Iris Python Debug Scripting
User Guide
Copyright © 2018–2020 Arm Limited or its affiliates. All rights reserved.

Release Information

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100-00</td>
<td>23 November 2018</td>
<td>Non-Confidential</td>
<td>New document.</td>
</tr>
<tr>
<td>0100-01</td>
<td>05 September 2019</td>
<td>Non-Confidential</td>
<td>Update for v11.8.</td>
</tr>
<tr>
<td>0100-02</td>
<td>12 March 2020</td>
<td>Non-Confidential</td>
<td>Update for v11.10.</td>
</tr>
</tbody>
</table>

Non-Confidential 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, third party 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.**

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. 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 subsidiaries) 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 [http://www.arm.com/company/policies/trademarks](http://www.arm.com/company/policies/trademarks).

Copyright © 2018–2020 Arm Limited (or its affiliates). All rights reserved.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349
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.

Web Address
www.arm.com
Contents

Iris Python Debug Scripting User Guide

Preface
About this book ........................................................................................................  7

Chapter 1 Getting started
1.1 Setting up the environment ...................................................................................... 1-10
1.2 Connecting to and running a model ........................................................................ 1-11

Chapter 2 Migrating from fm.debug to iris.debug
2.1 Changes when connecting to a model ...................................................................... 2-13
2.2 Changes to methods defined in Model.py and in Target.py .................................. 2-14

Chapter 3 Upgrading MxScripts to Python
3.1 Major differences between MxScript and Python .................................................. 3-16
3.2 Model connection and configuration ......................................................................... 3-18
3.3 Execution control ...................................................................................................... 3-19
3.4 Breakpoints ............................................................................................................. 3-21
3.5 Model resource access ............................................................................................. 3-22

Chapter 4 API reference
4.1 NetworkModel ........................................................................................................ 4-24
4.2 Model ..................................................................................................................... 4-25
4.3 Target ..................................................................................................................... 4-28
4.4 EventCallbackManager ......................................................................................... 4-38
4.5 Breakpoint ............................................................................................................. 4-40

Copyright © 2018–2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential
Preface

This preface introduces the *Iris Python Debug Scripting User Guide*. It contains the following:

• *About this book* on page 7.
About this book

This book describes the iris.debug Python module. iris.debug is a Python scripting interface to Fast Models which uses Iris as its backend. It allows you to interact with models, including connecting to and configuring them, performing execution control, and accessing registers and memory.

Using this book

This book is organized into the following chapters:

**Chapter 1 Getting started**
This chapter describes setting up Iris Python Debug Scripting and using it to run a model.

**Chapter 2 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.

**Chapter 3 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.

**Chapter 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.

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.

Typographic conventions

**italic**
Introduces special terminology, denotes cross-references, and citations.

**bold**
Highlights interface elements, such as menu names. Denotes signal 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.

monospace **bold**
Denotes language keywords when used outside example code.

<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.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:
• The product name.
• The product revision or version.
• An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:
• The title Iris Python Debug Scripting User Guide.
• The number 101421_0100_02_en.
• If applicable, the page number(s) to which your comments refer.
• A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

• Arm® Developer.
• Arm® Information Center.
• Arm® Technical Support Knowledge Articles.
• Technical Support.
• Arm® Glossary.
Chapter 1
Getting started

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

It contains the following sections:
• 1.1 Setting up the environment on page 1-10.
• 1.2 Connecting to and running a model on page 1-11.
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:
  ```bash
  export PYTHONPATH=$IRIS_HOME/Python:$PYTHONPATH
  ```
- tcsh:
  ```bash
  setenv PYTHONPATH $IRIS_HOME/Python:$PYTHONPATH
  ```

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

On Windows:

```bash
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:

```python
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 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, you must specify -I when launching the model. This option starts the Iris server. It might be useful to also specify -p to display the port number to use.

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 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:

```python
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 4.2 Model on page 4-25 and 4.3 Target on page 4-28.

--- Note ---

For some example scripts that demonstrate how to use iris.debug, see $PVLIB_HOME/Iris/Python/examples/.

**Related information**

- Iris examples
Chapter 2
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.

It contains the following sections:

• 2.1 Changes when connecting to a model on page 2-13.
• 2.2 Changes to methods defined in Model.py and in Target.py on page 2-14.
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.

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

Procedure

1. Run the ISIM with the additional options:
   - -I to start the Iris server
   - -p to print the port number to which the Iris server is listening

2. Use the Python client to connect to the model through the network, as follows:

   ```python
   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. 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().
Chapter 3
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.

It contains the following sections:
• 3.1 Major differences between MxScript and Python on page 3-16.
• 3.2 Model connection and configuration on page 3-18.
• 3.3 Execution control on page 3-19.
• 3.4 Breakpoints on page 3-21.
• 3.5 Model resource access on page 3-22.
### 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:

  ```python
  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:

  ```python
  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:

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

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

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

  This one is written in Python:

  ```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:

  ```python
  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:

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

  This is written in Python:

  ```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:

  ```python
  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 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>
<thead>
<tr>
<th>MxScript function</th>
<th>iris.debug equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connectToModel(port)</code></td>
<td><code>model = NetworkModel(host, port)</code></td>
</tr>
<tr>
<td></td>
<td><em>Note</em> This function does not select the target.</td>
</tr>
<tr>
<td><code>closeModel()</code></td>
<td><code>model.release()</code></td>
</tr>
<tr>
<td><code>debugIsim(isim)</code></td>
<td>Not implemented</td>
</tr>
<tr>
<td><code>debugSystemC(simulation)</code></td>
<td>Not implemented</td>
</tr>
<tr>
<td><code>getParameter(name)</code></td>
<td><code>target.parameters[&quot;name&quot;]</code></td>
</tr>
<tr>
<td><code>setParameter(name, value)</code></td>
<td><code>target.parameters[&quot;name&quot;] = value</code></td>
</tr>
<tr>
<td><code>getTargetList(filename)</code></td>
<td><code>model.get_target_info()</code></td>
</tr>
<tr>
<td><code>getTargetName()</code></td>
<td><code>target.instance_name</code></td>
</tr>
<tr>
<td><code>selectTarget(name)</code></td>
<td>Either of the following:</td>
</tr>
<tr>
<td></td>
<td>• <code>target = model.get_target(name)</code></td>
</tr>
<tr>
<td></td>
<td>• <code>cpus = model.get_cpus()</code></td>
</tr>
<tr>
<td><code>loadApp(filename)</code></td>
<td><code>target.load_application(filename)</code></td>
</tr>
<tr>
<td><code>saveState(filename)</code></td>
<td>Not implemented</td>
</tr>
<tr>
<td><code>restoreState(filename)</code></td>
<td>Not implemented</td>
</tr>
<tr>
<td><code>saveSession(filename)</code></td>
<td>Not implemented</td>
</tr>
<tr>
<td><code>openSession(filename)</code></td>
<td>Not implemented</td>
</tr>
<tr>
<td><code>setStateFile(filename)</code></td>
<td>Not implemented</td>
</tr>
</tbody>
</table>
### 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>
<thead>
<tr>
<th>MxScript function</th>
<th>iris.debug equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>run()</td>
<td>Either of the following:</td>
</tr>
<tr>
<td></td>
<td>model.run()</td>
</tr>
<tr>
<td></td>
<td>This function blocks until the target stops.</td>
</tr>
<tr>
<td></td>
<td>model.run(blocking=False)</td>
</tr>
<tr>
<td></td>
<td>This function is nonblocking.</td>
</tr>
<tr>
<td>runUntil(address)</td>
<td>Not implemented</td>
</tr>
<tr>
<td>runToLine(file, line)</td>
<td>Not implemented</td>
</tr>
<tr>
<td>stop()</td>
<td>model.stop()</td>
</tr>
<tr>
<td>getCurrentSourceFile()</td>
<td>Not implemented</td>
</tr>
<tr>
<td>getCurrentSourceLine()</td>
<td>Not implemented</td>
</tr>
<tr>
<td>getCurrentSourceColumn()</td>
<td>Not implemented</td>
</tr>
<tr>
<td>hardReset()</td>
<td>model.reset()</td>
</tr>
<tr>
<td>reset()</td>
<td>model.reset()</td>
</tr>
<tr>
<td></td>
<td>target.load_application(filename)</td>
</tr>
<tr>
<td>pause()</td>
<td>Not implemented</td>
</tr>
<tr>
<td>cont()</td>
<td>Not implemented</td>
</tr>
<tr>
<td>getStopCond()</td>
<td>Either of the following:</td>
</tr>
<tr>
<td></td>
<td>• target.get_hit_breakpoints()</td>
</tr>
<tr>
<td></td>
<td>• Return value of blocking model.run()</td>
</tr>
<tr>
<td>isSimStopped()</td>
<td>not target.is_running</td>
</tr>
<tr>
<td>restart()</td>
<td>model.reset()</td>
</tr>
<tr>
<td></td>
<td>target.load_application(filename)</td>
</tr>
<tr>
<td>goToMain()</td>
<td>Not implemented</td>
</tr>
<tr>
<td>step()</td>
<td>Not implemented</td>
</tr>
<tr>
<td>stepOver()</td>
<td>Not implemented</td>
</tr>
<tr>
<td>stepOut()</td>
<td>Not implemented</td>
</tr>
<tr>
<td>istep(count)</td>
<td>model.step()</td>
</tr>
<tr>
<td>getInstCount()</td>
<td>Not implemented</td>
</tr>
<tr>
<td>cycleStep(cycles)</td>
<td>Not implemented</td>
</tr>
<tr>
<td>enableStepBack(bool)</td>
<td>Not implemented</td>
</tr>
<tr>
<td>MxScript function</td>
<td>iris.debug equivalent</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------</td>
</tr>
</tbody>
</table>
| 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>
<thead>
<tr>
<th>MxScript function</th>
<th>iris.debug equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>bpAdd(address)</td>
<td><code>bp = target.add_bpt_prog(address)</code></td>
</tr>
<tr>
<td>bpAdd(file, line)</td>
<td>Not implemented</td>
</tr>
<tr>
<td>bpAddReg(reg_name)</td>
<td><code>bp = target.add_bpt_reg(reg_name)</code></td>
</tr>
<tr>
<td>bpAddMem(address)</td>
<td><code>bp = target.add_bpt_mem(address)</code></td>
</tr>
<tr>
<td>bpRemove(id)</td>
<td><code>bp.delete()</code></td>
</tr>
</tbody>
</table>
| 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>
<thead>
<tr>
<th>MxScript function</th>
<th>iris.debug equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>regWrite(name, value)</td>
<td>target.write_register(name, value)</td>
</tr>
<tr>
<td>regRead(name)</td>
<td>target.read_register(name)</td>
</tr>
<tr>
<td>memWrite(memspace, address, value)</td>
<td>target.write_memory(address, value[, memspace])</td>
</tr>
<tr>
<td></td>
<td>If memspace is not specified, the current memory space is used.</td>
</tr>
<tr>
<td>memRead(memspace, address, count)</td>
<td>target.read_memory(address, count[, memspace])</td>
</tr>
<tr>
<td></td>
<td>If memspace is not specified, the current memory space is used.</td>
</tr>
<tr>
<td>disassemble(address)</td>
<td>target.disassemble(address)</td>
</tr>
<tr>
<td>memStoreToFile(...)</td>
<td>with open(&quot;tempmem.bin&quot;, &quot;wb&quot;) as f:</td>
</tr>
<tr>
<td></td>
<td>mem = cpu.read_memory(0, count=1024)</td>
</tr>
<tr>
<td></td>
<td>f.write(mem)</td>
</tr>
<tr>
<td>memLoadFromFile(...)</td>
<td>with open(&quot;tempmem.bin&quot;, &quot;rb&quot;) as f:</td>
</tr>
<tr>
<td></td>
<td>mem = bytearray(f.read(1024))</td>
</tr>
<tr>
<td></td>
<td>cpu.write_memory(0, mem)</td>
</tr>
</tbody>
</table>
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.

It contains the following sections:

- 4.1 NetworkModel on page 4-24.
- 4.2 Model on page 4-25.
- 4.3 Target on page 4-28.
- 4.4 EventCallbackManager on page 4-38.
- 4.5 Breakpoint on page 4-40.
- 4.6 Exceptions on page 4-42.
4.1 NetworkModel

```python
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.

This section contains the following subsection:
• 4.1.1 __init__() on page 4-24.
```

4.1.1 __init__()

```python
__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.
4.2 Model

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

An Iris platform model.

This section contains the following subsections:

• 4.2.1 *get_target()* on page 4-25.
• 4.2.2 *get_targets()* on page 4-25.
• 4.2.3 *get_target_info()* on page 4-25.
• 4.2.4 *get_cpus()* on page 4-25.
• 4.2.5 *run()* on page 4-25.
• 4.2.6 *stop()* on page 4-26.
• 4.2.7 *step()* on page 4-26.
• 4.2.8 *reset()* on page 4-26.
• 4.2.9 *release()* on page 4-27.
• 4.2.10 *save_state()* on page 4-27.
• 4.2.11 *restore_state()* on page 4-27.

4.2.1 *get_target()

get_target(instance_name)

Obtain an interface to a target.

**Parameters**

instance_name

The instance name that corresponds to the target.

4.2.2 *get_targets()

get_targets()

Generator function to iterate over all targets in the simulation.

4.2.3 *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.

4.2.4 *get_cpus()

get_cpus()

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

4.2.5 *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.

### 4.2.6 stop()

```python
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.

### 4.2.7 step()

```python
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.

### 4.2.8 reset()

```python
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.

4.2.9 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.

4.2.10 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.

4.2.11 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.
4.3 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:

```python
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.

This section contains the following subsections:

- 4.3.1 load_application() on page 4-28.
- 4.3.2 add_bpt_prog() on page 4-29.
- 4.3.3 add_bpt_mem() on page 4-29.
- 4.3.4 add_bpt_reg() on page 4-29.
- 4.3.5 get_hit_breakpoints() on page 4-30.
- 4.3.6 clear_bpts() on page 4-30.
- 4.3.7 get_execution_state() on page 4-30.
- 4.3.8 set_execution_state() on page 4-30.
- 4.3.9 read_memory() on page 4-30.
- 4.3.10 write_memory() on page 4-30.
- 4.3.11 has_register() on page 4-31.
- 4.3.12 read_register() on page 4-32.
- 4.3.13 write_register() on page 4-32.
- 4.3.14 get_register_info() on page 4-32.
- 4.3.15 get_disass_modes() on page 4-33.
- 4.3.16 disassemble() on page 4-33.
- 4.3.17 get_steps() on page 4-33.
- 4.3.18 set_steps() on page 4-34.
- 4.3.19 get_instruction_count() on page 4-34.
- 4.3.20 get_pc() on page 4-34.
- 4.3.21 supports_tables() on page 4-34.
- 4.3.22 has_table() on page 4-34.
- 4.3.23 read_table() on page 4-34.
- 4.3.24 write_table() on page 4-35.
- 4.3.25 get_table_info() on page 4-35.
- 4.3.26 get_event_info() on page 4-36.
- 4.3.27 add_event_callback() on page 4-36.
- 4.3.28 remove_event_callback() on page 4-36.
- 4.3.29 handle_semihost_io() on page 4-36.
- 4.3.30 save_state() on page 4-36.
- 4.3.31 restore_state() on page 4-37.
- 4.3.32 Target properties on page 4-37.

### 4.3.1 load_application()

```python
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.

4.3.2 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.

4.3.3 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.

4.3.4 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.

4.3.5 get_hit_breakpoints()

get_hit_breakpoints()

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

4.3.6 clear_bpts()

clear_bpts()

Reset the state of all breakpoints.

4.3.7 get_execution_state()

get_execution_state()

Return True if execution state is enabled.

Exceptions

ValueError

Cannot get the execution state.

4.3.8 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.

4.3.9 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.

4.3.10 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.

4.3.11 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"

4.3.12 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.

4.3.13 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.

4.3.14 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:

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

```python
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()`.

### 4.3.15 get_disass_modes()

```python
get_disass_modes()
```
Return the disassembly modes for this target.

### 4.3.16 disassemble()

```python
disassemble(address, count = 1, mode = None, memory_space = None)
```
Disassemble instructions.

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.

### 4.3.17 get_steps()

```python
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.

4.3.18 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.

4.3.19 get_instruction_count()
get_instruction_count()
Return the current instruction count of the Target.

4.3.20 get_pc()
get_pc()
Return the current value of the program counter.

4.3.21 supports_tables()
supports_tables()
Return true if the target has any tables.

4.3.22 has_table()
has_table(name)
Return true if the target has the named table.

Parameters
name
The name of the table.
4.3.23 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.

4.3.24 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.

4.3.25 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.

4.3.26 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.

4.3.27 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.

4.3.28 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.

4.3.29 handle_semihost_io()

handle_semihost_io()

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

4.3.30 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.

4.3.31 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.

4.3.32 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.
4.4 EventCallbackManager

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

Manages user event callbacks for a particular target instance.

This section contains the following subsections:
- 4.4.1 get_info() on page 4-38.
- 4.4.2 get_evSrcId() on page 4-38.
- 4.4.3 add_callback() on page 4-38.
- 4.4.4 remove_callback_func() on page 4-38.
- 4.4.5 remove_callback_evSrcId() on page 4-38.

4.4.1 get_info()

```python
def get_info()
```

Yield EventSourceInfo for all events in the target instance.

4.4.2 get_evSrcId()

```python
def get_evSrcId(name)
```

Get the event source id for the named event.

**Parameters**

- name
  - Name of the event.

4.4.3 add_callback()

```python
def 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.

4.4.4 remove_callback_func()

```python
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.

4.4.5 remove_callback_evSrcId()

```python
remove_callback_evSrcId(evSrcId)
```

Remove a registered callback by event source id.
Parameters

`evSrcId`

The event source id for the callback function to remove.
4.5 Breakpoint

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

Provides a high level interface to breakpoints.

This section contains the following subsections:

• 4.5.1 enable() on page 4-40.
• 4.5.2 disable() on page 4-40.
• 4.5.3 delete() on page 4-40.
• 4.5.4 wait() on page 4-40.
• 4.5.5 Breakpoint properties on page 4-40.

4.5.1 enable()

enable()

Enable the breakpoint if the model supports it.

4.5.2 disable()

disable()

Disable the breakpoint if the model supports it.

4.5.3 delete()

delete()

Remove the breakpoint from the target.

4.5.4 wait()

wait(timeout = None)

Block until the breakpoint is triggered or the timeout expires.

Return True if the breakpoint was triggered, False otherwise.

4.5.5 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.
4.6 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.