Iris Python Debug Scripting

Version 1.0

User Guide

Non-Confidential

Issue 07
101421_0100_07_en

Iris Python Debug Scripting

User Guide

Release Information

Document history

Issue	Date	Confidentiality	Change
0100-00	23 November 2018	Non-Confidential	New document.
0100-01	5 September 2019	Non-Confidential	Update for v11.8.
0100-02	12 March 2020	Non-Confidential	Update for v11.10.
0100-03	22 September 2020	Non-Confidential	Update for v11.12.
0100-04	6 October 2021	Non-Confidential	Update for v11.16.
0100-05	16 February 2022	Non-Confidential	Update for v11.17.
0100-06	15 June 2022	Non-Confidential	Update for v11.18.
0100-07	14 September 2022	Non-Confidential	Update for v11.19.

Proprietary Notice

This document is protected by copyright and other related rights and the practice 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. 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 implementations infringe any third party patents.

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, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

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.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at https://www.arm.com/company/policies/trademarks.

Copyright © 2018–2022 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

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

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.

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.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

This book describes the iris.debug Python package.

1.1 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

Convention	Use
italic	Citations.
bold	Highlights interface elements, such as menu names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace`	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`<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	Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

1.2 Other information

See the Arm® website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.

- Arm® Glossary.

1.3 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.

Arm product resources	Document ID	Confidentiality
Iris User Guide	101196	Non-Confidential
Model Debugger for Fast Models User Guide	100968	Non-Confidential
MxScript v1.3 for Fast Models Reference Manual	DUI 0840	Non-Confidential
Python Debug Scripting for Fast Models Reference Manual	DUI 0851	Non-Confidential

Non-Arm resources	Document ID	Organization
https://www.python.org	-	Python Software Foundation

Note

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com

---

2. Getting started

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

2.1 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
```

2.2 Connecting to and running a model

This example shows how to connect to a model, load an application onto it, 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 5.2 Model on page 19 and 5.3 Target on page 23.

Note

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

Related information

Iris examples

3. Migrating from fm.debug to iris.debug

fm.debug is a Python client interface to Fast Models that is implemented using CADI. It is deprecated in Fast Models 11.10. To continue using a Python client with Fast Models, you must switch to using the Iris Python client, iris.debug, instead.

This chapter describes how to migrate from fm.debug to iris.debug.

3.1 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>)
```

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

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

Note

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

`save_state()` and `restore_state()`

These methods are defined in `Model.py` and in `Target.py`. In fm.debug, the input argument *`checkpoint_dir`* could either be a checkpoint directory or a stream object. In iris.debug, this argument can only be a checkpoint directory, which means that it must be a string and not a stream object.

`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()`.

4. Upgrading MxScripts to Python

This chapter describes the major differences between the MxScript language and Python, and gives the iris.debug equivalents to various MxScript functions for interacting with a model.

Note

Arm deprecates MxScript in favor of Python Debug Scripting.

4.1 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.

4.2 Model connection and configuration

MxScript has the concept 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 multiple 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 equivalent:

Table 4-1: 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

4.3 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 4-2: 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

MxScript function	iris.debug equivalent
`stop()`	`model.stop()`
`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

4.4 Breakpoints

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

Table 4-3: 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`

4.5 Model resource access

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

Table 4-4: 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)
```

5. 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 `LibraryModel` class, which is used to access a CADI model.

5.1 NetworkModel

`class iris.debug.Model.NetworkModel(`*`host, port, timeoutInMs, client_name, verbose`*`)`

Bases: `iris.debug.Model.Model`

Use this class to connect an Iris model to a running Iris server. It enables you to access components of the model, which are referred to as targets, and to control the execution of the model.

5.1.1 __init__()

```
__init__(host = "localhost", port = 0, timeoutInMs = 1000, client_name =
"client.iris_debug", verbose=False)
```

Connect to an initialized Iris server.

Parameters

host
: Hostname or IP address of the host running the model.

port
: Port number that the model is listening on. When `port` is 0, this means scan the port range 7100-7109 for Iris servers and connect to the first one found.

timeoutInMs
: Time limit in milliseconds for the connection to wait for a response from the server. By default, 1000ms.

client_name
: Hierarchical name of the client instance.

verbose
: If True, extra debugging information is printed.

5.2 Model

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

An Iris platform model.

5.2.1 get_target()

`get_target(instance_name)`

Obtain an interface to a target.

Parameters

`instance_name`

The instance name that corresponds to the target.

5.2.2 get_targets()

`get_targets()`

Generator function to iterate over all targets in the simulation.

5.2.3 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.

5.2.4 get_cpus()

`get_cpus()`

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

5.2.5 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.

5.2.6 stop()

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

Stop the model executing.

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.

TargetBusyError

The model is already stopped.

5.2.7 step()

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

Execute the target for *`count`* steps. Cores are stepping individually and sequentially. This is intrusive debugging as it permutes the scheduling order of the cores.

Parameters

`count`

The number of processor cycles 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.

5.2.8 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 `allowPartialReset` to true, you acknowledge that not all components will be reset and accept the consequences.

5.2.9 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.

5.2.10 save_state()

`save_state(stream_directory, save_all=True)`

Save the state of the simulation to a directory. Returns True if all components were saved successfully.

Parameters

`stream_directory`

String that is treated as the name of the directory to which to save the simulation state.

`save_all`

If True, save the state of the simulation and all targets in it that support checkpointing. If False, only save the simulation state. This parameter defaults to True.

Exceptions

NotImplementedError

`save_all` is False, and the simulation does not support checkpointing.

5.2.11 restore_state()

`restore_state(stream_directory, restore_all=True)`

Restore the state of the simulation from a directory. Returns True if all components were restored successfully.

Parameters

`stream_directory`

String that is treated as the name of the directory from which to restore the simulation state.

`restore_all`

If True, restore the state of the simulation and all targets in it that support checkpointing. If False, only restore the simulation state. This parameter defaults to True.

Exceptions

NotImplementedError

`restore_all` is False, and the simulation does not support checkpointing.

5.3 Target

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

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

You can access memory, registers, and breakpoints using methods provided by this object, 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 currently set by using the dictionary `Target.breakpoints`, which maps from breakpoint numbers to `Breakpoint` objects.

5.3.1 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.

5.3.2 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.

5.3.3 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.

5.3.4 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 one 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.

5.3.5 get_hit_breakpoints()

`get_hit_breakpoints()`

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

5.3.6 clear_bpts()

`clear_bpts()`

Reset the state of all breakpoints.

5.3.7 get_execution_state()

`get_execution_state()`

Return True if execution state is enabled.

Exceptions

`ValueError`

Cannot get the execution state.

5.3.8 set_execution_state()

`set_execution_state(enable)`

Set the execution state.

Parameters

enable

True to enable the component to execute instructions, false to disable it.

Exceptions

`ValueError`

Cannot set the execution state.

5.3.9 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.

5.3.10 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. 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.

5.3.11 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 to read from. This can take the following forms:

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

5.3.12 read_register()

```
read_register(name, side_effects = False)
```

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"`

`side_effects`

Deprecated.

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.

5.3.13 write_register()

`write_register(name, value, side_effects = False)`

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"`

`value`

The value to write to the register.

`side_effects`

Deprecated.

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.

5.3.14 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()`.

5.3.15 get_disass_modes()

`get_disass_modes()`

Return the disassembly modes for this target.

5.3.16 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.

5.3.17 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.

'cycle'

A step is one cycle.

Exceptions

`ValueError`

Cannot get the remaining steps.

5.3.18 set_steps()

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

Set the remaining number of steps.

Parameters

unit

Steps unit. Must be either:

'instruction'

A step is one executed instruction.

'cycle'

A step is one cycle.

Exceptions

`ValueError`

Cannot set the remaining number of steps.

5.3.19 get_instruction_count()

`get_instruction_count()`

Return the current instruction count of the Target.

5.3.20 get_pc()

`get_pc()`

Return the current value of the program counter.

5.3.21 supports_tables()

`supports_tables()`

Return true if the target has any tables.

5.3.22 has_table()

`has_table(`*`name`*`)`

Return true if the target has the named table.

Parameters

name

The name of the table.

5.3.23 read_table()

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

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

```
{index : {<col_name> : <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.

5.3.24 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

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

rowdata

is 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 `name` does not exist.

5.3.25 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.

5.3.26 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.

5.3.27 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 occurs.

Parameters

event_name
The name of the event.

func
A callback to be called when the event occurs.

fields
A list of event fields that the callback should provide.

5.3.28 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`
This can either be the name of an event or a callable object that was previously added to this target as an event callback.

5.3.29 handle_semihost_io()

`handle_semihost_io()`

Request that semihosted input and output are handled for this target using this Iris client.

5.3.30 save_state()

`save_state(checkpoint_dir)`

Save the state of the target to a directory.

Parameters

`checkpoint_dir`
Directory to which to save the target state.

5.3.31 restore_state()

`restore_state(checkpoint_dir)`

Restore the state of the target from a directory.

Parameters

`checkpoint_dir`
Directory in which the target state was stored.

5.3.32 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.

`parameters`
Dictionary of target's run-time parameters.

`pc_info`
Information about the PC register as a dictionary.

`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.

5.4 EventCallbackManager

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

Manages user event callbacks for a particular target instance.

5.4.1 get_info()

`get_info()`

Yield `EventSourceInfo` for all events in the target instance.

5.4.2 get_evSrcId()

`get_evSrcId(name)`

Get the event source id for the named event.

Parameters

name

Name of the event.

5.4.3 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

Callback function for the event.

fields

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

5.4.4 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.

5.4.5 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.

5.5 Breakpoint

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

Provides a high level interface to breakpoints.

5.5.1 enable()

`enable()`

Enable the breakpoint if the model supports it.

5.5.2 disable()

`disable()`

Disable the breakpoint if the model supports it.

5.5.3 delete()

`delete()`

Remove the breakpoint from the target.

5.5.4 wait()

`wait(`*`timeout`*` = None)`

Block until the breakpoint is triggered or the timeout expires.

Return True if the breakpoint was triggered, False otherwise.

5.5.5 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 on which this breakpoint is set. Only valid for register breakpoints.

5.6 Exceptions

iris.debug defines the following exception classes:

exception iris.debug.TargetError

Bases: `exceptions.Exception`

An error occurred while accessing the target.

exception iris.debug.TargetBusyError

Bases: `iris.debug.Exceptions.TargetError`

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.

exception iris.debug.SecurityError

Bases: `iris.debug.Exceptions.TargetError`

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.

exception iris.debug.TimeoutError

Bases: `iris.debug.Exceptions.TargetError`

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

exception iris.debug.SimulationEndedError

Bases: `iris.debug.Exceptions.TargetError`

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