:

```
from artiq.experiment import *

class SubkernelExperiment(EnvExperiment):
    def build(self):
        self.setattr_device("core")
        self.setattr_device("ttl0")
        self.setattr_device("ttl8") # assuming it's on satellite

    @subkernel(destination=1)
    def add_and_pulse(self, a: TInt32, b: TInt32) -> TInt32:
        c = a + b
        self.pulse_ttl(c)
        return c

    @subkernel(destination=1)
    def pulse_ttl(self, delay: TInt32) -> TNone:
        self.ttl8.pulse(delay*us)

    @kernel
    def run(self):
        subkernel_preload(self.add_and_pulse)
        self.core.reset()
        delay(10*ms)
        self.add_and_pulse(2, 2)
        self.ttl0.pulse(15*us)
        result = subkernel_await(self.add_and_pulse)
        assert result == 4
        self.pulse_ttl(20)
```

In this case, without the preload, the delay after the core reset would need to be longer. Depending on the connection, the call may still take some time in itself. Notice that the method `pulse_ttl()` can be called both within a subkernel and on its own.

Note

Subkernels can call subkernels on any other satellite, not only their own. Care should however be taken that different kernels do not call subkernels on the same satellite, or only very cautiously. If, e.g., a newer call overrides a subkernel that another caller is awaiting, unpredictable timeouts or locks may result, as the original subkernel will never return. There is no mechanism to check whether a particular satellite is ‘busy’; it is up to the programmer to handle this correctly.

11.3.3 Message passing

Apart from arguments and returns, subkernels can also pass messages between each other or the master with built-in `subkernel_send()` and `subkernel_recv()` functions. This can be used for communication between subkernels, to pass additional data, or to send partially computed data. Consider the following example:

```
from artiq.experiment import *

@subkernel(destination=1)
def simple_message() -> TInt32:
    data = subkernel_recv("message", TInt32)
    return data + 20

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

    @kernel
    def run(self):
        simple_self()
        subkernel_send(1, "message", 150)
        result = subkernel_await(simple_self)
        assert result == 170
```

The `subkernel_send(destination, name, value)` function requires three arguments: a destination, a name for the message (to be used for identification in the corresponding `subkernel_recv()`), and the passed value.

The `subkernel_recv(name, type, [timeout])` function requires two arguments: message name (matching exactly the name provided in `subkernel_send`) and expected type. Optionally, it also accepts a third argument, a timeout for the operation in milliseconds. As with `subkernel_await`, the default behavior is to wait as long as necessary, and a negative argument is ignored.

A message can only be received while a subkernel is running, and is placed into a buffer to be retrieved when required. As a result `send` executes independently of any receive and never deadlocks. However, a receive function may timeout or lock (wait forever) if no message with the correct name and destination is ever sent.

ENVIRONMENT

ARTIQ experiments exist in an environment, which consists of devices, arguments, and datasets. Access to the environment is handled through the *HasEnvironment* manager provided by the *EnvExperiment* class that experiments should derive from.

12.1 The device database

Information about available devices is provided to ARTIQ through a file called the device database, typically called `device_db.py`, which many of the ARTIQ front-end tools require access to in order to run. The device database specifies:

- what devices are available to an ARTIQ installation
- what drivers to use
- what controllers to use
- how and where to contact each device, i.e.
 - at which RTIO channel(s) each local device can be reached
 - at which network address each controller can be reached

as well as, if present, how and where to contact the core device itself (e.g., its IP address, often by a field named `core_addr`).

This is stored in a Python dictionary whose keys are the device names, which the file must define as a global variable with the name `device_db`. Examples for various system configurations can be found inside the subfolders of `artiq/examples`. A typical device database entry looks like this:

```
"led": {
    "type": "local",
    "module": "artiq.coredevice.ttl",
    "class": "TTLOut",
    "arguments": {"channel": 19}
},
```

Note that the key (the name of the device) is `led` and the value is itself a Python dictionary. Names will later be used to gain access to a device through methods such as `self.setattr_device("led")`. While in this case `led` can be replaced with another name, provided it is used consistently, some names (in particular, `core`) are used internally by ARTIQ and will cause problems if changed. It is often more convenient to use aliases for renaming purposes, see below.

Note

The device database is generated and stored in the memory of the master when the master is first started. Changes to the `device_db.py` file will not immediately affect a running master. In order to update the device database, right-click in the Explorer window and select ‘Scan device database’, or run the command `artiq_client scan-devices`.

Warning

It is important to understand that the device database does not *set* your system configuration, only *describe* it. If you change the devices available to your system, it is usually necessary to edit the device database, but editing the database will not change what devices are available to your system.

Remote (normally, non-realtime) devices must have accessible, suitable controllers and drivers; see *Developing a Network Device Support Package (NDSP)* for more information, including how to add entries for new remote devices to your device database. Local devices (normally, real-time, e.g. your Sinar hardware) must be connected to your system, and more importantly, your gateway and firmware must have been compiled to account for them, and to expect them at those ports.

While controllers can be added and removed to your device database on an *ad hoc* basis, in order to make new real-time hardware accessible, it is generally also necessary to recompile and reflash your gateway and firmware. (If you purchase your hardware from M-Labs, you will be provided with new binaries and necessary assistance.) See *Building and developing ARTIQ*.

Adding or removing new real-time hardware is a difference in *system configuration*, which must be specified at compilation time of gateway and firmware. For Kasli and Kasli-SoC, this is managed in the form of a JSON usually called the *system description file*. The device database generally provides that information to ARTIQ which can change from instance to instance ARTIQ is run, e.g., device names and aliases, network addresses, clock frequencies, and so on. The system configuration defines that information which is *not* permitted to change, e.g., what device is associated with which EEM port or RTIO channels. Insofar as data is duplicated between the two, the device database is obliged to agree with the system description, not the other way around.

If you obtain your hardware from M-Labs, you will always be provided with a `device_db.py` to match your system configuration, which you can edit as necessary to add controllers, aliases, and so on. In the relatively unlikely case that you are writing a device database from scratch, the `artiq_ddb_template` utility can be used to generate a template device database directly from the JSON system description used to compile your gateway and firmware. This is the easiest way to ensure that details such as the allocation of RTIO channel numbers will be represented in the device database correctly. See also the corresponding entry in *Utilities*.

12.1.1 Local devices

Local device entries are dictionaries which contain a `type` field set to `local`. They correspond to device drivers that are created locally on the master as opposed to using the controller mechanism; this is normally the real-time hardware of the system, including the core, which is itself considered a local device. The `led` example above is a local device entry.

The fields `module` and `class` determine the location of the Python class of the driver. The `arguments` field is another (possibly empty) dictionary that contains arguments to pass to the device driver constructor. `arguments` is often used to specify the RTIO channel number of a peripheral, which must match the channel number in gateway.

On Kasli and Kasli-SoC, the allocation of RTIO channels to EEM ports is done automatically when the gateway is compiled, and while conceptually simple (channels are assigned one after the other, from zero upwards, for each device entry in the system description file) it is not entirely straightforward (different devices require different numbers of RTIO channels). Again, the easiest way to handle this when writing a new device database is automatically, using `artiq_ddb_template`.

12.1.2 Controllers

Controller entries are dictionaries which contain a `type` field 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. For an example, see [the *NDSP development page*](#).

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 (matching the `port` field) and an appropriate bind address for the controller's listening socket.

An optional `best_effort` boolean field determines whether to use `sipyco.pc_rpc.Client` or `sipyco.pc_rpc.BestEffortClient`. `BestEffortClient` is very similar to `Client`, but suppresses network errors and automatically retries connections in the background. If no `best_effort` field is present, `Client` is used by default.

12.1.3 Aliases

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

12.2 Arguments

Arguments are values that parameterize the behavior of an experiment. ARTIQ supports both interactive arguments, requested and supplied at some point while an experiment is running, and submission-time arguments, requested in the build phase and set before the experiment is executed. For more on arguments in practice, see the tutorial section [Adding arguments](#). For supported argument types, see the reference for [artiq.language.environment](#); for specific methods, see the reference for [HasEnvironment](#).

12.3 Datasets

Datasets are values that are read and written by experiments kept in a key-value store. They exist to facilitate the exchange and preservation of information between experiments, from experiments to the management system, and from experiments to long-term storage. Datasets may be either scalars (`bool`, `int`, `float`, or NumPy scalar) or NumPy arrays. For basic use of datasets, see the [data interfaces tutorial](#).

A dataset may be broadcast (`broadcast=True`), that is, distributed to all clients connected to the master. This is useful e.g. for the ARTIQ dashboard to plot results while an experiment is in progress and give rapid feedback to the user. Broadcasted datasets live in a global key-value store owned by the master. Care should be taken that experiments use distinctive real-time result names in order to avoid conflicts. Broadcasted datasets may be used to communicate values across experiments; for instance, a periodic calibration experiment might update a dataset read by payload experiments.

Broadcasted datasets are replaced when a new dataset with the same key (name) is produced. By default, they are erased when the master halts. Broadcasted datasets may be made persistent (`persistent=True`, which also implies `broadcast=True`), in which case the master stores them in a LMDB database typically called `dataset_db.mdb`, where they are saved across master restarts.

By default, datasets are archived in the `results` HDF5 output for that run, although this can be opted against (`archive=False`). They can be viewed and analyzed with the ARTIQ browser, or with an HDF5 viewer of your choice.

12.3.1 Datasets and units

Datasets accept metadata for numerical formatting with the `unit`, `scale` and `precision` parameters of `set_dataset`.

Note

In experiment code, values are assumed to be in the SI base unit. Setting a dataset with a value of 1000 and the unit kV represents the quantity 1 kV. It is recommended to use the globals defined by `artiq.language.units` and write `1*kV` instead of 1000 for the value.

In dashboards and clients these globals are not available. However, setting a dataset with a value of 1 and the unit kV simply represents the quantity 1 kV.

`precision` refers to the max number of decimal places to display. This parameter does not affect the underlying value, and is only used for display purposes.

COMPILER

The ARTIQ compiler transforms the Python code of the kernels into machine code executable on the core device. For limited purposes (normally, obtaining executable binaries of idle and startup kernels), it can be accessed through `artiq_compile`. Otherwise it is invoked automatically whenever a function with an applicable decorator is called.

ARTIQ kernel code accepts *nearly*, but not quite, a strict subset of Python 3. The necessities of real-time operation impose a harsher set of limitations; as a result, many Python features are necessarily omitted, and there are some specific discrepancies (see also *Pitfalls*).

In general, ARTIQ Python supports only statically typed variables; it implements no heap allocation or garbage collection systems, essentially disallowing any heap-based data structures (although lists and arrays remain available in a stack-based form); and it cannot use runtime dispatch, meaning that, for example, all elements of an array must be of the same type. Nonetheless, technical details aside, a basic knowledge of Python is entirely sufficient to write ARTIQ experiments.

Note

The ARTIQ compiler is now in its second iteration. The third generation, known as NAC3, is *currently in development*, and available for pre-alpha experimental use. NAC3 represents a major overhaul of ARTIQ compilation, and will feature much faster compilation speeds, a greatly improved type system, and more predictable and transparent operation. It is compatible with ARTIQ firmware starting at ARTIQ-7. Instructions for installation and basic usage differences can also be found *on the M-Labs Forum*. While NAC3 is a work in progress and many important features remain unimplemented, installation and feedback is welcomed.

13.1 ARTIQ Python code

A variety of short experiments can be found in the subfolders of `artiq/examples`, especially under `kc705_nist_clock/repository` and `no_hardware/repository`. Reading through these will give you a general idea of what ARTIQ Python is capable of and how to use it.

13.1.1 Functions and decorators

The ARTIQ compiler recognizes several specialized decorators, which determine the way the decorated function will be compiled and handled.

`@kernel` (see `kernel()`) designates kernel functions, which will be compiled for and executed on the core device; the basic setup and background for kernels is detailed on the *Getting started with the core device* page. `@subkernel` (`subkernel()`) designates subkernel functions, which are largely similar to kernels except that they are executed on satellite devices in a DRTIO setting, with some associated limitations; they are described in more detail on the *Using DRTIO and subkernels* page.

`@rpc` (`rpc()`) designates functions to be executed on the host machine, which are compiled and run in regular Python, outside of the core device's real-time limitations. Notably, functions without decorators are assumed to be host-bound by default, and treated identically to an explicitly marked `@rpc`. As a result, the explicit decorator is only really necessary when specifying additional flags (for example, `flags={"async"}`, see below).

`@portable` (`portable()`) designates functions to be executed *on the same device they are called*. In other words, when called from a kernel, a portable is executed as a kernel; when called from a subkernel, it is executed as a kernel, on the same satellite device as the calling subkernel; when called from a host function, it is executed on the host machine.

`@host_only` (`host_only()`) functions are executed fully on the host, similarly to `@rpc`, but calling them from a kernel as an RPC will be refused by the compiler. It can be used to mark functions which should only ever be called by the host.

Warning

ARTIQ goes to some lengths to cache code used in experiments correctly, so that experiments run according to the state of the code when they were started, even if the source is changed during the run time. Python itself annoyingly fails to implement this (see also [issue #401](#)), necessitating a workaround on ARTIQ's part. One particular downstream limitation is that the ARTIQ compiler is unable to recognize decorators with path prefixes, i.e.:

```
import artiq.experiment as aq

[...]

@aq.kernel
def run(self):
    pass
```

will fail to compile. As long as `from artiq.experiment import *` is used as in the examples, this is never an issue. If prefixes are strongly preferred, a possible workaround is to import decorators separately, as e.g. `from artiq.language.core import kernel`.

13.1.2 ARTIQ types

Python/NumPy types correspond to ARTIQ types as follows:

Python	ARTIQ
<code>NoneType</code>	<code>TNone</code>
<code>bool</code>	<code>TBool</code>
<code>int</code>	<code>TInt32</code> or <code>TInt64</code>
<code>float</code>	<code>TFloat</code>
<code>str</code>	<code>TStr</code>
<code>bytes</code>	<code>TBytes</code>
<code>bytearray</code>	<code>TByteArray</code>
list of <code>T</code>	<code>TList(T)</code>
NumPy array	<code>TArray(T, num_dims)</code>
tuple of <code>(T1, T2, ...)</code>	<code>TTuple([T1, T2, ...])</code>
<code>range</code>	<code>TRange32</code> , <code>TRange64</code>
<code>numpy.int32</code>	<code>TInt32</code>
<code>numpy.int64</code>	<code>TInt64</code>
<code>numpy.float64</code>	<code>TFloat</code>

Integers are 32-bit by default but may be converted to 64-bit with `numpy.int64`.

The ARTIQ compiler can be thought of as overriding all built-in Python types, and types in kernel code cannot always be assumed to behave as they would in host Python. In particular, normally heap-allocated types such as arrays, lists, and strings are very limited in what they support. Strings must be constant and lists and arrays must be of constant size. Methods like `append`, `push`, and `pop` are unavailable as a matter of principle, and will not compile. Certain types, notably dictionaries, have no ARTIQ implementation and cannot be used in kernels at all.

Tip

Instead of pushing or appending, preallocate for the maximum number of elements you expect with a list comprehension, i.e. `x = [0 for _ in range(1024)]`, and then keep a variable `n` noting the last filled element of the array. Afterwards, `x[0:n]` will give you a list with that number of elements.

Multidimensional arrays are allowed (using NumPy syntax). Element-wise operations (e.g. `+`, `/`), matrix multiplication (`@`) and multidimensional indexing are supported; slices and views (currently) are not.

Tuples may contain a mixture of different types. They cannot be iterated over or dynamically indexed, although they may be indexed by constants and multiple assignment is supported.

User-defined classes are supported, provided their attributes are of other supported types (attributes that are not used in the kernel are ignored and thus unrestricted). 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.

13.1.3 Basic ARTIQ Python

Basic Python features can broadly be used inside kernels without further compunctions. This includes loops (`for` / `while` / `break` / `continue`), conditionals (`if` / `else` / `elif`), functions, exceptions, `try` / `except` / `else` blocks, and statically typed variables of any supported types.

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
```

Tip

Multiple variables of different types can be sent in one RPC call by returning a tuple, e.g.

```
def return_many() -> TTuple([TInt32, TFloat, TStr]):
    return (4, 12.34, "hello",)
```

Which can be retrieved from a kernel with `(a, b, c) = return_many()`

Kernels can freely modify attributes of objects shared with the host. However, by necessity, these modifications are actually applied to local copies of the objects, as the latency of immediate writeback would be unworkable in a real-time environment. Instead, modifications are written back *when the kernel completes*; notably, this means RPCs called by a kernel itself will only have access to the unmodified host version of the object, as the kernel hasn't finished execution yet. In some cases, accessing data on the host is better handled by calling RPCs specifically to make the desired modifications.

Warning

Kernels *cannot and should not* return lists, arrays, or strings they have created, or any objects containing them; in the absence of a heap, the way these values are allocated means they cannot outlive the kernels they are created in. Trying to do so will normally be discovered by lifetime tracking and result in compilation errors, but in certain cases lifetime tracking will fail to detect a problem and experiments will encounter memory corruption at runtime. For example:

```
def func(a):
    return a

class ProblemReturn1(EnvExperiment):
    def build(self):
        self.setattr_device("core")

    @kernel
    def run(self):
        # results in memory corruption
        return func([1, 2, 3])
```

will compile, **but corrupts at runtime**. On the other hand, lists, arrays, or strings can and should be used as inputs for RPCs, and this is the preferred method of returning data to the host. In this way the data is sent before the kernel completes and there are no allocation issues.

13.1.4 Available built-in functions

ARTIQ makes various useful built-in and mathematical functions from Python, NumPy, and SciPy available in kernel code. They are not guaranteed to be perfectly equivalent to their host namesakes (for example, `numpy rint()` normally rounds-to-even, but in kernel code rounds toward zero) but their behavior should be basically predictable.

Reference	Functions
Python built-ins	<ul style="list-style-type: none"> • <code>len()</code>, <code>round()</code>, <code>abs()</code>, <code>min()</code>, <code>max()</code> • <code>print()</code> (with caveats; see below) • all basic type conversions (<code>int()</code>, <code>float()</code> etc.)
NumPy mathematic utilities	<ul style="list-style-type: none"> • <code>sqrt()</code>, <code>cbrt()</code> • <code>fabs()</code>, <code>fmax()</code>, <code>fmin()</code> • <code>floor()</code>, <code>ceil()</code>, <code>trunc()</code>, <code>rint()</code>
NumPy exponents and logarithms	<ul style="list-style-type: none"> • <code>exp()</code>, <code>exp2()</code>, <code>expm1()</code> • <code>log()</code>, <code>log2()</code>, <code>log10()</code>
NumPy trigonometric and hyperbolic functions	<ul style="list-style-type: none"> • <code>sin()</code>, <code>cos()</code>, <code>tan()</code>, • <code>arcsin()</code>, <code>arccos()</code>, <code>arctan()</code> • <code>sinh()</code>, <code>cosh()</code>, <code>tanh()</code> • <code>arcsinh()</code>, <code>arccosh()</code>, <code>arctanh()</code> • <code>hypot()</code>, <code>arctan2()</code>
NumPy floating point routines	<ul style="list-style-type: none"> • <code>copysign()</code>, <code>nextafter()</code>
SciPy special functions	<ul style="list-style-type: none"> • <code>erf()</code>, <code>erfc()</code> • <code>gamma()</code>, <code>gammaLn()</code> • <code>j0()</code>, <code>j1()</code>, <code>y0()</code>, <code>y1()</code>

Basic NumPy array handling (`np.array()`, `numpy.transpose()`, `numpy.full()`, `@`, element-wise operation, etc.) is also available. NumPy functions are implicitly broadcast when applied to arrays.

13.1.5 Print and logging functions

ARTIQ offers two native built-in logging functions: `rtio_log()`, which prints to the *RTIO log*, as retrieved by `artiq_coreanalyzer`, and `core_log()`, which prints directly to the core log, regardless of context or network connection status. Both exist for debugging purposes, especially in contexts where a `print()` RPC is not suitable, such as in idle/startup kernels or when debugging delicate RTIO slack issues which may be significantly affected by the overhead of `print()`.

`print()` itself is in practice an RPC to the regular host Python `print()`, i.e. with output either in the terminal of `artiq_run` or in the client logs when using `artiq_dashboard` or `artiq_compile`. This means on one hand that it should not be used in idle, startup, or subkernels, and on the other hand that it suffers of some of the timing limitations of any other RPC, especially if the RPC queue is full. Accordingly, it is important to be aware that the timing of `print()` outputs can't reliably be used to debug timing in kernels, and especially not the timing of other RPCs.

13.2 Pitfalls

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

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 can't fit into

32 bits, the entire expression will be upgraded to 64-bit arithmetic, but if all constants are small, 32-bit arithmetic is 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.

13.3 Flags and 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.

13.3.1 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)
```

13.3.2 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 are true:

- Arguments and results will not be Not-a-Number or infinite;
- 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.

13.3.3 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):
        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 see through the `self.worker.work()` function call, and thus can 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()
```


MANAGEMENT SYSTEM

Note

The ARTIQ management system as described here is optional. Experiments can be run one-by-one using *artiq_run*, and controllers can be run 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. For an introduction to the system and how to use it, see *Using the management system*.

14.1 Components

See also overview for a visual idea of the management system.

14.1.1 Master

The *ARTIQ master* is responsible for managing the dataset and device databases, the experiment repository, scheduling and running experiments, archiving results, and distributing real-time results. It is a headless component, and one or several clients (command-line or GUI) use the network to interact with it.

The master expects to be given a directory on startup, the experiment repository, containing these experiments which are automatically tracked and communicated to clients. By default, it simply looks for a directory called `repository`. The `-r` flag can be used to substitute an alternate location. Subdirectories in `repository` are also read, and experiments stored in them are known to the master. They will be displayed as folders in the dashboard's explorer.

It also expects access to a `device_db.py`, with a corresponding flag `--device-db` to substitute a different file name. Additionally, it will reference or create certain files in the directory it is run in, among them `dataset_db.mdb`, the LMDB database containing persistent datasets, `last_rid.pyon`, which simply holds the last used RID, and the `results` directory. For more on the device and dataset databases, see also *Environment*.

Note

Because the other parts of the management system often display knowledge of the information stored in these files, confusion can sometimes result about where it is really stored and how it is distributed. Device databases, datasets, results, and experiments are all solely kept and administered by the master, which communicates with dashboards, clients, and controller managers over the network whenever necessary.

Notably, clients and dashboards normally do not *send in* experiments to the master. Rather, they make requests from the list of experiments the master already knows about, primarily those in `repository`, but also in the master's local file system. This is true even if `repository` is configured as a Git repository and cloned onto other machines.

The only exception is the command line client's `--content` flag, which allows submission by content, i.e. sending in experiment files which may be otherwise unknown to the master. This feature however has some important

limitations; see below in *Submission from the raw filesystem*.

The ARTIQ master should not be confused with the ‘master’ device in a DRTIO system, which is only a designation for the particular core device acting as central node in a distributed configuration of ARTIQ. The two concepts are otherwise unrelated.

14.1.2 Clients

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

The *dashboard* connects to the master and is the main method of interacting with it. The main roles 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.

The dashboard remembers and restores GUI state (window/dock positions, last values entered by the user, etc.) in between instances. This information is stored in a file called `artiq_dashboard_{server}_{port}.pyon` in the configuration directory (e.g. generally `~/.config/artiq` for Unix, same as data directory for Windows), distinguished in subfolders by ARTIQ version.

Note

To find where the configuration files are stored on your machine, try the command:

```
python -c "from artiq.tools import get_user_config_dir; print(get_user_config_dir())"
```

14.1.3 Browser

The *browser* is used to read ARTIQ results HDF5 files and run experiment *analyze()* functions, in particular to retrieve previous result databases, process them, and display them in ARTIQ applets. The browser also remembers and restores its GUI state; this is stored in a file called simply `artiq_browser.pyon`, kept in the same configuration directory as the dashboard.

The browser *can* connect to the master, specifically in order to be able to access the master’s store of datasets and to upload new datasets to it, but it does not require such a connection and can also be run completely standalone. However, it requires filesystem access to the `results` files to be of much use.

14.1.4 Controller manager

The controller manager is provided in the `artiq-comtools` package (which is also made available separately from mainline ARTIQ, to allow independent use with minimal dependencies) and started with the `artiq_ctlmgr` command. It is responsible for running and stopping controllers on a machine. One controller manager must be run by each network node that runs controllers.

A controller manager connects to the master and accesses the device database through it to determine what controllers need to be run. The local network address of the connection is used to filter for only those controllers allocated to the current node. Hostname resolution is supported. Changes to the device database are tracked upon rescan and controllers will be stopped and started accordingly.

14.2 Git integration

The master may use a Git repository to store experiment source code. Using Git rather than the bare filesystem has many advantages. For example, each HDF5 result file contains the commit ID corresponding to the exact source code it was produced by, making results more reproduceable. See also *Setting up Git integration*. Generally, it is recommended to use a bare repository (i.e. `git init --bare`), to easily support push transactions from clients, but both bare and non-bare repositories are supported.

Tip

If you are not familiar with Git, you may find the idea of the master reading experiment files from a bare repository confusing. A bare repository does not normally contain copies of the objects it stores; that is to say, you won't be able to find your experiment files listed in it. What it *does* contain is Git's internal data structures, i.e., `hooks`, `objects`, `config`, and so forth. Among other things, this structure also stores, in compressed form, the full contents of every commit made to the repository. It is this compressed data which the master has access to and can read the experiments from. It is not meant to be directly edited, but it is updated every time new commits are received.

It may be useful to note that a normal Git repository, created with `git init`, contains all the same internal data, kept in a hidden directory called `.git` to protect it from accidental modifications. Unlike a bare repository, it *also* normally contains working copies of all the files tracked by Git. When working with a non-bare repository, it is important to understand that the master still takes its image of the available experiments from the internal data, and *not* from the working copies. This is why, even in a non-bare repository, changes are only reflected once they are committed. The working copies are simply ignored.

Other important files – the device database, the dataset database, the `results` directory, and so on – are normally kept outside of the experiment repository, and in this case, they are not stored or handled by Git at all. The master accesses them through the regular filesystem, not through Git, and other ARTIQ components access them through the master. This can be seen visualized in the overview.

With a bare repository, a Git `post-receive` hook can be used to trigger a repository scan every time the repository is pushed to (i.e. updated), as described in the tutorial. This removes the need to trigger repository rescans manually. If you plan to run your ARTIQ system from a single PC, without distributed clients, you may also consider using a non-bare repository and the `post-commit` hook instead. In this workflow, changes can be drafted directly in the master's repository, but the master continues to submit from the last completed commit until a new commit is made (and the repository is rescanned).

Behind the scenes, when scanning the repository, the master 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 use of the Git backend is triggered when the master is started with the `-g` flag. Otherwise the raw filesystem is read and Git-based features will not be available.

14.2.1 Submission from the raw filesystem

By default, the dashboard runs experiments from the repository, that is, the master's temporary checkout folder, whereas the command-line client (`artiq_client submit`) runs experiments from the raw filesystem. This is convenient in order to be able to run working drafts without first committing them.

Be careful with this behavior, however, as it is rather particular. *The raw filesystem* means the immediate local filesystem of the running master. If the client is being run remotely, and you want to submit an experiment from the *client's* local filesystem, e.g. an uncommitted draft in a clone of the experiment repository, use the `--content` flag. If you would like to submit an experiment from the repository, in the same way the dashboard does, use the flag `--repository / -R`.

To be precise:

- `artiq_client submit` should be given a file path that is relative to the location of the master, that is, if the master is run in the directory above its repository, an experiment can be submitted as `repository/experiment_file.py`. Keep in mind that when working with a bare repository, there may be no copies of experiment files in the raw local filesystem. In this case, files can still be made accessible to the master by network filesystem share or some other method for testing.
- `artiq_client submit --repository` should be given a file path relative to the root of the repository, that is, if the experiment is directly within repository, it should be submitted as `experiment_file.py`. Just as in the dashboard, this file is taken from the last completed commit.
- `artiq_client submit --content` should be given a file path that is relative to the location of the client, whether that is local or remote to the master; the contents of the file will be submitted directly to the master to be run. This essentially transfers a raw string, and will not work if the experiment imports or otherwise accesses other files.

Other flags can also be used, such as `--class-name / -c` to select a class name in an experiment which contains several, or `--revision / -r` to use a particular revision. See the reference of `artiq_client` in *Main front-end tools*.

In order to run from the raw filesystem when using the dashboard, right-click in the Explorer window and select the option ‘Open file outside repository’. This will open a file explorer window displaying the master’s local filesystem, which can be used to select and submit experiments outside of the chosen repository directory. There is no GUI support for submission by content. It is recommended to simply use the command-line client for this purpose.

14.3 Experiment scheduling

14.3.1 Basics

To make more efficient use of resources, experiments are generally split into three phases and pipelined. While one experiment has control of the specialized hardware, others may carry out pre-computation or post-analysis in parallel. There are three stages of a standard experiment users may write code for:

1. The **preparation** stage, which pre-fetches and pre-computes any data that is 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.
2. The **run** stage, which corresponds to the body of the experiment. Users *must* implement this stage and overload the `run()` method. In this stage, the experiment has the right to run kernels and access hardware.
3. The **analysis** stage, where raw results collected in the running stage can be post-processed and/or saved. This stage may be implemented by overloading the `analyze()` method. It is not permitted to access hardware in this stage.

See also

These steps are implemented in `artiq.language.environment.Experiment`. User-written experiments should usually derive from (sub-class) `artiq.language.environment.EnvExperiment`, which additionally provides access to the methods of `HasEnvironment`.

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). Preparation and analysis stages are forbidden from accessing hardware so as not to interfere with a potential concurrent run stage. Note that they are not *prevented* from doing so, and it is up to the programmer to respect these guidelines.

Consecutive experiments are automatically pipelined by the ARTIQ master’s scheduler: first experiment A executes its preparation stage, then experiment A executes its running stage while experiment B executes its preparation stage, and

so on.

Note

An experiment A can exit its `run()` method before all its RTIO events have been executed, i.e., while those events are still ‘waiting’ in the RTIO core buffers. If the next experiment entering the running stage uses `reset()`, those buffers will be cleared, and any remaining events discarded, potentially including those scheduled by A.

This is a deliberate feature of seamless handover, but can cause problems if the events scheduled by A were important and should not have been skipped. In those cases, it is recommended to ensure the `run()` method of experiment A does not return until *all* its scheduled events have been executed, or that it is followed only by experiments which do not perform a core reset. See also *RTIO Synchronization*.

14.3.2 Priorities and timed runs

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

1. Experiments may be scheduled with a due date. This is considered the *earliest possible* time of their execution (rather than a deadline, or latest possible – ARTIQ makes no guarantees about experiments being started or completed before any specified time). If a due date is set and it has not yet been reached, the experiment is not eligible for preparation.
2. The integer priority value specified by the user.
3. The due date itself. The earliest (reached) due date will be scheduled first.
4. The run identifier (RID), an integer that is incremented at each experiment submission. This ensures that, all else being equal, experiments are scheduled in the same order as they are submitted.

14.3.3 Multiple pipelines

Experiments must be placed into a pipeline at submission time, set by the “Pipeline” field. The master supports multiple simultaneous pipelines, which will operate in parallel. Pipelines are identified by their names, and are automatically created (when an experiment is scheduled with a pipeline name that does not yet exist) and destroyed (when they run empty). By default, all experiments are submitted into the same pipeline, `main`.

When using multiple pipelines it is the responsibility of the user to ensure that experiments scheduled in parallel will never conflict with those of another pipeline over resources (e.g. attempt to use the same devices simultaneously).

14.3.4 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 experiment has been newly submitted, or reached its due date and become eligible for execution), the higher-priority experiments are executed first, and then `pause()` returns. If there are no such experiments, `pause()` returns immediately. To check whether `pause()` would in fact *not* return immediately, use `check_pause()`.

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

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 any other device, with `get_device()` or `setattr_device()`. See also the detailed reference on the *Management system interface* page.

Note

For maximum compatibility, the scheduler virtual device can also be accessed when running experiments with `artiq_run`. However, since there is no `Scheduler` backend, the methods are replaced by simple dummies, e.g. `check_pause()` simply returns false, and requests are printed into the console. Much the same is true of client control broadcasts (see again *Management system interface*).

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

14.3.5 Scheduler attributes

The scheduler virtual device also exposes information about an experiment's scheduling status through the attributes `rid`, `pipeline_name`, `priority`, and `expid`. This allows e.g. access to an experiment's current RID as `self.scheduler.rid`.

14.4 Internal details

Internally, the ARTIQ management system uses Simple Python Communications, or `SiPyCo`, which was originally written as part of ARTIQ and later split away as a generic communications library. The `SiPyCo` manual is hosted [here](#). The core of the management system is largely contained within `artiq.master`, which contains the `Scheduler`, the various environment and filesystem databases, and the worker processes that execute the experiments themselves.

By default, the master communicates with other processes over four network ports, see *Default network ports*, for logging, broadcasts, notifications, and control. All four of these can be customized by using the `--port` flags, see *the front-end reference*.

- The logging port is occupied by a `sipyco.logging_tools.Server`, and used only by the worker processes to transmit exceptions and other information to the master.
- The broadcast port is occupied by a `sipyco.broadcast.Broadcaster`, which inherits from `sipyco.pc_rpc.AsyncioServer`. Both the dashboard and the client automatically connect to this port, using `sipyco.broadcast.Receiver` to receive logs and CCB messages.
- The notification port is occupied by a `sipyco.sync_struct.Publisher`. The dashboard and client automatically connect to this port, using `sipyco.sync_struct.Subscriber`. Several objects are given to the `Publisher` to monitor, among them the experiment schedule, the device database, the dataset database, and the experiment list. It notifies the subscribers whenever these objects are modified.
- The control port is occupied by a `sipyco.pc_rpc.Server`, which when running can be queried with `sipyco.sipyco_rpctool` like any other source of RPC targets. Multiple concurrent calls to target methods are supported. Through this server, the clients are provided with access to control methods to access the various databases and repositories the master handles, through classes like `artiq.master.databases.DeviceDB`, `artiq.master.databases.DatasetDB`, and `artiq.master.experiments.ExperimentDB`.

The experiment database is supported by `artiq.master.experiments.GitBackend` when Git integration is active, and `artiq.master.experiments.FilesystemBackend` if not.

14.4.1 Experiment workers

The `artiq_run` tool makes use of many of the same databases and handlers as the master (whereas the scheduler and CCB manager are replaced by dummies, as mentioned above), but also directly runs the build, run, and analyze stages of the experiments. On the other hand, within the management system, the master's `Scheduler` spawns a new worker process for each experiment. This allows for the parallelization of stages and pipelines described above in *Experiment scheduling*.

The master and the worker processes communicate through IPC, Inter Process Communcation, implemented with `sipyco.pipe_ipc`. Specifically, it is `artiq.master.worker_impl` which is spawned as a new process for each experiment, and the class `artiq.master.worker.Worker` which manages the IPC requests of the workers, including access to `Scheduler` but also to devices, datasets, arguments, and CCBs. This allows the worker to support experiment `build()` methods and the *management system interfaces*.

The worker process also executes the experiment code itself. Within the experiment, kernel decorators – `kernel`, `subkernel`, etc. – call the ARTIQ compiler as necessary and trigger core device execution.

DRTIO SYSTEM

DRTIO is the 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 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 traffic, which is low-priority and non-realtime, e.g., to override and monitor TTL I/Os. Auxiliary traffic never interrupts or delays the main traffic, so it cannot cause unexpected latencies or exceptions (e.g. RTIO underflows).

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

Time transfer and clock synchronization is typically done over the serial link alone. The DRTIO code is written 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.

15.1 Terminology

In a system of interconnected DRTIO devices, each RTIO core (controlling a certain set of associated RTIO channels) is assigned a number and 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.

15.2 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 generated and manipulated with the utility `artiq_route`, see [Configuring the routing table](#). The binary file is 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.

15.3 Internal details

Bits 16-24 of the RTIO channel number (assigned to a respective device in the initial *system description JSON*, and specified again for use of the ARTIQ front-end in the device database) define the destination. Bits 0-15 of the RTIO channel number select the channel within the destination.

15.3.1 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.

15.3.2 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.

15.3.3 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.

15.3.4 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.

15.3.5 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.

15.3.6 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.

CORE DEVICE

The core device is a FPGA-based hardware component that contains a softcore or hardcore CPU tightly coupled with the so-called RTIO core, which runs in gateway and provides precision timing. The CPU executes Python code that is statically compiled by the ARTIQ compiler and communicates with peripherals (TTL, DDS, etc.) through the RTIO core, as described in *ARTIQ Real-Time I/O concepts*. 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 the other parts of ARTIQ (controllers, master, GUI, dataset management, etc.) without a core device, most use cases will require it.

16.1 Configuration storage

The core device reserves some storage space (either flash or directly on SD card, depending on target board) to store configuration data. The configuration data is organized as a list of key-value records, accessible either through *artiq_mkfs* and *artiq_flash* or, preferably in most cases, the `config` option of the *artiq_coremgmt* core management tool (see below). Information can be stored to keys of any name, but the specific keys currently used and referenced by ARTIQ are summarized below:

idle_kernel

Stores (compiled `.tar` or `.elf` binary of) idle kernel. See *Configuring the core device*.

startup_kernel

Stores (compiled `.tar` or `.elf` binary of) startup kernel. See *Configuring the core device*.

ip

Sets IP address of core device. For this and other networking options see also *Setting up core device networking*.

mac

Sets MAC address of core device. Unnecessary on Kasli or Kasli-SoC, which can obtain it from EEPROM.

ipv4_default_route

Sets IPv4 default route.

ip6

Sets IPv6 address of core device. Can be set irrespective of and used simultaneously as IPv4 address above.

ipv6_default_route

Sets IPv6 default route.

sed_spread_enable

If set to 1, will activate *Event spreading* in this core device. Needs to be set separately for satellite devices in a DRTIO setting.

log_level

Sets core device log level. Possible levels are TRACE, DEBUG, INFO, WARN, ERROR, and OFF. Note that enabling higher log levels will produce some core device slowdown.

uart_log_level

Sets UART log level, with same options. Printing a large number of messages to UART log will produce a significant slowdown.

rtio_clock

Sets the RTIO clock; see *Clocking*.

routing_table

Sets the routing table in DRTIO systems; see *Configuring the routing table*. If not set, a star topology is assumed.

device_map

If set, allows the core log to connect RTIO channels to device names and use device names as well as channel numbers in log output. A correctly formatted table can be automatically generated with *artiq_rtiomap*, see *Utilities*.

net_trace

If set to 1, will activate net trace (print all packets sent and received to UART and core log). This will considerably slow down all network response from the core. Not applicable for ARTIQ-Zynq (Kasli-SoC, ZC706).

panic_reset

If set to 1, core device will restart automatically. Not applicable for ARTIQ-Zynq.

no_flash_boot

If set to 1, will disable flash boot. Network boot is attempted if possible. Not applicable for ARTIQ-Zynq.

boot

Allows full firmware/gateway (`boot.bin`) to be written with *artiq_coremgmt*, on ARTIQ-Zynq systems only.

16.2 Common configuration commands

To write, then read, 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 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'
```

You can write several records at once:

```
$ artiq_coremgmt config write -s key1 value1 -f key2 filename -s key3 value3
```

You can also write entire files in a record using the `-f` option. This is useful for instance to write the startup and idle kernels into the flash storage:

```
$ artiq_coremgmt config write -f idle_kernel idle.elf
$ artiq_coremgmt config read idle_kernel | head -c9
b'\x7fELF'
```

The same option is used to write `boot.bin` in ARTIQ-Zynq. Note that the `boot` key is write-only.

See also the full reference of *artiq_coremgmt* in *Utilities*.

16.3 Clocking

The core device generates the RTIO clock using a PLL locked either to an internal crystal or to an external frequency reference. If choosing the latter, external reference must be provided (via front panel SMA input on Kasli boards). Valid configuration options include:

- `int_100` - internal crystal reference is used to synthesize a 100MHz RTIO clock,
- `int_125` - internal crystal reference is used to synthesize a 125MHz RTIO clock (default option),
- `int_150` - internal crystal reference is used to synthesize a 150MHz RTIO clock.
- `ext0_synth0_10to125` - external 10MHz reference clock used to synthesize a 125MHz RTIO clock,
- `ext0_synth0_80to125` - external 80MHz reference clock used to synthesize a 125MHz RTIO clock,
- `ext0_synth0_100to125` - external 100MHz reference clock used to synthesize a 125MHz RTIO clock,
- `ext0_synth0_125to125` - external 125MHz reference clock used to synthesize a 125MHz RTIO clock.

The selected option can be observed in the core device boot logs and accessed using `artiq_coremgmt config` with key `rtio_clock`.

As of ARTIQ 8, it is now possible for Kasli and Kasli-SoC configurations to enable WRPLL – a clock recovery method using DDMTD and Si549 oscillators – both to lock the main RTIO clock and (in DRTIO configurations) to lock satellites to master. This is set by the `enable_wrp11` option in the *JSON description file*. Because WRPLL requires slightly different gateway and firmware, it is necessary to re-flash devices to enable or disable it in extant systems. If you would like to obtain the firmware for a different WRPLL setting through AFWS, write to the `helpdesk@` email.

If phase noise performance is the priority, it is recommended to use `ext0_synth0_125to125` over other `ext0` options, as this bypasses the (noisy) MMCM.

If not using WRPLL, PLL can also be bypassed entirely with the options

- `ext0_bypass` (input clock used directly)
- `ext0_bypass_125` (explicit alias)
- `ext0_bypass_100` (explicit alias)

Bypassing the PLL ensures the skews between input clock, downstream clock outputs, and RTIO clock are deterministic across reboots of the system. This is useful when phase determinism is required in situations where the reference clock fans out to other devices before reaching the master.

16.4 Board details

16.4.1 FPGA board ports

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

16.4.2 Kasli and Kasli-SoC

Kasli and Kasli-SoC are versatile core devices designed for ARTIQ as part of the open-source Sinara family of boards. All support interfacing to various EEM daughterboards (TTL, DDS, ADC, DAC...) through twelve onboard EEM ports. Kasli is based on a Xilinx Artix-7 FPGA, and Kasli-SoC, which runs on a separate Zynq port of the ARTIQ firmware, is based on a Zynq-7000 SoC, notably including an ARM CPU allowing for much heavier software computations at high speeds. They are architecturally very different but supply similar feature sets. Kasli itself exists in two versions, of which the improved Kasli v2.0 is now in more common use, but the original v1.0 remains supported by ARTIQ.

Kasli can be connected to the network using a 1000Base-X SFP module, installed into the SFP0 cage. Kasli-SoC features a built-in Ethernet port to use instead. If configured as a DRTIO satellite, both boards instead reserve SFP0 for the upstream DRTIO connection; remaining SFP cages are available for downstream connections. Equally, if used as a DRTIO master, all free SFP cages are available for downstream connections (i.e. all but SFP0 on Kasli, all four on Kasli-SoC).

The DRTIO line rate depends upon the RTIO clock frequency running, e.g., at 125MHz the line rate is 2.5Gbps, at 150MHz 3.0Gbps, etc. See below for information on RTIO clocks.

16.4.3 KC705 and ZC706

Two high-end evaluation kits are also supported as alternative ARTIQ core device target boards, respectively the Kintex7 [KC705](#) and Zynq-SoC [ZC706](#), both from Xilinx. ZC706, like Kasli-SoC, runs on the ARTIQ-Zynq port. Both are supported in several set variants, namely NIST CLOCK and QC2 (FMC), either available in master, satellite, or standalone variants. See also [Building and developing ARTIQ](#) for more on system variants.

Common KC705 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.

16.5 Variant details

16.5.1 NIST CLOCK

With the KC705 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.

The ZC706 variant is identical except for the following differences:

- The SMA GPIO on channel 18 is replaced by an Input+Output capable PMOD1_0 line.
- Since there is no SDIO on the programmable logic side, channel 26 is instead occupied by an additional SPI:

RTIO channel	CS_N	MOSI	MISO	CLK
26	PMOD_SPI_CS_N	PMOD_SPI_MOSI	PMOD_SPI_MISO	PMOD_SPI_CLK

16.5.2 NIST QC2

With the KC705 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.

The ZC706 is identical except for the following differences:

- The SMA GPIO is once again replaced with PMOD1_0.
- The first four TTLs also have edge counters, on channels 52, 53, 54, and 55.

DEVELOPING A NETWORK DEVICE SUPPORT PACKAGE (NDSP)

Besides the kind of specialized real-time hardware most of ARTIQ is concerned with the control and management of, ARTIQ also easily handles more conventional ‘slow’ devices. This is done through *controllers*, based on *SiPyCo* (manual hosted [here](#)), which expose remote procedure call (RPC) interfaces to the network. This allows experiments to issue RPCs to the controllers as necessary, without needing to do direct I/O to the devices. Some advantages of this architecture include:

- Controllers/drivers can be run on different machines, alleviating cabling issues and OS compatibility problems.
- Reduces the impact of driver crashes.
- Reduces the impact of driver memory leaks.

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 support infrastructure may be kernels executed on the core device.

See also

Some NDSPs for particular devices have already been written and made available to the public. A (non-exhaustive) list can be found on the page *List of available NDSPs*.

17.1 Components of a NDSP

Full support for a specific device, called a network device support package or NDSP, requires 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 should be 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 should be 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.aqli_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, which would be unwieldy to pass as `sipyco_rpctool` command-line parameters, must be transferred over the network API.
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.

17.2 The driver and controller

As an example, we will develop a controller 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 (and offer as an RPC) is writing a message to that console.

To use RPCs, 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, we would place this class definition into a separate Python module called `driver`. In this example, for simplicity, we can include it in the controller module.

For the controller itself, we will turn this method into a server using `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 a `setuptools` entry point as well.

The parameters `::1` and `3249` are respectively the address to bind the server to (in this case, we use IPv6 localhost) and the TCP port. Add a line:

```
#!/usr/bin/env python3
```

at the beginning of the file, save it as `aqctl_hello.py`, and set its execution permissions:

```
$ chmod 755 aqctl_hello.py
```

Run it as:

```
$ ./aqctl_hello.py
```

In a different console, verify that you can connect to the TCP port:

```
$ telnet ::1 3249
Trying ::1...
Connected to ::1.
Escape character is '^]'.
```

Tip

To exit telnet, use the escape character combination (Ctrl +]) to access the `telnet>` prompt, and then enter `quit` or `close` to close the connection.

Also verify that a target (i.e. available service for RPC) named “hello” exists:

```
$ sipyco_rpctool :::1 3249 list-targets
Target(s):  hello
```

17.3 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. You have already used it above in the form of `list-targets`. 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, making sure the controller is running first. You should see the message appear in the controller’s terminal:

```
$ ./aqctl_hello.py
message: Hello World!
```

We see that the client has made a request to the server, which has, through the driver, performed the requisite I/O with the “device” (its console), resulting in the operation we wanted. Success!

Warning

Note that 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.

To access this driver in an experiment, we can retrieve the `Client` instance with the `get_device` and `set_device` methods of `artiq.language.environment.HasEnvironment`, and then use it like any other device (provided the controller is running and accessible at the time).

17.4 Integration with ARTIQ experiments

Generally we will want to add the device to our *device database* so that we can add it to an experiment with `self.setattr_device` and so the controller can be started and stopped automatically by a controller manager (the `artiq_ctlmgr` utility from `artiq-comtools`). To do so, add an entry to your device database in this format:

```
device_db.update({
    "hello": {
        "type": "controller",
        "host": "::1",
        "port": 3249,
        "command": "python /abs/path/to/aqctl_hello.py -p {port}"
    },
})
```

Now it can be added using `self.setattr_device("hello")` in the `build()` phase of the experiment, and its methods accessed via:

```
self.hello.message("Hello world!")
```

Note

In order to be correctly started and stopped by a controller manager, your controller must additionally implement a `ping()` method, which should simply return true, e.g.

```
def ping(self):
    return True
```

17.5 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`.

17.6 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():
```

(continues on next page)

(continued from previous page)

```

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.

17.7 Logging

For debug, information and warning messages, use the `logging` Python module and print the log on the standard error output (the default setting). As in other areas, there are five logging levels, from most to least critical, `CRITICAL`, `ERROR`, `WARNING`, `INFO`, and `DEBUG`. By default, the logging level starts at `WARNING`, meaning it will print messages of level `WARNING` and above (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 your controller. 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, then more messages (`INFO` and above) will be printed. If only one `-q` is present in the arguments, then `ERROR` and above will be printed. If `-qq` is present in the arguments, then only `CRITICAL` will be printed.

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",
                        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")

```

(continues on next page)

(continued from previous page)

```
logger.error("this is an error message")
logger.critical("this is a critical message")

if __name__ == "__main__":
    main()
```

17.8 Additional guidelines

17.8.1 Command line and options

- Controllers should be able to operate in “simulation” mode, specified with `--simulation`, 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 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 names.
- 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.

17.8.2 Style

- 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.
- Use docstrings for all public methods of the driver (note that those will be retrieved by `sipyco_rpcctool`).
- Choose a free default TCP port and add it to the *default port list* in this manual.

17.9 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*.

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	MSYS2 package	Documentation	URL
PDQ2	Not available	Not available	HTML	Gitea
Lab Brick Digital Attenuator	lda	lda	Not available	Gitea
Novatech 4098B	novatech409b	novatech409b	Not available	Gitea
Thorlabs T-Cubes	thorlabs_tcube	thorlabs_tcube	Not available	Gitea
Korad KA3005P	korad_ka3005p	korad_ka3005p	Not available	Gitea
Newfocus 8742	newfocus8742	newfocus8742	Not available	GitHub
Princeton Instruments PICam	Not available	Not available	Not available	GitHub
Anel HUT2 power distribution	hut2	Not available	Not available	Gitea
TOPTICA lasers	toptica-lasersdk-artiq	Not available	Not available	GitHub
HighFinesse wavemeters	highfinesse-net	Not available	Not available	GitHub
InfluxDB database	Not available	Not available	HTML	GitLab

MSYS2 packages all start with the `mingw-w64-clang-x86_64-` prefix.

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
Core analyzer proxy (proxy)	1385
Core analyzer proxy (control)	1386
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

MAIN FRONT-END TOOLS

These are the top-level commands used to run and manage ARTIQ experiments. Not all of the ARTIQ front-end is described here (many additional useful commands are presented in this manual in *Utilities*) but these together comprise the main points of access for using ARTIQ as a system.

20.1 `artiq.frontend.artiq_run`

Local experiment running tool

```
usage: artiq_run [-h] [--version] [-v] [-q] [--device-db DEVICE_DB]
                [--dataset-db DATASET_DB] [-c CLASS_NAME] [-o HDF5]
                FILE [ARGUMENTS ...]
```

20.1.1 Positional Arguments

FILE	file containing the experiment to run
ARGUMENTS	run arguments, use format KEY=VALUE

20.1.2 Named Arguments

--version	print the ARTIQ version number
--device-db	device database file (default: "device_db.py")
--dataset-db	dataset file (default: "dataset_db.mdb")
-c, --class-name	name of the class to run
-o, --hdf5	write results to specified HDF5 file (default: print them)

20.1.3 verbosity

-v, --verbose	increase logging level
-q, --quiet	decrease logging level

20.2 `artiq.frontend.artiq_master`

ARTIQ master

```
usage: artiq_master [-h] [--version] [--bind BIND] [--no-localhost-bind]
                  [--port-notify PORT_NOTIFY] [--port-control PORT_CONTROL]
                  [--port-logging PORT_LOGGING]
                  [--port-broadcast PORT_BROADCAST] [--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]
```

20.2.1 Named Arguments

- version** print the ARTIQ version number
- name** friendly name, displayed in dashboards to identify master instead of server address
- log-submissions** log experiment submissions to specified file

20.2.2 network server

- bind** additional hostname or IP address to bind to; use '*' to bind to all interfaces (default: [])
- no-localhost-bind** do not implicitly also bind to localhost addresses
- port-notify** TCP port for notifications connections (default: 3250)
- port-control** TCP port for control connections (default: 3251)
- port-logging** TCP port for remote logging connections (default: 1066)
- port-broadcast** TCP port for broadcasts connections (default: 1067)

20.2.3 databases

- device-db** device database file (default: 'device_db.py')
- dataset-db** dataset file (default: 'dataset_db.mdb')

20.2.4 repository

- g, --git** use the Git repository backend (default: False)
- r, --repository** path to the repository (default: 'repository')
- experiment-subdir** path to the experiment folder from the repository root (default: '')

20.2.5 logging

- v, --verbose** increase logging level of the master process
- q, --quiet** decrease logging level of the master process
- log-file** store logs in rotated files; set the base filename
- 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)

20.3 artiq.frontend.artiq_client

ARTIQ CLI client

```
usage: artiq_client [-h] [-s SERVER] [--port PORT] [--version] [-v] [-q]
                  {submit,delete,set-dataset,del-dataset,supply-interactive,cancel-
↪ interactive,show,scan-devices,scan-repository,ls,terminate}
                  ...
```

20.3.1 Positional Arguments

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

20.3.2 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 (default: None)
--version print the ARTIQ version number

20.3.3 verbosity

-v, --verbose increase logging level
-q, --quiet decrease logging level

20.3.4 Sub-commands

submit

submit an experiment

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

Positional Arguments

FILE file containing the experiment to run
ARGUMENTS run arguments, use format KEY=VALUE

Named Arguments

-p, --pipeline pipeline to run the experiment in (default: ‘main’)
-P, --priority priority (higher value means sooner scheduling, default: 0)
-t, --timed set a due date for the experiment (default: None)
-f, --flush flush the pipeline before preparing the experiment
-R, --repository use the experiment repository

- r, --revision** use a specific repository revision (defaults to head, ignored without -R)
- devarg-override** specify device arguments to override
- content** submit by content
- 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

set-dataset

add or modify a dataset

```
artiq_client set-dataset [-h] [--unit UNIT] [--scale SCALE]
                        [--precision PRECISION] [-p | -n]
                        NAME VALUE
```

Positional Arguments

- NAME** name of the dataset
- VALUE** value in PYON format

Named Arguments

- unit** physical unit of the dataset
- scale** factor to multiply value of dataset in displays
- precision** maximum number of decimals to print in displays
- p, --persist** make the dataset persistent
- n, --no-persist** make the dataset non-persistent

del-dataset

delete a dataset

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

Positional Arguments

name	name of the dataset
-------------	---------------------

supply-interactive

supply interactive arguments

```
artiq_client supply-interactive [-h] RID [ARGUMENTS ...]
```

Positional Arguments

RID	RID of target experiment
ARGUMENTS	interactive arguments

cancel-interactive

cancel interactive arguments

```
artiq_client cancel-interactive [-h] RID
```

Positional Arguments

RID	RID of target experiment
------------	--------------------------

show

show schedule, log, devices or datasets

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

Positional Arguments

WHAT	Possible choices: schedule, log, ccb, devices, datasets, interactive-args select object to show: ['schedule', 'log', 'ccb', 'devices', 'datasets', 'interactive-args']
-------------	---

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

ls

list a directory on the master

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

Positional Arguments

directory

terminate

terminate the ARTIQ master

```
artiq_client terminate [-h]
```

20.4 artiq.frontend.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]
                        [--analyzer-proxy-timeout ANALYZER_PROXY_TIMEOUT]
                        [--analyzer-proxy-timer ANALYZER_PROXY_TIMER]
                        [--analyzer-proxy-timer-backoff ANALYZER_PROXY_TIMER_BACKOFF]
                        [-v] [-q]
```

20.4.1 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 (default: None)

-p, --load-plugin Python module to load on startup

--analyzer-proxy-timeout connection timeout to core analyzer proxy (default: 5)

- analyzer-proxy-timer** retry timer to core analyzer proxy (default: 5)
- analyzer-proxy-timer-backoff** retry timer backoff multiplier to core analyzer proxy, (default: 1.1)

20.4.2 verbosity

- v, --verbose** increase logging level
- q, --quiet** decrease logging level

20.5 artiq.frontend.artiq_browser

ARTIQ Browser

```
usage: artiq_browser [-h] [--version] [--db-file DB_FILE]
                   [--browse-root BROWSE_ROOT] [-s SERVER] [--port PORT]
                   [-v] [-q]
                   [SELECT]
```

20.5.1 Positional Arguments

- SELECT** directory to browse or file to load

20.5.2 Named Arguments

- version** print the ARTIQ version number
- db-file** database file for local browser settings (default: None)
- browse-root** root path for directory tree (default ‘')
- s, --server** hostname or IP of the master to connect to when uploading datasets
- port** TCP port to use to connect to the master

20.5.3 verbosity

- v, --verbose** increase logging level
- q, --quiet** decrease logging level

20.6 artiq.frontend.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.

When supplying arguments to individual front-end tools, use `=` to avoid ambiguity in argument parsing, e.g.:

```
artiq_session -m=g -m--device-db=alternate_device_db.py -c=v
```

and so on.

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

20.6.1 Named Arguments

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

20.7 artiq_comtools.artiq_ctlmgr

ARTIQ controller manager. Supplied in the separate package `artiq-comtools`, which is included with a standard ARTIQ installation but can also be [installed standalone](#), with the intention of making it easier to run controllers and controller managers on machines where a full ARTIQ installation may not be necessary or convenient.

```
usage: artiq_ctlmgr [-h] [-v] [-q] [-s SERVER] [--port-notify PORT_NOTIFY]
                  [--port-logging PORT_LOGGING]
                  [--retry-master RETRY_MASTER] [--host-filter HOST_FILTER]
                  [--bind BIND] [--no-localhost-bind]
                  [--port-control PORT_CONTROL]
```

20.7.1 Named Arguments

-s, --server	hostname or IP of the master to connect to
--port-notify	TCP port to connect to for notifications
--port-logging	TCP port to connect to for logging
--retry-master	retry timer for reconnecting to master
--host-filter	IP address of controllers to launch (local address of master connection by default)

20.7.2 verbosity

-v, --verbose	increase logging level
-q, --quiet	decrease logging level

20.7.3 network server

--bind	additional hostname or IP address to bind to; use '*' to bind to all interfaces (default: [])
--no-localhost-bind	do not implicitly also bind to localhost addresses
--port-control	TCP port for control connections (default: 3249)

CORE LANGUAGE AND ENVIRONMENT

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 *`.

21.1 `artiq.language.core` module

Core ARTIQ extensions to the Python language.

`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 `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.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 for specifying additional flags.

`artiq.language.core.subkernel`(*arg=None, destination=0, flags={}*)

This decorator marks an object's method or function for execution on a satellite device. Destination must be given, and it must be between 1 and 255 (inclusive).

Subkernels behave similarly to kernels, with few key differences:

- they are started from main kernels,
- they do not support RPCs,
- but they can call other kernels or subkernels.

Subkernels can accept arguments and return values. However, they must be fully annotated with ARTIQ types.

To call a subkernel, call it like a normal function.

To await its finishing execution, call `subkernel.await(subkernel, [timeout])`. The timeout parameter is optional, and by default is equal to 10000 (milliseconds). This time can be adjusted for subkernels that take a long time to execute.

The compiled subkernel is copied to satellites, but not yet to the kernel core until it's called. For bigger subkernels it may take some time before they actually start running. To help with that, subkernels can be preloaded, with `subkernel_preload(subkernel)` function. A call to a preloaded subkernel will take less time, but only one subkernel can be preloaded at a time.

`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.

`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_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.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 parallel/sequential blocks, delays, etc. and provides a time-stamped logging facility for events.

exception `artiq.language.core.TerminationRequested`

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

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

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

`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.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).

21.2 artiq.language.environment module

class `artiq.language.environment.NoDefault`

Represents the absence of a default value.

exception `artiq.language.environment.DefaultMissing`

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

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

An argument that can be any PYON-serializable value.

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

A boolean argument.

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

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.
- **quickstyle** – Enables the choices to be displayed in the GUI as a list of buttons that submit the experiment when clicked.

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

An argument that can take a numerical value.

If `type=="auto"`, the result will be a `float` unless `precision = 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 `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.
- **precision** – The maximum number of decimals a UI should use.
- **type** – Type of this number. Accepts "`float`", "`int`" or "`auto`". Defaults to "`auto`".

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

A string argument.

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

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

```
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

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).

```
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.

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.

interactive(*title=""*)

Request arguments from the user interactively.

This context manager returns a namespace object on which the method *setattr_argument()* should be called, with the usual semantics.

When the context manager terminates, the experiment is blocked and the user is presented with the requested argument widgets. After the user enters values, the experiment is resumed and the namespace contains the values of the arguments.

If the interactive arguments request is cancelled, raises *CancelledArgsError*.

get_device_db()

Returns the full contents of the device database.

get_device(*key*)

Creates and returns a device driver.

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.

set_dataset(*key*, *value*, * (*Keyword-only parameters separator (PEP 3102)*), *unit=None*, *scale=None*, *precision=None*, *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

- **unit** – A string representing the unit of the value.
- **scale** – A numerical factor that is used to adjust the value of the dataset to match the scale or units of the experiment's reference frame when the value is displayed.
- **precision** – The maximum number of digits to print after the decimal point. Set *precision=None* to print as many digits as necessary to uniquely specify the value. Uses IEEE unbiased rounding.
- **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).

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).

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.

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

Returns the metadata of a dataset.

Returns dictionary with items describing the dataset, including the units, scale and precision.

This function is used to get additional information for displaying the dataset.

See `set_dataset()` for documentation of metadata items.

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.

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

Sets the default scheduling options.

This function should only be called from `build`.

class `artiq.language.environment.Experiment`

Base class for top-level experiments.

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

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()`.

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.

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.

exception `artiq.language.environment.CancelledArgsError`

Raised by the `interactive()` context manager when an interactive arguments request is cancelled.

21.3 artiq.language.scan module

Implementation and management of scan objects.

class `artiq.language.scan.ScanObject`

Represents a one-dimensional sweep of a numerical range. Multi-dimensional scans are constructed by combining several scan objects, for example using `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.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.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.Scannable(default=<class 'artiq.language.environment.NoDefault'>, unit="", *, scale=None, global_step=None, global_min=None, global_max=None, precision=2, ndecimals=None)`

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.
- **precision** – The maximum number of decimals a UI should use.

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). Iteration 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.

21.4 `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.

CORE REAL-TIME DRIVERS

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

22.1 System drivers

22.1.1 `artiq.coredevice.core` module

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

class `artiq.coredevice.core.Core`(*dmgr, host, ref_period, analyzer_proxy=None, analyze_at_run_end=False, ref_multiplier=8, target='rv32g', satellite_cpu_targets={}*)

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 1 ns. The machine time unit (`mu`) 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).
- **analyzer_proxy** – name of the core device analyzer proxy to trigger (optional).
- **analyze_at_run_end** – automatically trigger the core device analyzer proxy after the Experiment's run stage finishes.

`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.

`close()`

Disconnect core device and close sockets.

`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 available.

mu_to_seconds(*mu*)

Convert machine units (fine 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 and subkernels.

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 transferred 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 (fine RTIO cycles).

Parameters

seconds – time (in seconds) to convert.

trigger_analyzer_proxy()

Causes the core analyzer proxy to retrieve a dump from the device, and distribute it to all connected clients (typically dashboards).

Returns only after the dump has been retrieved from the device.

Raises `IOError` if no analyzer proxy has been configured, or if the analyzer proxy fails. In the latter case, more details would be available in the proxy log.

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.

22.1.2 `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 when 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.

exception `artiq.coredevice.exceptions.SubkernelError`

Raised when an operation regarding a subkernel is invalid or cannot be completed.

exception `artiq.coredevice.exceptions.UnwrapNoneError`

Raised when unwrapping a none Option.

22.1.3 `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 offloads the overhead of managing the handles onto the programmer.

record(*name*, *enable_ddma=False*)

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.

In DRTIO context, distributed DMA can be toggled with *enable_ddma*. Enabling it allows running DMA on satellites, rather than sending all events from the master.

Keeping it disabled it may improve performance in some scenarios, e.g. when there are many small satellite buffers.

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.

22.1.4 `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 `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

22.2 Digital I/O drivers

22.2.1 `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 gateway (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 `t1l_input`, up to the current timeline position:

```
t1l_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 = t1l_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 = t1l_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:

```
t1l_input.count(t1l_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.

A delay of at least one RTIO clock cycle is necessary 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.

A delay of at least one RTIO clock cycle is necessary 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_norrt()

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 results in negative slack.

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.

22.2.2 `artiq.coredevice.edge_counter` module

Driver for RTIO-enabled TTL edge counter.

As 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)
```

(continues on next page)

(continued from previous page)

```
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 RTIO 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 gateway 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)

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

See the sources of `artiq.gateware.rtio.phy.edge_counter` and `artiq.gateware.eem.DIO.add_std()` for the gateway 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, gateway_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 gateway phy.
- **gateway_width** – The width of the gateway counter register, in bits. This is only used for overflow handling; to change the size, the gateway 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=np.int64(-1)*)

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 gateway 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).

22.2.3 `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 results in collision errors.

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 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.

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 gateway core. Framing and chip selection can also be handled independently through other means, e.g. `TTLOut`.
- If there is a `mi_so` wire in the pads supplied in the gateway, 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 `mi_so` 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 `mi_so` 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; $f_{rtio_clk}/f_{spi} == div$. If `div` is odd, the setup phase of the SPI clock is one coarse RTIO clock cycle longer than the hold phase. (minimum=2, reset=2)
- **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.

22.2.4 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 are 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 are 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 are 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 per 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).

22.3 RF generation drivers

22.3.1 `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 \geq 1.3 valid options are: 0 - internal 100MHz XO; 1 - front-panel SMA; 2 internal MMCX. For hardware revision \leq 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 gateway 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 gateway 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.

22.3.2 `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 $f_{\text{ref}} / \text{clk_div} * \text{pll_n}$ 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 *AD9910.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 *CPLD.get_channel_att*.

Returns

Attenuation in dB.

get_att_mu() → numpy.int32

Get digital step attenuator value in machine units. See also *CPLD.get_channel_att*.

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 [AD9910.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 = np.int64(-1), profile: numpy.int32 = 7, ram_destination: numpy.int32 = -1*) → *float*

Set DDS data in SI units.

See also [AD9910.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 [CPLD.get_channel_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 [CPLD.get_channel_att](#).

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 and sizes.

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 and sizes.

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 = np.int64(-1), 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.

See also

`AD9910.set_phase_mode()` for a definition of the different phase modes.

Warning

Deterministic phase control depends on correct alignment of operations to a 4ns grid (SYNC_CLK). This function uses `now_mu()` to ensure such alignment automatically. When replayed over DMA, however, the ensuing event sequence *must* be started at the same offset relative to SYNC_CLK, or unstable SYNC_CLK cycle assignment (i.e. inconsistent delays of exactly 4ns) will result.

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 known 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. See also AD9910 datasheet.

Parameters

- **start** – Profile start address in RAM (10-bit).
- **end** – Profile end address in RAM, inclusive (10-bit).
- **step** – Profile time step, counted in DDS sample clock cycles, typically 4 ns (16-bit, 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 data bits
- **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 in advance and separately using [set_profile_ram\(\)](#) and the parent CPLD [set_profile\(\)](#).

Parameters

data – Data to be written to RAM.

22.3.3 `artiq.coredevice.ad9912` module

class `artiq.coredevice.ad9912.AD9912`(*dmgr, chip_select, cpld_device, sw_device=None, pll_n=10, pll_en=1*)

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 $f_{\text{ref}} / \text{clk_div} * \text{pll_n}$ 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.

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 [AD9912.get_mu\(\)](#).

Returns

A tuple (frequency, phase).

get_att() → float

Get digital step attenuator value in SI units.

See also *get_channel_att()*.

Returns

Attenuation in dB.

get_att_mu() → numpy.int32

Get digital step attenuator value in machine units.

See also *get_channel_att_mu()*.

Returns

Attenuation setting, 8-bit digital.

get_mu()

Get the frequency tuning word and phase offset word.

See also *AD9912.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 *AD9912.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 `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 `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)

22.3.4 `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 results in collision errors.

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).

22.3.5 `artiq.coredevice.mirny` module

RTIO driver for Mirny (4-channel GHz PLLs)

```
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 `Mirny.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.

22.3.6 `artiq.coredevice.almazny` module

class `artiq.coredevice.almazny.AlmaznyChannel`(*dmgr, host_mirny, channel*)

Driver for one Almazny channel.

Almazny is a mezzanine for the Quad PLL RF source Mirny that exposes and controls the frequency-doubled outputs. This driver requires Almazny hardware revision v1.2 or later and Mirny CPLD gateway v0.3 or later. Use `AlmaznyLegacy` for Almazny hardware v1.1 and earlier.

Parameters

- **host_mirny** – Mirny CPLD device name
- **channel** – channel index (0-3)

set(*att, enable, led=False*)

Set attenuation, RF switch, and LED state (SI units).

Parameters

- **att** – attenuator setting in dB (0-31.5)
- **enable** – RF switch state (bool)
- **led** – LED state (bool)

set_mu(*mu*)

Set channel state (machine units).

Parameters**mu** – channel state in machine units.**to_mu**(*att, enable, led*)

Convert an attenuation in dB, RF switch state and LED state to machine units.

Parameters

- **att** – attenuator setting in dB (0-31.5)
- **enable** – RF switch state (bool)
- **led** – LED state (bool)

Returns

channel setting in machine units

class `artiq.coredevice.almazny.AlmaznyLegacy`(*dmgr, host_mirny*)

Almazny (High-frequency mezzanine board for Mirny)

This applies to Almazny hardware v1.1 and earlier. Use [AlmaznyChannel](#) for Almazny v1.2 and later.**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)

22.3.7 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 [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

$$mod1 = 2^{*}24 \text{ and } mod2 \leq 2^{*}28$$

Parameters

- **f_vco** – target VCO frequency
- **f_pfd** – PFD frequency

Returns

(*n*, *frac1*, (*frac2_msb*, *frac2_lsb*), (*mod2_msb*, *mod2_lsb*))

22.3.8 artiq.coredevice.phaser module

`class artiq.coredevice.phaser.Miqro(channel)`

Miqro pulse generator.

A Miqro instance represents one RF output. The DSP components are fully contained in the Phaser gateway. The output is generated by with the following data flow:

Oscillators

- There are `n_osc = 16` oscillators with oscillator IDs `0... n_osc-1`.
- Each oscillator outputs one tone at any given time
 - I/Q (quadrature, a.k.a. complex) 2x16-bit signed data at `tau = 4 ns` sample intervals, 250 MS/s, Nyquist 125 MHz, bandwidth 200 MHz (from `f = -100..+100 MHz`, taking into account the interpolation anti-aliasing filters in subsequent interpolators),
 - 32-bit frequency (*f*) resolution ($\sim 1/16$ Hz),
 - 16-bit unsigned amplitude (*a*) resolution
 - 16-bit phase offset (*p*) resolution
- The output phase *p'* of each oscillator at time *t* (boot/reset/initialization of the device at `t=0`) is then $p' = f * t + p \pmod{1 \text{ turn}}$ where *f* and *p* are the (currently active) profile frequency and phase offset.

Note

The terms “phase coherent” and “phase tracking” are defined to refer to this choice of oscillator output phase *p'*. Note that the phase offset *p* is not relative to (on top of previous phase/profiles/oscillator history). It is “absolute” in the sense that frequency *f* and phase offset *p* fully determine oscillator output phase *p'* at time *t*. This is unlike typical DDS behavior.

- Frequency, phase, and amplitude of each oscillator are configurable by selecting one of `n_profiles = 32` profiles `0... n_profile-1`. This selection is fast and can be done for each pulse. The phase coherence defined above is guaranteed for each profile individually.

- Note: one profile per oscillator (usually profile index 0) should be reserved for the NOP (no operation, identity) profile, usually with zero amplitude.
- Data for each profile for each oscillator can be configured individually. Storing profile data should be considered “expensive”.

Note

To refer to an operation as “expensive” does not mean it is impossible, merely that it may take a significant amount of time and resources to execute, such that it may be impractical when used often or during fast pulse sequences. They are intended for use in calibration and initialization.

Summation

- The oscillator outputs are added together (wrapping addition).
- The user must ensure that the sum of oscillators outputs does not exceed the data range. In general that means that the sum of the amplitudes must not exceed one.

Shaper

- The summed complex output stream is then multiplied with a the complex-valued output of a triggerable shaper.
- Triggering the shaper corresponds to passing a pulse from all oscillators to the RF output.
- Selected profiles become active simultaneously (on the same output sample) when triggering the shaper with the first shaper output sample.
- The shaper reads (replays) window samples from a memory of size $n_window = 1 \ll 10$.
- The window memory can be segmented by choosing different start indices to support different windows.
- Each window memory segment starts with a header determining segment length and interpolation parameters.
- The window samples are interpolated by a factor (rate change) between 1 and $r = 1 \ll 12$.
- The interpolation order is constant, linear, quadratic, or cubic. This corresponds to interpolation modes from rectangular window (1st order CIC) or zero order hold) to Parzen window (4th order CIC or cubic spline).
- This results in support for single shot pulse lengths (envelope support) between τ and a bit more than $r * n_window * \tau = (1 \ll 12 + 10) \tau \sim 17 \text{ ms}$.
- Windows can be configured to be head-less and/or tail-less, meaning, they do not feed zero-amplitude samples into the shaper before and after each window respectively. This is used to implement pulses with arbitrary length or CW output.

Overall properties

- The DAC may upconvert the signal by applying a frequency offset f_1 with phase p_1 .
- In the Upconverter Phaser variant, the analog quadrature upconverter applies another frequency of f_2 and phase p_2 .
- The resulting phase of the signal from one oscillator at the SMA output is $(f + f_1 + f_2) * t + p + s(t - t_0) + p_1 + p_2 \pmod{1 \text{ turn}}$ where $s(t - t_0)$ is the phase of the interpolated shaper output, and t_0 is the trigger time (fiducial of the shaper). Unsurprisingly the frequency is the derivative of the phase.
- Group delays between pulse parameter updates are matched across oscillators, shapers, and channels.

- The minimum time to change profiles and phase offsets is ~128 ns (estimate, TBC). This is the minimum pulse interval. The sustained pulse rate of the RTIO PHY/Fastlink is one pulse per Fastlink frame (may be increased, TBC).

encode(*window, profiles, data*)

Encode window and profile selection.

Parameters

- **window** – Window start address (0 to 0x3ff)
- **profiles** – List of profile indices for the oscillators. Maximum length 16. Unused oscillators will be set to profile 0.
- **data** – List of integers to store the encoded data words into. Unused entries will remain untouched. Must contain at least three elements if all oscillators are used and should be initialized to zeros.

Returns

Number of words from *data* used.

pulse(*window, profiles*)

Emit a pulse

This encodes the window and profiles (see [encode\(\)](#)) and emits them (see [pulse_mu\(\)](#)).

Parameters

- **window** – Window start address (0 to 0x3ff)
- **profiles** – List of profile indices for the oscillators. Maximum length 16. Unused oscillators will select profile 0.

pulse_mu(*data*)

Emit a pulse (encoded)

The pulse fiducial timing resolution is 4 ns.

Parameters

data – List of up to 3 words containing an encoded MIQRO pulse as returned by [encode\(\)](#).

reset()

Establish no-output profiles and no-output window and execute them.

This establishes the first profile (index 0) on all oscillators as zero amplitude, creates a trivial window (one sample with zero amplitude, minimal interpolation), and executes a corresponding pulse.

set_profile(*oscillator, profile, frequency, amplitude, phase=0.0*)

Store an oscillator profile.

Parameters

- **oscillator** – Oscillator index (0 to 15)
- **profile** – Profile index (0 to 31)
- **frequency** – Frequency in Hz (passband -100 to 100 MHz). Interpreted in the Nyquist sense, i.e. aliased.
- **amplitude** – Amplitude in units of full scale (0. to 1.)
- **phase** – Phase in turns. See [Miqro](#) for a definition of phase in this context.

Returns

The quantized 32-bit frequency tuning word

set_profile_mu(*oscillator, profile, ftw, asf, pow_=0*)

Store an oscillator profile (machine units).

Parameters

- **oscillator** – Oscillator index (0 to 15)
- **profile** – Profile index (0 to 31)
- **ftw** – Frequency tuning word (32-bit signed integer on a 250 MHz clock)
- **asf** – Amplitude scale factor (16-bit unsigned integer)
- **pow** – Phase offset word (16-bit integer)

set_window(*start, iq, period=4e-09, order=3, head=1, tail=1*)

Store a window segment.

Parameters

- **start** – Window start address (0 to 0x3ff)
- **iq** – List of IQ window samples. Each window sample is a pair of two float numbers -1 to 1, one for each I and Q in units of full scale. The maximum window length is 0x3fe. The user must ensure that this window does not overlap with other windows in the memory.
- **period** – Desired window sample period in SI units ($4*ns$ to $(4 \ll 12)*ns$).
- **order** – Interpolation order from 0 (corresponding to constant/zero-order-hold/1st order CIC interpolation) to 3 (corresponding to cubic/Parzen/4th order CIC interpolation)
- **head** – Update the interpolator settings and clear its state at the start of the window. This also implies starting the envelope from zero.
- **tail** – Feed zeros into the interpolator after the window samples. In the absence of further pulses this will return the output envelope to zero with the chosen interpolation.

Returns

Actual sample period in SI units

set_window_mu(*start, iq, rate=1, shift=0, order=3, head=1, tail=1*)

Store a window segment (machine units).

Parameters

- **start** – Window start address (0 to 0x3ff)
- **iq** – List of IQ window samples. Each window sample is an integer containing the signed I part in the 16 LSB and the signed Q part in the 16 MSB. The maximum window length is 0x3fe. The user must ensure that this window does not overlap with other windows in the memory.
- **rate** – Interpolation rate change (1 to $1 \ll 12$)
- **shift** – Interpolator amplitude gain compensation in powers of 2 (0 to 63)
- **order** – Interpolation order from 0 (corresponding to constant/rectangular window/zero-order-hold/1st order CIC interpolation) to 3 (corresponding to cubic/Parzen window/4th order CIC interpolation)
- **head** – Update the interpolator settings and clear its state at the start of the window. This also implies starting the envelope from zero.
- **tail** – Feed zeros into the interpolator after the window samples. In the absence of further pulses this will return the output envelope to zero with the chosen interpolation.

Returns

Next available window memory address after this segment.

```
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,
                                     gw_rev=1, 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 RTIO PHY and the Phaser gateway come in different modes that have different features. Phaser mode and coredevice PHY mode are both selected at their respective gateway compile-time and need to match.

Phaser gateway	Coredevice PHY	Features per <i>PhaserChannel</i>
Base <= v0.5	Base	Base (5 <i>PhaserOscillator</i>)
Base >= v0.6	Base	Base + Servo
Miqro >= v0.6	Miqro	<i>Miqro</i>

The coredevice driver (this class and *PhaserChannel*) exposes the superset of all functionality regardless of the Coredevice RTIO PHY or Phaser gateway modes. This is to evade type unification limitations. Features absent in Coredevice PHY/Phaser gateway will not work and should not be accessed.

Base mode

The coredevice produces 2 IQ (in-phase and quadrature) data streams with 25 MS/s and 14 bits 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 may 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()*).

Miqro mode

See *Miqro*. Here the DAC operates in 4x interpolation.

Analog flow

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.

Servo

Each phaser output channel features a servo to control the RF output amplitude using feedback from an ADC. The servo consists of a first order IIR (infinite impulse response) filter fed by the ADC and a multiplier that scales the I and Q datastreams from the DUC by the IIR output. The IIR state is updated at the 3.788 MHz ADC sampling rate.

Each channel IIR features 4 profiles, each consisting of the [b0, b1, a1] filter coefficients as well as an output offset. The coefficients and offset can be set for each profile individually and the profiles each have their own y0, y1 output registers (the x0, x1 inputs are shared). To avoid transient effects, care should be taken to not update the coefficients in the currently selected profile.

The servo can be en- or disabled for each channel. When disabled, the servo output multiplier is simply bypassed and the datastream reaches the DAC unscaled.

The IIR output can be put on hold for each channel. In hold mode, the filter still ingests samples and updates its input x0 and x1 registers, but does not update the y0, y1 output registers.

After power-up the servo is disabled, in profile 0, with coefficients [0, 0, 0] and hold is enabled. If older gateware without the servo is loaded onto the Phaser FPGA, the device simply behaves as if the servo is disabled and none of the servo functions have any effect.

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 instances of *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 int32s containing the pattern

Returns

Bit error mask (16-bit)

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 μ s SPI clock) to read the temperature register.

dac_sync()

Trigger DAC synchronisation for both output channels.

The DAC `si_f_sync` is de-asserted, 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 *Phaser*).

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 bits 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 since 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 bits 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 *Phaser*).

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 in machine units.

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.

write16(*addr, data: numpy.int32*)

Write 16 bits to a sequence of FPGA registers.

write32(*addr, data: numpy.int32*)

Write 32 bits 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 *PhaserOscillator* (in the coredevice phy),
- an interpolation chain and digital upconverter (DUC) on Phaser,
- a *Miqro* instance 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 instances of *PhaserOscillator*.
- `miqro`: A *Miqro*.

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. *Miqro* is not affected by this, but both the oscillators and *Miqro* can be affected by intrinsic overshoot of the interpolator on the DAC.

`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 across 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_iir(*profile, kp, ki=0.0, g=0.0, x_offset=0.0, y_offset=0.0*)

Set servo profile IIR coefficients.

Avoid setting the IIR parameters of the currently active profile.

Gains are given in units of output full per scale per input full scale.

Note

Due to inherent constraints of the fixed point datatypes and IIR filters, the `x_offset` (setpoint) resolution depends on the selected gains. Low `ki` gains will lead to a low `x_offset` resolution.

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 (0-3)
- **kp** – Proportional gain. This is usually negative (closed loop, positive ADC voltage, positive setpoint). When 0, this implements a pure I controller.
- **ki** – Integrator gain (rad/s). Equivalent to the gain at 1 Hz. 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`.
- **x_offset** – IIR input offset. Used as the negative setpoint when stabilizing to a desired input setpoint. Will be converted to an equivalent output offset and added to `y_offset`.
- **y_offset** – IIR output offset.

`set_iir_mu(profile, b0, b1, a1, offset)`

Load a servo profile consisting of the three filter coefficients and an output offset.

Avoid setting the IIR parameters of the currently active profile.

The recurrence relation is (all data signed and MSB aligned):

$$a_0 y_n = a_1 y_{n-1} + b_0 x_n + b_1 x_{n-1} + o$$

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^{14}
- a_1 is the feedback gain
- b_0 and b_1 are the feedforward gains for the two delays

See also `PhaserChannel.set_iir()`.

Parameters

- **profile** – Profile to set (0 to 3)
- **b0** – b0 filter coefficient (16-bit signed)
- **b1** – b1 filter coefficient (16-bit signed)
- **a1** – a1 filter coefficient (16-bit signed)
- **offset** – Output offset (16-bit signed)

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 [Phaser](#)).

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 [Phaser](#)).

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 `syncsel_mixerxx` fields in the dac configuration dictionary (see [Phaser](#)).

Use of the DAC-NCO requires the DAC mixer and NCO to be enabled. These can be configured via the dac configuration dictionary.

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 `syncsel_mixerxx` fields in the dac configuration dictionary (see [Phaser](#)).

Use of the DAC-NCO requires the DAC mixer and NCO to be enabled. These can be configured via the dac configuration dictionary.

Parameters

pow – NCO phase offset word (16-bit)

set_servo(*profile=0, enable=0, hold=0*)

Set the servo configuration.

Parameters

- **enable** – 1 to enable servo, 0 to disable servo (default). If disabled, the servo is bypassed and hold is enforced since the control loop is broken.
- **hold** – 1 to hold the servo IIR filter output constant, 0 for normal operation.
- **profile** – Profile index to select for channel. (0 to 3)

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.Phaser0oscillator`(*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)

22.4 DAC/ADC drivers

22.4.1 `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 results in a collision 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 (default: 8192). Device dependent and must be set correctly for correct voltage-to-mu conversions. Knowledge of this state is not transferred between experiments.
- **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
- **vzs** – Measured voltage with the DAC set to zero-scale (0x0000)
- **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 $\leq 1.5\mu\text{s}$ “BUSY” period (t_{10}) 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: $[-2*vref, +2*vref - 1 \text{ LSB}] + \text{voltage offset}$.

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: $[-2*vref, +2*vref - 1 \text{ LSB}] + \text{voltage offset}$.
- **offset_dacs** – Register value for the two offset DACs (default: 0x2000)
- **vref** – DAC reference voltage (default: 5.)

Returns

The 16-bit DAC register value

22.4.2 `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 results in a collision 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

22.4.3 `artiq.coredevice.sampler` module

```
class artiq.coredevice.sampler.Sampler(dmgr, spi_adc_device, spi_pgia_device, cnv_device, div=8,
                                       gains=0, hw_rev='v2.2', 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.
- **hw_rev** – Sampler’s hardware revision string (default ‘v2.2’)
- **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 *Sampler.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, corrected_fs=True)

Convert ADC data in machine units to volts.

Parameters

- **data** – 16-bit signed ADC word
- **gain** – PGIA gain setting (0: 1, ..., 3: 1000)
- **corrected_fs** – use corrected ADC FS reference. Should be True for Sampler revisions after v2.1. False for v2.1 and earlier.

Returns

Voltage in volts

22.4.4 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 :exc:RTIOOverflow exceptions.

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 *Novogorny.sample_mu()*.

Parameters

- **next_ctrl** – ADC control word for the next sample

Returns

The ADC result packet (volts)

```
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 (volts).

22.4.5 `artiq.coredevice.fastino` module

RTIO driver for the Fastino 32-channel, 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 `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 (gateway) 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 (gateway).

apply_cic(channel_mask)

Apply the staged interpolator configuration on the specified channels.

Each Fastino channel starting with gateway v0.2 includes a fourth order (cubic) CIC interpolator with variable rate change and variable output gain compensation (see `stage_cic()`).

Fastino gateway 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 continuous 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.

Warning

On Fastino gateway 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.

22.4.6 `artiq.coredevice.shuttler` module

class `artiq.coredevice.shuttler.ADC`(*dmgr*, *spi_device*, *core_device='core'*)

Shuttler AFE ADC (AD4115) driver.

Parameters

- **spi_device** – SPI bus device name.
- **core_device** – Core device name.

calibrate(*volts*, *trigger*, *config*, *samples*=[-5.0, 0.0, 5.0])

Calibrate the Shuttler waveform generator using the ADC on the AFE.

Finds the average slope rate and average offset by samples, and compensates by writing the pre-DAC gain and offset registers in the configuration registers.

Note

If the pre-calibration slope rate is less than 1, the calibration procedure will introduce a pre-DAC gain compensation. However, this may saturate the pre-DAC voltage code (see [Config](#) notes). Shuttler cannot cover the entire +/- 10 V range in this case. See also [Config.set_gain\(\)](#) and [Config.set_offset\(\)](#).

Parameters

- **volts** – A list of all 16 cubic DC-bias splines. (See *DCBias*)
- **trigger** – The Shuttler spline coefficient update trigger.
- **config** – The Shuttler Core configuration registers.
- **samples** – A list of sample voltages for calibration. There must be at least 2 samples to perform slope rate calculation.

power_down()

Place the ADC in power-down mode.

The ADC must be reset before returning to other modes.

Note

The AD4115 datasheet suggests placing the ADC in standby mode before power-down. This is to prevent accidental entry into the power-down mode. See also *standby()* and *power_up()*.

power_up()

Exit the ADC power-down mode.

The ADC should be in power-down mode before calling this method.

See also *power_down()*.

read16(addr: numpy.int32) → numpy.int32

Read from 16-bit register.

Parameters

addr – Register address.

Returns

Read-back register content.

read24(addr: numpy.int32) → numpy.int32

Read from 24-bit register.

Parameters

addr – Register address.

Returns

Read-back register content.

read8(addr: numpy.int32) → numpy.int32

Read from 8-bit register.

Parameters

addr – Register address.

Returns

Read-back register content.

read_ch(channel: numpy.int32) → float

Sample a Shuttler channel on the AFE.

Performs a single conversion using profile 0 and setup 0 on the selected channel. The sample is then recovered and converted to volts.

Parameters

channel – Shuttler channel to be sampled.

Returns

Voltage sample in volts.

read_id() → `numpy.int32`

Read the product ID of the ADC.

The expected return value is 0x38DX, the 4 LSbs are don't cares.

Returns

The read-back product ID.

reset()

AD4115 reset procedure.

Performs a write operation of 96 serial clock cycles with DIN held at high. This resets the entire device, including the register contents.

Note

The datasheet only requires 64 cycles, but reasserting CS_n right after the transfer appears to interrupt the start-up sequence.

single_conversion()

Place the ADC in single conversion mode.

The ADC returns to standby mode after the conversion is complete.

standby()

Place the ADC in standby mode and disable power down the clock.

The ADC can be returned to single conversion mode by calling `single_conversion()`.

write16(addr: `numpy.int32`, data: `numpy.int32`)

Write to 16-bit register.

Parameters

- **addr** – Register address.
- **data** – Data to be written.

write24(addr: `numpy.int32`, data: `numpy.int32`)

Write to 24-bit register.

Parameters

- **addr** – Register address.
- **data** – Data to be written.

write8(addr: `numpy.int32`, data: `numpy.int32`)

Write to 8-bit register.

Parameters

- **addr** – Register address.
- **data** – Data to be written.

```
class artiq.coredevice.shuttler.Config(dmgr, channel, core_device='core')
```

Shuttler configuration registers interface.

The configuration registers control waveform phase auto-clear, pre-DAC gain and offset values for calibration with ADC on the Shuttler AFE card.

To find the calibrated DAC code, the Shuttler Core first multiplies the output data with pre-DAC gain, then adds the offset.

Note

The DAC code is capped at 0x7fff and 0x8000.

Parameters

- **channel** – RTIO channel number of this interface.
- **core_device** – Core device name.

```
get_gain(channel)
```

Return the pre-DAC gain value of a Shuttler Core channel.

Parameters

channel – The Shuttler Core channel.

Returns

Pre-DAC gain value. See [set_gain\(\)](#).

```
get_offset(channel)
```

Return the pre-DAC offset value of a Shuttler Core channel.

Parameters

channel – The Shuttler Core channel.

Returns

Pre-DAC offset value. See [set_offset\(\)](#).

```
set_clr(clr)
```

Set/Unset waveform phase clear bits.

Each bit corresponds to a Shuttler waveform generator core. Setting a clear bit forces the Shuttler Core to clear the phase accumulator on waveform trigger (See [Trigger](#) for the trigger method). Otherwise, the phase accumulator increments from its original value.

Parameters

clr – Waveform phase clear bits. The MSB corresponds to Channel 15, LSB corresponds to Channel 0.

```
set_gain(channel, gain)
```

Set the 16-bits pre-DAC gain register of a Shuttler Core channel.

The *gain* parameter represents the decimal portion of the gain factor. The MSB represents 0.5 and the sign bit. Hence, the valid total gain value (1 +/- 0.gain) ranges from 0.5 to 1.5 - LSB.

Parameters

- **channel** – Shuttler Core channel to be configured.
- **gain** – Shuttler Core channel gain.

`set_offset(channel, offset)`

Set the 16-bits pre-DAC offset register of a Shuttler Core channel.

See also `shuttler_volt_to_mu()`.

Parameters

- **channel** – Shuttler Core channel to be configured.
- **offset** – Shuttler Core channel offset.

`class artiq.coredevice.shuttler.DCBias(dmgr, channel, core_device='core')`

Shuttler Core cubic DC-bias spline.

A Shuttler channel can generate a waveform $w(t)$ that is the sum of a cubic spline $a(t)$ and a sinusoid modulated in amplitude by a cubic spline $b(t)$ and in phase/frequency by a quadratic spline $c(t)$, where

$$w(t) = a(t) + b(t) * \cos(c(t))$$

and t corresponds to time in seconds. This class controls the cubic spline $a(t)$, in which

$$a(t) = p_0 + p_1t + \frac{p_2t^2}{2} + \frac{p_3t^3}{6}$$

and $a(t)$ is measured in volts.

Parameters

- **channel** – RTIO channel number of this DC-bias spline interface.
- **core_device** – Core device name.

`set_waveform(a0: numpy.int32, a1: numpy.int32, a2: numpy.int64, a3: numpy.int64)`

Set the DC-bias spline waveform.

Given $a(t)$ as defined in `DCBias`, the coefficients should be configured by the following formulae:

$$T = 8 * 10^{-9}$$

$$a_0 = p_0$$

$$a_1 = p_1T + \frac{p_2T^2}{2} + \frac{p_3T^3}{6}$$

$$a_2 = p_2T^2 + p_3T^3$$

$$a_3 = p_3T^3$$

a_0 , a_1 , a_2 and a_3 are 16, 32, 48 and 48 bits in width respectively. See `shuttler_volt_to_mu()` for machine unit conversion.

Note

The waveform is not updated to the Shuttler Core until triggered. See `Trigger` for the update triggering mechanism.

Parameters

- **a0** – The a_0 coefficient in machine unit.
- **a1** – The a_1 coefficient in machine unit.
- **a2** – The a_2 coefficient in machine unit.
- **a3** – The a_3 coefficient in machine unit.

`class artiq.coredevice.shuttler.DDS(dmgr, channel, core_device='core')`

Shuttler Core DDS spline.

A Shuttler channel can generate a waveform $w(t)$ that is the sum of a cubic spline $a(t)$ and a sinusoid modulated in amplitude by a cubic spline $b(t)$ and in phase/frequency by a quadratic spline $c(t)$, where

$$w(t) = a(t) + b(t) * \cos(c(t))$$

and t corresponds to time in seconds. This class controls the cubic spline $b(t)$ and quadratic spline $c(t)$, in which

$$b(t) = g * (q_0 + q_1 t + \frac{q_2 t^2}{2} + \frac{q_3 t^3}{6})$$

$$c(t) = r_0 + r_1 t + \frac{r_2 t^2}{2}$$

$b(t)$ is in volts, $c(t)$ is in number of turns. Note that $b(t)$ contributes to a constant gain of $g = 1.64676$.

Parameters

- **channel** – RTIO channel number of this DC-bias spline interface.
- **core_device** – Core device name.

`set_waveform(b0: numpy.int32, b1: numpy.int32, b2: numpy.int64, b3: numpy.int64, c0: numpy.int32, c1: numpy.int32, c2: numpy.int32)`

Set the DDS spline waveform.

Given $b(t)$ and $c(t)$ as defined in [DDS](#), the coefficients should be configured by the following formulae.

$$T = 8 * 10^{-9}$$

$$b_0 = q_0$$

$$b_1 = q_1 T + \frac{q_2 T^2}{2} + \frac{q_3 T^3}{6}$$

$$b_2 = q_2 T^2 + q_3 T^3$$

$$b_3 = q_3 T^3$$

$$c_0 = r_0$$

$$c_1 = r_1 T + \frac{r_2 T^2}{2}$$

$$c_2 = r_2 T^2$$

b_0 , b_1 , b_2 and b_3 are 16, 32, 48 and 48 bits in width respectively. See [shuttler_volt_to_mu\(\)](#) for machine unit conversion. c_0 , c_1 and c_2 are 16, 32 and 32 bits in width respectively.

Note: The waveform is not updated to the Shuttler Core until triggered. See [Trigger](#) for the update triggering mechanism.

Parameters

- **b0** – The b_0 coefficient in machine units.
- **b1** – The b_1 coefficient in machine units.
- **b2** – The b_2 coefficient in machine units.
- **b3** – The b_3 coefficient in machine units.
- **c0** – The c_0 coefficient in machine units.
- **c1** – The c_1 coefficient in machine units.
- **c2** – The c_2 coefficient in machine units.

class `artiq.coredevice.shuttler.Relay`(*dmgr*, *spi_device*, *core_device='core'*)

Shuttler AFE relay switches.

This class controls the AFE relay switches and the LEDs. Switch the relay on to enable AFE output; off to disable the output. The LEDs indicate the relay status.

Note

The relay does not disable ADC measurements. Voltage of any channels can still be read by the ADC even after switching off the relays.

Parameters

- **spi_device** – SPI bus device name.
- **core_device** – Core device name.

enable(*en*: *numpy.int32*)

Enable/disable relay switches of corresponding channels.

Each bit corresponds to the relay switch of a channel. Asserting a bit turns on the corresponding relay switch; deasserting the same bit turns off the switch instead.

Parameters

en – Switch enable bits. The MSB corresponds to Channel 15, LSB corresponds to Channel 0.

init()

Initialize SPI device.

Configures the SPI bus to 16 bits, write-only, simultaneous relay switches and LED control.

class `artiq.coredevice.shuttler.Trigger`(*dmgr*, *channel*, *core_device='core'*)

Shuttler Core spline coefficients update trigger.

Parameters

- **channel** – RTIO channel number of the trigger interface.
- **core_device** – Core device name.

trigger(*trig_out*)

Triggers coefficient update of (a) Shuttler Core channel(s).

Each bit corresponds to a Shuttler waveform generator core. Setting `trig_out` bits commits the pending coefficient update (from `set_waveform` in `DCBias` and `DDS`) to the Shuttler Core synchronously.

Parameters

trig_out – Coefficient update trigger bits. The MSB corresponds to Channel 15, LSB corresponds to Channel 0.

`artiq.coredevice.shuttler.shuttler_volt_to_mu`(*volt*)

Return the equivalent DAC code. Valid input range is from -10 to 10 - LSB.

22.5 Miscellaneous

22.5.1 `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 μ s 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 known 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 known 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 [Channel.set_dds\(\)](#).

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 μ s) before allowing IIR updates after invoking `set()`. This is rounded to the nearest number of servo cycles ($\sim 1.2 \mu$ s). 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 `Channel.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 μ s, 0-255).

`set_y(profile, y)`

Set a profile's IIR state (filter output, Y0).

The IIR state is also known 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 known 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,
                                       sampler_hw_rev='v2.2', 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 gateway 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.
- **sampler_hw_rev** – Sampler’s revision string
- **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

enable (*int*) – 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 μ s) 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 μ s) 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, corrected_fs=True)`

Convert servo ADC data from machine units to volts.

`artiq.coredevice.suservo.y_mu_to_full_scale(y)`

Convert servo Y data from machine units to units of full scale.

22.5.2 `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*, *timeout_mu=-1*)

Retrieves the accumulated values for one frame from the ROI engines. Blocks until values are available or timeout is reached.

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.

If the timeout is reached before data is available, the exception `GrabberTimeoutException` is raised.

Parameters

timeout_mu – Timestamp at which a timeout will occur. Set to -1 (default) to disable timeout.

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.GrabberTimeoutException`

Raised when a timeout occurs while attempting to read Grabber RTIO input events.

exception `artiq.coredevice.grabber.OutOfSyncException`

Raised when an incorrect number of ROI engine outputs has been retrieved from the RTIO input FIFO.

MANAGEMENT SYSTEM INTERFACE

ARTIQ makes certain provisions to allow interactions between different components when using the *management system*. An experiment may make requests of the master or clients using virtual devices to represent the necessary line of communication; applets may interact with databases, the dashboard, and directly with the user (through argument widgets). This page collects the references for these features.

23.1 In experiments

23.1.1 scheduler virtual device

The scheduler is exposed to the experiments via a virtual device called `scheduler`. It can be requested like any other device, and the methods below, as well as `pause()`, can be called on the returned object.

The scheduler virtual device also contains the attributes `rid`, `pipeline_name`, `priority` and `expid`, which 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.

23.1.2 ccb virtual device

Client control broadcasts (CCBs) 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 especially 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.

23.2 In applets

23.2.1 Applet request interfaces

Applet request interfaces allow applets to perform actions on the master database and set arguments in the dashboard. Applets may inherit from `artiq.applets.simple.SimpleApplet` and call the methods defined below through the `req` attribute.

Embedded applets should use `AppletRequestIPC`, while standalone applets use `AppletRequestRPC`. `SimpleApplet` automatically chooses the correct interface on initialization.

class `artiq.applets.simple._AppletRequestInterface`**append_to_dataset**(*key, value*)

Append to a dataset. See documentation of `append_to_dataset()`.

mutate_dataset(*key, index, value*)

Mutate a dataset. See documentation of `mutate_dataset()`.

set_argument_value(*expurl, key, value*)

Temporarily set the value of an argument in an experiment in the dashboard. The value resets to default value when recomputing the argument.

Parameters

- **expurl** – Experiment URL identifying the experiment in the dashboard. Example: ‘repo:ArgumentsDemo’.
- **key** – Name of the argument in the experiment.
- **value** – Object representing the new temporary value of the argument. For *Scannable* arguments, this parameter should be a *ScanObject*. The type of the *ScanObject* will be set as the selected type when this function is called.

set_dataset(*key, value, unit=None, scale=None, precision=None, persist=None*)

Set a dataset. See documentation of `set_dataset()`.

23.2.2 Applet entry area

Argument widgets can be used in applets through the *EntryArea* class. Below is a simple example code snippet:

```
entry_area = EntryArea()

# Create a new widget
entry_area.setattr_argument("bl", BooleanValue(True))

# Get the value of the widget (output: True)
print(entry_area.bl)

# Set the value
entry_area.set_value("bl", False)

# False
print(entry_area.bl)
```

The *EntryArea* object can then be added to a layout and integrated with the applet GUI. Multiple *EntryArea* objects can be used in a single applet.

class `artiq.gui.applets.EntryArea`**setattr_argument**(*name, proc, group=None, tooltip=None*)

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

Parameters

- **name** – Argument name
- **proc** – Argument processor, for example *NumberValue*
- **group** – Used to group together arguments in the GUI under a common category

- **tooltip** – Tooltip displayed when hovering over the entry widget

get_value(*name*)

Get the value of an entry widget.

Parameters

name – Argument name

get_values()

Get all values in the *EntryArea* as a dictionary. Names are stored as keys, and argument values as values.

set_value(*name*, *value*)

Set the value of an entry widget. The change is temporary and will reset to default when the reset button is clicked.

Parameters

- **name** – Argument name
- **value** – Object representing the new value of the argument. For *Scannable* arguments, this parameter should be a *ScanObject*. The type of the *ScanObject* will be set as the selected type when this function is called.

set_values(*values*)

Set multiple values from a dictionary input. Calls *set_value()* for each key-value pair.

Parameters

values – Dictionary with names as keys and new argument values as values.

24.1 ARTIQ Firmware Service (AFWS) client

This tool serves as a client for building tailored firmware and gateware from M-Lab's servers and downloading the binaries in ready-to-flash format. It is necessary to have a valid subscription to AFWS to use it. Subscription also includes general helpdesk support and can be purchased or extended by contacting sales@. One year of support is included with any Kasli carriers or crates containing them purchased from M-Labs. Additional one-time use is generally provided with purchase of additional cards to facilitate the system configuration change.

```
usage: afws_client [-h] [--server SERVER] [--port PORT] [--cert CERT]
                  username {build,passwd,get_variants,get_json} ...
```

24.1.1 Positional Arguments

username	user name for logging into AFWS
action	Possible choices: build, passwd, get_variants, get_json

24.1.2 Named Arguments

--server	server to connect to (default: 'afws.m-labs.hk')
--port	port to connect to (default: 80)
--cert	SSL certificate file used to authenticate server (default: use system certificates)

24.1.3 Sub-commands

build

build and download firmware

```
afws_client build [-h] [--major-ver MAJOR_VER] [--rev REV] [--log]
                  [--experimental EXPERIMENTAL]
                  directory [variant]
```

Positional Arguments

directory	output directory
variant	variant to build (can be omitted if user is authorised to build only one)

Named Arguments

--major-ver	ARTIQ major version
--rev	revision to build (default: currently installed ARTIQ revision)
--log	Display the build log
--experimental	enable an experimental feature (can be repeatedly specified to enable multiple features)

passwd

change password

Warning

After receiving your credentials from M-Labs, it is recommended to change your password as soon as possible. It is your responsibility to set and remember a secure password. If necessary, passwords can be reset by contacting helpdesk@.

```
afws_client passwd [-h]
```

get_variants

get available variants and expiry dates

```
afws_client get_variants [-h]
```

get_json

get JSON description file of variant

```
afws_client get_json [-h] [-o OUT] [-f] [variant]
```

Positional Arguments

variant	variant to get (can be omitted if user is authorised to build only one)
----------------	---

Named Arguments

-o, --out	output JSON file
-f, --force	overwrite file if it already exists

24.2 Static compiler

Compiles an experiment into an ELF file (or a TAR file if the experiment involves subkernels). It is primarily used to prepare binaries for the startup and idle kernels, 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.

```
usage: artiq_compile [-h] [--version] [-v] [-q] [--device-db DEVICE_DB]
                   [--dataset-db DATASET_DB] [-c CLASS_NAME] [-o OUTPUT]
                   FILE [ARGUMENTS ...]
```

24.2.1 Positional Arguments

FILE	file containing the experiment to compile
ARGUMENTS	run arguments

24.2.2 Named Arguments

--version	print the ARTIQ version number
--device-db	device database file (default: "device_db.py")
--dataset-db	dataset file (default: "dataset_db.mdb")
-c, --class-name	name of the class to compile
-o, --output	output file

24.2.3 verbosity

-v, --verbose	increase logging level
-q, --quiet	decrease logging level

24.3 Flash storage image generator

Compiles key/value pairs (e.g. configuration information) into a binary image suitable for flashing into the storage space of the core device. It can be used in combination with `artiq_flash` to configure the core device, but this is normally necessary at most to set the `ip` field; once the core device is reachable by network it is preferable to use `artiq_coremgmt` config. Not applicable to ARTIQ-Zynq, where preconfiguration is better achieved by loading `config.txt` onto the SD card.

```
usage: artiq_mkfs [-h] [-s KEY STRING] [-f KEY FILENAME] output
```

24.3.1 Positional Arguments

output	output file
---------------	-------------

24.3.2 Named Arguments

-s	add string
-f	add file contents

24.4 Flashing/Loading tool

Allows for flashing and loading of various files onto the core device. Not applicable to ARTIQ-Zynq, where gateway and firmware should be loaded onto the core device with `artiq_coremgmt`, directly copied onto the SD card, or (for developers) using the `ARTIQ netboot` utility.

```
usage: artiq_flash [-h] [--version] [-v] [-q] [-n] [-H HOSTNAME] [-J JUMP]
                  [-t TARGET] [-I PREINIT_COMMAND] [-f STORAGE] [-d DIR]
                  [--srcbuild]
                  [ACTION ...]
```

24.4.1 Positional Arguments

ACTION actions to perform, default: flash everything

24.4.2 Named Arguments

--version print the ARTIQ version number

-n, --dry-run only show the openocd script that would be run

-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 efc kc705

-I, --preinit-command add a pre-initialization OpenOCD command. Useful for selecting a board when several are connected.

-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

24.4.3 verbosity

-v, --verbose increase logging level

-q, --quiet decrease logging level

Valid actions:

- gateway: write main gateway bitstream to flash
- bootloader: write bootloader to flash
- storage: write storage image to flash
- firmware: write firmware to flash
- load: load main gateway bitstream into device (volatile but fast)
- erase: erase flash memory
- start: trigger the target to (re)load its gateway bitstream from flash. If your core device is reachable by network, prefer 'artiq_coremgmt reboot'.

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.

24.5 Core device management tool

The core management utility gives remote access to the core device logs, the *core device flash storage*, and other management functions.

To use this tool, it is necessary to specify the IP address your core device can be contacted at. If no option is used, the utility will assume there is a file named `device_db.py` in the current directory containing the *device database*; otherwise, a device database file can be provided with `--device-db` or an address directly with `--device` (see also below).

```
usage: artiq_coremgmt [-h] [--version] [-v] [-q] [--device-db DEVICE_DB]
                    [-D DEVICE]
                    {log,config,reboot,debug} ...
```

24.5.1 Positional Arguments

tool Possible choices: log, config, reboot, debug

24.5.2 Named Arguments

--version print the ARTIQ version number
--device-db device database file (default: "device_db.py")
-D, --device use specified core device address instead of reading device database

24.5.3 verbosity

-v, --verbose increase logging level
-q, --quiet decrease logging level

24.5.4 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

-f, --file key and file whose content to be written to core device config

remove

remove key from core device config

```
artiq_coremgmt config remove [-h] ...
```

Positional Arguments

KEY key to be removed from core device config

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]
```

24.6 Device database template generator

This tool generates a basic template for a *device database* given the JSON description file(s) for the system. Entries for *controllers* are not generated.

```
usage: artiq_ddb_template [-h] [--version] [-o OUTPUT]
                        [-s DESTINATION DESCRIPTION]
                        PRIMARY_DESCRIPTION
```

24.6.1 Positional Arguments

PRIMARY_DESCRIPTION JSON system description file for the primary (standalone or master) node

24.6.2 Named Arguments

--version print the ARTIQ version number
-o, --output output file, defaults to standard output if omitted

-s, --satellite add DRTIO satellite at the given destination number with devices from the given JSON description

24.7 RTIO channel name map tool

This tool encodes the map of RTIO channel numbers to names in a format suitable for writing to the config key `device_map`. See *Set up resolving RTIO channels to their names*.

```
usage: artiq_rtiomap [-h] [--version] [-v] [-q] [--device-db DEVICE_DB]
                   [--show]
                   FILE
```

24.7.1 Positional Arguments

FILE write the result into the specified file, or read from it to show the map (see `--show`)

24.7.2 Named Arguments

--version print the ARTIQ version number
--device-db device database file (default: `“device_db.py”`)
--show show the channel mapping from the specified file, instead of writing to it

24.7.3 verbosity

-v, --verbose increase logging level
-q, --quiet decrease logging level

24.8 Core device RTIO analyzer tool

This tool retrieves core device RTIO logs either as raw data or as VCD waveform files, which are readable by third-party tools such as GtkWave. See *RTIO analyzer* for an example, or `artiq.test.coredevice.test_analyzer` for a relevant unit test.

Using the management system, the respective functionality is provided by `aqctl_coreanalyzer_proxy` and the dashboard’s ‘Waveform’ tab; see *Waveform*.

```
usage: artiq_coreanalyzer [-h] [-v] [-q] [--device-db DEVICE_DB]
                          [-r READ_DUMP] [-p] [-w WRITE_VCD] [-d WRITE_DUMP]
                          [-u]
```

24.8.1 Named Arguments

--device-db device database file (default: `“device_db.py”`)
-r, --read-dump read raw dump file instead of accessing device
-p, --print-decoded print raw decoded messages
-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

24.8.2 verbosity

-v, --verbose increase logging level
-q, --quiet decrease logging level

24.9 DRTIO routing table manipulation tool

This tool allows for manipulation of a DRTIO routing table file, which can be transmitted to the core device using `artiq_coremgmt config write`; see *Configuring the routing table*.

```
usage: artiq_route [-h] FILE {init,show,set} ...
```

24.9.1 Positional Arguments

FILE	target file
action	Possible choices: init, show, set

24.9.2 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

24.10 ARTIQ RTIO monitor

Command-line interface for monitoring RTIO channels, as in the Monitor capacity of dashboard MonInj. See *Using MonInj*.

```
usage: artiq_rtiomon [-h] CORE_ADDR CHANNEL [CHANNEL ...]
```

24.10.1 Positional Arguments

CORE_ADDR hostname or IP address of the core device
CHANNEL channel(s) to monitor

24.11 MonInj proxy

ARTIQ moninj proxy

```
usage: aqctl_moninj_proxy [-h] [-v] [-q] [--bind BIND] [--no-localhost-bind]
                          [--port-proxy PORT_PROXY]
                          [--port-control PORT_CONTROL]
                          CORE_ADDR
```

24.11.1 Positional Arguments

CORE_ADDR hostname or IP address of the core device

24.11.2 verbosity

-v, --verbose increase logging level
-q, --quiet decrease logging level

24.11.3 network server

--bind additional hostname or IP address to bind to; use '*' to bind to all interfaces (default: [])
--no-localhost-bind do not implicitly also bind to localhost addresses
--port-proxy TCP port for proxying connections (default: 1383)
--port-control TCP port for control connections (default: 1384)

24.12 Core device RTIO analyzer proxy

ARTIQ core analyzer proxy

```
usage: aqctl_coreanalyzer_proxy [-h] [-v] [-q] [--bind BIND]
                                 [--no-localhost-bind]
                                 [--port-proxy PORT_PROXY]
                                 [--port-control PORT_CONTROL]
                                 CORE_ADDR
```

24.12.1 Positional Arguments

CORE_ADDR hostname or IP address of the core device

24.12.2 verbosity

-v, --verbose increase logging level

-q, --quiet decrease logging level

24.12.3 network server

--bind additional hostname or IP address to bind to; use '*' to bind to all interfaces (default: [])

--no-localhost-bind do not implicitly also bind to localhost addresses

--port-proxy TCP port for proxying connections (default: 1385)

--port-control TCP port for control connections (default: 1386)

24.13 Core device logging controller

ARTIQ controller for core device logs

```
usage: aqctl_corelog [-h] [-v] [-q] [--bind BIND] [--no-localhost-bind]
                   [-p PORT] [--simulation]
                   CORE_ADDR
```

24.13.1 Positional Arguments

CORE_ADDR hostname or IP address of the core device

24.13.2 Named Arguments

--simulation Simulation - does not connect to device

24.13.3 verbosity

-v, --verbose increase logging level

-q, --quiet decrease logging level

24.13.4 network server

--bind additional hostname or IP address to bind to; use '*' to bind to all interfaces (default: [])

--no-localhost-bind do not implicitly also bind to localhost addresses

-p, --port TCP port to listen on (default: 1068)

NIXOS FOR ARTIQ (PREINSTALL HANDBOOK)

This handbook assumes you are starting with a running [NixOS](#) system using the [defenestrate](#) NixOS configuration for ARTIQ/Sinara. Notably, if you've recently ordered an ARTIQ/Sinara crate from M-Labs together with an accompanying preinstalled computer, **this will be how your machine is set up**. Welcome! What follows is a short guide to the configuration itself, how to adapt it to your needs, and how to work with Nix and NixOS in general.

Tip

If you did not purchase a preinstalled computer from M-Labs, you can nonetheless reproduce the environment described here by installing NixOS and adopting the same configuration files. Specifically, the relevant files will be `configuration.nix` and `flake.nix` in the [defenestrate](#) repository linked above. See below for more information on different configuration files.

25.1 Getting started

Unless requested otherwise, ARTIQ NixOS machines ship with the desktop environment [GNOME](#), a popular choice across many Linux distributions. If you have used Linux before, you're likely already familiar with the interface. A set of basic desktop applications (Firefox, LibreOffice, Gimp) as well as some more specialized tools (Spyder, Jupyter, GTKWave, Wireshark) are also pre-installed and pre-configured. Later you will find that this list of installed packages is precisely specified in the `configuration.nix` file, and is very simple to edit to your needs.

Note

You will find multiple preinstalled options for a terminal emulator, including XTerm and GNOME's Console. The configuration's default shell is the [friendly interactive shell](#) or `fish`. This handbook will assume you have a basic familiarity with a Linux environment and commands like `sudo`, so if not, it's probably worth finding some materials on these things before you continue.

The hardware M-Labs ships for pre-installed ARTIQ machines is usually the [ASUS NUC 14RVK Pro Kit](#), with Intel i7-155H CPU, 32GB RAM, and a 1GB NVMe. Other options may also be available. Contact sales@ for details.

25.1.1 Logging in

The **default username** of the configuration is `rabi` (named after the physicist [Isidor Rabi](#).) The **default password** (also applicable for the root user) is `rabi`. Initially, the system is configured for autologin and will not require a password for either login or use of `sudo`.

Naturally, it's recommended to change this password immediately. You can use your preferred utility (e.g. the GNOME settings manager or the `passwd` command) to set a new password. You can also create new users as necessary. To turn off autologin, or require a password for `sudo`, see below in [Customizing your configuration](#).

25.1.2 Using ARTIQ

The most recent release version of ARTIQ is already pre-installed on your machine, not through flakes or profiles (as described in *Installing ARTIQ*) but in the NixOS configuration itself. Using ARTIQ is as simple as opening any terminal and typing any of the usual ARTIQ front-end or utility commands.

Note

To flash a core device by JTAG, or connect to the UART log, note the USB I/O on the front of the ASUS NUC, and follow the instructions in *Connecting to the UART log*. Pyserial and OpenOCD are pre-installed, and the default user is already included in the `plugdev` group.

25.1.3 Updating your system

To update all software system-wide, it isn't necessary to directly edit any of the configuration files. Instead, you can use `nix flake update`, e.g.:

```
$ cd /etc/nixos
$ sudo nix flake update
```

and build a new *generation*:

```
$ sudo nixos-rebuild boot
```

A reboot will be required for changes to take effect. See below to learn more about generations.

In order to update to a **new release version of ARTIQ** (or to the beta) system-wide, first change the source used in `flake.nix`:

```
inputs.artiq.url = git+https://git.m-labs.hk/M-Labs/artiq.git?ref=release-<number>;
```

(Remove `?ref=release-<number>` entirely for the beta branch). Then update and rebuild as above. Note however that `sudo nix flake update` and a rebuild will be required every time to incorporate new commits, and for easy access to the beta it may be better to use a flake, as described in *Installing ARTIQ*.

Tip

The `inputs.artiq.url` key can also be replaced with a path to a local clone of the ARTIQ repository on your system, e.g.:

```
path:/absolute/path/to/your/ARTIQ
```

if you'd like to be able to edit the source for your system-wide installation. Again, however, for such purposes it may be better to use flakes, rather than requiring a full rebuild for each update.

25.2 Nix and declarative configuration

Before you start customizing your configuration, it may be useful to take some time to understand why NixOS functions the way it does. The central concept which makes Nix different from most package managers, and NixOS from most Linux distributions, is a *declarative* philosophy of package and environment management. In practice, what this means is that rather than simply installing packages into your system, you *define upfront* the list of packages available in your environment (or for particular, specialized environments) in a centralized configuration file. Nix then evaluates those configuration files to see what packages it should *make* available in specific contexts.

Since Nix goes to considerable lengths to build packages in isolation from each other, and to specify precisely the versions used in any given environment or build, this means, first of all, that environments are (almost) perfectly replicable. It's always possible to recreate, for example, your user environment, or the ARTIQ development environment, in any other context, simply by copying or referencing the files which declare them. Isolation means installed packages can't 'clutter up' your system, and can always be removed or upgraded independently without affecting each other.

Finally, reproducibility also applies to your *past* configurations. That is, as long as you save your previous configuration files, Nix allows for 'perfect' rollback. If you make changes that break your system, you can always revert to an older configuration, and you'll be able to return to precisely the environment you had before you made changes.

25.2.1 NixOS configuration files

Your system-wide configuration files are stored in `/etc/nixos`. Because the *defenestrate* configuration uses `flakes`, there will be three files: `hardware-configuration.nix`, `flake.nix`, and `configuration.nix`.

The first, `hardware-configuration.nix`, is automatically generated by analyzing the actual hardware of the machine your NixOS is running. This is normally the only file which will differ if you choose to run the same NixOS configuration on different physical machines. Generally speaking, you shouldn't touch this file at all, unless you know what you're doing.

The second, `flake.nix`, is the root of the configuration. In *defenestrate*, it notably configures the URL for the ARTIQ repository.

The third, `configuration.nix`, is the usual configuration file for all NixOS variants. This is where most of the relevant settings for your environment are defined, including installed software.

Note

The `hardware-configuration.nix` file is usually generated, along with a mostly-blank `configuration.nix`, using the command `nixos-generate-config`. Generally this is done during installation, but in some situations, especially if you change your partitioning scheme, it's recommended to regenerate it rather than editing it manually. To look over the output that would be generated without actually changing any files, use:

```
$ nixos-generate-config --show-hardware-config
```

25.2.2 Generations and rollback

In order to change your configuration, simply editing `configuration.nix` is not enough. Once you're satisfied with your changes, it's necessary to *trigger a rebuild*, which creates a new 'generation', i.e., a new iterated version of your personal NixOS. Notably, when you boot your machine, you'll notice that the `systemd-boot` bootloader offers you a choice between all your saved generations before starting the operating system.

Tip

The `systemd` boot menu appears automatically for several seconds upon startup, but will relatively quickly continue with the default choice if not instructed otherwise. To prolong the time to make a choice, hit any arrow key and the timer will stop. Use arrow keys to make a selection and `ENTER` to continue.

To create a new generation, it's recommended to use the following command:

```
$ sudo nixos-rebuild boot
```

This will build the new generation *and* set it as default in the bootloader, but not activate it (switch into it) immediately. This is somewhat preferable for stability reasons. There are other options, including `nixos-rebuild switch`,

which activates the new generation immediately (though some settings will still require a reboot to take effect), and `nixos-rebuild test`, which generates and activates a rebuild but doesn't set it as default. See `nixos-rebuild --help` for more.

To rollback to an old generation, simply select it in the boot menu and boot into it.

Warning

Like most Nix artifacts, old generations will be saved indefinitely unless garbage collected manually. On the other hand, if you run garbage collection regularly, old generations will also be deleted. See [Garbage collection](#).

25.2.3 Declarative vs. imperative

As a further feature of NixOS's style of system management, it's useful to understand that a declarative configuration doesn't necessarily make it impossible to change elements of your system imperatively, i.e., conventionally.

Generally this is allowed purposefully. For example, in *defenestrate*, for ease of use, the `mutableUsers` option in `configuration.nix` is set to `true`; this means that passwords, groups, and additional users are all imperatively manageable, and the settings changed will be preserved across generation rebuilds. There are also many common system settings, such as network connections, available keyboard layouts, and aesthetic features like desktop backgrounds, which *can* be managed declaratively – in fact almost everything can be – but are usually left to conventional settings managers (e.g. GNOME Settings), which operate imperatively.

Normally, these imperative settings will carry across seamlessly to newly built generations, since they are stored on your disk, in exactly the way you are probably used to. On the other hand, they will **not** transfer if you rebuild your configuration on a different machine, and they also can't be rolled back through Nix.

Warning

Generally speaking, declarative configurations will overwrite imperative configurations upon a rebuild. For example, if you set `mutableUsers` to `false` in order to define users purely declaratively, the imperatively defined user list will be overwritten, and will not be retrievable by booting into an earlier generation.

25.3 Customizing your configuration

To customize your configuration, open `configuration.nix` in the editor of your choice. For changes to take effect, run `sudo nixos-rebuild boot` and reboot, as described above.

Note that `configuration.nix` is written in the *Nix language*, just as flakes are. You may therefore run into evaluation errors when rebuilding if you've misused the syntax. For relatively simple changes, the error messages should help you along; if you're interested in making more complex changes, you'll want to look into Nix itself. See also [Further resources](#) below.

For all `configuration.nix` options and their effects, see the [list of options](#) in the official NixOS manual. (This list is very long, and not worth reading through in its entirety. Use CTRL+F to search for specific options).

25.3.1 Basic settings

Some basic settings should be immediately readable, such as timezone, host name, default locale, default key layout, and so on. If you ordered your machine from M-Labs, they will likely already be adapted to your requests, but you can also find them in `configuration.nix`, under the following names:

```
networking.hostName = "artiq";

console.font = "Lat2-Terminus16";
console.keyMap = "us";
i18n.defaultLocale = "en_US.UTF-8";

time.timeZone = "UTC";
```

(not necessarily in this order, or this proximity – the order in which options are defined is arbitrary, and makes no difference to the system).

25.3.2 User management and login

Autologin is defined via the following options:

```
services.displayManager.autoLogin.enable = true;
services.displayManager.autoLogin.user = "rabi";
```

Set `enable` to `false` or delete both lines to disable autologin.

To change the behavior of `sudo`, use the following option:

```
security.sudo.wheelNeedsPassword = false;
```

Setting it to `true` will require a password for use of `sudo`. The default password for root is also `rabi`. The default username can also be changed by editing the relevant options. Other changes (new users, etc.) can safely be made imperatively.

See also

Alternatively, you may choose to do all your user management declaratively. See [User Management](#) in the NixOS manual.

You may also be interested in changing the default user shell, if you prefer a particular alternative to `fish`. This is done with the `users.defaultUserShell` option.

25.3.3 Installing additional packages

Broadly speaking, the software available in your system is supplied via a list assigned to a single option, `environment.systemPackages`. A simple example may look like:

```
environment.systemPackages = with pkgs; [
  vim
  wget
  gitAndTools.gitFull
  firefox
  gnome3.gnome-tweaks
  libreoffice-fresh
  vscodium
];
```

Note

with `pkgs` is a convenience of the Nix language signifying that all these packages originate in the `pkgs` input, which is here specified in `flake.nix` as:

```
inputs.nixpkgs.url = github:NixOS/nixpkgs/nixos-24.05;
```

i.e. the central Nixpkgs repository for all Nix packages. Without `with pkgs` the list would have to be given as `pkgs.wget`, `pkgs.vim`, `pkgs.gitAndTools.gitFull`, and so on.

To find out what packages are available, you can use [Nixpkgs search](#) in your browser. To install it in your environment, simply add the name of the package to the `systemPackages` list and rebuild. You can always test the package in your system first by obtaining packages ad-hoc with `nix-shell`; see below.

Rarely, some software may require additional options to be set in order to function as expected. In particular, if a service or daemon needs to be configured, or if additional permissions or groups are necessary, it's often the case that the corresponding NixOS module needs to be enabled. For example:

```
services.openssh.enable = true;
```

both installs the `openssh` package and configures and starts the `sshd` service.

25.4 Nix and NixOS tips

25.4.1 Installing additional packages (ad-hoc)

Often you might find yourself wanting access to a tool or a piece of software for a single use case or a limited time, without really wanting to install it into your system permanently. Nix provides a very convenient solution for this. Any package in the [Nixpkgs repository](#) can be installed temporarily, into a particular shell, with the command:

```
$ nix-shell -p <pkg-name>
```

Warning

Somewhat unfortunately, the two commands `nix-shell` and `nix shell` both exist in NixOS and **are not synonymous**. Note in this case the use of the hyphen. To start a shell from a *flake*, use `nix shell` with no hyphen. A similar distinction applies for `nix-build` and `nix build`.

Much as with flakes, the command may take some time at first use for large packages, but if you end up calling repeatedly, subsequent invocations will reference the Nix store and run almost instantly. Also much as with flakes, the package is not 'installed' in a permanent sense and will disappear once the shell is closed. This is also useful in order to be able to test packages quickly before installing them into the environment.

25.4.2 Flakes and builds

Running NixOS is perfectly compatible with the other Nix features used by ARTIQ. In particular, using `nix shell` or `nix develop` with the various ARTIQ flakes, as described in [Installing ARTIQ](#) and [\(Re\)flashing your core device](#), can continue to be very useful, especially to access different versions of ARTIQ, even simultaneously. It is also perfectly possible to install packages into your Nix profile. All of these are ways to define and make use of certain environments, with access to certain sets of packages, none of which will overwrite each other.

Note that, for instance, if you have installed ARTIQ `release-8` into your environment, but run `nix shell` on the ARTIQ `release-7` flake, that *specific shell* will run `release-7` commands, whereas all others will continue to use

the system default environment, with `release-8`. You can check this kind of behavior yourself with `--version`.

25.4.3 Garbage collection

As also noted in *Installing ARTIQ*, Nix stores all packages it encounters in `/nix/store`, which generally ensures that any given version of a package only needs to be fetched and built once, even when reused repeatedly or in different environments. Old NixOS generations are also stored, and will remain available in the boot menu upon future boots. With time, however, this can start to occupy large amounts of storage space. To clear out the Nix store and free up more space, run:

```
$ sudo nix-collect-garbage
```

To clear old generations, it's also necessary to run:

```
$ sudo nix-collect-garbage --delete-old
```

If you'd like to preserve the possibility of rollback, one option is to use `--delete-older-than`, for example:

```
$ sudo nix-collect-garbage --delete-older-than=30d
```

which will only delete generations older than thirty days. To save configurations in a more permanent way, you can save old versions of `configuration.nix`, `flake.nix`, and `flake.lock`.

25.4.4 Further resources

- [The Nix package lookup](#)
- [The NixOS manual](#), in particular the [list of configuration options](#)
- If you are interested in learning to use itself Nix in more detail, [nix.dev](#) and [noogle](#)
- Various official NixOS sources, including an [official wiki](#)

FAQ (HOW DO I...)

26.1 use this documentation?

The content of this manual is arranged in rough reading order. If you start at the beginning and make your way through section by section, you should form a pretty good idea of how ARTIQ works and how to use it. Otherwise:

If you are just starting out, and would like to get ARTIQ set up on your computer and your core device, start with *Installing ARTIQ*, *(Re)flashing your core device*, and *Networking and configuration*, in that order.

If you have a working ARTIQ setup (or someone else has set it up for you), start with the tutorials: read *ARTIQ Real-Time I/O concepts*, then progress to *Getting started with the core device*, *Using the management system*, and *Data and user interfaces*. If your system is in a DRTIO configuration, *DRTIO and subkernels* will also be helpful.

Pages like *Management system* and *Core device* describe **specific components of the ARTIQ ecosystem** in more detail. If you want to understand more about device and dataset databases, for example, read the *Environment* page; if you want to understand the ARTIQ Python dialect and everything it does or does not support, read the *Compiler* page.

Reference pages, like *Main front-end tools* and *Core real-time drivers*, contain the detailed documentation of the individual methods and command-line tools ARTIQ provides. They are heavily interlinked throughout the rest of the documentation: whenever a method, tool, or exception is mentioned by name, like *artiq_run*, *now_mu()*, or *RTIOUnderflow*, it can normally be clicked on to directly access the reference material. Notice also that the online version of this manual is searchable; see the ‘Search docs’ bar at left.

26.2 build this documentation?

To generate this manual from source, you can use `nix build` directives, for example:

```
$ nix build git+https://git.m-labs.hk/M-Labs/artiq.git\?ref=release-[number]#artiq-  
↪manual-html
```

Substitute `artiq-manual-pdf` to get the LaTeX PDF version. The results will be in `result`.

The manual is written in **reStructured Text**; you can find the source files in the ARTIQ repository under `doc/manual`. If you spot a mistake, a typo, or something that’s out of date or missing – in particular, if you want to add something to this FAQ – feel free to clone the repository, edit the source RST files, and make a pull request with your version of an improvement. (If you’re not a fan of or not familiar with command-line Git, Gitea supports making edits and pull requests directly in the web interface; tutorial materials are easy to find online.) The second best thing is to open an issue to make M-Labs aware of the problem.

26.3 roll back to older versions of ARTIQ, or obtain it through other installation methods?

At all times, three versions of ARTIQ are actively supported by M-Labs, released through the beta, stable, and legacy channels. See [ARTIQ Releases](#).

If you are trying to rollback to stable or legacy, the process should be accordingly simple. See the respective [Installing ARTIQ](#) page in the respective version of the manual. If you've previously used the version you are rolling back to, you can likely use the rollback methods described in [Upgrading ARTIQ](#); otherwise you can always treat it as a fresh install. Remember that it will also be necessary to reflash core devices with corresponding legacy binaries.

Regarding pre-legacy releases, note that being actively supported simply means that M-Labs makes prebuilt packages and binaries for these versions available via the supported installation methods and through AFWS. Outdated versions aren't automatically built or offered over these channels, but their source code remains available in the Git repository, and you are free to use it or adapt it in accordance with the terms of the license, including building whatever packages you prefer. In general, though, newer releases of ARTIQ offer more features, more stability, better performance, and better support. The legacy release is supported simply as a convenience for users who haven't been able to upgrade yet. For normal purposes, it is recommended to use the current stable release of ARTIQ if at all possible, or the beta to gain access to new features and improvements that are still in development.

For more details, see also [Clarifications regarding the ARTIQ release model and AFWS](#).

Tip

If you're particularly concerned with being able to precisely reproduce older experiments, even when you've moved on to newer ARTIQ versions, upgrade carefully and make your own local backups to be able to rollback to older versions of your system. Make sure to keep copies of older firmware binaries in order to be able to reflash your hardware. Older versions of ARTIQ will always continue working if left untouched, and you won't need to worry about rebuilding from the source if you keep your own prebuilt versions around.

26.4 troubleshoot networking problems?

Diagnosis aids:

- Can you ping the device?
- Is the Ethernet LED on?
- Is the ERROR LED on?
- Is there anything unusual recorded in *the UART log*?

Some things to consider:

- Is the `core_addr` field of your `device_db.py` set correctly?
- Did your device flash and boot successfully? Were the binaries generated for the correct board hardware version?
- Are your core device's IP address and networking configurations definitely set correctly? Check the UART log to confirm, and talk to your network administrator about what the correct choices are.
- Is your core device configured for an external reference clock? If so, it cannot function correctly without one. Is the external reference clock plugged in?
- Are Ethernet and (on Kasli only) SFP0 plugged in all the way? Are they working? Try different cables and SFP adapters; M-Labs tests with CAT6 cables, but lower categories should be supported too.
- Are your PC and your crate in the same subnet?

- Is some other device in your network already using the configured IP address? Turn off the core device and try pinging the configured IP address; if it responds, you have a culprit. One of the two will need a different networking configuration.
- Are there restrictions or issues in your router or subnet that are preventing the core device from connecting? It may help to try connecting the core device to your PC directly.

26.5 fix ‘no startup kernel found’ / ‘no idle kernel found’ in the core log?

Don't. Note that these are INFO messages, and not ERROR or even WARN. If you haven't flashed an idle or startup kernel yet, this is normal, and will not cause any problems; between experiments the core device will simply do nothing. The same applies to most other messages in the style of ‘no configuration found’ or ‘falling back to default’. Your system will generally run just fine on its defaults until you get around to setting these configurations, though certain features may be limited until properly set up. See *Networking and configuration* and the list of keys in core-device-flash-storage.

26.6 fix ‘Mismatch between gateway and software versions’?

Either reflash your core device with a newer version of ARTIQ (see *(Re)flashing your core device*) or update your software (see *Upgrading ARTIQ*), depending on which is out of date.

Note

You can check the specific versions you are using at any time by comparing the gateway version given in the core startup log and the output given by adding `--version` to any of the standard ARTIQ front-end commands. This is especially useful when e.g. seeking help in the forum or at the helpdesk, where your running ARTIQ version is often crucial information to diagnose a problem.

Minor version mismatches are common, even in stable ARTIQ versions, but should not cause any issues. The ARTIQ release system ensures breaking changes are strictly limited to new release versions, or to the beta branch (which explicitly makes no promises of stability.) Updates that *are* applied to the stable version are usually bug fixes, documentation improvements, or other quality-of-life changes. As long as gateway and software are using the same stable release version of ARTIQ, even if there is a minor mismatch, no warning will be displayed.

26.7 change configuration settings of satellite devices?

Currently, it is not possible to reach satellites through `artiq_coremgmt config`, although this is being worked on. On Kasli, use `artiq_mkfs` and `artiq_flash`; on Kasli-SoC, preload the SD card with a `config.txt`, formatted as a list of `key=value` pairs, one per line.

Don't worry about individually flashing idle or startup kernels. If your idle or startup kernel contains subkernels, it will automatically compile as a `.tar`, which you only need to flash to the master.

26.8 fix unreliable DRTIO master-satellite links?

Inconsistent DRTIO connections, especially with odd or absent errors in the core logs, are often a symptom of overheating either in the master or satellite boards. Check the core device fans for failure or defects. Improve air circulation around the crate or attach additional fans to see if that improves or resolves the issue. In the long term, fan trays to be rack-mounted together with the crate are a clean solution to these kinds of problems.

26.9 add or remove EEM peripherals or DRTIO satellites?

Adding new real-time hardware to an ARTIQ system almost always means reflashing the core device; if you are adding new satellite core devices, they will have to be flashed as well. If you have obtained your upgrades from M-Labs or QUARTIQ, updated binaries and reflashing support will normally be offered to you directly. In any other case, track down your JSON system description file(s), bring them up to date with the updated state of your system, and see *Building and developing ARTIQ*.

Once you have an updated set of binaries, reflash the core device, following the instructions in *(Re)flashing your core device*. Be sure to update your device database before starting experimentation; run `artiq_ddb_template` on your system description(s) to update the local devices, and copy over any aliases or entries for NDSP controllers you may have been using. Note that the device database is a Python file, and the generated file of local devices can also simply be imported into the final version, allowing for dynamic modifications, especially in complex systems that may have multiple device databases in use.

26.10 see command-line help?

Like most if not almost all terminal utilities, ARTIQ commands, tools and applets print their help messages directly into the terminal and exit when run with the flag `--help` or `-h`:

```
$ artiq_run -h
```

This is the simplest and most direct way of accessing the same usage and reference material that is replicated in this manual on the pages *Main front-end tools* and *Utilities*.

26.11 find ARTIQ examples?

The official examples are stored in the `examples` folder of the ARTIQ package. You can find the location of the ARTIQ package on your machine with:

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

Copy the `examples` folder from that path into your home or user directory, and start experimenting! (Note that some examples have dependencies not included with a standard ARTIQ install, like `matplotlib` and `numba`. To run those examples properly, make sure those modules are accessible.)

If you have progressed past this level and would like to see more in-depth code or real-life examples of how other groups have handled running experiments with ARTIQ, see the “Community code” directory on the M-labs [resources page](#).

26.12 fix failed to connect to moninj in the dashboard?

This and other similar messages almost always indicate that your device database lists controllers (for example, `aqctl_moninj_proxy`) that either haven’t been started or aren’t reachable at the given host and port. See *Non-RTIO devices and the controller manager*, or simply run:

```
$ artiq_ctlmgr
```

to let the controller manager start the necessary controllers automatically.

26.13 fix address already in use when running ARTIQ commands?

A message like `OSError: [Errno 98] error while attempting to bind on address ('127.0.0.1', 1067): [errno 98] address already in use` indicates that the IP address and port number combination you're trying to use is already occupied by some other process. Often this simply means that the ARTIQ process you're trying to start is in fact already running. Note for example that trying to start a controller which is already being run by a controller manager will generally fail for this reason.

Note

ARTIQ management system communications, whether distributed or local, run over TCP/IP, using TCP port numbers to identify their destinations. Generally speaking, client processes like the dashboard don't require fixed ports of their own, since they can simply reach out to the master when they want to establish a connection. Running multiple dashboards will never cause a port conflict. On the other hand, server processes like the ARTIQ master have to be 'listening' at a fixed, open port in order to be able to receive incoming connections. For more details, look into [ports in computer networking](#).

Most management system processes belong to the second category, and are bound to one or several fixed communication ports while they're running. See also [Default network ports](#).

You can use the command `netstat` to list the ports currently in use on your system. To check the status of a specific port on Linux, try either of:

```
$ netstat -anp --inet | grep "<port-number>"
$ lsof -i:<port-number>
```

On Windows, you can list ports with:

```
$ netstat -ano -p TCP
```

Use your preferred method to search through the output; suitable commands will vary by environment (e.g. `grep` in an MSYS2 shell, `Select-String` in PowerShell, `find` in the Windows command line, etc.)

In all cases, if there are no results, the port isn't in use and should be free for new processes.

Tip

While it is possible to run, for example, two identical ARTIQ controllers on the same machine, they can't be bound to the same port numbers at the same time. If you're intentionally running multiple copies of the same ARTIQ processes, use the command-line `--port` options to set alternate ports for at least one of the two. See [Main front-end tools](#) and [Utilities](#) for exact flags to use. Controllers should have similar flags available and will also require updated [device database entries](#). Note that alternate ports must be consistent to be useful, e.g., a master and dashboard must have the same `--port-notify` set in order to communicate with each other!

Otherwise, either the running process must be stopped, or you'll have to set different port numbers for the process you're trying to start. In some cases it might happen that a process is no longer accessible or has become unresponsive but is still occupying its ports. The easiest way to free the ports is to kill the process manually. On Linux, you can use the `kill` command with `lsof`:

```
$ kill $(lsof -t -i:<port-number>)
```

On Windows, use `netstat` again to identify the process ID, and then feed it into `taskkill`, e.g.:

```
$ netstat -ano -p TCP
$ taskkill /F /PID <process-ID>
```

26.14 diagnose and fix sequence errors?

Go through your code, keeping manual track of SED lanes. See the following example:

```
@kernel
def run(self):
    self.core.reset()
    with parallel:
        self.ttl0.on() # lane0
        self.ttl_sma.pulse(800*us) # lane1(rising) lane1(falling)
    with sequential:
        self.ttl1.on() # lane2
        self.ttl2.on() # lane3
        self.ttl3.on() # lane4
        self.ttl4.on() # lane5
        delay(800*us)
        self.ttl1.off() # lane5
        self.ttl2.off() # lane6
        self.ttl3.off() # lane7
        self.ttl4.off() # lane0
    self.ttl0.off() # lane1 -> clashes with the falling edge of ttl_sma,
                    # which is already at +800us
```

In most cases, as in this one, it's relatively easy to rearrange the generation of events so that they will be better spread out across SED lanes without sacrificing actual functionality. One possible solution for the above sequence looks like:

```
@kernel
def run(self):
    self.core.reset()
    self.ttl0.on() # lane0
    self.ttl_sma.on() # lane1
    self.ttl1.on() # lane2
    self.ttl2.on() # lane3
    self.ttl3.on() # lane4
    self.ttl4.on() # lane5
    delay(800*us)
    self.ttl1.off() # lane5
    self.ttl2.off() # lane6
    self.ttl3.off() # lane7
    self.ttl4.off() # lane0 (no clash: new timestamp is higher than last)
    self.ttl_sma.off() # lane1
    self.ttl0.off() # lane2
```

In this case, the `pulse()` is split up into its component `on()` and `off()` so that events can be generated more linearly. It can also be worth keeping in mind that delaying by even a single coarse RTIO cycle between events avoids switching SED lanes at all; in contexts where perfect simultaneity is not a priority, this is an easy way to avoid sequencing issues. See again *Sequence errors*.

26.15 understand applet commands?

The ‘Command’ field contains the exact terminal command used to open and operate the applet. The default `{artiq_applet}` prefix simply translates to something to the effect of `python -m artiq.applets.`, intended to be immediately followed by the applet module name. The options suffixed after the module name are the same used in the command line, and a list of them can be shown by using the standard command line `-h` help flag:

```
$ python -m artiq.applets.plot_xy -h
```

in any terminal.

26.16 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.

26.17 organize applets in groups?

Create groups by left-clicking within the applet list and selecting ‘New Group’. Move applets in and out of groups by dragging them with the mouse. To unselect an applet or a group, use `CTRL+click`.

26.18 organize experiment windows in the dashboard?

Experiment windows can be organized by using the following hotkeys:

- `CTRL+SHIFT+T` to tile experiment windows
- `CTRL+SHIFT+C` to cascade experiment windows

The windows will be organized in the order they were last interacted with.

26.19 fix errors when restarting management system after a crash?

On Windows in particular, abnormal shutdowns such as power outages or bluescreens sometimes corrupt the organizational files used by the management system, resulting in errors to the tune of `ValueError: source code string cannot contain null bytes` when restarting. The easiest way to handle these problems is to delete the corrupted files and start from scratch. Note that GUI configuration `.pyon` files are kept in the user configuration directory, see below at [find the dashboard and browser configuration files?](#)

26.20 create and use variable-length arrays in kernels?

You can’t, in general; see the corresponding notes under *ARTIQ types*. ARTIQ kernels do not support heap allocation, meaning in particular that lists, arrays, and strings must be of constant size. One option is to preallocate everything, as mentioned on the Compiler page; another option is to chunk it and e.g. read 100 events per function call, push them upstream and retry until the gate time closes.

26.21 understand how best to send data between kernel and host?

See also *Basic ARTIQ Python*. Let’s run down the options for kernel-host data transfer:

- Kernels can return single values directly. They *cannot* return lists, arrays or strings, because of the way these values are allocated, which prevents values of these types from outliving the kernel they

are created in. This is still true when the values in question are wrapped in functions or objects, in which case they may be missed by lifetime tracking and accepted by the compiler, but will cause memory corruption when run.

- Kernels can freely make changes to attributes of objects shared with the host, including `self`. However, these changes will be made to a kernel-owned copy of the object, which is only synchronized with the host copy when the kernel completes. This means that host-side operations executed during the runtime of the kernel, including RPCs, will be handling an unmodified version of the object, and modifications made by those operations will simply be overwritten when the kernel returns.

Note

Attribute writeback happens *once per kernel*, that is, if your experiment contains many separate kernels called from the host, modifications will be written back when each separate kernel completes. This is generally not suitable for data transfer, however, as new kernels are costly to create, and experiments often try to avoid doing so. It is also important to specify that kernels called *from* a kernel will not write back to the host upon completion. Attribute writeback is only executed upon return to the host.

- Kernels can interact with datasets, either as attributes (if `setattr_dataset()` is used) or by RPC of the get and set methods (`get_dataset()`, `set_dataset()`, etc.). In this case note that, like certain other host-side methods, `get_dataset()` will not actually be accepted by the compiler, because its return type is not specified. To call it as an RPC, simply wrap it in another function which *does* specify a return type. `set_dataset()` can be similarly wrapped to make it asynchronous.
- Kernels can of course also call arbitrary RPCs. When sending data to the host, these can be asynchronous, and this is normally the recommended way of transferring data back to the host, resulting in a relatively minor amount of delay in the kernel. Keep in mind however that asynchronous RPCs may still block execution for some time if the arguments are very large or if many RPCs are submitted in close succession. When receiving data from the host, RPCs must be synchronous, which is still considerably faster than starting a new kernel. Note that if data is being both (asynchronously) sent and received, there is a small possibility of minor race conditions (i.e. retrieved data may not yet show updates sent in an earlier RPC).

Kernel attributes and data transfer remain somewhat of an open area of development. Many such developments are or will be implemented in **NAC3**, the next-generation ARTIQ compiler. The overhead for starting new kernels, which is largely dominated by compile time, should be significantly reduced (NAC3 can be expected to complete compilations 6x - 30x faster than currently).

26.22 write part of my experiment as a coroutine/asyncio task/generator?

You cannot 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.

26.23 determine the pyserial URL to connect 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
```

This will give you the `/dev/ttyUSBxx` (or `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 for selecting the serial device by its USB vendor ID, product ID and/or serial number. These 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>"` ` .
```

26.24 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:

- `t1_out`: any output-only TTL.
- `t1_out_serdes`: any output-only TTL that uses a SERDES (i.e. has a fine timestamp). Can be aliased to `t1_out`.
- `loop_out`: any output-only TTL. Must be physically connected to `loop_in`. Can be aliased to `t1_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.

26.25 find the dashboard and browser configuration files?

```
python -c "from artiq.tools import get_user_config_dir; print(get_user_config_dir())"
```


ADDITIONAL RESOURCES

27.1 Other related documentation

- the [Sinara wiki](#)
- the [SiPyCo manual](#)
- the [Migen manual](#)
- in a pinch, the [M-labs internal docs](#)

For more advanced questions, sometimes the [list of publications](#) about experiments performed using ARTIQ may be interesting. See also the official M-Labs [resources](#) page, especially the section on community code.

27.2 “Help, I’ve done my best and I can’t get any further!”

- If you have an active M-Labs AFWS/support subscription, you can email helpdesk@ at any time for personalized assistance. Please include the following information:
 - Your installed ARTIQ version (add `--version` to any of the standard ARTIQ commands)
 - The variant name of your system (refer to the sticker on the crate if you aren’t sure)
 - The recent output of your core log, either through `artiq_coremgmt` (if you’re able to contact your device by network), or over UART following [the guide here](#)
 - How your problem happened, and what you’ve already tried to fix it
- Compare your materials with the examples; see also [finding ARTIQ examples](#) above.
- Check the list of [active issues](#) on the ARTIQ Gitea repository for possible known problems with ARTIQ. Search through the closed issues to see if your question or concern has been addressed before.
- Search the [M-Labs forum](#) for similar problems, or make a post asking for help yourself.
- Look into the [Mattermost live chat](#) or the bridged IRC channel.
- Read the open source code and its docstrings and figure it out.
- If you’re reasonably certain you’ve identified a bug, or if you’d like to suggest a feature that should be included in future ARTIQ releases, [file a Gitea issue](#) yourself, following one of the provided templates.
- In some odd cases, you may want to see the [mailing list archive](#); the ARTIQ mailing list was shut down at the end of 2020 and was last regularly used during the time of ARTIQ-2 and 3, but for some older ARTIQ features, or to understand a development thought process, you may still find relevant information there.

In any situation, if you found the manual unclear or unhelpful, you might consider following the [directions for contribution](#) and editing it to be more helpful for future readers.

PYTHON MODULE INDEX

a

- artiq.coredevice.ad53xx, 188
- artiq.coredevice.ad9910, 154
- artiq.coredevice.ad9912, 164
- artiq.coredevice.ad9914, 166
- artiq.coredevice.adf5356, 171
- artiq.coredevice.almazny, 169
- artiq.coredevice.cache, 138
- artiq.coredevice.core, 135
- artiq.coredevice.dma, 137
- artiq.coredevice.edge_counter, 143
- artiq.coredevice.exceptions, 137
- artiq.coredevice.fastino, 195
- artiq.coredevice.grabber, 211
- artiq.coredevice.i2c, 149
- artiq.coredevice.mirny, 168
- artiq.coredevice.novogorny, 193
- artiq.coredevice.phaser, 173
- artiq.coredevice.sampler, 192
- artiq.coredevice.shuttler, 198
- artiq.coredevice.spi2, 146
- artiq.coredevice.suservo, 205
- artiq.coredevice.ttl, 139
- artiq.coredevice.urukul, 151
- artiq.coredevice.zotino, 192
- artiq.frontend.afws_client, 217
- artiq.frontend.aqctl_coreanalyzer_proxy, 226
- artiq.frontend.aqctl_corelog, 227
- artiq.frontend.aqctl_moninj_proxy, 226
- artiq.frontend.artiq_browser, 125
- artiq.frontend.artiq_client, 121
- artiq.frontend.artiq_compile, 218
- artiq.frontend.artiq_coreanalyzer, 224
- artiq.frontend.artiq_coremgmt, 220
- artiq.frontend.artiq_dashboard, 124
- artiq.frontend.artiq_ddb_template, 223
- artiq.frontend.artiq_flash, 219
- artiq.frontend.artiq_master, 119
- artiq.frontend.artiq_mkfs, 219
- artiq.frontend.artiq_route, 225
- artiq.frontend.artiq_rtiomap, 224
- artiq.frontend.artiq_rtiomon, 225
- artiq.frontend.artiq_run, 119
- artiq.frontend.artiq_session, 125
- artiq.language.core, 127
- artiq.language.environment, 129
- artiq.language.scan, 133
- artiq.language.units, 134
- artiq_comtools.artiq_ctlmgr, 126

Symbols

`_AppletRequestInterface` (class in `artiq.applets.simple`), 214

A

`AD53xx` (class in `artiq.coredevice.ad53xx`), 188

`ad53xx_cmd_read_ch()` (in module `artiq.coredevice.ad53xx`), 191

`ad53xx_cmd_write_ch()` (in module `artiq.coredevice.ad53xx`), 191

`AD9910` (class in `artiq.coredevice.ad9910`), 154

`AD9912` (class in `artiq.coredevice.ad9912`), 164

`AD9914` (class in `artiq.coredevice.ad9914`), 166

`ADC` (class in `artiq.coredevice.shuttler`), 198

`adc_channel()` (in module `artiq.coredevice.novogorny`), 195

`adc_ctrl()` (in module `artiq.coredevice.novogorny`), 195

`adc_data()` (in module `artiq.coredevice.novogorny`), 195

`adc_mu_to_volt()` (in module `artiq.coredevice.sampler`), 193

`adc_mu_to_volts()` (in module `artiq.coredevice.suservo`), 211

`adc_softspan()` (in module `artiq.coredevice.novogorny`), 195

`adc_value()` (in module `artiq.coredevice.novogorny`), 195

`ADF5356` (class in `artiq.coredevice.adf5356`), 171

`AlmaznyChannel` (class in `artiq.coredevice.almazny`), 169

`AlmaznyLegacy` (class in `artiq.coredevice.almazny`), 170

`amplitude_to_asf()` (`artiq.coredevice.ad9910.AD9910` method), 155

`amplitude_to_asf()` (`artiq.coredevice.ad9914.AD9914` method), 166

`amplitude_to_ram()` (`artiq.coredevice.ad9910.AD9910` method), 155

`analyze()` (`artiq.language.environment.Experiment` method), 133

`append_to_dataset()` (`artiq.applets.simple._AppletRequestInterface` method), 215

`append_to_dataset()` (`artiq.language.environment.HasEnvironment` method), 132

`AppletsCCBDock` (class in `artiq.dashboard.applets_ccb`), 214

`apply_cic()` (`artiq.coredevice.fastino.Fastino` method), 195

`artiq.coredevice.ad53xx` module, 188

`artiq.coredevice.ad9910` module, 154

`artiq.coredevice.ad9912` module, 164

`artiq.coredevice.ad9914` module, 166

`artiq.coredevice.adf5356` module, 171

`artiq.coredevice.almazny` module, 169

`artiq.coredevice.cache` module, 138

`artiq.coredevice.core` module, 135

`artiq.coredevice.dma` module, 137

`artiq.coredevice.edge_counter` module, 143

`artiq.coredevice.exceptions` module, 137

`artiq.coredevice.fastino` module, 195

`artiq.coredevice.grabber` module, 211

`artiq.coredevice.i2c` module, 149

`artiq.coredevice.mirny` module, 168

- artiq.coredevice.novogorny module, 193
 - artiq.coredevice.phaser module, 173
 - artiq.coredevice.sampler module, 192
 - artiq.coredevice.shuttler module, 198
 - artiq.coredevice.spi2 module, 146
 - artiq.coredevice.suservo module, 205
 - artiq.coredevice.ttl module, 139
 - artiq.coredevice.urukul module, 151
 - artiq.coredevice.zotino module, 192
 - artiq.frontend.afws_client module, 217
 - artiq.frontend.aqctl_coreanalyzer_proxy module, 226
 - artiq.frontend.aqctl_corelog module, 227
 - artiq.frontend.aqctl_moninj_proxy module, 226
 - artiq.frontend.artiq_browser module, 125
 - artiq.frontend.artiq_client module, 121
 - artiq.frontend.artiq_compile module, 218
 - artiq.frontend.artiq_coreanalyzer module, 224
 - artiq.frontend.artiq_coremgmt module, 220
 - artiq.frontend.artiq_dashboard module, 124
 - artiq.frontend.artiq_ddb_template module, 223
 - artiq.frontend.artiq_flash module, 219
 - artiq.frontend.artiq_master module, 119
 - artiq.frontend.artiq_mkfs module, 219
 - artiq.frontend.artiq_route module, 225
 - artiq.frontend.artiq_rtiomap module, 224
 - artiq.frontend.artiq_rtiomon module, 225
 - artiq.frontend.artiq_run module, 119
 - artiq.frontend.artiq_session module, 125
 - artiq.gui.applets.EntryArea (*built-in class*), 215
 - artiq.language.core module, 127
 - artiq.language.environment module, 129
 - artiq.language.scan module, 133
 - artiq.language.units module, 134
 - artiq_comtools.artiq_ctlmgr module, 126
 - asf_to_amplitude() (*artiq.coredevice.ad9910.AD9910 method*), 155
 - asf_to_amplitude() (*artiq.coredevice.ad9914.AD9914 method*), 167
 - at_mu() (*in module artiq.language.core*), 129
 - att_to_mu() (*artiq.coredevice.almazny.AlmaznyLegacy method*), 170
 - att_to_mu() (*artiq.coredevice.mirny.Mirny method*), 169
 - att_to_mu() (*artiq.coredevice.urukul.CPLD method*), 152
- ## B
- BooleanValue (*class in artiq.language.environment*), 129
 - break_realttime() (*artiq.coredevice.core.Core method*), 135
 - build() (*artiq.language.environment.HasEnvironment method*), 130
 - burst_mu() (*artiq.coredevice.novogorny.Novogorny method*), 194
- ## C
- CacheError, 137
 - cal_trf_vco() (*artiq.coredevice.phaser.PhaserChannel method*), 183
 - calculate_pll() (*in module artiq.coredevice.adf5356*), 173
 - calibrate() (*artiq.coredevice.ad53xx.AD53xx method*), 188
 - calibrate() (*artiq.coredevice.shuttler.ADC method*), 198
 - call_child_method() (*artiq.language.environment.HasEnvironment method*), 130
 - CancelledArgsError, 133
 - ccb_create_applet() (*artiq.dashboard.applets_ccb.AppletsCCBDock method*), 214

- `ccb_disable_applet()` (*artiq.dashboard.applets_ccb.AppletsCCBDock method*), 214
- `ccb_disable_applet_group()` (*artiq.dashboard.applets_ccb.AppletsCCBDock method*), 214
- `CenterScan` (class in *artiq.language.scan*), 133
- `cfg_sw()` (*artiq.coredevice.ad9910.AD9910 method*), 155
- `cfg_sw()` (*artiq.coredevice.ad9912.AD9912 method*), 164
- `cfg_sw()` (*artiq.coredevice.urukul.CPLD method*), 152
- `cfg_switches()` (*artiq.coredevice.urukul.CPLD method*), 152
- `cfg_write()` (*artiq.coredevice.urukul.CPLD method*), 152
- `Channel` (class in *artiq.coredevice.suservo*), 205
- `check_pause()` (*artiq.master.scheduler.Scheduler method*), 213
- `check_termination()` (*artiq.master.scheduler.Scheduler method*), 213
- `clear_dac_alarms()` (*artiq.coredevice.phaser.Phaser method*), 179
- `clear_smp_err()` (*artiq.coredevice.ad9910.AD9910 method*), 155
- `ClockFailure`, 137
- `close()` (*artiq.coredevice.core.Core method*), 135
- `CompileError`, 135
- `Config` (class in *artiq.coredevice.shuttler*), 200
- `configure()` (*artiq.coredevice.novogorny.Novogorny method*), 194
- `Core` (class in *artiq.coredevice.core*), 135
- `CoreCache` (class in *artiq.coredevice.cache*), 138
- `CoreDMA` (class in *artiq.coredevice.dma*), 137
- `CoreException` (class in *artiq.coredevice.exceptions*), 137
- `count()` (*artiq.coredevice.ttl.TTLInOut method*), 140
- `CounterOverflow`, 144
- `CPLD` (class in *artiq.coredevice.urukul*), 151
- ## D
- `dac_iotest()` (*artiq.coredevice.phaser.Phaser method*), 179
- `dac_read()` (*artiq.coredevice.phaser.Phaser method*), 179
- `dac_sync()` (*artiq.coredevice.phaser.Phaser method*), 179
- `dac_tune_fifo_offset()` (*artiq.coredevice.phaser.Phaser method*), 179
- `dac_write()` (*artiq.coredevice.phaser.Phaser method*), 179
- `DCBias` (class in *artiq.coredevice.shuttler*), 202
- `DDS` (class in *artiq.coredevice.shuttler*), 202
- `dds_offset_to_mu()` (*artiq.coredevice.suservo.Channel method*), 205
- `DefaultMissing`, 129
- `delay()` (in module *artiq.language.core*), 129
- `delay_mu()` (in module *artiq.language.core*), 129
- `delete()` (*artiq.master.scheduler.Scheduler method*), 213
- `disable_output()` (*artiq.coredevice.adf5356.ADF5356 method*), 171
- `DMAError`, 137
- `DMARecordContextManager` (class in *artiq.coredevice.dma*), 138
- `duc_stb()` (*artiq.coredevice.phaser.Phaser method*), 180
- ## E
- `EdgeCounter` (class in *artiq.coredevice.edge_counter*), 144
- `en_trf_out()` (*artiq.coredevice.phaser.PhaserChannel method*), 183
- `enable()` (*artiq.coredevice.shuttler.Relay method*), 204
- `enable_output()` (*artiq.coredevice.adf5356.ADF5356 method*), 171
- `encode()` (*artiq.coredevice.phaser.Miqro method*), 175
- `EnumerationValue` (class in *artiq.language.environment*), 129
- `EnvExperiment` (class in *artiq.language.environment*), 133
- `erase()` (*artiq.coredevice.dma.CoreDMA method*), 137
- `exit_x()` (*artiq.coredevice.ad9914.AD9914 method*), 167
- `Experiment` (class in *artiq.language.environment*), 132
- `ExplicitScan` (class in *artiq.language.scan*), 134
- ## F
- `f_pfd()` (*artiq.coredevice.adf5356.ADF5356 method*), 171
- `f_vco()` (*artiq.coredevice.adf5356.ADF5356 method*), 171
- `Fastino` (class in *artiq.coredevice.fastino*), 195
- `fetch_count()` (*artiq.coredevice.edge_counter.EdgeCounter method*), 144
- `fetch_timestamped_count()` (*artiq.coredevice.edge_counter.EdgeCounter method*), 145
- `frequency_to_div()` (*artiq.coredevice.spi2.SPIMaster method*), 147
- `frequency_to_ftw()` (*artiq.coredevice.ad9910.AD9910 method*), 156
- `frequency_to_ftw()` (*artiq.coredevice.ad9912.AD9912 method*), 156

- 164
frequency_to_ftw() (*artiq.coredevice.ad9914.AD9914* method), 167
- frequency_to_ftw()* (*artiq.coredevice.ttl.TTLClockGen* method), 139
- frequency_to_ram()* (*artiq.coredevice.ad9910.AD9910* method), 156
- frequency_to_xftw()* (*artiq.coredevice.ad9914.AD9914* method), 167
- ftw_to_frequency()* (*artiq.coredevice.ad9910.AD9910* method), 156
- ftw_to_frequency()* (*artiq.coredevice.ad9912.AD9912* method), 164
- ftw_to_frequency()* (*artiq.coredevice.ad9914.AD9914* method), 167
- ftw_to_frequency()* (*artiq.coredevice.ttl.TTLClockGen* method), 139
- ## G
- gate_both()* (*artiq.coredevice.edge_counter.EdgeCounter* method), 145
- gate_both()* (*artiq.coredevice.ttl.TTLInOut* method), 140
- gate_both_mu()* (*artiq.coredevice.edge_counter.EdgeCounter* method), 145
- gate_both_mu()* (*artiq.coredevice.ttl.TTLInOut* method), 140
- gate_falling()* (*artiq.coredevice.edge_counter.EdgeCounter* method), 145
- gate_falling()* (*artiq.coredevice.ttl.TTLInOut* method), 141
- gate_falling_mu()* (*artiq.coredevice.edge_counter.EdgeCounter* method), 145
- gate_falling_mu()* (*artiq.coredevice.ttl.TTLInOut* method), 141
- gate_rising()* (*artiq.coredevice.edge_counter.EdgeCounter* method), 145
- gate_rising()* (*artiq.coredevice.ttl.TTLInOut* method), 141
- gate_rising_mu()* (*artiq.coredevice.edge_counter.EdgeCounter* method), 145
- gate_rising_mu()* (*artiq.coredevice.ttl.TTLInOut* method), 141
- gate_roi()* (*artiq.coredevice.grabber.Grabber* method), 211
- gate_roi_pulse()* (*artiq.coredevice.grabber.Grabber* method), 211
- get()* (*artiq.coredevice.ad9910.AD9910* method), 156
- get()* (*artiq.coredevice.ad9912.AD9912* method), 164
- get()* (*artiq.coredevice.cache.CoreCache* method), 138
- get()* (*artiq.coredevice.i2c.PCF8574A* method), 149
- get_adc()* (*artiq.coredevice.suservo.SUServo* method), 209
- get_adc_mu()* (*artiq.coredevice.suservo.SUServo* method), 209
- get_amplitude()* (*artiq.coredevice.ad9910.AD9910* method), 156
- get_argument()* (*artiq.language.environment.HasEnvironment* method), 130
- get_asf()* (*artiq.coredevice.ad9910.AD9910* method), 156
- get_att()* (*artiq.coredevice.ad9910.AD9910* method), 156
- get_att()* (*artiq.coredevice.ad9912.AD9912* method), 165
- get_att_mu()* (*artiq.coredevice.ad9910.AD9910* method), 156
- get_att_mu()* (*artiq.coredevice.ad9912.AD9912* method), 165
- get_att_mu()* (*artiq.coredevice.phaser.PhaserChannel* method), 183
- get_att_mu()* (*artiq.coredevice.urukul.CPLD* method), 152
- get_channel_att()* (*artiq.coredevice.urukul.CPLD* method), 152
- get_channel_att_mu()* (*artiq.coredevice.urukul.CPLD* method), 152
- get_crc_err()* (*artiq.coredevice.phaser.Phaser* method), 180
- get_dac_alarms()* (*artiq.coredevice.phaser.Phaser* method), 180
- get_dac_data()* (*artiq.coredevice.phaser.PhaserChannel* method), 183
- get_dac_temperature()* (*artiq.coredevice.phaser.Phaser* method), 180
- get_dataset()* (*artiq.language.environment.HasEnvironment* method), 132
- get_dataset_metadata()* (*artiq.language.environment.HasEnvironment* method), 132
- get_device()* (*artiq.language.environment.HasEnvironment* method), 131
- get_device_db()* (*artiq.language.environment.HasEnvironment* method), 131
- get_frequency()* (*artiq.coredevice.ad9910.AD9910* method), 156

- [get_ftw\(\)](#) (*artiq.coredevice.ad9910.AD9910* method), 157
[get_gain\(\)](#) (*artiq.coredevice.shuttler.Config* method), 201
[get_gains_mu\(\)](#) (*artiq.coredevice.sampler.Sampler* method), 192
[get_handle\(\)](#) (*artiq.coredevice.dma.CoreDMA* method), 137
[get_mu\(\)](#) (*artiq.coredevice.ad9910.AD9910* method), 157
[get_mu\(\)](#) (*artiq.coredevice.ad9912.AD9912* method), 165
[get_next_frame_mu\(\)](#) (*artiq.coredevice.phaser.Phaser* method), 180
[get_offset\(\)](#) (*artiq.coredevice.shuttler.Config* method), 201
[get_phase\(\)](#) (*artiq.coredevice.ad9910.AD9910* method), 157
[get_pow\(\)](#) (*artiq.coredevice.ad9910.AD9910* method), 157
[get_profile_mu\(\)](#) (*artiq.coredevice.suservo.Channel* method), 205
[get_rtio_counter_mu\(\)](#) (*artiq.coredevice.core.Core* method), 135
[get_rtio_destination_status\(\)](#) (*artiq.coredevice.core.Core* method), 136
[get_sta\(\)](#) (*artiq.coredevice.phaser.Phaser* method), 180
[get_status\(\)](#) (*artiq.coredevice.suservo.SUServo* method), 210
[get_status\(\)](#) (*artiq.master.scheduler.Scheduler* method), 213
[get_value\(\)](#) (*artiq.gui.applets.EntryArea* method), 216
[get_values\(\)](#) (*artiq.gui.applets.EntryArea* method), 216
[get_y\(\)](#) (*artiq.coredevice.suservo.Channel* method), 205
[get_y_mu\(\)](#) (*artiq.coredevice.suservo.Channel* method), 205
[Grabber](#) (class in *artiq.coredevice.grabber*), 211
[GrabberTimeoutException](#), 212
- ## H
- [HasEnvironment](#) (class in *artiq.language.environment*), 130
[host_only\(\)](#) (in module *artiq.language.core*), 128
- ## I
- [i2c_poll\(\)](#) (in module *artiq.coredevice.i2c*), 150
[i2c_read_byte\(\)](#) (in module *artiq.coredevice.i2c*), 150
[i2c_read_many\(\)](#) (in module *artiq.coredevice.i2c*), 150
[i2c_write_byte\(\)](#) (in module *artiq.coredevice.i2c*), 150
[i2c_write_many\(\)](#) (in module *artiq.coredevice.i2c*), 151
[I2CError](#), 137
[I2CSwitch](#) (class in *artiq.coredevice.i2c*), 149
[info\(\)](#) (*artiq.coredevice.adf5356.ADF5356* method), 171
[init\(\)](#) (*artiq.coredevice.ad53xx.AD53xx* method), 189
[init\(\)](#) (*artiq.coredevice.ad9910.AD9910* method), 157
[init\(\)](#) (*artiq.coredevice.ad9912.AD9912* method), 165
[init\(\)](#) (*artiq.coredevice.ad9914.AD9914* method), 167
[init\(\)](#) (*artiq.coredevice.adf5356.ADF5356* method), 172
[init\(\)](#) (*artiq.coredevice.fastino.Fastino* method), 196
[init\(\)](#) (*artiq.coredevice.mirny.Mirny* method), 169
[init\(\)](#) (*artiq.coredevice.phaser.Phaser* method), 180
[init\(\)](#) (*artiq.coredevice.sampler.Sampler* method), 193
[init\(\)](#) (*artiq.coredevice.shuttler.Relay* method), 204
[init\(\)](#) (*artiq.coredevice.suservo.SUServo* method), 210
[init\(\)](#) (*artiq.coredevice.urukul.CPLD* method), 153
[init_sync\(\)](#) (*artiq.coredevice.ad9914.AD9914* method), 167
[input\(\)](#) (*artiq.coredevice.ttl.TTLInOut* method), 141
[input_mu\(\)](#) (*artiq.coredevice.grabber.Grabber* method), 211
[interactive\(\)](#) (*artiq.language.environment.HasEnvironment* method), 131
[InternalError](#), 137
[io_rst\(\)](#) (*artiq.coredevice.urukul.CPLD* method), 153
- ## K
- [kernel\(\)](#) (in module *artiq.language.core*), 127
[kernel_from_string\(\)](#) (in module *artiq.language.core*), 128
- ## L
- [load\(\)](#) (*artiq.coredevice.ad53xx.AD53xx* method), 189
- ## M
- [measure_frame_timestamp\(\)](#) (*artiq.coredevice.phaser.Phaser* method), 180
[measure_io_update_alignment\(\)](#) (*artiq.coredevice.ad9910.AD9910* method), 157
[Miqro](#) (class in *artiq.coredevice.phaser*), 173
[Mirny](#) (class in *artiq.coredevice.mirny*), 168
[module](#)
 - [artiq.coredevice.ad53xx](#), 188
 - [artiq.coredevice.ad9910](#), 154
 - [artiq.coredevice.ad9912](#), 164
 - [artiq.coredevice.ad9914](#), 166
 - [artiq.coredevice.adf5356](#), 171
 - [artiq.coredevice.almazny](#), 169
 - [artiq.coredevice.cache](#), 138

- artiq.coredevice.core, 135
 - artiq.coredevice.dma, 137
 - artiq.coredevice.edge_counter, 143
 - artiq.coredevice.exceptions, 137
 - artiq.coredevice.fastino, 195
 - artiq.coredevice.grabber, 211
 - artiq.coredevice.i2c, 149
 - artiq.coredevice.mirny, 168
 - artiq.coredevice.novogorny, 193
 - artiq.coredevice.phaser, 173
 - artiq.coredevice.sampler, 192
 - artiq.coredevice.shuttler, 198
 - artiq.coredevice.spi2, 146
 - artiq.coredevice.suservo, 205
 - artiq.coredevice.ttl, 139
 - artiq.coredevice.urukul, 151
 - artiq.coredevice.zotino, 192
 - artiq.frontend.afws_client, 217
 - artiq.frontend.aqctl_coreanalyzer_proxy, 226
 - artiq.frontend.aqctl_corelog, 227
 - artiq.frontend.aqctl_moninj_proxy, 226
 - artiq.frontend.artiq_browser, 125
 - artiq.frontend.artiq_client, 121
 - artiq.frontend.artiq_compile, 218
 - artiq.frontend.artiq_coreanalyzer, 224
 - artiq.frontend.artiq_coremgmt, 220
 - artiq.frontend.artiq_dashboard, 124
 - artiq.frontend.artiq_ddb_template, 223
 - artiq.frontend.artiq_flash, 219
 - artiq.frontend.artiq_master, 119
 - artiq.frontend.artiq_mkfs, 219
 - artiq.frontend.artiq_route, 225
 - artiq.frontend.artiq_rtiomap, 224
 - artiq.frontend.artiq_rtiomon, 225
 - artiq.frontend.artiq_run, 119
 - artiq.frontend.artiq_session, 125
 - artiq.language.core, 127
 - artiq.language.environment, 129
 - artiq.language.scan, 133
 - artiq.language.units, 134
 - artiq_comtools.artiq_ctlmgr, 126
 - mu_to_att() (artiq.coredevice.almazny.AlmaznyLegacy method), 170
 - mu_to_att() (artiq.coredevice.urukul.CPLD method), 153
 - mu_to_seconds() (artiq.coredevice.core.Core method), 136
 - MultiScanManager (class in artiq.language.scan), 134
 - mutate_dataset() (artiq.applets.simple._AppletRequestInterface method), 215
 - mutate_dataset() (artiq.language.environment.HasEnvironment method), 131
- ## N
- NoDefault (class in artiq.language.environment), 129
 - NoScan (class in artiq.language.scan), 133
 - Novogorny (class in artiq.coredevice.novogorny), 193
 - now_mu() (in module artiq.language.core), 129
 - NRTSPIMaster (class in artiq.coredevice.spi2), 146
 - NumberValue (class in artiq.language.environment), 129
- ## O
- off() (artiq.coredevice.ttl.TTLInOut method), 141
 - off() (artiq.coredevice.ttl.TTLOut method), 143
 - on() (artiq.coredevice.ttl.TTLInOut method), 141
 - on() (artiq.coredevice.ttl.TTLOut method), 143
 - OutOfSyncException, 212
 - output() (artiq.coredevice.ttl.TTLInOut method), 142
 - output_divider() (artiq.coredevice.adf5356.ADF5356 method), 172
 - output_power_mu() (artiq.coredevice.adf5356.ADF5356 method), 172
 - output_toggle() (artiq.coredevice.almazny.AlmaznyLegacy method), 170
- ## P
- PCF8574A (class in artiq.coredevice.i2c), 149
 - Phaser (class in artiq.coredevice.phaser), 177
 - PhaserChannel (class in artiq.coredevice.phaser), 182
 - PhaserOscillator (class in artiq.coredevice.phaser), 187
 - playback() (artiq.coredevice.dma.CoreDMA method), 138
 - playback_handle() (artiq.coredevice.dma.CoreDMA method), 138
 - pll_frac1() (artiq.coredevice.adf5356.ADF5356 method), 172
 - pll_frac2() (artiq.coredevice.adf5356.ADF5356 method), 172
 - pll_mod2() (artiq.coredevice.adf5356.ADF5356 method), 172
 - pll_n() (artiq.coredevice.adf5356.ADF5356 method), 172
 - portable() (in module artiq.language.core), 127
 - pow_to_turns() (artiq.coredevice.ad9910.AD9910 method), 157
 - pow_to_turns() (artiq.coredevice.ad9912.AD9912 method), 165
 - pow_to_turns() (artiq.coredevice.ad9914.AD9914 method), 167
 - power_down() (artiq.coredevice.ad9910.AD9910 method), 157

- power_down() (*artiq.coredevice.shuttler.ADC method*), 199
- power_up() (*artiq.coredevice.shuttler.ADC method*), 199
- precompile() (*artiq.coredevice.core.Core method*), 136
- prepare() (*artiq.language.environment.EnvExperiment method*), 133
- prepare() (*artiq.language.environment.Experiment method*), 132
- pulse() (*artiq.coredevice.phaser.Miqro method*), 175
- pulse() (*artiq.coredevice.ttl.TTLInOut method*), 142
- pulse() (*artiq.coredevice.ttl.TTLOut method*), 143
- pulse_mu() (*artiq.coredevice.phaser.Miqro method*), 175
- pulse_mu() (*artiq.coredevice.ttl.TTLInOut method*), 142
- pulse_mu() (*artiq.coredevice.ttl.TTLOut method*), 143
- put() (*artiq.coredevice.cache.CoreCache method*), 138
- PYONValue (*class in artiq.language.environment*), 129
- ## R
- RangeScan (*class in artiq.language.scan*), 133
- read() (*artiq.coredevice.ad9912.AD9912 method*), 165
- read() (*artiq.coredevice.fastino.Fastino method*), 196
- read() (*artiq.coredevice.spi2.SPIMaster method*), 147
- read() (*artiq.coredevice.suservo.SUServo method*), 210
- read16() (*artiq.coredevice.ad9910.AD9910 method*), 158
- read16() (*artiq.coredevice.shuttler.ADC method*), 199
- read24() (*artiq.coredevice.shuttler.ADC method*), 199
- read32() (*artiq.coredevice.ad9910.AD9910 method*), 158
- read32() (*artiq.coredevice.phaser.Phaser method*), 180
- read64() (*artiq.coredevice.ad9910.AD9910 method*), 158
- read8() (*artiq.coredevice.phaser.Phaser method*), 181
- read8() (*artiq.coredevice.shuttler.ADC method*), 199
- read_ch() (*artiq.coredevice.shuttler.ADC method*), 199
- read_id() (*artiq.coredevice.shuttler.ADC method*), 200
- read_muxout() (*artiq.coredevice.adf5356.ADF5356 method*), 172
- read_ram() (*artiq.coredevice.ad9910.AD9910 method*), 158
- read_reg() (*artiq.coredevice.ad53xx.AD53xx method*), 189
- read_reg() (*artiq.coredevice.mirny.Mirny method*), 169
- record() (*artiq.coredevice.dma.CoreDMA method*), 138
- ref_counter() (*artiq.coredevice.adf5356.ADF5356 method*), 172
- Relay (*class in artiq.coredevice.shuttler*), 203
- request_termination() (*artiq.master.scheduler.Scheduler method*), 213
- reset() (*artiq.coredevice.core.Core method*), 136
- reset() (*artiq.coredevice.phaser.Miqro method*), 175
- reset() (*artiq.coredevice.shuttler.ADC method*), 200
- rpc() (*in module artiq.language.core*), 127
- RTIODestinationUnreachable, 137
- RTI00verflow, 137
- RTI0Underflow, 137
- run() (*artiq.language.environment.Experiment method*), 132
- ## S
- sample() (*artiq.coredevice.novogorny.Novogorny method*), 194
- sample() (*artiq.coredevice.sampler.Sampler method*), 193
- sample_get() (*artiq.coredevice.ttl.TTLInOut method*), 142
- sample_get_nonrt() (*artiq.coredevice.ttl.TTLInOut method*), 142
- sample_input() (*artiq.coredevice.ttl.TTLInOut method*), 142
- sample_mu() (*artiq.coredevice.novogorny.Novogorny method*), 194
- sample_mu() (*artiq.coredevice.sampler.Sampler method*), 193
- Sampler (*class in artiq.coredevice.sampler*), 192
- Scannable (*class in artiq.language.scan*), 134
- ScanObject (*class in artiq.language.scan*), 133
- Scheduler (*class in artiq.master.scheduler*), 213
- seconds_to_mu() (*artiq.coredevice.core.Core method*), 136
- set() (*artiq.coredevice.ad9910.AD9910 method*), 158
- set() (*artiq.coredevice.ad9912.AD9912 method*), 165
- set() (*artiq.coredevice.ad9914.AD9914 method*), 167
- set() (*artiq.coredevice.almazny.AlmaznyChannel method*), 170
- set() (*artiq.coredevice.i2c.I2CSwitch method*), 149
- set() (*artiq.coredevice.i2c.PCF8574A method*), 149
- set() (*artiq.coredevice.i2c.TCA6424A method*), 150
- set() (*artiq.coredevice.suservo.Channel method*), 206
- set() (*artiq.coredevice.ttl.TTLClockGen method*), 139
- set_all_att_mu() (*artiq.coredevice.urukul.CPLD method*), 153
- set_amplitude() (*artiq.coredevice.ad9910.AD9910 method*), 158
- set_amplitude_phase() (*artiq.coredevice.phaser.PhaserOscillator method*), 187
- set_amplitude_phase_mu() (*artiq.coredevice.phaser.PhaserOscillator method*), 187
- set_argument_value() (*artiq.applets.simple._AppletRequestInterface method*), 215

set_asf() (*artiq.coredevice.ad9910.AD9910* method), 159
 set_att() (*artiq.coredevice.ad9910.AD9910* method), 159
 set_att() (*artiq.coredevice.ad9912.AD9912* method), 165
 set_att() (*artiq.coredevice.adf5356.ADF5356* method), 172
 set_att() (*artiq.coredevice.almazny.AlmaznyLegacy* method), 171
 set_att() (*artiq.coredevice.mirny.Mirny* method), 169
 set_att() (*artiq.coredevice.phaser.PhaserChannel* method), 183
 set_att() (*artiq.coredevice.urukul.CPLD* method), 153
 set_att_mu() (*artiq.coredevice.ad9910.AD9910* method), 159
 set_att_mu() (*artiq.coredevice.ad9912.AD9912* method), 166
 set_att_mu() (*artiq.coredevice.adf5356.ADF5356* method), 172
 set_att_mu() (*artiq.coredevice.almazny.AlmaznyLegacy* method), 171
 set_att_mu() (*artiq.coredevice.mirny.Mirny* method), 169
 set_att_mu() (*artiq.coredevice.phaser.PhaserChannel* method), 184
 set_att_mu() (*artiq.coredevice.urukul.CPLD* method), 153
 set_cfg() (*artiq.coredevice.fastino.Fastino* method), 196
 set_cfg() (*artiq.coredevice.phaser.Phaser* method), 181
 set_cfr1() (*artiq.coredevice.ad9910.AD9910* method), 159
 set_cfr2() (*artiq.coredevice.ad9910.AD9910* method), 159
 set_clr() (*artiq.coredevice.shuttler.Config* method), 201
 set_config() (*artiq.coredevice.edge_counter.EdgeCounter* method), 145
 set_config() (*artiq.coredevice.spi2.SPIMaster* method), 147
 set_config() (*artiq.coredevice.suservo.SUServo* method), 210
 set_config_mu() (*artiq.coredevice.spi2.NRTSPIMaster* method), 146
 set_config_mu() (*artiq.coredevice.spi2.SPIMaster* method), 148
 set_continuous() (*artiq.coredevice.fastino.Fastino* method), 196
 set_dac() (*artiq.coredevice.ad53xx.AD53xx* method), 189
 set_dac() (*artiq.coredevice.fastino.Fastino* method), 197
 set_dac_cmix() (*artiq.coredevice.phaser.Phaser* method), 181
 set_dac_mu() (*artiq.coredevice.ad53xx.AD53xx* method), 189
 set_dac_mu() (*artiq.coredevice.fastino.Fastino* method), 197
 set_dac_test() (*artiq.coredevice.phaser.PhaserChannel* method), 184
 set_dataset() (*artiq.applets.simple._AppletRequestInterface* method), 215
 set_dataset() (*artiq.language.environment.HasEnvironment* method), 131
 set_dds() (*artiq.coredevice.suservo.Channel* method), 206
 set_dds_mu() (*artiq.coredevice.suservo.Channel* method), 206
 set_dds_offset() (*artiq.coredevice.suservo.Channel* method), 206
 set_dds_offset_mu() (*artiq.coredevice.suservo.Channel* method), 207
 set_default_scheduling() (*artiq.language.environment.HasEnvironment* method), 132
 set_duc_cfg() (*artiq.coredevice.phaser.PhaserChannel* method), 184
 set_duc_frequency() (*artiq.coredevice.phaser.PhaserChannel* method), 184
 set_duc_frequency_mu() (*artiq.coredevice.phaser.PhaserChannel* method), 184
 set_duc_phase() (*artiq.coredevice.phaser.PhaserChannel* method), 184
 set_duc_phase_mu() (*artiq.coredevice.phaser.PhaserChannel* method), 184
 set_fan() (*artiq.coredevice.phaser.Phaser* method), 181
 set_fan_mu() (*artiq.coredevice.phaser.Phaser* method), 181
 set_frequency() (*artiq.coredevice.ad9910.AD9910* method), 160
 set_frequency() (*artiq.coredevice.adf5356.ADF5356* method), 172
 set_frequency() (*artiq.coredevice.phaser.PhaserOscillator* method), 187
 set_frequency_mu() (*artiq.coredevice.phaser.PhaserOscillator* method), 188
 set_ftw() (*artiq.coredevice.ad9910.AD9910* method),

- 160
- `set_gain()` (*artiq.coredevice.shuttler.Config* method), 201
- `set_gain_mu()` (*artiq.coredevice.novogorny.Novogorny* method), 194
- `set_gain_mu()` (*artiq.coredevice.sampler.Sampler* method), 193
- `set_group()` (*artiq.coredevice.fastino.Fastino* method), 197
- `set_group_mu()` (*artiq.coredevice.fastino.Fastino* method), 197
- `set_hold()` (*artiq.coredevice.fastino.Fastino* method), 197
- `set_iir()` (*artiq.coredevice.phaser.PhaserChannel* method), 184
- `set_iir()` (*artiq.coredevice.suservo.Channel* method), 207
- `set_iir_mu()` (*artiq.coredevice.phaser.PhaserChannel* method), 185
- `set_iir_mu()` (*artiq.coredevice.suservo.Channel* method), 207
- `set_leds()` (*artiq.coredevice.fastino.Fastino* method), 197
- `set_leds()` (*artiq.coredevice.phaser.Phaser* method), 181
- `set_leds()` (*artiq.coredevice.zotino.Zotino* method), 192
- `set_mu()` (*artiq.coredevice.ad9910.AD9910* method), 160
- `set_mu()` (*artiq.coredevice.ad9912.AD9912* method), 166
- `set_mu()` (*artiq.coredevice.ad9914.AD9914* method), 167
- `set_mu()` (*artiq.coredevice.almazny.AlmaznyChannel* method), 170
- `set_mu()` (*artiq.coredevice.ttl.TTLClockGen* method), 139
- `set_nco_frequency()` (*artiq.coredevice.phaser.PhaserChannel* method), 186
- `set_nco_frequency_mu()` (*artiq.coredevice.phaser.PhaserChannel* method), 186
- `set_nco_phase()` (*artiq.coredevice.phaser.PhaserChannel* method), 186
- `set_nco_phase_mu()` (*artiq.coredevice.phaser.PhaserChannel* method), 186
- `set_offset()` (*artiq.coredevice.shuttler.Config* method), 201
- `set_output_power_mu()` (*artiq.coredevice.adf5356.ADF5356* method), 172
- `set_pgia_mu()` (*artiq.coredevice.suservo.SUServo* method), 210
- `set_phase()` (*artiq.coredevice.ad9910.AD9910* method), 161
- `set_phase_mode()` (*artiq.coredevice.ad9910.AD9910* method), 161
- `set_phase_mode()` (*artiq.coredevice.ad9914.AD9914* method), 168
- `set_pow()` (*artiq.coredevice.ad9910.AD9910* method), 162
- `set_profile()` (*artiq.coredevice.phaser.Miqro* method), 175
- `set_profile()` (*artiq.coredevice.urukul.CPLD* method), 153
- `set_profile_mu()` (*artiq.coredevice.phaser.Miqro* method), 175
- `set_profile_ram()` (*artiq.coredevice.ad9910.AD9910* method), 162
- `set_servo()` (*artiq.coredevice.phaser.PhaserChannel* method), 186
- `set_sync()` (*artiq.coredevice.ad9910.AD9910* method), 162
- `set_sync_div()` (*artiq.coredevice.urukul.CPLD* method), 154
- `set_sync_dly()` (*artiq.coredevice.phaser.Phaser* method), 182
- `set_time_manager()` (in module *artiq.language.core*), 128
- `set_value()` (*artiq.gui.applets.EntryArea* method), 216
- `set_values()` (*artiq.gui.applets.EntryArea* method), 216
- `set_waveform()` (*artiq.coredevice.shuttler.DCBias* method), 202
- `set_waveform()` (*artiq.coredevice.shuttler.DDS* method), 203
- `set_window()` (*artiq.coredevice.phaser.Miqro* method), 176
- `set_window_mu()` (*artiq.coredevice.phaser.Miqro* method), 176
- `set_x()` (*artiq.coredevice.ad9914.AD9914* method), 168
- `set_x_mu()` (*artiq.coredevice.ad9914.AD9914* method), 168
- `set_y()` (*artiq.coredevice.suservo.Channel* method), 208
- `set_y_mu()` (*artiq.coredevice.suservo.Channel* method), 208
- `setattr_argument()` (*artiq.gui.applets.EntryArea* method), 215
- `setattr_argument()` (*artiq.language.environment.HasEnvironment* method), 131
- `setattr_dataset()` (*artiq.language.environment.HasEnvironment* method), 132

- setattr_device()** (*artiq.language.environment.HasEnvironment method*), 131
setup_roi() (*artiq.coredevice.grabber.Grabber method*), 211
shuttler_volt_to_mu() (*in module artiq.coredevice.shuttler*), 204
single_conversion() (*artiq.coredevice.shuttler.ADC method*), 200
spi_cfg() (*artiq.coredevice.phaser.Phaser method*), 182
spi_read() (*artiq.coredevice.phaser.Phaser method*), 182
spi_write() (*artiq.coredevice.phaser.Phaser method*), 182
SPIError, 137
SPIMaster (*class in artiq.coredevice.spi2*), 146
sta_read() (*artiq.coredevice.urukul.CPLD method*), 154
stage_cic() (*artiq.coredevice.fastino.Fastino method*), 197
stage_cic_mu() (*artiq.coredevice.fastino.Fastino method*), 198
standby() (*artiq.coredevice.shuttler.ADC method*), 200
stop() (*artiq.coredevice.ttl.TTLClockGen method*), 139
StringValue (*class in artiq.language.environment*), 130
subkernel() (*in module artiq.language.core*), 127
SubkernelError, 137
submit() (*artiq.master.scheduler.Scheduler method*), 213
SUServo (*class in artiq.coredevice.suservo*), 208
sync() (*artiq.coredevice.adf5356.ADF5356 method*), 173
syscall() (*in module artiq.language.core*), 128
- T**
- TCA6424A** (*class in artiq.coredevice.i2c*), 150
TerminationRequested, 129
timestamp_mu() (*artiq.coredevice.ttl.TTLInOut method*), 142
to_mu() (*artiq.coredevice.almazny.AlmaznyChannel method*), 170
trf_read() (*artiq.coredevice.phaser.PhaserChannel method*), 187
trf_write() (*artiq.coredevice.phaser.PhaserChannel method*), 187
Trigger (*class in artiq.coredevice.shuttler*), 204
trigger() (*artiq.coredevice.shuttler.Trigger method*), 204
trigger_analyzer_proxy() (*artiq.coredevice.core.Core method*), 136
TTLClockGen (*class in artiq.coredevice.ttl*), 139
TTLInOut (*class in artiq.coredevice.ttl*), 139
TTLOut (*class in artiq.coredevice.ttl*), 143
- tune_io_update_delay()** (*artiq.coredevice.ad9910.AD9910 method*), 162
tune_sync_delay() (*artiq.coredevice.ad9910.AD9910 method*), 163
turns_amplitude_to_ram() (*artiq.coredevice.ad9910.AD9910 method*), 163
turns_to_pow() (*artiq.coredevice.ad9910.AD9910 method*), 163
turns_to_pow() (*artiq.coredevice.ad9912.AD9912 method*), 166
turns_to_pow() (*artiq.coredevice.ad9914.AD9914 method*), 168
turns_to_ram() (*artiq.coredevice.ad9910.AD9910 method*), 163
- U**
- unset()** (*artiq.coredevice.i2c.I2CSwitch method*), 149
UnwrapNoneError, 137
update() (*artiq.coredevice.fastino.Fastino method*), 198
update_xfer_duration_mu() (*artiq.coredevice.spi2.SPIMaster method*), 148
urukul_cfg() (*in module artiq.coredevice.urukul*), 154
urukul_sta_ifc_mode() (*in module artiq.coredevice.urukul*), 154
urukul_sta_pll_lock() (*in module artiq.coredevice.urukul*), 154
urukul_sta_proto_rev() (*in module artiq.coredevice.urukul*), 154
urukul_sta_rf_sw() (*in module artiq.coredevice.urukul*), 154
urukul_sta_smp_err() (*in module artiq.coredevice.urukul*), 154
- V**
- voltage_group_to_mu()** (*artiq.coredevice.fastino.Fastino method*), 198
voltage_to_mu() (*artiq.coredevice.ad53xx.AD53xx method*), 190
voltage_to_mu() (*artiq.coredevice.fastino.Fastino method*), 198
voltage_to_mu() (*in module artiq.coredevice.ad53xx*), 191
- W**
- wait_until_mu()** (*artiq.coredevice.core.Core method*), 136
watch_done() (*artiq.coredevice.ttl.TTLInOut method*), 142
watch_stay_off() (*artiq.coredevice.ttl.TTLInOut method*), 143
watch_stay_on() (*artiq.coredevice.ttl.TTLInOut method*), 143

write() (*artiq.coredevice.ad9912.AD9912 method*), 166
 write() (*artiq.coredevice.fastino.Fastino method*), 198
 write() (*artiq.coredevice.spi2.SPIMaster method*), 148
 write() (*artiq.coredevice.suservo.SUServo method*),
 211
 write16() (*artiq.coredevice.ad9910.AD9910 method*),
 163
 write16() (*artiq.coredevice.phaser.Phaser method*),
 182
 write16() (*artiq.coredevice.shuttler.ADC method*), 200
 write24() (*artiq.coredevice.shuttler.ADC method*), 200
 write32() (*artiq.coredevice.ad9910.AD9910 method*),
 163
 write32() (*artiq.coredevice.phaser.Phaser method*),
 182
 write64() (*artiq.coredevice.ad9910.AD9910 method*),
 163
 write8() (*artiq.coredevice.phaser.Phaser method*), 182
 write8() (*artiq.coredevice.shuttler.ADC method*), 200
 write_dac() (*artiq.coredevice.ad53xx.AD53xx*
 method), 190
 write_dac_mu() (*artiq.coredevice.ad53xx.AD53xx*
 method), 190
 write_ext() (*artiq.coredevice.mirny.Mirny method*),
 169
 write_gain_mu() (*artiq.coredevice.ad53xx.AD53xx*
 method), 190
 write_offset() (*artiq.coredevice.ad53xx.AD53xx*
 method), 190
 write_offset_dacs_mu() (*ar-*
 tiq.coredevice.ad53xx.AD53xx method),
 190
 write_offset_mu() (*artiq.coredevice.ad53xx.AD53xx*
 method), 191
 write_ram() (*artiq.coredevice.ad9910.AD9910*
 method), 164
 write_reg() (*artiq.coredevice.mirny.Mirny method*),
 169

X

xftw_to_frequency() (*ar-*
 tiq.coredevice.ad9914.AD9914 method),
 168

Y

y_mu_to_full_scale() (*in module ar-*
 tiq.coredevice.suservo), 211

Z

Zotino (*class in artiq.coredevice.zotino*), 192