Python Debug Scripting for Fast Models

Version 1.0

Reference Manual

ARM DUI0851B

CONTENTS

1 Getting Started **1**
- 1.1 Installation . . . 1
- 1.2 Loading a model . . . 2

2 Upgrading MxScripts to Python **3**
- 2.1 Major Differences Between MxScript and Python . . . 3
- 2.2 Model connection and configuration . . . 4
- 2.3 Execution Control . . . 5
- 2.4 Breakpoints . . . 7
- 2.5 Model Resource Access . . . 8

3 API Reference **9**
- 3.1 Model . . . 9
- 3.2 Target . . . 11
- 3.3 Breakpoints . . . 16
- 3.4 Exceptions . . . 17

CHAPTER

ONE

GETTING STARTED

1.1 Installation

1.1.1 Prerequisites

fm.debug requires an existing installation of Python 2.7.*. It will not work with Python 2.6.* or earlier, or with Python 3.*. Python is available from http://www.python.org/getit/.

> NOTE: fm.debug relies on a Python extension module implemented in C and must be used with 'CPython' (the original Python implementation from python.org). Other Python implementations such as Jython do not support the C interface used by the extension module.

1.1.2 Linux

In order to use the fm.debug Python module you need to tell the Python interpreter where to find it. This can be done by adding the directory containing it to the PYTHONPATH environment variable.

sh:

```
export PYTHONPATH=$PVLIB_HOME/lib/python2.7:$PYTHONPATH
```

tcsh:

```
setenv PYTHONPATH $PVLIB_HOME/lib/python2.7:$PYTHONPATH
```

This will be done for you by the setup_all setup scripts.

Alternatively you can add the directory to the Python path from within the script before importing the module:

```
import sys, os
sys.path.append(os.path.join(os.environ['PVLIB_HOME'], 'lib', 'python2.7'))

import fm.debug
```

1.1.3 Windows

In order to use the fm.debug Python module you need to tell the Python interpreter where to find it. This can be done by adding the directory containing it to the PYTHONPATH environment variable:

```
set PYTHONPATH=%PVLIB_HOME%\lib\python27;%PYTHONPATH%
```

Alternatively you can add the directory to the Python path from within the script before importing the module:

```
import sys, os
sys.path.append(os.path.join(os.environ['PVLIB_HOME'], 'lib', 'python27'))

import fm.debug
```

1.2 Loading a model

The following example code shows how to load a model from a library file, load an application into the model and run the model:

```
import fm.debug

model = fm.debug.LibraryModel("/path/to/model.so")
cpu = model.get_cpus()[0]
cpu.load_application("/path/to/application.axf")
model.run()
```

This code creates two variables, `model` and `cpu`. The first is a `Model` object which represents the entire simulated system, composed of a number of targets including cores and memories. The model object can be used to access these targets and to start, stop and step the model. The second variable, `cpu`, is a `Target` object, in this case the first cpu in the model. This can be used to read and write the core's memory and registers, and to set and clear breakpoints.

For documentation on the operations that can be performed on models and targets, see *Model* and *Target*.

1.2.1 32 and 64-bit models

Models can be loaded in two ways. A CADI model library can be loaded in process using `fm.debug.LibraryModel` or an already running CADI remote simulation can be connected to using `fm.debug.NetworkModel`.

In order to load a 32-bit model library, the script must be run by a 32-bit Python interpreter (and similarly a 64-bit model requires a 64-bit Python interpreter).

Connecting to a remote simulation using `fm.debug.NetworkModel` does not have this limitation.

CHAPTER

TWO

UPGRADING MXSCRIPTS TO PYTHON

This page describes the major differences between the MxScript and Python languages, and the fm.debug equivalents of MxScript functions for interacting with a model. It is recommended that you also consult the Python documentation at http://www.python.org/doc/, for an introduction to the language features and the Python standard library.

2.1 Major Differences Between MxScript and Python

Each Python script which uses fm.debug must have the following line near the top:

```
from fm.debug import *
```

In MxScript, comment lines begin with `//`, whereas in Python they begin with `#`.

In Python, indentation is used to represent scope, instead of curly braces. This means that your indentation must be correct and consistent, and curly braces should 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 (such as if, for and while) end with a colon, and the block of code that they apply to is represented by it being indented relative to the flow-control statement. An empty block can be created using the `pass` statement, if needed. The `elif` statement can be used to check for multiple conditions, only one of which is true.

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

In Python, `for` loops always iterate over a list. The `range` function is provided to create a list of integers, as follows:

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

The following two loops, the first in MxScript and the second in Python, have the same function:

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

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

`while` loops behave similarly to their MxScript equivalents, but using the Python syntax rules of ending a flow control statement with a colon, and using indentation to represent scope:

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

Python does not have an equivalent of 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, the first in MxScript and the second in Python, are equivalent:

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

a = 5
b = "hello"
```

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

```
import fm.debug
from fm.debug import LibraryModel, NetworkModel
from fm.debug import *
```

- The first loads the fm.debug module (Python file, or collection of files) and adds fm.debug to the current namespace.
- The second loads the fm.debug module, and adds LibraryModel and NetworkModel to the current namespace, without making fm.debug or any other contents of it available.
- The third form adds the entire contents of the fm.debug module to the current namespace.

2.2 Model connection and configuration

In MxScript, there is 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 is used to switch between multiple targets.

In contrast to this, fm.debug uses an object-oriented design, where there are objects which represent models and targets, and these have methods to interact with them. This makes it much more practical to work with multiple targets or even models. An example of where this would be useful is debugging a multi-processor system, where it is necessary to interact with multiple CPU targets.

MxScript function	fm.debug equivalent
loadModel(filename,...)	`model = LibraryModel(filename)` (Note: does not select target)
connectModel(port)	`model = NetworkModel(host, port)` (Note: does not select target)
closeModel()	`model.release()`
debugIsim()	Not implemented
debugSystemC()	Not implemented

getParameter(name)	Before connecting to model:
	`LibraryModel.get_model_parameters (filename)["name"]`
	After connecting to model:
	`target.parameters["name"]`

setParameter(name, value)	Before connecting to model:
	`model = LibraryModel(filename, {"name": value})`
	After connecting to model:
	`target.parameters["name"] = value`

getTargetList(filename)	`model.get_target_info()`
getTargetName()	`target.instance_name`
selectTarget(name)	• `target = model.get_target(name)` • `cpus = model.get_cpus()`
loadApp(filename)	`target.load_application(filename)`
saveState(filename)	Not implemented
loadState(filename)	Not implemented
saveSession(filename)	Not implemented
openSession(filename)	Not implemented
setStateFile(filename)	Not implemented

2.3 Execution Control

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

MxScript function	fm.debug equivalent
run()	• `model.run()` - blocks until target stops • `model.run(blocking=False)` - non-blocking
runUntil()	Not implemented
runToLine()	Not implemented
stop()	`model.stop()`
getCurrentSourceFile()	Not implemented
getCurrentSourceLine()	Not implemented
getCurrentSourceColumn()	Not implemented
hardReset()	`target.reset()`
reset()	`target.reset()` `target.load_application()`
pause()	Not implemented
cont()	Not implemented
getStopCond()	• `target.get_hit_breakpoints()` • `return value of blocking model.run()`
isSimStopped()	`not target.is_running`
restart()	`target.reset()` `target.load_application()`
goToMain()	Not implemented
step()	Not implemented
stepOver()	Not implemented
stepOut()	Not implemented
istep()	`step()`
getInstCount()	Not implemented
cycleStep()	Not implemented
enableStepBack()	Not implemented
sleep(seconds)	`import time` `time.sleep(seconds)`
msleep(milliseconds)	`import time` `time.sleep(milliseconds * 1000)`
getCycleCount()	Not implemented

2.4 Breakpoints

MxScript function	fm.debug equivalent
bpAdd(address)	`bp = target.add_bpt_prog(address)`
bpAdd(file, line)	Not implemented
bpAddReg(register_name)	`bp = target.add_bpt_reg(register_name)`
bpAddMem(address)	`bp = target.add_bpt_mem(address)`
bpRemove(id)	`bp.delete()`

bpRemoveAll()	`for bp in target.breakpoints.values():`
	`  try:`
	`  bp.delete()`
	`  except ValueError:`
	`  # Global breakpoints`
	`  # cannot be deleted`
	`  pass`

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

MxScript function	fm.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 it is not specified, the current memory space is used
memRead(memspace, address, count)	`target.read_memory(address, count[, memspace])` If it 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

THREE

API REFERENCE

3.1 Model

There are two classes that can be used to access a CADI model: LibraryModel and NetworkModel. These are used, respectively, for creating a new instance of a model by loading it from a library, and connecting to a currently running CADI server. These classes are identical apart from their constructors, and provide methods for accessing components of the model (refered to as targets), and controlling the execution of the model.

class fm.debug.**LibraryModel**(*filename*, *parameters=None*, *verbose=False*)
Bases: fm.debug.Model.Model

A CADI model loaded from a DLL

__init__(*filename*, *parameters=None*, *verbose=False*)
Loads a model from a DLL, and initialises it

Parameters

- **filename** – The path of the file to load the model from
- **parameters** – Dictionary containing parameters of the model to set. The keys of this dictionary should be strings containing the names of the parameters, and the values should be strings, bools or ints containing the values of the parameters
- **verbose** – If True, extra debugging information is printed

classmethod get_model_parameters(*filename*)
Get a dictionary containing the default parameters for the model

Parameters **filename** – The path of the file to load the model parameters from

class fm.debug.**NetworkModel**(*host*, *port*, *verbose=False*)
Bases: fm.debug.Model.Model

A CADI model connected to a CADI server

__init__(*host*, *port*, *verbose=False*)
Connects to an already initalised CADI server

Parameters

- **host** – Hostname or IP address of the computer running the model.
- **port** – Port number that the model is listening on.
- **verbose** – If True, extra debugging information is printed

class fm.debug.Model.**Model**(*simulation*, *verbose*)
Bases: object

This class wraps a CADI 2.0 model.

get_target(*target_name*)
Obtain an interface to a target

Parameters **target_name** – The instance name corresponding to the desired target

get_cpus()
Returns all targets which have executes_software set

get_target_info()
Returns an iterator over namedtuples containing information about all of the target instances contained in the model

run(*blocking=True*, *timeout=None*)
Starts the model executing

Parameters

- **blocking** – If True, this call will block until the model stops executing (typically due to a breakpoint). If False, this call will return once that target has started executing.
- **timeout** – If None, this call will wait 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.

Raises TimeoutError If the timeout expires.

stop(*timeout=None*)
Stops the model executing

Parameters **timeout** – If None, this call will wait 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.

Raises TimeoutError If the timeout expires.

step(*count=1*, *timeout=None*)
Execute the target for *count* steps.

Parameters

- **count** – The number of processor cycles to execute.
- **timeout** – If None, this call will wait 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.

Raises TimeoutError If the timeout expires.

release(*shutdown=False*)
End the simulation and release the model.

Parameters **shutdown** – If True, the simulation will be shutdown and any other scripts or debuggers will have to disconnect.

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

is_checkpointable
Returns True if the Simulation has a checkpointing interface

save_state(*stream_delegate_or_directory*, *save_all=True*)
Save Simulation-wide state.

Parameters

- **stream_delegate_or_directory** – If this is a string it is treated as a directory name and the default checkpoint delegate is used to produce a checkpoint that is compatible with the –restore option in model_shell or an isim system.

  If this is not a string it is treated as a checkpoint delegate object. For an example of how to create a custom checkpoint delegate see the checkpointing example at: ${PVLIB_HOME}/examples/python2.7/checkpointing.py

- **save_all** – If True, save the state of the simulation and all targets inside the simulation that support checkpointing. If False, only save the simulation state. This parameter defaults to True.

If save_all is False and the simulation engine does not support checkpointing, this method will raise NotImplementedError

Returns True if all components saved successfully

restore_state(*stream_delegate_or_directory*, *restore_all=True*)

Restore Simulation-wide state.

Parameters

- **stream_delegate_or_directory** – If this is a string it is treated as a directory name and the default checkpoint delegate is used to read a checkpoint that is compatible with the –save option in model_shell or an isim system.

  If this is not a string it is treated as a checkpoint delegate object. For an example of how to create a custom checkpoint delegate see the checkpointing example at: ${PVLIB_HOME}/examples/python2.7/checkpointing.py

- **restore_all** – If True, restore the state of the simulation and all targets inside the simulation that support checkpointing. If False, only restore the simulation state. This parameter defaults to True.

If restore_all is False and the simulation engine does not support checkpointing, this method will raise NotImplementedError

Returns True if all components restored successfully

3.2 Target

class fm.debug.Target.**Target**(*cadi*, *model*)

Wraps a CADI object providing a simplified interface to common tasks.

Memory and registers can be accessed using methods, for example:

```
cpu.read_memory(0x1234, count=8)
cpu.write_register("Core.R5", 1000)
```

Breakpoints can be set using methods of this object:

```
cpu.add_bpt_mem(0x1234, memory_space="Secure", on_read=False)
cpu.bpt_add_reg("Core.CPSR")
```

These methods return Breakpoint objects, which allow enabling, disabling and deleting the breakpoint. The breakpoints currently set are accessible using the dictionary `Target.breakpoints`, which maps from breakpoint numbers to Breakpoint objects.

load_application(*filename*, *loadData=True*, *verbose=False*, *parameters=None*)

Load an application to run on the model.

Parameters

- **filename** – The filename of the application to load.
- **loadData** – 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** – A list of command line parameters to pass to the application or None.

reset()
Reset the simulation

This will reset the model to a state equivalent to the state immediately after instantiation.

add_bpt_prog(*address*, *memory_space=None*)
Set a new breakpoint, which will be hit when program execution reaches a 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

add_bpt_mem(*address*, *memory_space=None*, *on_read=True*, *on_write=True*, *on_modify=True*)
Set a new breakpoint, which will be hit when a 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 will be triggered when the memory location is read from.
- **on_write** – If True, the breakpoint will be triggered when the memory location is written to.
- **on_modify** – If True, the breakpoint will be triggered when the memory location is modified.

add_bpt_reg(*reg_name*, *on_read=True*, *on_write=True*, *on_modify=True*)
Set a new breakpoint, which will be hit when a 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 will be triggered when the register is read from.
- **on_write** – If True, the breakpoint will be triggered when the register is written to.
- **on_modify** – If True, the breakpoint will be triggered when the register is modified.

get_hit_breakpoints()
Returns the list of breakpoints that were hit last time the target was running

is_running
Returns True if the target is currently running

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

Parameters

- **address** – Address to begin reading from.
- **memory_space** – Name of the memory space to read or None which will read the core's current memory space.
- **size** – Size of memory access unit in bytes. Must be one of 1, 2, 4 or 8. Note that not all values are supported by all models. Note that the data is always returned as bytes, so calling with size=4, count=1 will return a byte array of length 4.
- **count** – Number of units to read.
- **do_side_effects** – If True, the target must perform any side-effects normally triggered by the read, for example clear-on-read.

Returns an integer if count is 1, otherwise returns a bytearray of length size*count

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

Parameters

- **address** – Address to begin reading from
- **data** – The data to write. If count is 1, this must be an integer Otherwise it must be a bytearray with length >= size*count
- **memory_space** – memory space to read. Default is None which will read the core's current memory space.
- **size** – Size of memory access unit in bytes. Must be one of 1, 2, 4 or 8. Note that 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** – If True, the target must perform any side-effects normally triggered by the write, for example triggering an interrupt.

has_register(*name*)
Does the target have the named 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>`

Returns True if the register exists and has an unambiguous name, or False otherwise.

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** – If True, perform side effects associated with the access. Default is False

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

write_register(*name*, *value*, *side_effects=False*)
Write a value to 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>`

- **value** – the value to write to the register
- **side_effects** – If True, perform side effects associated with the access. Default is False

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

get_register_info(*name=None*)
This method can be used to retrieve information about about the registers present in this Target.

It is used in two ways:

- **`get_register_info(name)`** will return the info for the named register
- **`get_register_info()`** The function acts as a generator and will yield information about all registers.

Parameters **name** – The name of the register to provide info for.

If None, it will yield information about all registers.

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

disass_mode
Returns the current disassembly mode for this target

get_disass_modes()
Returns the disassembly modes for this target

disassemble(*address*, *count=1*, *mode=None*, *memory_space=None*)
Disassemble intructions.

If count=1 this method will return a 3-tuple of: address, opcode, disassembly

where: address is the address of the instruction opcode is a string containing the instruction opcode at that address disassembly is a string containing the disassembled representation of the instruction

If count is > 0, this method will behave like a generator function which yields one 3-tuple per instruction disassembled.

Parameters

- **address** – Address to start disassembling from.
- **count** – Number of instructions to disassemble. Default value is 1.
- **mode** – Disassembly mode to use.Must be either None (the target's current mode will be used) or one of the values returned by get_disass_modes(). Default value 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 will be used. Default value is None

Raises NotImplementedError if the target does not support disassembly Raises ValueError if mode is not valid Raises ValueError is count is < 1

This method may yield fewer than count times if an error occurs during disassembly.

is_checkpointable
Returns True if the target has a checkpointing interface

save_state(*stream_delegate_or_directory*)
Save target's simulation state.

Parameters **stream_delegate_or_directory** – If this is a string it is treated as a directory name and the default checkpoint delegate is used to produce a checkpoint that is compatible with the –save and –restore options in model_shell or an isim system.

If this is not a string it is treated as a checkpoint delegate object. For an example of how to create a custom checkpoint delegate see the checkpointing example at: ${PVLIB_HOME}/examples/python2.7/checkpointing.py

If the target does not support checkpointing, NotImplementedError will be raised

Returns True on success

restore_state(*stream_delegate_or_directory*)
Restore target's simulation state.

Parameters **stream_delegate_or_directory** – If this is a string it is treated as a directory name and the default checkpoint delegate is used to read a checkpoint that is compatible with the –save and –restore options in model_shell or an isim system.

If this is not a string it is treated as a checkpoint delegate object. For an example of how to create a custom checkpoint delegate see the checkpointing example at: ${PVLIB_HOME}/examples/python2.7/checkpointing.py

If the target does not support checkpointing, NotImplementedError will be raised

Returns True on success

parameters
Dictionary of target's run-time parameters

stdin
Target's semihosting stdin

stdout
Target's semihosting stdout

stderr
Target's semihosting stderr

3.3 Breakpoints

class fm.debug.Breakpoint.**Breakpoint**(*target*, *cadi*, *req*, *bptNumber=None*, *description=None*)
Provides a high level interface to breakpoints.

enable()
Enable the breakpoint if supported by the model.

disable()
Disable the breakpoint if supported by the model.

delete()
Remove the breakpoint from the target

This breakpoint should not be accessed after this method is called

wait(*timeout=None*)
Block until the breakpoint is triggered or the timeout expires. Returns True if the breakpoint was triggered, False otherwise.

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

memory_space
The name of the memory space that this breakpoint is set in

Only valid for program and memory breakpoints

address
The memory address that this breakpoint is set on

Only valid for program and memory breakpoints

register
The name of the register that this breakpoint is set on

Only valid for register breakpoints

on_read
True if this breakpoint is triggered on read

Only valid for register and memory breakpoints

on_write
True if this breakpoint is triggered on write

Only valid for register and memory breakpoints

on_modify
True if this breakpoint is triggered on modify

Only valid for register and memory breakpoints

bpt_type
The name of the breakpoint type

Valid values are:

- Program
- Memory
- Register
- InstStep
- ProgramRange
- Exception
- UserDefined

Not all types are supported by all models, and only Program, Memory and Register are fully supported by fm.debug.

number
Identification number of this breakpoint

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

This is only valid until the breakpoint is deleted, and breakpoint numbers can be re-used.

enabled
True if the breakpoint is currently enabled

3.4 Exceptions

exception `fm.debug.`**`TargetError`**
Bases: `exceptions.Exception`

An error occurred while accessing the target

exception `fm.debug.`**`TargetBusyError`**
Bases: `fm.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 repeat the original call after the target reaches a stable state.

exception `fm.debug.`**`SecurityError`**
Bases: `fm.debug.Exceptions.TargetError`

Method failed because of an access being denied

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

exception `fm.debug.`**`TimeoutError`**
Bases: `fm.debug.Exceptions.TargetError`

Timeout expired while waiting for a target to enter the desired state

exception fm.debug.**SimulationEndedError**

Bases: fm.debug.Exceptions.TargetError

Attempted to call a method on a simulation which has ended