Iris Python Debug Scripting

Version 1.0

User Guide

Non-Confidential

Issue 11
101421_0100_11_en

Iris Python Debug Scripting User Guide

This document (101421_0100_11_en) was issued on 2025-02-19. There might be a later issue at https://developer.arm.com/documentation/101421

The product version is 1.0.

See also: Proprietary Notice | Product and document information | Useful resources

Start Reading

If you prefer, you can skip to the start of the content.

Intended audience

This document is written for software developers writing Python scripts to debug Fast Models targets using the `iris.debug` Python module.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Contents

1. Getting started

This chapter describes setting up Iris Python Debug Scripting and using it to run a model.

1.1 Setting up the environment

You first need to set up your environment before using the `iris.debug` Python module.

`iris.debug` requires an existing installation of Python 3.*. Python is available from https://www.python.org/getit.

To use `iris.debug`, you first need to tell the Python interpreter where to find it. Add the directory that contains `iris.debug` to the `PYTHONPATH` environment variable. For example, on Linux:

- sh:

  ```
  export PYTHONPATH=$IRIS_HOME/Python:$PYTHONPATH
  ```

- tcsh:

  ```
  setenv PYTHONPATH $IRIS_HOME/Python:$PYTHONPATH
  ```

This step is done for you by the Fast Models setup scripts for Linux.

On Windows:

```
set PYTHONPATH=%IRIS_HOME%\Python;%PYTHONPATH%
```

Alternatively, add the directory that contains `iris.debug` to the Python path from within your script, before importing the module, as follows:

```
import sys, os
sys.path.append(os.path.join(os.environ['IRIS_HOME'], 'Python'))
import iris.debug
```

1.2 Connecting to and running a model

This example shows how to connect to a model, load an application, and run the model.

You can connect to a model by creating a `NetworkModel` instance, passing the IP address or hostname, and port number.

Note

- `iris.debug` only supports ISIM executables. It does not support models that have been built as shared libraries. This is a change in behavior from the `fm.debug` module which `iris.debug` replaces.

- If you are connecting to a Fast Model, specify `--iris-connect` when launching the model, to start the Iris server. For more information, see FVP command-line options.

The model is composed of multiple targets which represent the components in the system. A `Target` object can be obtained by calling `Model.get_target(name)` on an instantiated model, passing it the name of the target. A convenience method `Model.get_cpus()` is also provided, which returns a list of `Target` objects for all targets for which `componentType == 'Core'`, or that have the `executesSoftware` flag set.

This example assumes that the model has started an Iris server locally, listening to port 7100:

```
import iris.debug
model = iris.debug.NetworkModel("localhost",7100)
cpu = model.get_cpus()[0]
cpu.load_application("/path/to/application.axf")
model.run()
```

The code creates two variables:

model

A `Model` object which represents the entire simulated system. It is composed of various targets including cores and memories. The model object can be used to access these targets and to start, stop, and step the model.

cpu

A `Target` object, in this case the first CPU in the model. It can be used to read and write the memory and registers of the core and to set and clear breakpoints.

For documentation of the operations that can be performed on models and targets, see 4.4 class Model on page 17 and 4.5 class Target on page 20.

Note

Some example scripts that demonstrate how to use `iris.debug` are located in `$PVLIB_HOME/Iris/Python/examples/`.

Related information

Iris examples

2. Migrating from fm.debug to iris.debug

`fm.debug` is a deprecated Python client interface to Fast Models that was implemented using CADI. To continue using a Python client with Fast Models, users of `fm.debug` must migrate to the Iris Python client, `iris.debug`.

2.1 Changes when connecting to a model

If you previously used `fm.debug` with a model that was implemented as a shared library, `iris.debug` no longer supports this type of model.

About this task

With `iris.debug`, you must use an ISIM executable instead and follow these steps:

Procedure

1. Run the ISIM with the additional option `--iris-connect` to start the Iris server. For more information, see FVP command-line options.
2. Use the Python client to connect to the model through the network, as follows:

```
import iris.debug
model = iris.debug.NetworkModel('localhost',<port_number>)
```

2.2 Changes to methods defined in Model.py and in Target.py

`iris.debug` is designed to work in the same way as `fm.debug`, but there are differences in how some methods are called.

Note

All `Model` and `Target` class methods defined in `fm.debug`, apart from those listed here, are available in `iris.debug` and are unchanged.

`reset()`

In the `fm.debug` implementation, this method is called from a `Target` object. `iris.debug` implements this method in the `Model` class, which means that you can call `model.reset()` but can no longer call `target.reset()`.

3. Upgrading MxScripts to Python

Arm deprecates the MxScript language. Use Python Debug Scripting instead. This chapter describes the major differences between the MxScript and Python, and gives the `iris.debug` equivalents to various MxScript functions for interacting with a model.

3.1 Major differences between MxScript and Python

The main differences are as follows:

- Each Python script that uses `iris.debug` must have the following line near the top:

```
from iris.debug import *
```

- In MxScript, comment lines begin with `//`, whereas in Python they begin with `#`.
- In Python, indentation, not curly braces, is used to represent scope. Therefore, your indentation must be correct and consistent, and curly braces must not be used to represent scope.
- In Python, statements are not required to be delimited with semicolons. Instead, a new line is sufficient.
- In Python, flow control statements, for example `if`, `for`, and `while`, end with a colon, and the block of code that they apply to is indented. If necessary, an empty block can be created using the `pass` statement. To check for multiple conditions, only one of which is true, the `elif` statement can be used. For example:

```
if foo < 5:
    bar = 3
elif foo >= 17:
    bar += 2
else:
    bar = 7
```

- In Python, `for` loops always iterate over a list. To create a list of integers, the `range` function is used. For example:

```
>>> range(3)
[0, 1, 2]
```

The following two loops are equivalent. This loop is written in MxScript:

```
for (int i = 0; i < 3; i++) {
    // do nothing
}
```

This one is written in Python:

```
for i in range(3):
```

```
    pass
```

- `while` loops behave similarly to their MxScript equivalents. However, they use the Python syntax rule of ending a flow control statement with a colon, and use indentation to represent scope. For example:

```
while i > 1:
    i /= 2
```

- Python does not have an equivalent to the MxScript `do … while` loop.
- In Python, the logical operators `and`, `or`, and `not` are used instead of `&&`, `||`, and `!`.
- In Python, variables are not explicitly typed, so the following examples are equivalent. This code is written in MxScript:

```
int a = 5;
string b = "hello";
```

  This is written in Python:

```
a = 5
b = "hello"
```

- Unlike MxScript, Python does not have a preprocessor. Instead, the `import` statement can be used to access code from another file. This statement has the following forms:

  `import iris.debug`
  Loads the `iris.debug` module, and adds `iris.debug` to the current namespace.

  `from iris.debug import NetworkModel`
  Loads the `iris.debug` module and adds `NetworkModel` to the current namespace, without making `iris.debug` or any of its other contents available.

  `from iris.debug import *`
  Adds the entire contents of the `iris.debug` module to the current namespace.

3.2 Model connection and configuration

MxScript has the concepts of the current model, and the current target in that model. All functions operate on the current model or target, and the `selectTarget()` function switches between targets.

In contrast, `iris.debug` uses an object-oriented design, in which objects represent models and targets. These objects provide methods to interact with them. This design makes it much more practical to work with multiple targets or models. An example of where this design is useful is debugging a multi-processor system, where it is necessary to interact with multiple CPU targets.

The following table shows the MxScript functions that connect to and configure models, and their `iris.debug` equivalents:

Table 3-1: Model connection and configuration functions

MxScript function	iris.debug equivalent
`connectToModel(port)`	`model = NetworkModel(host, port)` **Note:** This function does not select the target.
`closeModel()`	`model.release()`
`debugIsim(isim)`	Not implemented
`debugSystemC(simulation)`	Not implemented
`getParameter(name)`	`target.parameters["name"]`
`setParameter(name, value)`	`target.parameters["name"] = value`
`getTargetList(filename)`	`model.get_target_info()`
`getTargetName()`	`target.instance_name`
`selectTarget(name)`	Either of the following: • `target = model.get_target(name)` • `cpus = model.get_cpus()`
`loadApp(filename)`	`target.load_application(filename)`
`saveState(filename)`	Not implemented
`restoreState(filename)`	Not implemented
`saveSession(filename)`	Not implemented
`openSession(filename)`	Not implemented
`setStateFile(filename)`	Not implemented

3.3 Execution control

`iris.debug` is not a full debugger. Therefore, it does not implement higher-level functions, such as those that require loading the source files or debug symbols that correspond to an application.

The following table shows the MxScript functions that control model execution, and their `iris.debug` equivalent:

Table 3-2: Execution control functions

MxScript function	iris.debug equivalent
`run()`	Either of the following: **`model.run()`** This function blocks until the target stops. **`model.run(blocking=False)`** This function is nonblocking.
`runUntil(<address>)`	Not implemented
`runToLine(<file>, <line>)`	Not implemented
`stop()`	`model.stop()`

MxScript function	iris.debug equivalent
getCurrentSourceFile()	Not implemented
getCurrentSourceLine()	Not implemented
getCurrentSourceColumn()	Not implemented
hardReset()	model.reset()
reset()	model.reset() target.load_application(<filename>)
pause()	Not implemented
cont()	Not implemented
getStopCond()	Either of the following: • target.get_hit_breakpoints() • Return value of blocking model.run()
isSimStopped()	not target.is_running
restart()	model.reset() target.load_application(<filename>)
goToMain()	Not implemented
step()	Not implemented
stepOver()	Not implemented
stepOut()	Not implemented
istep(*<count>*)	model.step()
getInstCount()	Not implemented
cycleStep(*<cycles>*)	Not implemented
enableStepBack(*<bool>*)	Not implemented
sleep(*<seconds>*)	import time time.sleep(<seconds>)
msleep(*<milliseconds>*)	import time time.sleep(<milliseconds * 1000>)
getCycleCount()	Not implemented

3.4 Breakpoints

The following table shows the MxScript functions that relate to breakpoints and their iris.debug equivalent:

Table 3-3: Breakpoints functions

MxScript function	iris.debug equivalent
bpAdd(*address*)	bp = target.add_bpt_prog(*address*)
bpAdd(*file, line*)	Not implemented

MxScript function	iris.debug equivalent
`bpAddReg(reg_name)`	`bp = target.add_bpt_reg(reg_name)`
`bpAddMem(address)`	`bp = target.add_bpt_mem(address)`
`bpRemove(id)`	`bp.delete()`
`bpRemoveAll()`	`for bp in target.breakpoints.values():` `    bp.delete()`
`bpEnable(id)`	`bp.enable()`
`bpDisable(id)`	`bp.disable()`
`bpEnableAll()`	`for bp in target.breakpoints.values():` `    bp.enable()`
`bpDisableAll()`	`for bp in target.breakpoints.values():` `    bp.disable()`
`bpList()`	`target.breakpoints`
`bpSetTriggerType()`	Not implemented
`bpSetIgnoreCount()`	Not implemented
`bpSetCond()`	Not implemented
`bpIsHit(id)`	`bp.is_hit`

3.5 Model resource access

The following table shows the MxScript functions that access model resources, and their `iris.debug` equivalent:

Table 3-4: Resource access functions

MxScript function	iris.debug equivalent
`regWrite(name, value)`	`target.write_register(name, value)`
`regRead(name)`	`target.read_register(name)`
`memWrite(memspace, address, value)`	`target.write_memory(address, value[, memspace])` If *memspace* is not specified, the current memory space is used.
`memRead(memspace, address, count)`	`target.read_memory(address, count[, memspace])` If *memspace* is not specified, the current memory space is used.
`disassemble(address)`	`target.disassemble(address)`

`memStoreToFile(…)`	`with open("tempmem.bin", "wb") as f:`
	`    mem = cpu.read_memory(0, count=1024)`
	`    f.write(mem)`

`memLoadFromFile(…)`	`with open("tempmem.bin", "rb") as f:`
	`    mem = bytearray(f.read(1024))`
	`    cpu.write_memory(0, mem)`

4. API reference

This chapter describes the public interface of `iris.debug`. Any members whose name starts with an underscore are internal and have not been documented.

Note

`iris.debug` does not support the `fm.debug` class `LibraryModel`, which is used to access a CADI model.

An existing model can be connected to by creating a new `NetworkModel`, passing either the IP address or hostname, and port number.

The model is comprised of multiple targets which represent the components in the system.

A `Target` object can be obtained by calling `Model.get_target(name)` on an instantiated model, passing it the name of the target.

A convenience method `Model.get_cpus()` is also provided which returns a list of `Target` objects for all targets for which `componentType == 'Core'` or that have the `executesSoftware` flag set. For example:

```
>>> model = NetworkModel(host = 'localhost', port = 7100)
>>> cpus = model.get_cpus()
>>> cpus[0].read_register("CPSR")
>>> model.run()
```

4.1 class NetworkModel

`iris.debug.NetworkModel(host='localhost', port=0, timeoutInMs=5000, client_name='client.iris_debug', verbose=False)`

An Iris model that is connected to an Iris server.

Parameters

`host`

Hostname or IP address of the computer running the model.

`port`

Port number that the model is listening on. If 0, it scans the port range 7100-7109 for Iris servers and connects to the first one found.

`timeoutInMs`

Time limit in milliseconds for the connection to wait for a response from the server. By default, 5000ms.

client_name

Hierarchical name of the client instance.

verbose

If True, extra debugging information is printed.

4.2 class NetworkModelInitializer

`class iris.NetworkModelInitializer(server_startup_command=None, host='localhost', port=None, timeout_in_ms=1000, synchronous=False, verbose=False)`

The `NetworkModelInitializer` class represents an established or pending connection between an Iris model debugger, accessible through the class `NetworkModel`, and an Iris server which is embedded either in an ISIM or in a simulation that uses an ISIM as a library.

Use the `NetworkModelFactory` class to create an instance of this class. After it is created, you can use it in one of two ways:

- In the following example, `network_model` is an instance of `NetworkModel`. All resources are automatically deallocated at the end of the `with` statement context:

```
with NetworkModelFactory.CreateNetworkToHost(host, port) as network_model:
    network_model.get_targets()
```

- In the following example, `network_model` is an instance of `NetworkModel`. Resources are not automatically deallocatted so you need to handle exceptions and force deallocation manually:

```
network_model_initializer = NetworkModelFactory.CreateNetworkToHost(host, port)
network_model = network_model_initializer.start()
    try:
        network_model.get_targets()
    finally:
        network_model_initializer.close()
```

A full working example is provided in `$PVLIB_HOME/Iris/Python/Examples/DemoNetworkInitializer.py`.

4.3 class NetworkModelFactory

`class iris.NetworkModelFactory`

Allows the creation of `NetworkModelInitializer`s. It contains only class methods.

4.3.1 CreateNetworkFromCommand()

`CreateNetworkFromCommand(command_line, timeout_in_ms=1000)`

Create a network model initializer for an Iris server to be started using a command line.

Parameters

`command_line`
The command line to launch the Iris server.

`timeout_in_ms`
Timeout in milliseconds for the connection between the client and the Iris TCP server.

Related information

class NetworkModelInitializer on page 15

4.3.2 CreateNetworkFromIsim()

`CreateNetworkFromIsim(isim_filename, parameters=None, timeout_in_ms=1000)`

Create a network model initializer for an ISIM that is not yet running.

Parameters

`isim_filename`
Full path of the ISIM to launch.

`parameters`
Parameters to pass to the ISIM.

`timeout_in_ms`
Timeout in milliseconds for the connection between the client and the Iris TCP server.

Related information

class NetworkModelInitializer on page 15

4.3.3 CreateNetworkFromLibrary()

`CreateNetworkFromLibrary(simulation_command, library_filename, parameters=None, timeout_in_ms=1000)`

Create a network model initializer for a simulation application that uses an ISIM as a library and is not yet running.

Parameters

`simulation_command`
The command to launch the simulation application.

`library_filename`
: Full path to the library.

`parameters`
: Parameters to pass to the simulation.

`timeout_in_ms`
: Timeout in milliseconds for the connection between the client and the Iris TCP server.

Related information

class NetworkModelInitializer on page 15

4.3.4 CreateNetworkToHost()

```
CreateNetworkToHost(hostname, port, timeout_in_ms=1000)
```

Create a network model initializer for an Iris server that is already running and is accessible at the given hostname and port.

Parameters

`hostname`
: Hostname that the Iris server is running on.

`port`
: Port number that the Iris server is listening on.

`timeout_in_ms`
: Timeout in milliseconds for the connection between the client and the Iris server.

Related information

class NetworkModelInitializer on page 15

4.4 class Model

```
iris.debug.Model(client, verbose)
```

This class wraps an Iris model.

4.4.1 get_cpus()

```
get_cpus()
```

Return all targets that have `executesSoftware` set or have `componentType = 'Core'`.

4.4.2 get_target()

```
get_target(instance_name)
```

Obtain an interface to a target.

Parameters

`instance_name`
: The instance name that corresponds to the target.

4.4.3 get_target_info()

```
get_target_info()
```

Return an iterator over named tuples that contain information about all of the target instances contained in the model.

4.4.4 get_targets()

```
get_targets()
```

Generator function to iterate over all targets in the simulation.

4.4.5 release()

```
release(shutdown=False)
```

End the simulation and release the model.

Parameters

`shutdown`
: If True, the simulation is shut down and any other scripts or debuggers must disconnect.

If False, a simulation might be kept alive after disconnection.

4.4.6 reset()

```
reset(allow_partial_reset=False)
```

Reset the simulation to exactly the same state it had after instantiation.

Parameters

`allow_partial_reset`

If true, perform a partial simulation reset for simulations that do not support a full reset. This might be because only the Fast Models components in a SystemC platform simulation can be reset. By setting `allow_partial_reset` to True, you acknowledge that not all components will be reset and accept the consequences.

For an ISIM model, which is built purely from Fast Models components, the whole platform can be reset.

4.4.7 run()

```
run(blocking = True, timeout = None)
```

Start executing the model.

Parameters

`blocking`

If True, this call blocks until the model stops executing, typically due to a breakpoint.

If False, this call returns when the target starts executing.

`timeout`

If None, this call waits indefinitely for the target to enter the correct state.

If set to a float or int, this parameter gives the maximum number of seconds to wait.

Exceptions

`TimeoutError`

The timeout expired.

`TargetBusyError`

The model is already running.

4.4.8 step()

```
step(count=1, timeout=None)
```

Step all cores in the system for `count` instructions each, blocking.

Cores are stepped individually and sequentially. The first core is stepped for `count` instructions. When that completes, the second core is stepped for `count` instructions and so on. This is intrusive debugging as it permutes the scheduling order of the cores and it generally lets more simulation time pass than indicated by `count`. Also, the number of steps executed is independent of the relative clock speeds of the CPUs.

Only cores that have `get_execution_state()==True` are processed by this function. Therefore it is possible to select which cores should be stepped by calling `set_execution_state()` beforehand.

Note

This is an exotic stepping function. Use `core.set_steps()` or `model.run()` for normal non-intrusive stepping.

Parameters

`count`

The number of processor instructions to execute.

`timeout`

If None, this call waits indefinitely for the target to enter the correct state.

If set to a float or int, this parameter gives the maximum number of seconds to wait.

Exceptions

`TimeoutError`

The timeout expired.

`TargetBusyError`

The model is running.

`ValueError`

No CPUs are present or not all CPUs support per-instance execution control.

4.4.9 stop()

```
stop(timeout = None)
```

Stop the model executing.

Silently returns if the model is already stopped.

Parameters

`timeout`

If None, this call waits indefinitely for the target to enter the correct state.

If set to a float or int, this parameter gives the maximum number of seconds to wait.

Exceptions

`TimeoutError`

The timeout expired.

4.5 class Target

`iris.debug.Target(instInfo, model)`

Wraps an Iris object, providing a simplified interface to common tasks.

You can access memory, registers, and breakpoints using methods defined in this class, for example:

```
cpu.read_memory(0x1234, count=8)
cpu.write_register("Core.R5", 1000)
cpu.add_bpt_mem(0x1234, memory_space="Secure", on_read=False)
cpu.add_bpt_reg("Core.CPSR")
```

The breakpoint-related methods return `Breakpoint` objects, which allow you to enable, disable, and delete the breakpoint. You can access the breakpoints that are set by using the dictionary `Target.breakpoints`, which maps from breakpoint numbers to `Breakpoint` objects.

4.5.1 add_bpt_mem()

`add_bpt_mem(address, memory_space=None, on_read=True, on_write=True, on_modify=None)`

Set a new data breakpoint which is hit when the specified memory location is accessed.

Parameters

`address`
The address to set the breakpoint on.

`memory_space`
The name of the memory space that `address` is in. If None, the current memory space of the core is used.

`on_read`
If True, the breakpoint is triggered when the memory location is read from.

`on_write`
If True, the breakpoint is triggered when the memory location is written to.

`on_modify`
Deprecated. If True, the breakpoint is triggered when the memory location is modified.

4.5.2 add_bpt_prog()

`add_bpt_prog(address, memory_space=None)`

Set a new code breakpoint which is hit when program execution reaches the specified memory address.

Parameters

`address`

The address to set the breakpoint on.

`memory_space`

The name of the memory space that `address` is in. If None, the current memory space of the core is used.

4.5.3 add_bpt_reg()

```
add_bpt_reg(reg_name, on_read=True, on_write=True, on_modify=None)
```

Set a new register breakpoint which is hit when the specified register is accessed.

Parameters

`reg_name`

The name of the register to set the breakpoint on. The name can be in any of the following formats:

- "*group.register*"
- "*group.register.field*"
- "*register*"
- "*register.field*"

The last two forms can only be used if the register name is unambiguous.

`on_read`

If True, the breakpoint is triggered when the register is read from.

`on_write`

If True, the breakpoint is triggered when the register is written to.

`on_modify`

Deprecated. If True, the breakpoint is triggered when the register is modified.

4.5.4 add_event_callback()

```
add_event_callback(event_name, func, fields=None)
```

Add a callback function for the named event. This function is called when the event fires.

Parameters

`event_name`

The name of the event.

func

A callback to be called when the event fires.

fields

A list of event fields that the callback provides.

4.5.5 clear_bpts()

```
clear_bpts()
```

Clear all breakpoints.

4.5.6 disassemble()

```
disassemble(address, count=1, mode=None, memory_space=None)
```

Disassemble intructions.

If `count`=1 this method returns a 3-tuple of `addr`, `opcode`, `disass`, where:

addr	is the address of the instruction.
opcode	is a string containing the instruction opcode at that address.
disass	is a string containing the disassembled representation of the instruction.

If `count`> 0, this method behaves like a generator function that yields one 3-tuple for each disassembled instruction.

Parameters

address

Address to start disassembling from.

count

Number of instructions to disassemble. Default is 1. This method might yield fewer than `count` results if an error occurs during disassembly.

mode

Disassembly mode to use. Must be either None, in which case the target's current mode is used, or one of the values returned by `get_disass_modes()`. Default is None.

memory_space

Memory space for `address`. Must be the name of a valid memory space for this target or None. If None, the current memory space is used. Default is None.

Exceptions

ValueError

The target does not support disassembly.

4.5.7 get_disass_modes()

`get_disass_modes()`

Return the disassembly modes for this `Target`.

4.5.8 get_event_info()

`get_event_info(name=None)`

Retrieve information about the event sources provided by this `Target`.

It is used in the following ways:

`get_event_info(name)`
: Return the information for the named event and its fields.

`get_event_info()`
: Act as a generator and yield information about all events.

Parameters

`name`
: The name of the event to provide information for. If None, yields information about all events.

4.5.9 get_execution_state()

`get_execution_state()`

Return True if execution state is enabled.

Exceptions

`ValueError`
: It cannot get the execution state.

4.5.10 get_hit_breakpoints()

`get_hit_breakpoints()`

Return the list of breakpoints that were hit the last time the `Target` was running.

4.5.11 get_instruction_count()

`get_instruction_count()`

Return the current instruction count of the `Target`.

4.5.12 get_pc()

`get_pc()`

Return the current value of the program counter.

4.5.13 get_register_info()

`get_register_info(name=None)`

Retrieve information about the registers that are present in this `Target`.

It is used in the following ways:

`get_register_info(name)`
: Return the information for the named register.

`get_register_info()`
: Act as a generator and yield information about all registers.

Parameters

`name`
: The name of the register to provide information for. If None, it yields information about all registers. It follows the same rules as the `name` parameter of `read_register()` and `write_register()`.

4.5.14 get_steps()

`get_steps(unit='instruction')`

Return the remaining number of steps.

Parameters

`unit`
: Steps unit. Must be either:

'instruction'
: A step is one executed instruction. This is the default.

'cycle'
A step is one cycle.

Exceptions

`ValueError`
Cannot get the remaining steps.

4.5.15 get_table_info()

```
get_table_info(name=None)
```

Retrieve information about the tables that are present in this `Target`.

It is used in the following ways:

`get_table_info(name)`
Return the information for the named table and its columns.

`get_table_info()`
Act as a generator and yield information about all tables.

Parameters

`name`
The name of the table to provide information for. If None, yields information about all tables.

4.5.16 handle_semihost_io()

```
handle_semihost_io()
```

Request that semihosted input and output are handled for this `Target` by this Iris client.

4.5.17 has_register()

```
has_register(name)
```

Return True if the named register exists and has an unambiguous name, False otherwise.

Parameters

`name`
The name of the register. It follows the same rules as the `name` parameter of `read_register()` and `write_register()`.

4.5.18 has_table()

```
has_table(name)
```

Return True if the `Target` has the named table.

Parameters

`name`
: The name of the table.

4.5.19 load_application()

```
load_application(filename, loadData=None, verbose=None, parameters=None)
```

Load an application to run on the model.

Parameters

`filename`
: The filename of the application to load.

`loadData`
: Deprecated.

If set to True, the target loads data, symbols, and code.

If set to False, the target does not reload the application code to its program memory. This can be used, for example, to either:

- Forward information about applications that are loaded to a target by other platform components.
- Change command-line parameters for an application that was loaded by a previous call.

`verbose`
: Set this to True to allow the target to print verbose messages.

`parameters`
: Deprecated.

A list of command-line parameters to pass to the application, or None.

4.5.20 read_memory()

```
read_memory(address, memory_space=None, size=1, count=1, do_side_effects=False)
```

Return a byte array of length `size*count`.

Parameters

address

Address to begin reading from.

memory_space

Name of the memory space to read or None, which reads the core's current memory space.

size

Size of the memory access unit in bytes. Must be one of 1, 2, 4, or 8.

Note

- Not all values are supported by all models.
- The data is always returned as bytes, so calling this function with `size`=4, `count`=1 returns a byte array of length 4.

count

Number of units to read.

do_side_effects

Deprecated.

If True, the `Target` must perform any side-effects that are normally triggered by the read, for example clear-on-read.

4.5.21 read_all_registers()

```
read_all_registers()
```

Read the current value of all registers. Returns a dictionary of register values keyed off register id.

Exceptions

ValueError

Failed to read a readable register.

4.5.22 read_register()

```
read_register(name=None, side_effects=False, rscId=None)
```

Read the current value of a register.

Parameters

name

The name of the register to read from. This can take the following forms:

- `"group.register"`

- "*group.register.field*"
- "*register*"
- "*register.field*"

If `rscId` is provided, `name` is ignored.

side_effects
Deprecated.

rscId
Resource id of the register to read from. If this is provided, `name` is ignored.

Exceptions

ValueError
The register name does not exist, or the group name is omitted and there are multiple registers in different groups with that name.

4.5.23 read_table()

```
read_table(name, index=None, count=1)
```

Read the specified rows from the named table. The rows are returned as a dictionary, in the form:

```
{index : {<colname> : <value>, …}, …}
```

Parameters

name
The name of the table to read from.

index
Row from which to start reading. Default is `minIndex` of the table.

count
Number of rows to read, starting from `index`. Default is 1.

Exceptions

ValueError
The table `name` does not exist, or `count` is less than 1.

4.5.24 remove_event_callback()

```
remove_event_callback(event_name_or_func)
```

Remove an event callback function that was previously added to this `Target`.

Parameters

event_name_or_func

Either the name of an event or a callable object that was previously added to this `Target` as an event callback.

4.5.25 remove_bpt()

```
remove_bpt(bptId)
```

Delete a breakpoint.

Parameters

bptId

Breakpoint id of the breakpoint to delete.

Exceptions

ValueError

`bptId` does not exist, or the deletion failed.

4.5.26 set_execution_state()

```
set_execution_state(enable)
```

Set the execution state of this `Target`.

Parameters

enable

True to enable execution of instructions, false to disable it.

Exceptions

ValueError

Cannot set the execution state.

4.5.27 set_steps()

```
set_steps(steps, unit='instruction')
```

Set the remaining number of steps.

Parameters

unit

Steps unit, either:

'instruction'

A step is one executed instruction. This is the default.

'cycle'

A step is one cycle.

Exceptions

`ValueError`

Cannot set the remaining number of steps.

4.5.28 supports_tables()

```
supports_tables()
```

Return True if the `Target` has any tables.

4.5.29 write_memory()

```
write_memory(address, data, memory_space=None, size=1, count=None,
do_side_effects=False)
```

Write a byte array of length `size`*`count` to memory.

Parameters

`address`

Address to begin writing to.

`data`

The data to write. If `count` is 1, this can be an integer. Otherwise it must be a byte array with length >= `size`*`count`.

`memory_space`

The memory space to write to. The default is None which reads the core's current memory space.

`size`

Size of the memory access unit in bytes. Must be one of 1, 2, 4, or 8.

Note

Not all values are supported by all models.

`count`

Number of units to write. If None, `count` is automatically calculated such that all data from the array is written to the target.

`do_side_effects`

Deprecated.

If True, the target must perform any side-effects normally triggered by the write, for example triggering an interrupt.

4.5.30 write_register()

```
write_register(name=False, value=None, side_effects=False, rscId=None)
```

Write a value to a register.

Parameters

`name`

The name of the register to write to. This can take the following forms:

- `"`*`group.register`*`"`
- `"`*`group.register.field`*`"`
- `"`*`register`*`"`
- `"`*`register.field`*`"`

If `rscId` is provided, `name` is ignored.

`value`

The value to write to the register.

`side_effects`

Deprecated.

`rscId`

Resource id of the register to write to. If this is provided, `name` is ignored.

Exceptions

`ValueError`

Neither a register name nor a resource id were provided.

4.5.31 write_table()

```
write_table(name, table_records)
```

Write specified records to a table.

Parameters

`name`

The name of the table to write to.

table_records

A dictionary in the form:

```
{ index : rowdata, …}
```

where:

index

The value of the row index where `rowdata` is written.

rowdata

The cells in dictionary form:

```
{ <col name> : <value>, ... }
```

The table record can have a subset of the cells in the row to which a write should take place.

This parameter has the same format as the return value of `read_table()`.

Exceptions

ValueError

The table does not exist.

4.5.32 Target properties

The `Target` class defines the following properties:

component_type

The type of a target component as a string.

description

The description of a target.

disass_mode

The current disassembly mode for this target.

executes_software

True if the component supports executing instructions.

instance_name

The instance name of the target.

is_running

True if the target is currently running.

stdin

The target's semihosting stdin.

stdout

The target's semihosting stdout.

stderr
The target's semihosting stderr.

target_name
The name of the target component.

4.6 class EventCallbackManager

```
class iris.debug.EventCallbackManager(client, target, verbose)
```

Manages user event callbacks for a particular target instance.

4.6.1 add_callback()

```
add_callback(evSrcId, func, fields=None)
```

Create an event stream for the specified event source which will call back `func()`.

Parameters

evSrcId
Event source id of the event.

func
Name of the callback function for the event.

fields
List of string names of event source fields to receive in the callback function.

Exceptions

ValueError
Unknown event source id, or unknown event field.

Exceptions.TargetError
Failed to add the event callback.

4.6.2 get_evSrcId()

```
get_evSrcId(name)
```

Get the event source id for the named event.

Parameters

name
Name of the event.

Exceptions

`ValueError`
The target does not support the named event.

4.6.3 get_info()

```
get_info()
```

Yield `EventSourceInfo` for all events that are supported by the target instance.

4.6.4 remove_callback_evSrcId()

```
remove_callback_evSrcId(evSrcId)
```

Remove a registered callback by event source id.

Parameters

`evSrcId`
The event source id for the callback function to remove.

4.6.5 remove_callback_func()

```
remove_callback_func(func_to_remove)
```

Remove a registered callback function.

Parameters

`func_to_remove`
Callback function to remove.

Exceptions

`ValueError`
No event stream is registered with this callback function.

4.7 class Breakpoint

```
class iris.debug.Breakpoint(target, bpt_info)
```

Provides a high level interface to a breakpoint in an Iris target.

4.7.1 delete()

`delete()`

Remove the breakpoint from the target.

4.7.2 disable()

`disable()`

Disable the breakpoint if the model supports it.

4.7.3 enable()

`enable()`

Enable the breakpoint if the model supports it.

4.7.4 wait()

`wait(timeout=None)`

Block until the breakpoint is triggered or the timeout expires.

Return True if the breakpoint was triggered, False otherwise.

4.7.5 Breakpoint properties

The `Breakpoint` class defines the following properties:

`address`
: The memory address at which this breakpoint is set. Only valid for code and data breakpoints.

`bpt_type`
: The name of the breakpoint type. Valid values are:

`Program`	Code breakpoint.
`Memory`	Data breakpoint.
`Register`	Register breakpoint.

`enabled`
: True if the breakpoint is currently enabled.

is_hit

True if the breakpoint was hit the last time the target was running.

memory_space

The name of the memory space in which this breakpoint is set.

Only valid for code and data breakpoints.

number

Identification number of this breakpoint.

This number is the same as the key in the `Target.breakpoints` dictionary.

If the number is non-negative, it is equal to the `bptId` and the breakpoint is enabled. If the number is negative, the breakpoint is disabled.

This number is only valid until the breakpoint is deleted, and breakpoint numbers can be reused and modified.

on_modify

Deprecated. True if this breakpoint is triggered on modify. Only valid for register and memory breakpoints.

on_read

True if this breakpoint is triggered by reads. Only valid for register and memory breakpoints.

on_write

True if this breakpoint is triggered by writes. Only valid for register and memory breakpoints.

register

The name of the register that this breakpoint is set on. Only valid for register breakpoints.

4.8 Exceptions

`iris.debug` defines the following exception classes:

iris.debug.SecurityError

Method failed because an access was denied.

This could be caused by, for example, writing to a read-only register or reading memory with restricted access.

iris.debug.SimulationEndedError

Attempted to call a method on a simulation that has ended.

iris.debug.TargetBusyError

The call could not be completed because the target is busy.

Registers and memories, for example, might not be writable while the target is executing application code.

The debugger can either wait for the target to reach a stable state or enforce a stable state by, for example, stopping a running target. The debugger can then repeat the original call when the target has reached a stable state.

`iris.debug.TargetError`

An error occurred while accessing the target.

`iris.debug.TimeoutError`

Timeout expired while waiting for a target to enter the new state.

Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in the Arm documents.

Product status

All products and Services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

The information in this document is Final, that is for a developed product.

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
0100-11	19 February 2025	Non-Confidential	Update for v11.28.
0100-10	13 March 2024	Non-Confidential	Update for v11.25.
0100-09	22 March 2023	Non-Confidential	Update for v11.21.
0100-08	7 December 2022	Non-Confidential	Update for v11.20.
0100-07	14 September 2022	Non-Confidential	Update for v11.19.
0100-06	15 June 2022	Non-Confidential	Update for v11.18.
0100-05	16 February 2022	Non-Confidential	Update for v11.17.
0100-04	6 October 2021	Non-Confidential	Update for v11.16.
0100-03	22 September 2020	Non-Confidential	Update for v11.12.
0100-02	12 March 2020	Non-Confidential	Update for v11.10.

Issue	Date	Confidentiality	Change
0100-01	5 September 2019	Non-Confidential	Update for v11.8.
0100-00	23 November 2018	Non-Confidential	New document.

For information about the functional changes to Fast Models, see the Fast Models Release Notes.

Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` <u>`underline`</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Caution

We recommend the following. If you do not follow these recommendations your system might not work.

Warning

Your system requires the following. If you do not follow these requirements your system will not work.

Danger

You are at risk of causing permanent damage to your system or your equipment, or of harming yourself.

Note

This information is important and needs your attention.

Tip

This information might help you perform a task in an easier, better, or faster way.

Remember

This information reminds you of something important relating to the current content.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Table 1: Arm publications

Document name	Document ID	Licensee only
Fast Models Tools User Guide	109415	No
Iris User Guide	101196	No
MxScript v1.3 for Fast Models Reference Manual	DUI 0840	No
Python Debug Scripting for Fast Models Reference Manual	DUI 0851	No

Table 2: Other publications

Document ID	Organization	Document name
-	Python Software Foundation	*Python™*