Iris Python Debug Scripting

Version 1.0

User Guide

arm

101421_0100_00_en

Iris Python Debug Scripting

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	23 November 2018	Non-Confidential	New document.

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

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

http://www.arm.com

Contents

Iris Python Debug Scripting User Guide

Preface

This preface introduces the *Iris Python Debug Scripting User Guide*.

It contains the following:

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 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 3 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_00_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

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

iris.debug requires an existing installation of Python 2.7.* or Python 3.*. Python is available from *https://www.python.org/getit*.

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

- sh:

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

- tcsh:

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

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

On Windows:

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

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

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

1.2 Connecting to and running a model

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

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

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

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 *3.2 Model* on page 3-21 and *3.3 Target* on page 3-24.

Chapter 2
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:

2.1 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.

2.2 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 2-1 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

2.3 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 2-2 Execution control functions

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

Table 2-2 Execution control functions (continued)

MxScript function	iris.debug equivalent
`sleep(`*`seconds`*`)`	`import time` `time.sleep(seconds)`
`msleep(`*`milliseconds`*`)`	`import time` `time.sleep(milliseconds * 1000)`
`getCycleCount()`	Not implemented

2.4 Breakpoints

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

Table 2-3 Breakpoints functions

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

2.5 Model resource access

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

Table 2-4 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)`

Chapter 3
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 following:

- The fm.debug `LibraryModel` class, which is used to access a CADI model.
- Checkpointing. `Target.save_state()` and `Target.restore_state()` raise `NotImplementedError`.

It contains the following sections:

3.1 NetworkModel

```
class iris.debug.Model.NetworkModel(host, port, verbose = False)
```

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:

- *3.1.1 __init__()* on page 3-20.

3.1.1 __init__()

```
__init__(host, port, 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.

`verbose`
: If True, extra debugging information is printed.

3.2 Model

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

An Iris platform model.

This section contains the following subsections:

3.2.1 get_target()

`get_target(instance_name)`

Obtain an interface to a target.

Parameters

`instance_name`

The instance name that corresponds to the target.

3.2.2 get_targets()

`get_targets()`

Generator function to iterate over all targets in the simulation.

3.2.3 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.

3.2.4 get_cpus()

`get_cpus()`

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

3.2.5 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.

3.2.6 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.

3.2.7 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.

3.2.8 reset()

```
reset()
```

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

3.2.9 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.

3.2.10 is_checkpointable

`is_checkpointable`

True if the simulation has a checkpointing interface.

3.3 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.

This section contains the following subsections:

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 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.

3.3.5 get_hit_breakpoints()

```
get_hit_breakpoints()
```

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

3.3.6 clear_bpts()

```
clear_bpts()
```

Reset the state of all breakpoints.

3.3.7 get_execution_state()

```
get_execution_state()
```

Return True if execution state is enabled.

Exceptions

ValueError
: Cannot get the execution state.

3.3.8 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.

3.3.9 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.

3.3.10 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.

3.3.11 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*"

3.3.12 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.

3.3.13 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.

3.3.14 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()`.

3.3.15 get_disass_modes()

```
get_disass_modes()
```

Return the disassembly modes for this target.

3.3.16 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.

3.3.17 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.

3.3.18 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.

3.3.19 get_instruction_count()

```
get_instruction_count()
```

Return the current instruction count of the Target.

3.3.20 get_pc()

```
get_pc()
```

Return the current value of the program counter.

3.3.21 supports_tables()

```
supports_tables()
```

Return true if the target has any tables.

3.3.22 has_table()

```
has_table(name)
```

Return true if the target has the named table.

Parameters

name

The name of the table.

3.3.23 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.

3.3.24 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.

3.3.25 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.

3.3.26 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.

3.3.27 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.

3.3.28 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.

3.3.29 stdin_write()

`stdin_write(value)`

Write semihosted stdin to the target.

Parameters

value

Semihosted input string.

Exceptions

`NotImplementedError`
Target does not support semihosted input.

3.3.30 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_checkpointable`
True if the target has a checkpointing interface.

`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.

`stdout`
The target's semihosting stdout.

`stderr`
The target's semihosting stderr.

`target_name`
The name of the target component.

3.4 EventCallbackManager

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

Manages user event callbacks for a particular target instance.

This section contains the following subsections:

3.4.1 get_info()

`get_info()`

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

3.4.2 get_evSrcId()

`get_evSrcId(name)`

Get the event source id for the named event.

Parameters

`name`
Name of the event.

3.4.3 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.

3.4.4 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.

3.4.5 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.

3.5 Breakpoint

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

Provides a high level interface to breakpoints.

This section contains the following subsections:

3.5.1 enable()

`enable()`

Enable the breakpoint if the model supports it.

3.5.2 disable()

`disable()`

Disable the breakpoint if the model supports it.

3.5.3 delete()

`delete()`

Remove the breakpoint from the target.

3.5.4 wait()

`wait(timeout = None)`

Block until the breakpoint is triggered or the timeout expires.

Return True if the breakpoint was triggered, False otherwise.

3.5.5 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.

3.6 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.

exception iris.debug.ObtainInterfaceFailed

Bases: `exceptions.Exception`

Failed to obtain an interface object.