Intel DAL Python Command Line Interface User Guide

Intel Top Secret

Revision: 1.0.7

Intel DAL Python Command Line Interface User Guide: Intel Top Secret

Revision History

Revision 1.0.7	06.29.2015	Elizabeth Yeager
automated command updating from CLI help against release candidate 1528		
Revision 1.0.6	04.25.2013	Elizabeth Yeager
Converting doc to structured doc format with automated command updating from CLI help		
Revision 1.0.5	12.05.2012	Stephen Lepisto
Added new control variables 'ishalted' and 'isstopped'.		
Revision 1.0.4	12.05.2012	Stephen Lepisto
1. Added waitforevent command. 2. Added string support for breakpointID for br(), brnew(), brchange(), brenable(), brdisable(),brget(),brremove() methods.		
Revision 1.0.3	06.04.2011	Stephen Lepisto
Fixed br() example so breakpoints are properly starting at 1, not 0.		
Revision 1.0.2	03.15.2011	Stephen Lepisto
1. releasealthreadswhilestepping, stepintoexception, and enableoxmdebug. 2. Added NodeState.find() method. 3. Added calls() command (this has been available for some time, just not documented here).		
Revision 1.0.1	02.23.2012	Stephen Lepisto
Added loadthis command		

Revision 1.0	01.02.2012	Stephen Lepisto
1. Added python runtime to diagram in the overview section 2. Added example to "auto-start" files in Appendix F and note that the files are not created by default.		
Revision 0.9.6	12.08.2011	Stephen Lepisto
1. reset target now accepts a debug port parameter. Also in multi-debug port configuration, a debug port must be specified for resettarget() and resettap() or an error is raised (change in default behavior). 2. search() command has been changed to specify number of matches to return instead of depth of search.		
Revision 0.9.5	11.28.2011	Stephen Lepisto
Documented globalpath and spath control variables		
Revision 0.9.4	11.07.2011	Stephen Lepisto
1. Removed operations from the Address class (it is moved entirely to .Net) 2. Added 'stepping' attribute as a common attribute on the Node class. 3. Added section on the Python CLI startup sequence. 4. Added Appendix F to describe command line switches and autostart files. 5. Updated Builder Example to use Icast() in all the right places so the example can be executed without issue in IronPython. 6. Added documentation for some functions on itpii.cmdsControl and in its own section. This includes some new functionality. 7. Added Tip to description of the itpii package about the __fromembeddedpython__ attribute. 8. Added usemode control variable. 9. Added reconnectmasterframe() and stopmasterframe().		
Revision 0.9.3	11.02.2011	Stephen Lepisto
Added vmexitbreak and vmlaunchbreak control variables.		
Revision 0.9.2	11.02.2011	Stephen Lepisto
Added hookstatus() command.		
Revision 0.9.1	10.18.2011	Stephen Lepisto
Corrected examples in go() command to prefix GoTilType enumeration with 'itp.'		
Revision 0.9.0	09.29.2011	Stephen Lepisto
1. Format fixes for clarity, consistency. Added more classes, functions. 2. Updated discussion on state to reflect latest implementation (no init(), for example). 3. Added NodeState.save() method. 4. Added note to Events section about how to detect a "halt" by examining the break message. 5. Fixed long example in drscan().		
Revision 0.5.11	08.23.2011	Stephen Lepisto
1. Updated builder example in section 2.2 2. Added note to event handling about halt actually being a break event.		
Revision 0.5.10	08.08.2011	Stephen Lepisto
1. Added Address and AddressType. 2. Added BitData. 3. Added vmcsinfo(), vmcsptr()		
Revision 0.5.9	06.28.2011	Stephen Lepisto
1. Updated descriptions for drscan(), irdrscan(), and msgdr() to explain how the writeData/writeValue parameter now works with packed integers.		

2. Added drive_obs() method, OBSPins and PinOperation enumeration. 3. Added State.search() command. 4. Added filename parameter to the forcereconfig() command.		
Revision 0.5.8	06.28.2011	Stephen Lepisto
1. Added Introduction section for Target State. 2. Added reference sections for State class, StateRoot class, and ThreadStateRoot class. 3. Added itpii.memload(), and itpii.memsave() commands. 4. Added itpii.loggerclear(), itpii.loggerecho(), itpii.loggerlevel() commands. 5. Added itpii.bits() command. 6. Added itpii.libcall() command.		
Revision 0.5.7	06.02.2012	Stephen Lepisto
1. Added note to asm() and asmmode control variable that the asmmode is ignored unless a linear or physical address is used with the asm() command. 2. Added Remarks to memblock() and memdump() to describe how to perform an optimized memory read/write.		
Revision 0.5.6	06.02.2011	Stephen Lepisto
1. Fixed usage examples on br(), brnew(), brchange() to make addresses linear. 2. Added subscribe/unsubscribe functions to the itpii namespace for receiving events. 3. Added example for subscribing to events. 4. Added definition for 'namespace'. 5. Added cpuid_eax(), cpuid_ebx(), cpuid_ecx(), and cpuid_edx(). 6. Changed use of 'handle.handle' to be 'itpii.Handle', to reflect addition of new Handle class. 7. Added section 4.7 on ITP Data Types. 8. Added itpversion and version control variables. 9. Added holdhook(), pulsehook().		
Revision 0.5.5	05.12.2013	Stephen Lepeisto
1. Added example (section 1.2.4) on accessing python CLI from another module. 2. Added clarifications to the description of modules in Appendix D). 3. Added manualscans control variable and updated irscan()/drscan() commands. 4. Added itpii.print_last_exception() 5. Added 'guestphysical' to Eval(), along with 'code', 'data', and 'stack' sub-types. 6. Added num_activeprocessors and targpower control variables. 7. Added display() command to non-thread-specific commands. 8. Added polling() and pulsepwrgood() commands.		
Revision 0.5.4	05.02.2011	Stephen Lepisto
Added autoscandr(), autoscandrraw() commands		
Revision 0.5.3	04.19.2011	Stephen Lepisto
Added eval(), msgtimedelay(), tapresetidcodes(), upload(), vmcsread(), vmcswrite(), commands.		
Revision 0.5.2	04.1.2011	Stephen Lepisto
1. Changed show() to info() on Node, NodeContainer 2. Changed show() to display() on Register examples. 3. Added NodeContainer.infoall(), clone(). 4. Added ItpII.setDomain(), getDomain() 5. Added irdrscan() command 6. Changed all commands ending in 'formatted' to remove the 'formatted suffix and changed al corresponding commands without the suffix to now have the 'raw' suffix. This means that these commands now by default output a formatted string and the 'raw' versions return the numerical values. 7. Added obsfromsyncin() and obstosyncout() commands.		

Revision 0.5.1	03.14.2011	Stephen Lepisto
Updated based on latest implementation		
Revision 0.5	03.04.2011	Stephen Lepisto
Updated based on peer review comments		
Revision 0.2.3	01.26.2013	Stephen Lepisto
Added commands reference		
Revision 0.2.2	01.26.2011	Stephen Lepisto
Added new sections 1,2,3		
Revision 0.2.1	12.08.2010	Stephen Lepisto
Added Memory example in Appendix C		
Revision 0.2.0	11.02.2010	Stephen Lepisto
Initial Release of this version		

Table of Contents

List of Figures

List of Tables

1 Introduction

The Intel® Dfx Abstraction Layer (DAL) provides a rich source of functionality for debugging target platforms and associated silicon. The DAL abstracts away the target-specific details on how to access the features of interest, such as model-specific registers (MSRs), entering and exiting probe mode, various forms of hardware break detects, and more. The Python Command Line Interface (CLI) provides access to this functionality in an easy-to-use form while still allowing access to the more advanced features of the DAL, all using the well-known and mature Python language.

The Python CLI has two basic goals: provide easy access to the power of the DAL and provide a migration path for those who are accustomed to the command line used in the ITP (In-Target Probe) software. Due to these different goals, there are generally two ways to accomplish most tasks in the Python CLI: 1) Use the legacy commands that are familiar to users of ITP and 2) embrace the new paradigm of the DAL and take control of your debugging efforts.

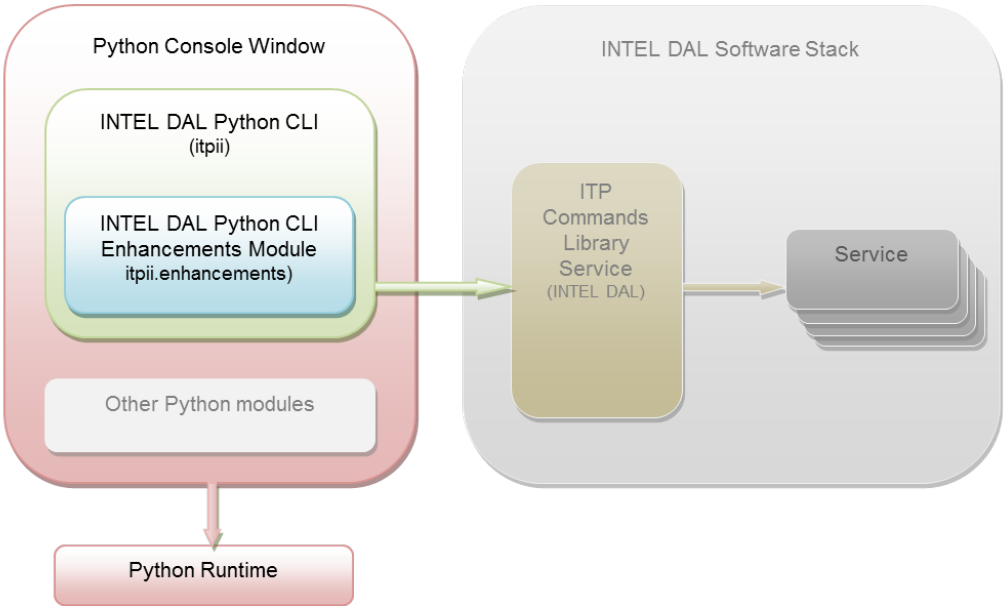
To support ITP users, most of the commands available on the ITP command line are available in the Python CLI. These commands offer easy access to the platform under test at the cost of being a discrete command with some overhead for each call. As mentioned before, the alternative is to access the power of the DAL directly, which is more efficient in terms of execution but more complex to accomplish. This overview is aimed at those who are accustomed to ITP in order to familiarize them with how the Python CLI works in conjunction with the DAL. A later section describes how to use the DAL directly from Python.

The python CLI supports command line switches to alter its behavior as well as "auto-start" files, which are processed before any other script. See Appendix F: Python CLI Command Line Switches for details.

1.1 Overview of the Python CLI

The Python CLI is a python package containing a set of Python modules that connect to the ITP Commands Library Service, which is part of the Intel DAL.

Figure 1.1. The Python Console and DAL Software Stack



The Python Console Window is the console provided with Intel DAL. Technically, any Python console should work; however, only the console window provided with Intel DAL has been fully tested.

The Intel DAL CLI Enhancements package is included in the Intel DAL CLI package. The enhancements are enabled by default if you start the Python CLI with the shortcut provided by the Intel DAL installation. Otherwise, importing the Python CLI modules manually does not enable the enhancements automatically. This allows the client to provide their own favorite enhancements.

The Intel DAL Python CLI is the entry point to accessing the Intel DAL software stack through the Intel DAL ITP Commands Library Service. It is assumed that the flavor of Python running in the Python Console Window can somehow access .Net assemblies (two such options are CPython with the Python.net add-on, and IronPython).

"Other Python modules" in the diagram is a placeholder to indicate that other Python modules that do not have direct connection with Intel DAL might be loaded at the same time.

1.2 For the ITP User in you

The Python CLI uses the Python language for everything. If you are not familiar with Python, please read Appendix D: A (Very) Brief Introduction to Python.

Python has many similarities to the ITP scripting language but there are also some fundamental differences. In addition, the DAL has a fundamentally different way of looking at target platforms and the devices on those targets. This affects how the Python CLI works. Whereas ITP provided a flat list of devices along with the concept of viewpoints to work with that list of devices, the Python CLI provides a tree of devices, with each device in the tree giving context to the commands that apply to just that device. Allow me to explain.

Figure 1.2. Typical device list from ITP (this one for an Emerald Lake platform using the Sandy Bridge CPU).

```
ID  DP TP SC Alias          Type         Step Idcode   BusType D/C /T/B Enabled
-------------------------------------------------------------------------------
0    0  0  0  SNDB_UC0       SNDB_UC      D2   6a688013 JTAG    0/- /-/- Yes
1    0  1  0  PROC0          SNDB         D2   7a768013 JTAG    0/0 /0/- Yes
2    0  1  0  PROC1          SNDB         D2   7a768013 JTAG    0/0 /1/- Yes
3    0  2  0  PROC2          SNDB         D2   7a768013 JTAG    0/1 /0/- Yes
4    0  2  0  PROC3          SNDB         D2   7a768013 JTAG    0/1 /1/- Yes
5    0  3  0  PROC4          SNDB         D2   7a768013 JTAG    0/2 /0/- Yes
6    0  3  0  PROC5          SNDB         D2   7a768013 JTAG    0/2 /1/- Yes
7    0  4  0  PROC6          SNDB         D2   7a768013 JTAG    0/3 /0/- Yes
8    0  4  0  PROC7          SNDB         D2   7a768013 JTAG    0/3 /1/- Yes
9    0  5  0  SNDB_GT0       SNDB_GT      D2   6a689013 JTAG    1/- /-/- Yes
```

Figure 1.3. Same set of devices in the Python CLI but expressed in tree form.

```
Default (LNodeDomain)
  None (LNodePkg)
    None (LNodeDie)
      SNB_UC0 (LNodeUncore)        ← Corresponds to Device ID 0
        SNB_GT0 (LNodeBox)         ← Corresponds to Device ID 9
      SNB_C0 (LNodeCore)
        SNB_C0_T0 (LNodeThread)    ← Corresponds to Device ID 1
        SNB_C0_T1 (LNodeThread)    ← Corresponds to Device ID 2
      SNB_C1 (LNodeCore)
        SNB_C1_T0 (LNodeThread)    ← Corresponds to Device ID 3
        SNB_C1_T1 (LNodeThread)    ← Corresponds to Device ID 4
      SNB_C2 (LNodeCore)
        SNB_C2_T0 (LNodeThread)    ← Corresponds to Device ID 5
        SNB_C2_T1 (LNodeThread)    ← Corresponds to Device ID 6
      SNB_C3 (LNodeCore)
        SNB_C3_T0 (LNodeThread)    ← Corresponds to Device ID 7
        SNB_C3_T1 (LNodeThread)    ← Corresponds to Device ID 8
```

This is a simple view that makes the tree nature clear. The 'LNodexxx' labels identify Logical Node types. So a logical domain contains one or more logical packages, which in turn contains one or more logical dies, and which in turn contains one or more logical uncores, cores, and chipsets. A logical uncore (or chipset and possibly core) can contain one or more logical collections of functionality called a "box" (a 'box' in this case is nothing more than a general label for a feature in the device, such as the Management Engine (ME) or Power Control Unit (PCU). In other words, any design unit that is not a core, uncore, or chipset is called a 'box'). A core contains one or more logical threads. Notice how the various elements relate to each other and the devices contained within each element. This is how the context mentioned earlier is made evident.

Figure 1.4. A more detailed view of the tree, showing the domain, package, die, uncore, and first core with threads, along with some details of each node in the tree

```
Node
   type        = domain
   name        = 'Default'
   Number of packages = 1
   Node
     type        = package
     name        = 'None'
     instance = 0
     Number of dies = 1
     Node
       type        = die
       name        = 'None'
       dieid       = None
       Number of uncores = 1
       Number of cores   = 4
       Node
         type        = uncore
         name        = 'SNB_UC0'
         did         = 0
         tapport     = 0
         Node
           type        = box
           name        = 'SNB_GT0'
           instance    = None
           did         = 9
       Node
         type        = core
         name        = 'SNB_C0'
         devicetype  = SNB
         stepping    = D2
         coreid      = 0
         irlength    = 8
         tapport     = 1
         Number of threads = 2
         Node
           type        = thread
           name        = 'SNB_C0_T0'
           did         = 1
           coreid      = 0
           threadid    = 0
           threadalias = 'P0'
           devicetype  = SNB
           stepping    = D2
         Node
           type        = thread
           name        = 'SNB_C0_T1'
           did         = 2
           coreid      = 0
           threadid    = 1
           threadalias = 'P1'
           devicetype  = SNB
           stepping    = D2
```

ITP used the concept of viewpoints to manage accessing the various devices on a platform. The user would set the current viewpoint then most of the commands entered would apply only to that viewpoint. Sometimes it was necessary to direct a command to a different device from the current one so it was possible to specify a "viewpoint override" that temporarily directed the command to that other device. And sometimes it was necessary to address all the viewpoints at once with a special viewpoint override called "[all]".

The DAL uses a device tree and as such, the Python CLI does not have the concept of viewpoints. Instead, commands either take a device node as a parameter or the device node has the command available directly from that node (only for thread nodes). If you want to target all applicable devices, use the command without parameters from the main namespace.

In the next section, the examples will get you used to using the Python CLI. Install the Intel DAL, hook up some hardware, and work through these examples.

1.2.1 The Great Runtime Example

Perhaps one of the single most common operations desired by users of ITP and the DAL is the ability to "halt" the processors or put them into "probe mode" so as to examine the state of the devices. When the processor is in probe mode, all its registers can be examined and even modified. Later, the processors are resumed so the platform can continue whatever it was doing before entering probe mode (assuming you didn't change some memory or register to alter the behavior of the processor).

This example is given as a series of steps that show how to start the Python CLI, and how to put one or all processors into probe mode and then take them out of probe mode. This is done from the Python CLI. A later example does something interesting with this ability to halt and resume a processor.

Note: This example assumes you are running on Windows with an installed version of the Intel DAL and you have attached to a suitable target platform.

Features demonstrated:

- Starting Python CLI
- Putting all processors into probe mode (halt)
- Putting selected processors into probe mode (halt)
- Taking all processors out of probe mode (go)
- Taking selected processor out of probe mode (go)
- How the Python prompt changes entering and exiting probe mode
- Using the 'breakall' control variable.

1. Start the Python CLI with one of the following options:

   1.a.Select the "Python Console" option in the Start menu (typically All Programs > Intel > Intel Dfx Abstraction Layer > Python Console")

1.b. Use Windows Explorer to navigate to the DAL's installation folder (typically "C:\Program Files (x86)\Intel\DAL") and double-click on the pythonconsole.cmd file).

1.c. After a short while, the command prompt will appear ('>>?').

2. Type itp.halt() and press Enter. This puts all the threads into probe mode. A series of messages describe how each thread entered probe mode and at what address they were executing at the time. Note how the prompt changed to '>>>'.

Figure 1.5. CLI output after halt command.

```
[SNB_C0_T0] Halt Command break at 0x38:000000007A4AA693
[SNB_C0_T1] BreakAll break at 0x38:000000007A480BA7
[SNB_C1_T0] BreakAll break at 0x38:000000007A480BA3
[SNB_C1_T1] BreakAll break at 0x38:000000007A480BA7
[SNB_C2_T0] BreakAll break at 0x38:000000007A480BA7
[SNB_C2_T1] BreakAll break at 0x38:000000007A480BA3
[SNB_C3_T0] BreakAll break at 0x38:000000007A480BA3
[SNB_C3_T1] BreakAll break at 0x38:000000007A480BA7
```

3. Type itp.go() and press Enter. Nothing appears to happen except the prompt changed back to '>>?'. However, the target is now running again.

4. Type itp.cv.breakall and press Enter. This displays the current breakall setting. The output will look something like this:

Figure 1.6. CLI output after querying the control variable breakall.

```
Global
```

This shows that breakall is set for the global mode, which causes all processors on the same debug port to be halted when one processor is halted.

5. Type itp.cv.breakall = "off" and press Enter. This step uses a control variable called 'breakall' to change the behavior of the DAL in regards to what happens to other threads when a specific thread is told to halt (that is, enter probe mode). In this case, we are telling the DAL to halt only one specific thread and leave the others alone. This control variable has global consequences (across all thread devices) and so it is accessed through the main command object ('itp').

6. Type itp.threads[0].halt() and press Enter. This tells the first thread to enter probe mode.

Figure 1.7. CLI output after halt command on a thread.

```
[SNB_C0_T0] Halt Command break at
0x38:000000007A8650CE
[SNB_C0_T1] Multithreaded break at
0x38:000000007A863CB8
```

Note how only these two threads are affected.

7. Type itp.status() and press Enter. This displays the current status of all threads (because this command is called from the main command object).

   Figure 1.8. CLI output after status command.

```
Status for       : SNB_C0_T0
  Processor      : Halted          <-
  Processor mode : Protected, Paged, SixtyFourBit
Status for       : SNB_C0_T1
  Processor      : Halted          <-
  Processor mode : Protected, Paged, SixtyFourBit
Status for       : SNB_C1_T0
  Processor      : Running
  Processor mode : Unavailable while running
Status for       : SNB_C1_T1
  Processor      : Running
  Processor mode : Unavailable while running
Status for       : SNB_C2_T0
  Processor      : Running
  Processor mode : Unavailable while running
Status for       : SNB_C2_T1
  Processor      : Running
  Processor mode : Unavailable while running
Status for       : SNB_C3_T0
  Processor      : Running
  Processor mode : Unavailable while running
Status for       : SNB_C3_T1
  Processor      : Running
  Processor mode : Unavailable while running
```

   Note how the first two threads, SNB_C0_T0 and SNC_C0_T1 are halted the red arrows) and the others are still running.

8. Type itp.threads[0].go() and press Enter. This tells the first thread to exit probe mode (resume execution). Because the second thread is sympathetic to the first thread, the second thread also resumes. You can confirm this by typing itp.status() again.

9. Type itp.cv.breakall = "global" and press Enter. This restores the breakall setting to the default so all threads halt and resume when one thread is halted or resumed.

Congratulations! You can now halt and resume processors on a target with the Python CLI!

1.2.2 The Reading Registers Example

This example shows how to halt a processor and then read some architectural registers from the threads on the processor

Features Demonstrated:

- Putting all processors into probe mode (halt)
- Taking all processors out of probe mode (go)

- Create alias to a thread device node.• Access architectural registers rip and rflags.
- Access a single sub-component of rflags using long syntax.
- Access a single sub-component of rflags using shortcut syntax.
- Display sub-components of architectural registers rip and rflags.
- Store the value of a sub-component of rflags into a variable.
- Change a sub-component of rflags.
- Restore the value of a sub-component of rflags saved in a variable.

1. Make sure the Python CLI is running.
2. Type itp.halt() and press Enter. This puts all threads into probe mode.

   Figure 1.9. CLI output after halt command.

   ```
   [SNB_C0_T0] Halt Command break at 0x38:000000007A4AA693
   [SNB_C0_T1] BreakAll break at 0x38:000000007A480BA7
   [SNB_C1_T0] BreakAll break at 0x38:000000007A480BA3
   [SNB_C1_T1] BreakAll break at 0x38:000000007A480BA7
   [SNB_C2_T0] BreakAll break at 0x38:000000007A480BA7
   [SNB_C2_T1] BreakAll break at 0x38:000000007A480BA3
   [SNB_C3_T0] BreakAll break at 0x38:000000007A480BA3
   [SNB_C3_T1] BreakAll break at 0x38:000000007A480BA7
   ```

3. Type p0 = itp.threads[0] and press Enter. This creates an alias to the first thread. Typing the alias is shorter than typing itp.threads[0].
4. Type p1 = itp.threads[1] and press Enter. This creates an alias for the second thread.
5. Type p0.state.regs.rip and press Enter. This displays the instruction pointer for the first thread.

   Figure 1.10. CLI output after read of rip register.

   ```
   [64b] 0x000000007A4AA693
   ```

6. Type p1.state.regs.rip and press Enter. This displays the instruction pointer for the second thread. The two values should be different since the threads were likely running different code.
7. Type p0.state.regs.rip.display() and press Enter. This displays the rip register and all sub-components of that register, along with the values in each sub-component.

   Figure 1.11. CLI output after p0.state.regs.display command.

   ```
   rip = 0x000000007a4aa693
   rip.eip = 0x7a4aa693
   rip.eip.ip = 0xa693
   ```

8. Type p0.state.regs.rflags.eflags.cf and press Enter. This displays the contents of a single sub-component of the rflags register (in this case, the carry flag bit).

   Figure 1.12. CLI output after p0.state.regs.rflags.eflags.cf command.

   ```
   [1b] 0x0
   ```

9. Type p1.state.regs.rip and press Enter. This displays the instruction pointer for the second thread. The two values should be different since the threads were likely running different code.

10. Type p0.state.regs.rip.display() and press Enter. This displays the rip register and all sub-components of that register, along with the values in each sub-component.

    Figure 1.13. CLI output after p0.state.regs.display command.

    ```
    rip = 0x000000007a4aa693
    rip.eip = 0x7a4aa693
    rip.eip.ip = 0xa693
    ```

11. Type p0.state.regs.rflags.eflags.cf and press Enter. This displays the contents of a single sub-component of the rflags register (in this case, the carry flag bit).

    Figure 1.14. CLI output after p0.state.regs.rflags.eflags.cf command.

    ```
    [1b] 0x0
    ```

12. Type p0.state.regs.rflags.cf and press Enter. This is a shortcut to the 'cf' sub-component, since there is only one 'cf' sub-component in the entire rflags register.

    Figure 1.15. CLI output after p0.state.regs.rflags.eflags.cf command.

    ```
    [1b] 0x0
    ```

13. Type savedRflags = p0.state.regs.rflags.value and press Enter. This stores the value of the rflags register into the variable v (in Python, a variable is defined the first time it is used). Important: To read the value from a register like this, you must use the 'value' attribute.

14. Type p0.state.regs.rflags = 0xffffffff and press Enter. This changes the rflags register to all 1's.

15. Type p0.state.regs.rflags.display() and press Enter. Note how all the sub-components are set.

16. Type p0.state.regs.rflags = savedRflags and press Enter. This restores the rflags register to its original value.

17. Type itp.go() and press Enter. The target is now running again.

Congratulations again! You have successfully created aliases for two threads, navigated into the architectural registers for two threads, discovered how to use shortcuts on the registers, and how to save, change, and restore the value of a register.

1.2.3 Climbing the Device Tree Example

In this example, we are going to explore the device tree and how to find and use devices from the tree.

Features Demonstrated:

- Display the entire tree with details.
- Display the details of a single device node.
- Navigate the device tree in Python using 'dotted' notation.
- Create alias to a thread device node
- Search the device tree for a device node of a specific type.
- Search the device tree for a device node with a specific name.
- Display information about a device node.

1. The device tree starts at itp.domains. Enter itp.domains.infoall() and press Enter. A very long list of devices nodes will scroll by.

Figure 1.16. Partial list of device nodes.

```
Node
  type            = domain
  name            = Default
  # packages      = 1
  Node
    type            = package
    name            = <NONE>
    instance        = 0
    # dies          = 1
    Node
      type            = die
      name            = <NONE>
      dieid           = 0
      # uncores       = 1
      # cores         = 4
      Node
        type            = uncore
        name            = SNB_UC0
        did             = 0
        tapport         = 0
        Node
          type            = box
          name            = SNB_GT0
          instance        = None
          did             = 9
```

2. Enter itp.domains[0].info() and press Enter.

Figure 1.17. Node info

```
Node
  type            = domain
  name            = Default
  # packages      = 1
```

Not too exciting so let's move on to something more interesting.

3. Enter itp.domains[0].packages[0].dies[0].cores[0].threads[0].info() and press Enter.

Figure 1.18. thread info

```
Node
  type            = thread
  name            = SNB_C0_T0
  did             = 1
  threadid        = 0
  threadalias     = P0
  coreid          = 0
  devicetype      = SNB
  stepping        = D2
```

Now that's a little more like it.

4. Enter p0 = itp.domains[0].packages[0].dies[0].cores[0].threads[0] and press Enter. This creates an alias for thread 0.

5. Enter p0.info() and press Enter. This demonstrates that the alias is actually the same thread object as was displayed in step 3.

6. Enter uncores = itp.domains.getAllByType("uncore") and press Enter. This call retrieves all uncore objects in all target domains. uncores is a list of those uncore devices.

7. Enter uncores.infoall() and press Enter. This displays the contents of the list of uncore devices.

Figure 1.19. uncore info

```
Node
  type            = uncore
  name            = SNB_UC0
  did             = 0
  tapport         = 0
  Node
    type            = box
    name            = SNB_GT0
    instance        = None
    did             = 9
```

8. Enter p1 = itp.domains.getByName("SNB_C0_T1") and press Enter. This searches all target domains and retrieves the first thread named "SNB_C0_T1". This works for any device, if you know the name of that device.

Congratulations! You now have a grasp of how to navigate the device tree in the Python CLI.

1.2.4 Python CLI in your own Python Module

A common scenario is to create a script using python CLI commands to accomplish some specific task. Python imposes certain rules on how script files or modules interact with each other. In brief, python puts each module in its own namespace when loaded. The name of the namespace is the name of the module without the .py extension. All types, functions, and code defined in that module live in that namespace. You can access those types and functions from another module but you have to go through the namespace to do so. It is possible to pull or import those types from another module into your module so you can use them without using the namespace.

This example creates a module files that demonstrates how to import the python CLI commands, import the BitData type, and use them.

1. Open Notepad (or other text editor of your choice), and enter the following code:

```
import sys, os
dalDir = r"c:\Intel\DAL"   # default location
if "DalInstallDir" in os.environ:
    dalDir = os.environ["DalInstallDir"]
if dalDir not in sys.path:
    sys.path.append(dalDir)

import itpii
itp = itpii.baseaccess()
from itpii import BitData


def verifyValue(value, valueName):
    """Verify the value is not None and is an integer.
       Raise exception if otherwise.
    """

    if value is None:
        raise ValueError("'%s' cannot be None"%valueName)
    if type(value) not in [int, long]:
        raise TypeError("'%s' must be an integer, not a %s."%(valueName, type(value)))


def composeData(lowValue, middleValue, highValue):
    """Create and return a 32-bit BitData object with lowValue at bits 8:0,
    middleValue at bits:15:8, and highValue at bits 31:16.
    """

    verifyValue(lowValue, "lowValue")
    verifyValue(middleValue, "middleValue")
    verifyValue(highValue, "highValue")

    data = BitData(32)
    # Use the general purpose bits() command.
    # Could also use methods on the BitData object itself.
    itpii.bits(data, 0, 8, lowValue & 0xff)
    itpii.bits(data, 8, 8, middleValue & 0xff)
    itpii.bits(data, 16, 16, highValue & 0xffff)
    return data


def getThread(thread):
    """Convert the thread to NodeThread object if possible.
       Raise exception to indicate possible error.

    thread could be an index into the itp.threads list, or it could be an alias or
    name of a thread, or it could be a ThreadNode object (that is, an object from the
    itp.threads list).
    """

    if thread is None:
        raise ValueError("Must specify a thread.")
    if type(thread) in [int, long]:
        # Treat parameter as an index into the itp.threads list.
        if thread < 0 or thread > len(itp.threads):
            raise ValueError("thread ID %d is out of bounds."%thread)
        return itp.threads[thread]
    elif type(thread) in [str, unicode]:
        # Treat parameter as an alias or name in the itp.threads list.
```

```
        return itp.threads[thread]
    elif type(thread) not in [NodeThread]:
        raise TypeError("Thread must be an integer or a NodeThread object.")
    # The parameter is already a NodeThread so just return it.
    return thread


def writeToAddress(thread, address, lowValue, middleValue, highValue):
    """Write the three values as a 32-bit number to the specified address
       in the specified thread's memory space.  If necessary, halt the thread
       first (if the thread was not halted on entry, it will be resumed on exit).
    """

    writeData = composeData(lowValue, middleValue, highValue)
    threadNode = getThread(thread)
    wasRunning = False
    if threadNode.cv.isrunning:
        threadNode.halt()
        wasRunning = True
    try:
        threadNode.mem(address, 4, writeData)
        readData = threadNode.mem(address, 4)
        if writeData!= readData:
            print("Data read (%s) is not the data written (%s)!"%(readData,
                                                                  writeData))
        else:
            print("Successfully wrote %s to address %s"%(writeData, address))
    finally:
        if wasRunning:
            threadNode.go()
```

2. Save the file to the name 'writememtest.py' to a known folder.

3. Open the python CLI. When the prompt is available, enter the following, setting the path to the folder containing the 'writememtest.py.' file in the append() command:

```
import sys

sys.path.append(r"PATH TO FOLDER CONTAINING writememtest.py>")

import writememtest

writememtest.writeToAddress(0, "0x1000P", 0x11, 0x22, 0x3333)
```

Congratulations! You have successfully created and executed your own custom python script that uses python CLI commands and the BitData type.

1.2.5 Listening for Events

Sometimes you want a python script to respond to certain events exposed by the DAL. In ITP, you would use the IItpCallback COM interface. The ITP Commands Library exposes similar events as received by the IItpCallback interface but how you connect to those events is very different. The Python CLI provides a simple mechanism for subscribing to those events.

The following example shows how to subscribe to hardware events to show a power loss and power restore event. A more complete example can be found in Example 2 in Appendix B.

1. Start the python CLI and enter (or copy and paste) the following:

```
def reportHardwareEvents(args):
    if args.iEvent == itpii.HardwareEvents.PowerFailed:
            print("Ahhh, we lost power.")
    elif args.iEvent == itpii.HardwareEvents.PowerRestored:
        print("Yay!  We have power!.")

itpii.subscribeToHardwareEvents(reportHardwareEvents)
itp.pulsepwrgood()
```

2. After the power is restored and you see the power has come back, enter the following to no longer receive hardware events:
   `itpii.unsubscribeFromHardwareEvents(reportHardwareEvents)`

Congratulations! You have just subscribed to and unsubscribed from hardware events. Remember to check out Example 2 in Appendix B, which shows how to subscribe to and recognize all of the events.

1.3 Target State

In the DAL, device nodes have properties known collectively as Target State. This target state represents values specific to a device at a specific moment in time. For example, architectural registers and their values are considered state as is the collection of tap command registers and their values. Some device nodes have many hundreds or even thousands of pieces of state. To organize all these properties, the DAL uses what is called a Target State Tree or State Tree.

A state tree is a hierarchical collection of state, organized into state nodes. Each state node can have one or more child nodes and each child node in turn can have one more children of its own. Each property in the state tree can be reached by specifying each node in the tree that leads to the desired property. For example, to get to the eax register in the state tree on thread 0, use 'itp.threads[0].state.archreg.rax.eax'. This 'dotted' notation is the python syntax for accessing attributes and methods of a class instance and is often called a 'path', since it describes the path to negotiate to get from starting point (the root of the tree) to a desired node. For state that has a number of frames or instances associated with it, use the array notation '[n]'. For example, 'itp.threads[0].state.mlc_cache.set[3].way[4]'.

Each state node has a value with 1 or more bits and this value is represented by a BitData object. This value can be seen as an accumulation of the values of all the state nodes below a particular state node. Or, it can be seen as the child nodes being just decoration (labels) for individual ranges of bits in a state node's value.

1.3.1 Accessing State

In the python CLI, state is available on the 'state' attribute for a device node. Each uncore, thread, and chipset has state (technically, all logical device nodes have state but only uncore, thread, and chipset devices are currently supported by the DAL for state access). Entering 'itp.threads[0].state.' and pressing Tab will load the first level of state. Note: The first access to state may take a little longer than normal while the device's state is read in.

Be aware that some state, such as architectural registers, cannot be accessed unless the CPU is in probe mode.

If a reconfiguration event occurs that causes the device trees to change (for example, a target is powered off and a core is enabled or disabled), all the state trees associated with the devices also become disabled and have to be re-initialized once again.

1.3.2 Showing State

State comes in two forms, decorated and undecorated. The decorated form just means that the state is broken down into fields and each field has a label associated with it. Undecorated means the raw value, as contained in a BitData object. Entering the name or path of the state node on the python CLI command line and pressing enter shows the decorated state. To get the undecorated form, use the 'value' attribute on the state node in question. For example, the following shows the decorated state for the rax register and then shows the undecorated value of that same register.

```
>>? itp.halt()
    [HSW_C0_T0] Halt Command break at 0x0:0000000000000000
    [HSW_C0_T1] Halt Command break at 0x0:0000000000000000
    [HSW_C1_T0] Halt Command break at 0x0:0000000000000000
    [HSW_C1_T1] Halt Command break at 0x0:0000000000000000
    [HSW_C2_T0] Halt Command break at 0x0:0000000000000000
    [HSW_C2_T1] Halt Command break at 0x0:0000000000000000
    [HSW_C3_T0] Halt Command break at 0x0:0000000000000000
    [HSW_C3_T1] Halt Command break at 0x0:0000000000000000
>>> itp.threads[0].state.archreg.rax           # DECORATED STATE
rax('63:0') = 0x0000000000000042
eax('31:0') = 0x00000042
ax('15:0') = 0x0042
  ah('7:0') = 0x00
  al('7:0') = 0x42
>>> itp.threads[0].state.archreg.rax.value     # UNDECORATED STATE
[64b] 0x0000000000000042
```

1.3.3 Using Local State Trees

Sometimes it is desirable to copy a portion of the state tree into what is called a local state tree. Changes made to this local state tree do not affect the main device state tree until the changes are committed. You create a local state tree with the get() method on the 'state' attribute and you commit the changes using the put() method.

Here is a trivial example showing a local state tree.

```
>>? itp.halt()
>>> p0 = itp.threads[0]
    [HSW_C0_T0] Halt Command break at 0x0:0000000000000000
    [HSW_C0_T1] Halt Command break at 0x0:0000000000000000
    [HSW_C1_T0] Halt Command break at 0x0:0000000000000000
    [HSW_C1_T1] Halt Command break at 0x0:0000000000000000
    [HSW_C2_T0] Halt Command break at 0x0:0000000000000000
    [HSW_C2_T1] Halt Command break at 0x0:0000000000000000
    [HSW_C3_T0] Halt Command break at 0x0:0000000000000000
    [HSW_C3_T1] Halt Command break at 0x0:0000000000000000
>>> rax = p0.state.archreg.rax.get()   # create local state tree
>>> rax
rax('63:0') = 0x0000000000000000
eax('31:0') = 0x00000000
ax('15:0') = 0x0000
  ah('7:0') = 0x00
  al('7:0') = 0x00
>>> rax.eax = 1
>>> rax   # show changes to local rax
rax('63:0') = 0x0000000000000001
eax('31:0') = 0x00000001
ax('15:0') = 0x0001
  ah('7:0') = 0x00
  al('7:0') = 0x01
>>> p0.state.archreg.rax   # show that main device tree is unchanged
rax('63:0') = 0x0000000000000000
eax('31:0') = 0x00000000
ax('15:0') = 0x0000
  ah('7:0') = 0x00
  al('7:0') = 0x00
>>> p0.state.archreg.rax.put(rax)    # commit changes back to main device tree
>>> p0.state.archreg.rax   # and show the changes in the main device tree
rax('63:0') = 0x0000000000000001
eax('31:0') = 0x00000001
ax('15:0') = 0x0001
  ah('7:0') = 0x00
  al('7:0') = 0x01
```

1.3.4 Searching State

There are times when you may know the name of a register or data node but you do not know precisely where that register is. Use the search() command to find everywhere that register is located. The search() command uses a simple wildcard matching scheme similar to how filenames are matched in a command line. The search() command displays the full state path to every matching node.

Examples:

Find a specific data register (in this case, the architectural register 'rax').

```
>>> itp.threads[0].state.search("rax")
HSW_C0_T0.archreg.rax
```

Find all data registers that start with 'r' and end with 'x' and have any character in the middle.

```
>>> itp.threads[0].state.search("r?x")
HSW_C0_T0.archreg.rcx
HSW_C0_T0.archreg.rbx
HSW_C0_T0.archreg.rdx
HSW_C0_T0.archreg.rax
```

Find all data registers that start with 'r' and end with 'x' with zero or more characters between them.

```
>>> itp.threads[0].state.search("r*x")
HSW_C0_T0.rs_ldmtx
HSW_C0_T0.rs_eumtx
HSW_C0_T0.archreg.rcx
HSW_C0_T0.archreg.rbx
HSW_C0_T0.archreg.rdx
HSW_C0_T0.archreg.rax
```

2 Accessing the DAL Directly

Accessing the DAL services directly takes some effort from Python but once mastered, this approach provides full access to the DAL functionality.

Here is a summary of what needs to be accomplished:

1. Tell Python where to find the assemblies of the DAL.
2. Tell Python which assemblies to load.
3. Import the appropriate namespaces and/or types from the DAL.
4. Get the desired service from the MasterFrame.
5. Call the appropriate methods on the service to accomplish the desired task.

A number of services are specific to the thread device node so those services are obtained from the device node itself. More general services are obtained from MasterFrame.

2.1 Get Execution State Example

This example displays the current execution state of all threads. This demonstrates a number of elements for accessing the DAL directly.

Features Demonstrated

- Tell Python where to find assemblies
- Tell Python which assemblies to load
- Get the MasterFrame
- Get a service from the MasterFrame
- Get a list of threads on the target
- Loop over the list of threads
- Get a service from each thread
- Get the state of the thread from the thread-specific service
- Display the name and state of each thread
- Convert an integer to an enumeration label using .Net libraries

```
# Tell Python where to find the assemblies of the DAL
import os
```

```
import sys
#change the path to your installation of the DAL
sys.path.append(os.path.abspath(r"C:\Intel\DAL"))
from itpii import Icast

# Tell Python which assemblies to load
import clr
clr.AddReference("Intel.DAL.MasterFrame")
clr.AddReference("Intel.DAL.Services.TargetTopology")
clr.AddReference("Intel.DAL.Services.ExecutionControl")

# Import the appropriate namespaces and/or types from the DAL
import System
from Intel.DAL.MasterFrame             import MasterFrame
from Intel.DAL.Services.TargetTopology import IServiceTargetTopology
from Intel.DAL.Services.ExecutionControl import (
                               IServiceExecutionControl, EExecutionState)

# Get the desired service from the MasterFrame.
mf = MasterFrame()
# Note: Use square brackets to indicate a call on a generic method.
topologyService = Icast(mf.GetService[IServiceTargetTopology](), IserviceTargetTopology)

# Call the appropriate methods on the service.
threads = topologyService.LogicalRoot.FindRelatedLNodeThreads()
for thread in threads:
    serviceInstance = thread.GetService[IServiceExecutionControl]()
    execService = Icast(serviceInstance, IserviceExecutionControl)
    state = System.Enum.GetName(EExecutionState, execService.ExecutionState)
    print("Thread %s state is %s"%(thread.Name, state))
```

The output looks something like this (it will vary depending on the target being accessed):

```
Thread SNB_C0_T0 state is Running
Thread SNB_C0_T1 state is Running
Thread SNB_C1_T0 state is Running
Thread SNB_C1_T1 state is Running
Thread SNB_C2_T0 state is Running
Thread SNB_C2_T1 state is Running
Thread SNB_C3_T0 state is Running
Thread SNB_C3_T1 state is Running
```

Congratulations! You now know the basics to getting a service from the MasterFrame and from a device node.

2.2 Using a Builder Example

A job is a set of operations that, together, accomplish some task. Executing jobs is at the heart of the DAL and most functionality in the DAL is accomplished through jobs. Jobs are constructed using builders. Each builder can add one or more operations to the job that are specific to that builder. After the job is constructed, the job is executed and the results read out.

This example shows how to get the ID code for a device using the Tap Command Register Builder.

Note: This example is designed to be run from python directly, and not using the python CLI.

Features Demonstrated:

- Obtain the job service from MasterFrame
- Use the job service to create a job.
- Use the job service to obtain a builder and attach the job to the builder.
- Use the builder to add operations to the job.
- Execute the job and wait for the job to complete.
- Get the results of the job.

```
import os
import sys
# Change the path to your installation of the DAL
sys.path.append(os.path.abspath(r"C:\Intel\DAL"))
from itpii import Icast

import clr
clr.AddReference("Intel.DAL.MasterFrame")
clr.AddReference("Intel.DAL.Services.TargetTopology")
clr.AddReference("Intel.DAL.Services.Job")
clr.AddReference("Intel.DAL.Builders.TapCmdReg")
clr.AddReference("Intel.DAL.Common.IState")

from Intel.DAL.Services.TargetTopology import IServiceTargetTopology, EEnabledType
from Intel.DAL.Services.Job            import IJobService, IJob
from Intel.DAL.Builders                import IBuilderTapCmdReg
from Intel.DAL.MasterFrame             import MasterFrame
from Intel.DAL.Common.IState           import IStateNav

mf = MasterFrame()

# Get a device for whose idcode we want to look at
topoService = Icast(mf.GetService[IServiceTargetTopology](), IServiceTargetTopology)
threads = topoService.LogicalRoot.FindRelatedLNodeThreads(EEnabledType.ENABLED_NODES)
thread0 = threads[0]

# Create a job, get a builder, add a builder operation, the execute the job.
jobService = Icast(mf.GetService[IJobService](), IJobService)
job = Icast(jobService.CreateJob("IDCodeJob"), IJob)
builder = Icast(jobService.GetBuilder(IBuilderTapCmdReg, job.Main), IBuilderTapCmdReg)
builder.RegRead("idcode")
jobReceipt = job.Execute(thread0).WaitCompletion()

# Get the requested data
data = Icast(jobReceipt.GetData(), IStateNav)
```

```
idcode = data.Read("idcode")
jobReceipt.Dispose()
print("Thread %s idcode is %s"%(thread0.Name, idcode))
```

The output of this example looks like the following (it will vary depending on the target being accessed):

```
Thread SNB_C0_T0 idcode is [32b] 0x7A768013
```

Congratulations! You now know the basics for using a builder to create a job and then executing that job to get some result.

2.3 Reusing a Job Example

There are times when you want to perform a set of operations on multiple device nodes. The Job Engines supports executing a job on more than one device without having to reconstruct the job. In other words, build the job once, and use it on multiple devices.

This example halts a thread, reads the rax register, and then resumes the thread (technically, the core on which the thread exists is halted (put into probe mode) and resumed). All of the threads are then iterated over, using the same job on each thread. This brings together ideas from the examples in section 2.1 and section 2.2.

Features Demonstrated:

- Obtain the job service from MasterFrame.
- Use the job service to create a job.
- Use the job service to obtain a builder and attach the job to the builder.
- Use the builder to add operations to the job.
- Execute the job and wait for the job to complete for all available threads
- Display the results of the job after each execution.

```
import os
import sys
# Change the path to your installation of the DAL
sys.path.append(os.path.abspath(r"C:\Intel\DAL"))
from itpii import Icast

import clr
clr.AddReference("Intel.DAL.MasterFrame")
clr.AddReference("Intel.DAL.Services.TargetTopology")
clr.AddReference("Intel.DAL.Services.Job")
clr.AddReference("Intel.DAL.Builders.ArchReg")
clr.AddReference("Intel.DAL.Builders.ExecutionControl")

from Intel.DAL.Services.TargetTopology import IServiceTargetTopology, Node, EEnabledType
```

```
from Intel.DAL.Services.Job            import IJobService
from Intel.DAL.Builders                import IBuilderArchReg, IBuilderExecutionControl
from Intel.DAL.MasterFrame             import MasterFrame
from System.Collection.Generics        import List

mf = MasterFrame()

# Get a list of enabled threads
topoService = Icast(mf.GetService[IServiceTargetTopology](), IServiceTargetTopology)
threadNodes = topoService.LogicalRoot.FindRelatedLNodeThreads(EEnabledType.ENABLED_NODES)

# Create a job and get the two builders needed
jobService = mf.GetService[IJobService]()
job = jobService.CreateJob("ReadRegJob")
execBuilder = jobService.GetBuilder(IBuilderExecutionControl, job.Main)
regBuilder = jobService.GetBuilder(IBuilderArchReg, job.Main)

# Add the builder operations: Halt (with pause), read register, Go
execBuilder.Halt()
execBuilder.Wait(500, True)
regBuilder.Read("rax")
execBuilder.Go()

# Loop over all enabled threads to execute the job and display the result
for thread in threadNodes:
    jobReceipt = job.Execute(thread).WaitCompletion()
    # Get the requested data and display
    data = jobReceipt.GetData()
    jobReceipt.Dispose()
    print("[%s] rax = %s"%(thread.Name, data.Read("rax")))
```

The output of this example looks like the following (it will vary depending on the target being accessed):

```
[SNB_C0_T0] rax = [64b] 0x0000000000000014
[SNB_C0_T1] rax = [64b] 0x0000000080000011
[SNB_C1_T0] rax = [64b] 0x0000000080000011
[SNB_C1_T1] rax = [64b] 0x0000000080000011
[SNB_C2_T0] rax = [64b] 0x0000000080000011
[SNB_C2_T1] rax = [64b] 0x0000000080000011
[SNB_C3_T0] rax = [64b] 0x0000000080000011
[SNB_C3_T1] rax = [64b] 0x0000000080000011
```

Congratulations! You now know the how to create a job and reuse it for different device nodes.

3 The Python CLI Start Sequence

The Python CLI performs a sequence of steps to get to the command prompt, depending on how it is started. This section details those steps. The python CLI can be invoked with command line options.

3.1 Invoking from the Python Console Shortcut

By selecting the Python Console Shortcut, the following sequence takes place.

- The pythonconsole.cmd script is loaded and executed by the Windows Command Processor.
- The pythonconsole.cmd script determines what python to run based on any command switches provided.
- The current folder is set based on the shortcut's "Start In" setting.
- The appropriate python is invoked and is given the itpii\startup.py script to execute.
- Python loads and initializes the itpii package.
    1. The itpii package initialization determines the color of python to run.
    2. Loads commands_other module into the itpii namespace.
    3. Loads the data types module into the itpii namespace.
- Python loads the startup module and executes it. The startup module then
    1. Parses the command line options.
    2. Displays version information for the DAL, python, .Net, pythonNet (if using) and pyreadline (if available)
    3. Sets the startup module's location as the first location to search for all python CLI files (this makes this Python CLI ignore all other copies of the DAL).
    4. Enables all enhancements
    5. Calls itpii.baseaccess() to create the one and only instance of the ItpII class and to complete the connection to the DAL and its services.
    6. Looks for dalstartup.py in the DAL install folder and imports it if found.
    7. Looks for daluserstartup.py in the user's home folder and imports it if found.
    8. Processes all files specified with the --mac command switch in the order they were given on the command line (see Appendix F for more information on command line switches).

- Done.

3.2 Invoking directly from the Command Prompt

This is for those who like to live on the command line.

- Open a Windows Command Prompt, if not already open.
- Navigate to the DAL install folder (typically, "C:\Intel\DAL")
- Type the following and press Enter. You can specify any command line switches, separated from the filename by at least one space. pythonconsole.cmd
- The startup sequence is now the same as described in section 3.1.

4 The DAL Python CLI API

4.1 itpii.package

Importing the itpii package loads the various commands and data types used in the python CLI. However, a connection to the Intel DAL has not yet been established. That is done by calling the itpii.baseaccess() command.

Note

All these commands and types are also exposed on the object returned from the itpii.baseaccess() function. For example, 'itp.Address', 'itp.log()', 'itp.libcall()', etc.

Tip

Because the connection to the Intel DAL is delayed until baseaccess() is called, it's possible to use the various data types and enhancements in the itpii package without requiring the Intel DAL to running. Only if you need to access functionality in the DAL itself (through the python CLI commands or directly through the DAL services), do you need to call itpii.baseaccess() to initialize the connection to the DAL.

Tip

If the python CLI is ever run in an embedded python session other than the DAL's Embedded Python Service, set the __fromembeddedpython__ variable in the __main__ namespace. This tells the python CLI not to import the signal and readline modules, both of which are not much use in an embedded python session.

For example, from C using CPython, the code could look like this:

```
PyObject* pyModule = PyImportModule("__main__");
PyObject_SetAttrString(pyModule, "__fromembeddedpython__", Py_None);
Py_XDECREF(pyModule);
```

4.1.1 Attributes

Table 4.1. itpii.package attributes

Attribute	Type	Description
Address	Intel.DAL.Services.CommandsLibrary.Common.Address	A class that represents an address to be communicated to the DAL. Exposed as itpii.datatypes Address.
Bitdata	itpii.datatypes.BitData	A class type representing an arbitrarily long stream of bits that can be manipulated in

Attribute	Type	Description
		various ways. All commands that return or are passed numbers use an instance of the BitData class.
Handle	itpii.datatypes.Handle	A class type that represents a handle type in legacy ITP. Used primarily with the message scan commands.
Icast	itpii.datatypes.Icast	Helper class that facilitates using .Net interfaces from python.
ItpII	itpii.ItpII	A class type representing the main Python CLI command interface
Node	itpii.node.Node	A class type used to represent various types of device nodes.
NodeContainer	itpii.node.NodeContainer	A class type used for containing a list of device nodes.
NodeState	itpii.NodeState.NodeState	A class type used for representing state access on a device node.
NodeHook	itpii.NodeHook.NodeHook	A class type used for representing DFX hooks access on a device node.

4.1.2 Functions

Table 4.2. itpii.package functions

itpii.commands	Description
append	Append a debug object to a file. Implementation of the ITP command 'append'.
baseaccess	Return an instance of the ItpII class to gain access to the Python CLI functionality.
bits	View or change a range of bits in a value. Can be used for in-place changes of BitData and ITP Data Types.
exit	Exit the Python CLI. Implementation of the ITP command 'exit'.
libcall	Create a wrapper for a function in a DLL. Implementation of the ITP command 'libcall'.

itpii.commands	Description
log	Turn on logging for all input and output to the command line. Implementation of the ITP command 'log'.
loggerecho	Enable or disable echoing the DAL log to the command line.
loggerlevel	Display or change the level of DAL logging.
nolog	Turns off logging and closes the log file. Implementation of the ITP command 'nolog'.
pause	Pause until the user presses Enter. Implementation of the ITP command 'pause'.
print_last_exception	Display details of the last exception raised (if any).
printf	Print a formatted string with arguments. Implementation of the ITP command 'printf'.
proc	Display the source of a user-defined function. Implementation of the ITP command 'proc'.
put	Put a debug object to a file. Implementation of the ITP command 'put'.
remove	Remove a debug object from memory. Implementation of the ITP command 'remove'.
sleep	Sleep for a specified number of seconds. Implementation of the ITP command 'sleep'.
subscribeToBreakEvents	Subscribe to break events.
subscribeToErrorEvents	Subscribe to error and warning events.
subscribeToHardwareEvents	Subscribe to hardware events (such as power loss, reset)
subscribeToReconfigureEvents	Subscribe to reconfiguration events.
subscribeToRunControlEvents	Subscribe to run control events (go, halt, breaks)
unsubscribeToBreakEvents	Unsubscribe from break events.
unsubscribeToErrorEvents	Unsubscribe from error and warning events.
unsubscribeToHardwareEvents	Unsubscribe from hardware events (such as power loss, reset)
unsubscribeToReconfigureEvents	Unsubscribe from reconfiguration events.
unsubscribeToRunControlEvents	Unsubscribe from run control events (go, halt, breaks)

4.1.2.1 append

Description: Appends source code for an object into a file.

Arguments: filename -- Name of the file to append to. Must be enclosed in quotes.

objectname -- Name of the object to get source for. Must be enclosed in quotes. See Remarks.

Returns: None.

Remarks: Attempts to obtain the source code for the specified object name (which can be a module, function, or type but not variable) and, if successful, ppends the source code to the specified file.

Raises a ValueError if the source code for the specified object cannot be obtained.

Raises an IOError if unable to write to file.

Usage:

```
>>? def myfunc():
...     print("Hello!")
...
>>? itpii.append("myfunc.py", "myfunc")
```

4.1.2.2 baseaccess

Description This command constructs and returns an ItpII class object that in turn initializes access to the Intel DAL software stack and modifies the global environment to add functions specific to Intel DAL. The Intel DAL automatically creates a device topology based on what is discoverable on the connected target system.

Syntax: itpobject = itpii.baseaccess()

Arguments: None.

Returns: An ItpII object. The same instance is returned for each call to baseaccess().

Remarks: AThe ItpII object returned from baseaccess() is a singleton, which means subsequent calls to baseaccess() returns the first ItpII object. This is true across the entire Python session, regardless of which module calls baseaccess(). What this means is a user can create a Python module, add the statements shown in the Example, then import that Python module into the Python CLI. The user's module then has access to the same instance of the ItpII object that is available in the Python CLI.

Tip

: You can assign a new attribute to the object returned from the baseaccess() command and that attribute will be visible in all mod-

ules loaded into the current python session. For example, itp.value = 42 will expose the 'value' attribute to all python modules in the current python session that call baseaccess() to get the ItpII object. Those modules in turn can change 'value' and those changes are seen in all python modules.

Usage:

```
import itpii
itp = itpii.baseaccess()
```

4.1.2.3 bits

Description View or change a range of bits in a value. Can be used for in-place changes of BitData and ITP Data Types.

Syntax: itpii.bits(obj, offset, size, assign=None)

Arguments: obj -- A BitData object or an instance of one of the ITP Data Types (see section 4.8) or a simple integer.

offset -- A valid expression yielding the bit index into the specified value where the bit field begins. This value must be less than the size of the integer, BitData object, or data type object.

size -- A valid expression yielding the size, in bits,
of the bit field. The size plus the offset must be less than the size of the integer, BitData object, or data type object.

assign -- If specified, a numerical value or BitData object to be assigned to the bit field. This expression, if it results in a value greater than possible in the bit field, will be truncated to the bit-size during assignment.

Returns: If assign is None, returns the requested bits in the same form as obj (for example, if obj is a BitData, then a BitData is returned; if obj is an integer, an integer is returned).

if assign is not None, return the updated value as the same data type as sent in. For BitData and ITP Data Types, the change is also made in the passed-in object, in which case the return value can be ignored.

Exceptions: ValueError: The offset or size values are out of bounds.

Remarks: This version of the bits() command works with BitData, ITP Data Types, and raw integers and supports in-place changing of BitData and ITP Data Types. To change the bits in an architectural register, use the thread-specific version of the bits() command (section 0).

Usage:

```
def example():
```

```
   value = BitData(32, 0x1234)
   print("Bits 4 through 8 of value = %s"%itpii.bits(value, 4, 4))
   print("Setting bits 4 through 8 of value
        = %s"%itpii.bits(value, 4, 4, 0xf))
   print("value (as BitData) = %s"%value)
   print("")
   value = Ord4(0x1234)
   print("Bits 4 through 8 of value = %s"%itpii.bits(value, 4, 4))
   print("Setting bits 4 through 8 of value
        = 0x%x"%itpii.bits(value, 4, 4, 0xf))
   print("value (as Ord4) = 0x%x"%value)
   print("")
   print("bits 4 through 8 of 0x1234 = 0x%x"%itpii.bits(0x1234, 4, 4))
   print("Setting bits 4 through 8 of 0x1234
        = 0x%x"%itpii.bits(0x1234, 4, 4, 0xf))

example()
```

The output from this example looks like this

```
                           Bits 4 through 8 of value = [4b] 0x3
Setting bits 4 through 8 of value = [32b] 0x000012F4
value (as BitData) = [32b] 0x000012F4

Bits 4 through 8 of value = 3
Setting bits 4 through 8 of value = 0x12f4
value (as Ord4) = 0x12f4

bits 4 through 8 of 0x1234 = 0x3
Setting bits 4 through 8 of 0x1234 = 0x12f4
```

4.2 itpii.cmdsControl module

This is an internal module but certain functions are exposed on it to provide some advanced control over the python CLI. Only documented fucntions should be used.

4.2.1 Functions

Table 4.3. itpii.cmdsControl functions

itpii.cmdsControl Commands	Description
asyncTimestampsDisable	Disable timestamps from appearing on asynchronous messages.
asyncTimestampsEnable	Enable timestamps on all asynchronous messages.
backgroundThreadForCommandsDisable	Disable use of background threads for running python CLI commands.
backgroundThreadForCommandsEnable	Enable use of background threads for running python CLI commands.

itpii.cmdsControl Commands	Description
getServiceRegistry	Retrieve an object through which DAL services can be retrieved.
hardwareEventsFrequencyFilterDisable	Disable filtering of hardware events based on frequency over an interval of time.
hardwareEventsFrequencyFilterEnable	Enable filtering of hardware events based on frequency over an interval of time.

4.2.1.1 asyncTimeStampDisable

Description: Disable timestamps on all asynchronous messages (the default state).

Syntax: itpii.cmdsControl.asyncTimestampsDisable()

Parameters: None.

Returns: None.

Remarks: There are times when there are many asynchronous messages from the DAL over a long period of time (reset events, break events, power events, and more). It is useful if these events had a time stamp on them to keep them all straight. The asyncTimestampsDisable() function turns off these timestamps for a less cluttered look.

Timestamps on asynchronous messages can be enabled with asyncTimestampsEnable().

This method cannot be nested: once called, all timestamps are disabled until asyncTimestampsEnable() is called.

Usage:

```
itpii.cmdsControl.asyncTimestampsDisable()
```

4.2.1.2 asyncTimeStampEnable

Description: Enable timestamps on all asynchronous messages.

Syntax: itpii.cmdsControl.asyncTimestampsEnable()

Parameters: None.

Returns: None.

Remarks: There are times when there are many asynchronous messages from the DAL over a long period of time (reset events, break events, power events, and more). It is useful if these events had a time stamp on them to keep them all straight. The asyncTimestampsEnable() function turns on these timestamps.

Timestamps on asynchronous messages can be disabled with asyncTimestampsDisable().

This method cannot be nested: once called, all timestamps are enabled until asyncTimestampsDisable() is called.

The timestamp attempts to conform to the current locale.

Usage:

```
itpii.cmdsControl.asyncTimestampsEnable()
```

Example Output: This is a series of halt messages with the timestamps enabled. The format of the timestamp may vary based on locale.

```
10/27/11 15:04:38   [SNB_C0_T0] Halt Command break ...
10/27/11 15:04:38   [SNB_C0_T1] Halt Command break ...
10/27/11 15:04:38   [SNB_C1_T0] Halt Command break ...
10/27/11 15:04:38   [SNB_C1_T1] Halt Command break ...
10/27/11 15:04:38   [SNB_C2_T0] Halt Command break ...
10/27/11 15:04:38   [SNB_C2_T1] Halt Command break ...
10/27/11 15:04:39   [SNB_C3_T0] Halt Command break ...
10/27/11 15:04:39   [SNB_C3_T1] Halt Command break ...
```

4.2.1.3 backgroundThreadForCommandsDisable

Description: Disable use of background threads for running python CLI commands.

Syntax: itpii.cmdsControl.backgroundThreadForCommandsDisable()

Parameters: None.

Returns: None.

Remarks: All python CLI commands are run on a background thread by default. This generally allows these commands to be interrupted with Ctrl-C but at a cost of performance. The disableBackgroundThread() function disables the use of the background thread so performance is better and complex code that uses python threads will behave better. However, python CLI commands can then no longer be interrupted with Ctrl-C.

The use of a background thread to run python CLI commands can be enabled with backgroundThreadForCommandsEnable().

This method cannot be nested: once called, the use of a background thread is disabled until backgroundThreadForCommandsEnable() is called.

Usage:

```
itpii.cmdsControl.backgroundThreadForCommandsDisable()
```

4.2.1.4 backgroundThreadForCommandsEnable

Description: Enable use of background threads for running python CLI commands.

Syntax: itpii.cmdsControl.backgroundThreadForCommandsEnable()

Parameters: None.

Returns: None.

Remarks: All python CLI commands are run on a background thread by default. This generally allows these commands to be interrupted with Ctrl-C but at a cost of performance. The enableBackgroundThread() function re-enables the use of the background thread so python CLI commands can once again be interrupted by Ctrl-C. The trade-off is some loss in performance and possible behavior problems with complex use of python threads.

The use of a background thread to run python CLI commands can be disabled with backgroundThreadForCommandsDisable().

This method cannot be nested: once called, the use of a background thread is enabled until backgroundThreadForCommandsDisable() is called.

Usage:

```
itpii.cmdsControl.backgroundThreadForCommandsEnable()
```

4.2.1.5 getServiceRegistry()

Description: Retrieve a python object from the DAL that can be used to retrieve DAL services.

Syntax: itpii.cmdsControl.getServiceRegistry()

Parameters: None.

Returns: None.

Remarks: This command is used to make it easier for a python script that is running with the python CLI to get access to DAL services. A typical python script will use the python CLI commands to do all the work. However, there are rare times when it is necessary to go directly to the DAL to perform some function that does not exist in the python CLI. The normal rule of thumb for such situations is to instantiate the MasterFrame class to get access to DAL services. However, the MasterFrame class is supposed to be instantiated only once per process and the python CLI already has done that. The getServiceRegistry() function provides access to the that MasterFrame object – a least for the purpose of getting services.

When the python CLI is running in an embedded python session (using the Embedded Python service), the getServiceRegistry() function returns a different object. However, the methods for getting a service are still the same. this is why calling functions on the object returned from the get-

ServiceRegistry() function should be confined to only the GetService() methods.

The possible GetService methods:

Generic Versions

- TService GetService[TService]()
- TService GetService[TService, TDiscriminator]()
- TService Getservice[TService, TDiscriminator](discriminatorInstance)

Non-generic Versions:

- object GetService(serviceType)
- object GetService(serviceType, discriminatorType)
- object GetService(serviceType, discriminatorType, discriminatorInstance)

Typically the GetService[TService]() and GetService(serviceType) are the only two methods of interest to outside developers.

Warning
If you don't know what all this means, then the getServiceRegistry() function is not for you.

Usage:

```
serviceRegistry = itpii.cmdsControl.getServiceRegistry()
```

4.2.1.6 hardwareEventsFrequencyFilterDisable

Description: Disable filtering of hardware events based on frequency over an interval of time. After this function is called, all hardware events are reported.

Syntax: itpii.hardwareEventsFrequencyFilterDisable()

Parameters: None.

Returns: None.

Remarks: There are times when too many hardware events come in from the target, swamping the command line with notifications. To deal with this flood of notifications, the hardware event handler can filter out repeated events that occur more than a certain amount in a period of time. This event filtering is enabled by default.

The hardwareEventsFrequencyFilterDisable() method disables this filtering so all hardware events are displayed, regardless of how many there are.

Turn this filtering on with hardwareEventsFrequencyFilterEnable().

Usage:

```
itpii.hardwareEventsFrequencyFilterDisable()
```

4.2.1.7 hardwareEventsFrequencyFilterEnable

Description: Enable filtering of hardware events based on frequency over an interval of time. As long as events of the same type keep occurring in less than frequencyInterval seconds, the events are not reported after the first time.

Syntax: itpii.hardwareEventsFrequencyFilterDisable()

Parameters: frequencyInterval -- The number of seconds between events of the same type that will cause subsequent events of that type to be silently counted instead of reported. Can be a floating point number. If this is None, defaults to 1.0 seconds. See Remarks.

Returns: None.

Remarks: There are times when too many hardware events come in from the target, swamping the command line with notifications. To deal with this flood of notifications, the hardware event handler can filter out repeated events that occur more than a certain amount in a period of time. This event filtering is enabled by default.

If an event comes in and it occurs within 2 * frequencyInterval seconds of the last event of the same type, the total number of events of that type are reported first then the event is reported.

If more than 2 * frequencyInterval seconds has passed since the last event of the same type, only the single event is reported.

Turn this filtering off with hardwareEventsFrequencyFilterDisable().

Usage:

```
itpii.hardwareEventsFrequencyFilterEnable()
itpii.hardwareEventsFrequencyFilterEnable(1.5)
```

4.3 itpii.ItpII Class

Constructing the ItpII class initiates a connection with Intel DAL and constructs the Python node tree based on the discovered or configured device tree (which will take some time). The resulting instance is used to access Intel DAL functionality.

Call the itpii.baseaccess() command to get an instance of the itpii.ItpII class.

Example:

```
import itpii
itp = itpii.baseaccess()
```

4.3.1 Attributes

Table 4.4. itpii.ItpII attributes

Attribute	Type	Description
Address	Intel.DAL.Services.CommandsLibrary.Common.Address	Represents an address type as used in the ITP Commands Library.
AddressType	Intel.DAL.Services.CommandsLibrary.Common.Address.AddressType	Enumeration of the types of addresses that can be constructed with the Address class. This enumeration is passed as a parameter to the Address class constructor (see section 5.1).
chipsets	NodeContainer	Collection of all chipsets (nowadays called Peripheral Control Hubs or PCH) across all domains. This list is read-only.
cv	ControlVariableContainer	A list of control variables. A control variable is used to access the state of or alter the behavior of debug ports, devices, or the Intel DAL stack itself.
debugports	NodeContainer	Collection of all debug ports across all domains. This list is read-only.
devicelist	DeviceContainer	A list of Device objects that have various properties of interest to legacy ITP users. This property is the replacement for the 'devicelist' command in ITP. This list is read-only.
domains	NodeContainer	A list of domain nodes based on the current configuration. A "domain" is a user-specified entity that can encompass part of a target system, the entire target system, or multiple target systems. The user controls the "domains"

Attribute	Type	Description
		that appear by using the Target Topology Service and a configuration utility to create a configuration that is then used when instantiating the ItpII class. This list is read-only.
handle	itpii.datatypes.Handle	Represents a handle ITP type. Used in message scan handles.
nodetypes	tuple (of strings)	A read-only list of strings defining the known device node types. The device node types can be passed to the getByType() and getAllByType() functions.
threads	NodeContainer	Collection of all threads on all cores across all domains, in the order they were discovered. This is used to help ease the conversion from the legacy ITP viewpoint system to the device-oriented system of the DAL. The ITP viewpoint is translated to an index into this threads collection. This list is read-only.
uncores	NodeContainer	Collection of all uncores across all domains. This list is read-only.
data types	(various)	See Data Types Section. These data types are also exported into the __main__ namespace for ease of access in the python CLI.

4.3.2 Functions

4.3.2.1 getDomain()

Description: Retrieve the name of the domain that is considered current (that is, the domain whose devices are in itp.devicelist and itp.threads).

Syntax: itpii.getDomain()

Parameters: None.

Returns: A string containing the name of the current domain.

Remarks: See ItpII.setDomain() for details on the concept of current domain.

Usage:

```
import itpii
itp = itpii.baseaccess()
itp.setDomain("domain1")
```

4.3.2.2 setDomain()

Description: Change the current domain whose devices are exposed through the ItpII.devicelist, ItpII.threads, and ItpII.domains containers. The contents of the device containers are replaced with devices from the new domain.

Syntax: itpii.setDomain()

Parameters: domain: Name of the domain to set or an explicit NodeDomain object. Nothing happens if the specified domain is already established.

Returns: None.

Remarks: The concept of a "current" target domain allows for backwards compatibility with legacy ITP scripts while allowing for future expansion when multiple target domains are used.

One limitation of this approach: since the current domain affects the containers on the ItpII class, only the devices from a single domain can be accessed at a time in a python session. You can cache the device nodes from the containers and they will still be useable. Or you can clone the container.

For example:

```
domain0_threads = itp.threads.clone()
domain0_devices = itp.devicelist.clone()
domain0_domains = itp.domains.clone()
itp.setDomain("domain1")
```

Usage:

```
import itpii
itp = itpii.baseaccess()
itp.setDomain("domain1")
```

4.3.2.3 reconnectmasterframe()

Description: Reinitialize the connection to the DAL.

Syntax: itpii.reconnectmasterframe()

Parameters: None.

Returns: None.

Remarks: This should be used only if the DAL has been stopped. Use the stopmasterframe() command (section 4.3.2.4) to stop the DAL from inside the python CLI.

Usage:

```
import itpii
itp = itpii.baseaccess()
itp.reconnectmasterframe()
```

4.3.2.4 stopmasterframe()

Description: Stop the DAL.

Syntax: itpii.stopmasterframe()

Parameters: None.

Returns: None.

Remarks: Once the DAL is stopped, all device and target access will fail. Use the reconnectmasterframe() command to restart the DAL and re-establish a connection.

Usage:

```
import itpii
itp = itpii.baseaccess()
itp.stopmasterframe()
```

4.3.3 ITP Commands

All of the ITP commands are available from the ItpII object returned from itpii.baseaccess(). However, until Intel DAL is configured (with a call to the baseaccess() function), none of the commands will work.

For commands that affect devices (and that is the majority of the commands), the "itp.command()" form with no parameters affects all devices. Some commands that can affect an individual device can accept that device as a parameter, thus focusing attention only on that device. For example, "itp.idcode(p0)".

See global commands section for commands that appear on the ItpII class.

Finally, for commands that affect threads, those commands are also added to the NodeThread class.

For example:

```
# Python
itp = itpii.baseaccess()
```

```
itp.halt()  ## Halts all devices

core0 = itp.domains[0].getByType("core")
core0.halt()        ## Halts all threads on the first core
```

4.4 itpii.BitData Class

4.4.1 Attributes

Table 4.5. itpii.BitData attributes

Attribute	Type	Description
BitCapacity	int	How many bits this instance of BitData can hold without expanding. For BitData, this value has a granularity of 32 bit chunks, which reflects how the bits are stored internally.
BitGranularity	int Always 1.	For internal use only.
BitSize	int	The number of bits actually in this BitData.
Data	int[]	The underlying raw bits in 32-bit chunks. This is normally not used.
IsReadOnly	bool Always returns False.	For internal use only.
IsResizable	bool Always returns True.	For internal use only.
IsSynchronized	bool	Always returns False. For internal use only.
LongBitSize	long	The number of bits actually in this BitData (for those times when more than 2^31 bits are in the BitData).
MaxBitSize	long	The maximum number of bits a BitData can hold.
Random	Random	Get or set the Random object (a .Net class) that is used to randomize the contents of a BitData with the Randomize() method. This is normally not changed.
SyncRoot	Object	Used for synchronizing among .Net threads. For internal use only.

4.4.2 Operators

The following operators are supported by the python CLI version of the Address class.

Table 4.6. itpii.BitData operators

Operator	Description
==	Compare a BitData to another BitData or an integer for equality. `>>> BitData(32, 10) == 10` `True`
!=	Compare a BitData to another BitData or an integer for inequality. `>>> BitData(32, 10) != 20` `True`
>	Determine if a BitData is greater than another BitData or integer. `>>> BitData(32, 10) > 9` `True`
<	Determine if a BitData is less than another BitData or integer. `>>> BitData(32, 10) < 11` `True`
>=	Determine if a BitData is greater than or equal to another BitData or integer. `>>> BitData(32, 10) >= 9` `True`
+=	Increment BitData by the specified amount. `>>> bd = BitData(32, 10)` `>>> bd += 2` `>>> bd` `[32b] 0x0000000C`
-=	Decrement BitData by the specified amount. `>>> bd = BitData(32, 10)` `>>> bd -= 2` `>>> bd` `[32b] 0x00000008` *=
<	Multiply BitData by the specified amount.

Operator	Description
	``>>> bd = BitData(32, 10) >>> bd *= 2 >>> bd [32b] 0x00000014```
/=	Determine if a BitData is less than another BitData or integer. ```>>> bd = BitData(32, 10) >>> bd /= 2 >>> bd [32b] 0x00000005```
%=	Divide the BitData by the specified amount and store the remainder in the BitData. ```>>> bd = BitData(32, 10) >>> bd %= 3 >>> bd [32b] 0x00000001```
&=	Perform a bitwise AND of the BitData by the specified amount and store the result in the BitData. ```>>> bd = BitData(32, 10) >>> bd &= 2 >>> bd [32b] 0x00000002```
|=	Perform a bitwise OR of the BitData by the specified amount and store the result in the BitData. ```>>> bd = BitData(32, 10) >>> bd |= 3 >>> bd [32b] 0x0000000B```
^=	Perform a bitwise XOR (exclusive-or) of the BitData by the specified amount and store the result in the BitData. ```>>> bd = BitData(32, 10) >>> bd ^= 2 >>> bd [32b] 0x00000008```
<<=	Shift the BitData to the left by the specified number of bits. ```>>> bd = BitData(32, 10)```

Operator	Description
	`>>> bd <<= 2` `>>> bd` `[32b] 0x00000028`
>>=	Shift the BitData to the right by the specified number of bits. `>>> bd = BitData(32, 10)` `>>> bd >>= 2` `>>> bd` `[32b] 0x00000002`
+	Add the specified amount to the BitData and return a new BitData `>>> BitData(32, 10) + 2` `[32b] 0x0000000C`
-	Subtract the specified amount to the BitData and return a new BitData `>>> BitData(32, 10) - 2` `[32b] 0x00000008`
*	Multiply the specified amount to the BitData and return a new BitData `>>> BitData(32, 10) * 2` `[32b] 0x00000014`
/	Divide the specified amount to the BitData and return a new BitData `>>> BitData(32, 10) / 2` `[32b] 0x00000005`
%	Divide the BitData by the specified amount and return the remainder as a new BitData. `>>> BitData(32, 10) & 2` `[32b] 0x00000002`
|	Perform a bitwise AND of the BitData with the specified mask and return the result as a new BitData. `>>> BitData(32, 10) | 3` `[32b] 0x0000000B`
^	Perform a bitwise XOR (exclusive-or) of the BitData with the specified mask and return the result as a new BitData.

Operator	Description
	`>>> BitData(32, 10) ^ 2` `[32b] 0x00000008`
<<	Shift the BitData left by the specified number of bits and return a new BitData. `>>> BitData(32, 10) << 2` `[32b] 0x00000028`
>>	Shift the BitData right by the specified number of bits and return a new BitData. `>>> BitData(32, 10) >> 2` `[32b] 0x00000002`

4.4.3 Constructor

Description: Construct and instance of the BitData class.

Syntax: BitData(*args)

Parameters: args One or more arguments, depending on how the BitData class is to be constructed. See Remarks.

Returns: Return a new instance of the BitData class.

Remarks: Since the python CLI BitData class derives from the ITP Commands Library BitData class, multiple ways of instantiating the class are possible. The following table describes the combination of argument types that are allowed. All other argument types will raise a TypeError: no constructor matches given arguments.

Table 4.7. itpii.BitData constructors

Constructor	Description
BitData(BitData)	Construct a BitData from another BitData
BitData(str)	Construct a BitData from a string. The string is expected to be a hexadecimal (prefixed by '0x') or binary string (prefixed by '0y') and can be an arbitrary length. For hexadecimal, each digit after the prefix contributes 4 bits to the final bit size. For binary, each digit after the prefix contributes 1 bit to the final bit size.

Constructor	Description
	Note A decimal string is treated as the number of bits to size the Bit-Data to and not the actual bits themselves. This is a side effect of how python implicitly converts strings consisting solely of decimal digits to an integer.
BitData(int)	Construct a BitData with the specified number of bits, which are all set to 0.
BitData(int, int)	Construct a BitData with the specified number of bits, using the second argument as the bit data itself (up to 32 bits). The supplied bits are padded or truncated to the specified number of bits.
BitData(int, long)	Construct a BitData with the specified number of bits, using the second argument as the bit data itself (up to 64 bits). The supplied bits are padded or truncated to the specified number of bits. **Note** Although a python long type contains an arbitrary number of bits, pythonNet and IronPython will use this constructor only if the number of bits is between 32 and 64 bits.
BitData(int, string)	Construct a BitData with the specified number of bits, using the second argument as the bit data itself. The supplied bits are padded or truncated to the specified number of bits. The string can be hexadecimal or binary and can be of any length. **Note** pythonNet will call this constructor if a long type is greater than 64 bits in size (due to how pythonNet marshals a long type).
BitData(int, int[])	Construct a BitData with the specified number of bits, using the second argu-

Constructor	Description
	ment as the bit data itself. The supplied bits are padded or truncated to the specified number of bits. The second argument is a list of integers, with the bits packed into each integer.
BitData(int, long[])	Construct a BitData with the specified number of bits, using the second argument as the bit data itself. The supplied bits are padded or truncated to the specified number of bits. The second argument is a list of 64-bit longs, with the bits packed into each long. **Note** Although the python long type can be longer than 64 bits, this constructor will only get called if the long values are confined to 64 bits.

4.4.4 Methods

Note

The following methods are .Net-specific and have little bearing on the python implementation. They are documented in the .Net Object base type if you are curious. They are listed here for completeness since they are visible from the python CLI.

- Equals()
- Finalize()
- GetHashCode()
- GetType()
- MemberwiseClone()
- ReferenceEquals()

4.4.4.1 Add()

Description: Add two values represented by bitfields (ranges of bits) in two BitData objects and deposit the result in the specified bitfield in this BitData.

Syntax: Add(o4DstOffset, o4DstSize, cbdSrcA, o4SrcAOffset, o4SrcASize, cbdSrcB, o4SrcBOffset, o4SrcBSize)

Parameters: o4DstOffset Bit offset in this BitData where the result will be stored.

o4DstSize Size of the bitfield in this BitData where the result will be stored.

cbdSrcA First BitData from which to get the first addend.

o4SrcAOffset Bit offset into first BitData from where to get the addend.

o4SrcSize Size of the bitfield in the first BitData from where to get the addend.

cbdSrcB Second BitData from which to get the second addend.

o4SrcBOffset Bit offset into the second BitData from where to get the addend.

o4SrcBSize Size of the bitfield in the second BitData from where to get the addend.

Returns: None.

Remarks: The result is padded or truncated to the destination bitfield.

Example

```
bd  = BitData(32, "0xff0f")
bdA = BitData(32, "0x0300")
bdB = BitData(32, "0x0002")
bd.Add(4, 4, bdA, 8, 4, bdB, 0, 4)
print(bd)
```

The output looks like this:

```
[32b] 0x0000FF5F
```

4.4.4.2 CompareTo()

Description: Compare this BitData to another BitData.

Syntax: CompareTo(other)

Parameters: other The BitData object to compare to.

Returns: Return an integer result:

-1 means 'A < B'

0 means 'A == B'

+1 means 'A > B'

Remarks: This method is not typically used in python as the python operators are typically easier to understand.

Example

```
bd1 = BitData("0x1f")
bd2 = BitData("0x1e")
result = bd1.CompareTo(bd2)
print("Result of comparing %s to %s = %d"%(bd1, bd2, result))
```

The output looks like this:

```
Result of comparing [8b] 0x1F to [8b] 0x1E = 1
```

4.4.4.3 Copy()

Description: Create a copy of this BitData or a portion of it as a new BitData or copy bits from one BitData to this BitData.

Syntax: Copy (extract) from this BitData

Copy()

Copy(indexList)

Copy(o4Offset, o4Size)

Copy from one place in this BitData to another

Copy(o4DstOffset, o4SrcOffset, o4Size)

Copy into this BitData from another BitData

Copy(bitOffsetDst, dataSrc, indicesSrc)

Copy(o4DstOffset, cbdSrc, o4SrcOffset, o4Size)

Copy(indicesDst, dataSrc, bitOffsetSrc)

Parameters: indexList-An IndexList object containing a collection of bit indices from which to get bits.

o4Offset-Offset into this BitData to a bitfield to copy from.

o4Size-Number of bits in the bitfield to copy from/to.

bitOffsetDst-Offset into this BitData to a bitfield to copy to.

o4DstOffset-Offset into this BitData to a bitfield to copy to.

o4SrcOffset-Offset into a source BitData (or this BitData) to a bitfield to copy from.

dataSrc-A source BitData from which to get bits.

cbdSrc-A source BitData from which to get bits.

indicesSrc-An IndexList object containing a collection of bit indices to read from a source BitData.

Returns: The first three forms return a new BitData object. The remaining forms return None.

Remarks: There are three basic forms of the Copy() method:

1. Extract bits from this BitData object and return them in a new BitData object.

2. Copy bits from one place in this BitData object to another place in this BitData object.

3. Copy bits from a source BitData object to this BitData object.

Within these three forms, the variations are based on the type of the parameters passed in.

The difference between Extract() and Copy() is the Extract() method returns the bits as an integer while the Copy() method returns the bits in a new BitData object.

The first form, (that takes no parameters) is functionally equivalent to the Clone() method.

Example

```
bd  = BitData(16, "0xff0f")
bd2 = BitData(16, "0xabcd")
bdClone = bd.Copy()
bdByte = bd.Copy(4, 8)
print("Copy of source: %s"%bdClone)
print("Copied byte   : %s"%bdByte)
bd.Copy(4, bd2, 8, 4)
print("Copied from outside: %s"%bd)
bd.Copy(4, 8, 4)
print("Copied from within : %s"%bd)
```

The output looks like this:

```
Copy of source: [16b] 0xFF0F
Copied byte   : [8b] 0xF0
Copied from outside: [16b] 0xFFBF
Copied from within : [16b] 0xFFFF
```

4.4.4.4 Delete()

Description: Remove a range of bits from this BitData. The bits to the left of the range are moved to the right.

Syntax: Delete(bitOffset, bitSize)

Parameters: bitOffset-Offset to first bit to remove.

bitSize-Number of bits to remove

Returns: None.

Remarks: None.

Example

```
bd = BitData("0xf1f")
bd.Delete(4, 4)
print(bd)
```

The output looks like this:

```
[8b] 0xFF
```

4.4.4.5 Deposit()

Description: Deposit a value into a specified bitfield of this BitData. The value is padded or truncated to fit.

Syntax: Deposit(o4StartBit, o4BitSize, o4Value)

Parameters: o4StartBit Offset to first bit in the bitfield to affect.

o4BitSize Number of bits in the bitfield to affect.

o4Value The value to deposit. Can be up to 64 bits in size.

Returns: None.

Remarks: None.

Example

```
bd = BitData("0xf0f")
bd.Deposit(4, 4, 0xe)
print(bd)
```

The output looks like this:

```
[8b] 0xFEF
```

4.4.4.6 Deposit32()

Description: Deposit a 32-bit value into a specified 32-bit bitfield of this BitData.

Syntax: Deposit32(o4StartBit, o4Value)

Parameters: o4StartBit Offset to first bit in the bitfield to affect.

o4Value The value to deposit that is treated as 32 bits in size.

Returns: None.

Remarks: None.

Example

```
bd = BitData("0xf00000000f")
bd.Deposit32(4, 0x12345678)
print(bd)
```

The output looks like this:

```
[40b] 0xF12345678F
```

4.4.4.7 Divide()

Description: Divide two values represented by bitfields (ranges of bits) in two BitData objects and deposit the result in the specified bitfield in this BitData.

Syntax: Divide(o4DstOffset, o4DstSize, cbdSrcA, o4SrcAOffset, o4SrcASize, cbdSrcB, o4SrcBOffset, o4SrcBSize)

Parameters: o4DstOffset Bit offset in this BitData where the result will be stored.

o4DstSize Size of the bitfield in this BitData where the result will be stored.

cbdSrcA First BitData from which to get the dividend.

o4SrcAOffset Bit offset into first BitData from where to get the dividend.

o4SrcSize Size of the bitfield in the first BitData from where to get the dividend.

cbdSrcB Second BitData from which to get the second divisor.

o4SrcBOffset Bit offset into the second BitData from where to get the divisor.

o4SrcBSize Size of the bitfield in the second BitData from where to get the divisor.

Returns: None.

Remarks: The result is padded or truncated to the destination bitfield.

> **Warning**
> Supports a maximum of 32 bits for dividend and divisor.

Example

```
bd  = BitData(32, "0xff0f")
bdA = BitData(32, "0x0800")
bdB = BitData(32, "0x0002")
bd.Divide(4, 4, bdA, 8, 4, bdB, 0, 4)
print(bd)
```

The output looks like this:

```
[32b] 0x0000FF4F
```

4.4.4.8 Extract()

Description: Extract up to 32 bits of data and return as an integer.

Syntax: Extract(o4StartBit, o4BitSize)

Parameters: o4StartBit Offset to first bit in the bitfield to read.

o4BitSize Number of bits in the bitfield to read.

Returns: Return an integer containing the value of the extracted bitfield.

Remarks: None.

Example

```
bd = BitData("0xf5f")
print("Extracted value = %d"%bd.Extract(4, 4))
```

The output looks like this:

```
Extracted value = 5
```

4.4.4.9 Extract32()

Description: Extract 32 bits of data and return as an integer.

Syntax: Extract32(o4StartBit)

Parameters: o4StartBit Offset to first bit in the bitfield to read.

Returns: Return an integer containing the 32-bit value of the extracted bitfield.

Remarks:

Warning

Warning: The 32-bit range of bits to extract must exist entirely within the BitData object; otherwise an error is raised: IndexOut-OfRangeException: Index was outside the bounds of the array.

Example

```
bd  = Bitbd = BitData("0xf12345678f")
print("Extracted value = 0x%x"%bd.Extract32(4))
print(bd)
```

The output looks like this:

```
Extracted value = 0x12345678
```

4.4.4.10 Extract64()

Description: Extract up to 64 bits of data and return as an integer.

Syntax: Extract64(o4StartBit, o4BitSize)

Parameters: o4StartBit Offset to first bit in the bitfield to read.

o4BitSize Number of bits in the bitfield to read.

Returns: Return a long containing the value of the extracted bitfield.

Remarks: None.

Example:

```
bd = BitData("0xffee1122334455667788aabb")
print("Extracted value = 0x%x"%bd.Extract64(0, 64))
```

The output looks like this:

```
Extracted value = 0x334455667788aabb
```

4.4.4.11 FindFirst()

Description: Find the offset of the first bit in the BitData (or part of the BitData) that is set.

Syntax: FindFirst()

FindFirst(o4StartBit, o4BitSize)

Parameters: o4StartBit Offset to first bit to start searching from.

o4BitSize Number of bits to search.

Returns: Return the bit offset of the first set bit; otherwise, returns -1 if no bits are set. The returned offset is relative to the starting bit.

Remarks: The first form of the method is called when no parameters are provided. In that case, the first bit is 0 and the number of bits to search is the number of bits in the BitData.

Example

```
bd = BitData(32, "0y0101101000")
print("First bit set at offset %d"%bd.FindFirst())
print("Another bit set at offset %d"%(bd.FindFirst(4, 32 - 4) + 4))
```

The output looks like this:

```
First bit set at offset 3
Another bit set at offset 5
```

4.4.4.12 FindLast()

Description: Find the offset of the last bit in the BitData (or part of the BitData) that is set.

Syntax: FindLast()

FindLast(o4StartBit, o4BitSize)

Parameters: o4StartBit Offset to first bit to start searching from.

o4BitSize Number of bits to search.

Returns: Return the bit offset of the last set bit; otherwise, returns -1 if no bits are set. The returned offset is relative to the starting bit.

Remarks: The first form of the method is called when no parameters are provided. In that case, the first bit is 0 and the number of bits to search is the number of bits in the BitData.

Example

```
bd = BitData(32, "0y0101101000")
print("Last bit set at offset %d"%bd.FindLast())
print("Another bit set at offset %d"%(bd.FindLast(0, 4)))
```

The output looks like this:

```
Last bit set at offset 8
```

```
Another bit set at offset 3
```

4.4.4.13 FromString()

Description: Set all or part of this BitData with bits taken from a string.

Syntax: FromString(strValue)

FromString(dstStart, dstSize, strValue)

Parameters: strValue The string containing the bits to use. See Remarks.

dstStart Offset to first bit of bitfield to set.

dstSize Number of bits of bitfield to set.

Returns: Return the number of text characters consumed from the string.

Remarks: The string can be decimal, hexadecimal (with prefix '0x') or binary (with prefix '0y'),

The first form of this method is called when only a string is provided. In that case, the string replaces the entire contents of the BitData object. The BitData object is resized to match the number of bits in the string.

The second form is called when a start and size value is provided in addition to the string. In that case, only that range of bits is replaced by bits from the string value.

Example

```
bd = BitData()
numCharsUsed = bd.FromString("0x112233")
print("Result = %s (used %d chars from string)"%(bd, numCharsUsed))
numCharsUsed = bd.FromString(4, 8, "0xbbaa")
print("Result = %s (used %d chars from string)"%(bd, numCharsUsed))
```

The output looks like this:

```
Result = [24b] 0x112233 (used 8 chars from string)
Result = [24b] 0x112AA3 (used 4 chars from string)
```

4.4.4.14 GenerateCRC()

Description: Generate a CRC value from the specified bitfield in this BitData.

Syntax: GenerateCRC(o4Offset, o4Size, crcDef)

Parameters: o4Offset Offset to first bit of bitfield to process.

o4Size Number of bits to process.

crcDef An instance of the Intel.DAL.Common.BinaryUtilies.Crc class.

Returns: Return a 32-bit value containing the generated CRC of the specified bit-field.

Remarks: None.

Example

```
import clr
clr.AddReference("Intel.DAL.Common.BinaryUtilities")
from Intel.DAL.Common.BinaryUtilities import Crc

bd = BitData("0x393837363534333231")
crcDef = Crc(4, 3, 0, True, True, 0)
crc = bd.GenerateCRC(0, bd.BitSize, crcDef)
print("Generated CRC = 0x%x"%crc)
```

The output looks like this:

```
Generated CRC = 0x7
```

4.4.4.15 GenerateECC()

Description: Generate an ECC value from the specified bitfield in this BitData given an ECC mask (HMAT) table.

Syntax: GenerateECC(o4Offset, o4Size, ao4EccMasks).

Parameters: o4Offset Offset to first bit of bitfield to process.

o4Size Number of bits in bitfield.

ao4EccMasks The ECC mask table. See Remarks.

Returns: Return a 32-bit value containing the generated CRC of the specified bit-field.

Remarks: The number of masks in the ECC mask table must be the same as the number of bits in the bit field.

Example

```
eccMasks = [3, 5]
bd = BitData("0x00010000")
ecc = bd.GenerateECC(16, 2, eccMasks)
print("Generated ECC = 0x%x"%ecc)
```

The output looks like this:

```
Generated ECC = 0x3
```

4.4.4.16 GetBit()

Description: Retrieve the value of a specified bit from this BitData.

Syntax: GetBit(o4Offset)

Parameters: o4Offset Offset to the bit to retrieve.

Returns: 0 if the bit is clear or 1 if the bit is set.

Remarks: None.

Example

```
bd = BitData("0y100100")
print("Bit at position 2 is %d"%bd.GetBit(2))
```

The output looks like this:

```
Bit at position 2 is 1
```

4.4.4.17 GrowSize()

Description: Increase the capacity of this BitData by the specified amount.

Syntax: GrowSize(iBits)

Parameters: iBits The number of bits of capacity to add.

Returns: None.

Remarks: The new capacity will be rounded up to the nearest 32 bits to accommodate the internal storage mechanism for the bits. The capacity will never go down.

Example

```
bd = BitData(32)
print("Original capacity = %d"%bd.BitCapacity)
bd.GrowSize(24)
print("New capacity = %d"%bd.BitCapacity)
```

The output looks like this:

```
Original capacity = 32
New capacity = 64
```

4.4.4.18 Initialize()

Description: DivInitialize a bitfield in this BitData to a specified value, up to 32 bits in size.

Syntax: Initialize(o4StartBit, o4BitSize, o4Value)

Parameters: o4DstOffset Offset to first bit in the bitfield to affect.

o4Size Number of bits in the bitfield to affect.

o4Value The value to use. Can be up to 32 bits in size.

Returns: None.

Remarks: This method is the same as the Deposit() method except that a maximum of 32 bits of value can be used.

Example

```
bd = BitData("0xf0f")
bd.Initialize(4, 4, 0xe)
print(bd)
```

The output looks like this:

```
[12b] 0xFEF
```

4.4.4.19 Insert()

Description: Insert a BitData into this BitData at the specified bit offset.

Syntax: Insert(data, bitOffset)

Parameters: data The BitData object to insert.

bitOffset The 64-bit bit offset into this BitData where to insert.

Returns: None.

Remarks: The bits to the left of the bit offset are pushed to the left to make room for the new BitData being inserted. The capacity of this BitData is increased as needed to accommodate the inserted bits.

Example

```
bd = BitData("0xff")
bd2 = BitData("0xe")
bd.Insert(bd2, 4)
print(bd)
```

The output looks like this:

```
[12b] 0xFEF
```

4.4.4.20 InvBit()

Description: Invert the bit's value in this BitData at the specified offset.

Syntax: InvBit(o4Offset)

Parameters: o4Offset Offset to the bit to invert.

Returns: None.

Remarks: None.

Example

```
bd = BitData("0xff")
bd.InvBit(3)
print(bd)
```

The output looks like this:

```
[8b] 0xF7
```

4.4.4.21 Invert()

Description: Perform an XOR (exclusive-or) on the specified bitfield in this BitData with the specified mask, with the result being written to the specified bitfield.

Syntax: Invert(o4DstOffset, o4Size, o4Value)

Parameters: o4DstOffset Offset to the first bit of the bitfield to change.

o4Size The number of bits in the bitfield.

o4Value The 32-bit value to use as the mask.

Returns: None.

Remarks: The number of bits actually affected is the lesser of the specified bit size or 32.

Example

```
bd = BitData("0xaa")
bd.Invert(4, 4, 0x55)
print(bd)
```

```
```

The output looks like this:

```
[8b] 0xFA
```

4.4.4.22 IsBitOff()

Description: Determine if the specified bit is off.

Syntax: IsBitOff(o4Offset)

Parameters: o4Offset Offset to the bit to test.

Returns: Return True if the bit is off (set to 0); otherwise, return False.

Remarks: You can also use the GetBit() method to get the state of the bit at a specific offset.

Example

```
bd = BitData("0y10110")
bd.IsBitOff(3)
```

The output looks like this:

```
True
```

4.4.4.23 IsBitOn()

Description: Determine if the specified bit is on.

Syntax: Determine if the specified bit is on.

Parameters: o4Offset Offset to the bit to test.

Returns: Return True if the bit is on (set to 1); otherwise, return False.

Remarks: You can also use the GetBit() method to get the state of the bit at a specific offset.

Example

```
bd = BitData("0y10110")
bd.IsBitOn(1)
```

The output looks like this:

```
True
```

4.4.4.24 IsValue()

Description: Determine if the specified bitfield in this BitData wholly contains a pattern.

Syntax: IsValue(o4Offset, o4Size, o4Value)

Parameters: o4Offset Offset to the first bit of the bitfield to test.

o4Size Number of bits in the bitfield to test.

o4Size The pattern to test against. See Remarks.

Returns: Return True if the bitfield contains the pattern; otherwise, returns False.

Remarks: The value is seen as a pattern that might be replicated throughout the bitfield. The IsValue() method determines if the value is actually replicated. This method can handle a partial pattern if the bitfield is smaller than 32 bits, in which case only the corresponding bits in the value are compared. This method can also handle a bitfield that is larger than 32 bits but is not an even multiple of 32 bits; the remaining bits are compared to the corresponding remaining bits in the value.

Example

```
bd = BitData("0xf1234567812345678f")
print("Pattern is repeated in %s: %s"%(bd, bd.IsValue(4, 64, 0x12345678)))
bd = BitData("0xf9234567812345678f")
print("Pattern is repeated in %s: %s"%(bd, bd.IsValue(4, 64, 0x12345678)))
```

The output looks like this:

```
Pattern is repeated in [72b] 0xF1234567812345678F: True
Pattern is repeated in [72b] 0xF9234567812345678F: False
```

4.4.4.25 MaskedMatch()

Description: Perform a masked bitwise comparison of a bitfield in this BitData to a specified value.

Syntax: MaskedMatch(bitOffset, bitSize, value, mask)

Parameters: bitOffset Offset to the first bit of the bitfield to test.

bitSize Number of bits in the bitfield to test.

value A BitData containing the bits to compare against.

mask A BitData containing a mask used to mask out unwanted bits in the comparison. The size of this BitData must be the same as the number of bits in the bitSize parameter.

Returns: Return True if the bitfield matches the value after masking out the unwanted bits.

Remarks: None.

Example

```
bd = BitData("0xfeef")
value = BitData("0x22")
mask = BitData("0x33")
bd.MaskedMatch(4, 8, value, mask)
```

The output looks like this:

```
True
```

4.4.4.26 MoveBit()

Description: Copy a single bit from a source BitData into this BitData.

Syntax: MoveBit(o4DstOffset, cbdSrc, o4SrcOffset)

Parameters: o4DstOffset Offset into this BitData where the bit will go.

cbdSrc The BitData from which to get the bit.

o4SrcOffset Offset into the source BitData from where to get the bit.

Returns: None.

Remarks: None.

Example

```
bd = BitData("0y101")
bd2 = BitData("0y1")
bd.MoveBit(1, bd2, 0)
print(bd)
```

The output looks like this:

```
[3b] 0x7
```

4.4.4.27 Multiply()

Description: Multiply two values represented by bitfields (ranges of bits) in two BitData objects and deposit the result in the specified bitfield in this BitData.

Syntax: Multiply(o4DstOffset, o4DstSize, cbdSrcA, o4SrcAOffset, o4SrcASize, cbdSrcB, o4SrcBOffset, o4SrcBSize)

Parameters: o4DstOffset Bit offset in this BitData where the result will be stored.

o4DstSize Size of the bitfield in this BitData where the result will be stored.

cbdSrcA First BitData from which to get the multiplicand.

o4SrcAOffset Bit offset into first BitData from where to get the multiplicand.

o4SrcSize Size of the bitfield in the first BitData from where to get the multiplicand.

cbdSrcB Second BitData from which to get the second multiplier.

o4SrcBOffset Bit offset into the second BitData from where to get the multiplier.

o4SrcBSize Size of the bitfield in the second BitData from where to get the multiplier.

Returns: None.

Remarks: The result is padded or truncated to the destination bitfield. Supports arbitrary limits (up to 2^31 bits).

Example

```
bd  = BitData(32, "0xff0f")
bdA = BitData(32, "0x0400")
bdB = BitData(32, "0x0002")
bd.Multiply(4, 4, bdA, 8, 4, bdB, 0, 4)
print(bd)
```

The output looks like this:

```
[32b] 0x0000FF8F
```

4.4.4.28 Parity()

Description: Retrieve the parity of this BitData or a bitfield within this BitData.

Syntax: Parity()

Parity(o4Offset, o4Size)

Parameters: o4Offset 32-bit offset to the first bit of the bitfield to examine.

o4Size 32-bit count of bits in the bitfield to examine.

Returns: Return 1 if the bitfield has an odd number of bits set; otherwise, return 0.

Remarks: The first form of this method performs a parity calculation on the entire BitData. The second form performs a parity calculation on a bitfield of this BitData.

Example

```
bd = BitData("0y111")
parity = bd.Parity()
print("Parity calculation for %s: %d (1 means odd number of bits)"%(bd, parity)
```

The output looks like this:

```
Parity calculation for [3b] 0x7: 1 (1 means odd number of bits)
```

4.4.4.29 PopCount()

Description: Retrieve the count of set bits in this BitData or a bitfield of this BitData.

Syntax: PopCount()

Popcount(o4SrcStart, o4Size)

Parameters: o4SrcStart 32-bit offset to the first bit of the bitfield to examine.

o4Size 32-bit count of bits in the bitfield to examine.

Returns: Return the number of bits that are set.

Remarks: The first form of this method looks at the bits in the entire BitData. The second form performs looks at the bits in a bitfield of this BitData.

Example

```
bd = BitData("0y111")
bitCount = bd.PopCount()
print("Number of bits set in %s: %d"%(bd, bitCount))
```

The output looks like this:

```
Number of bits set in [3b] 0x7: 3
```

4.4.4.30 Randomize()

Description: Generate a BitData with random data or randomize a bitfield within this BitData.

Syntax: Randomize()

Randomize(o4Offset, o4Size)

Parameters: o4Offset 32-bit offset to the first bit of the bitfield to change.

o4Size 32-bit count of bits in the bitfield to change.

Returns: The first form returns a copy of the BitData containing the randomized value. The original BitData is untouched. The second form changes this BitData (or the bitfield in this BitData) and returns nothing.

Remarks: The first form of this method creates a copy of this BitData then fills the copy with random data. The new BitData is returned, leaving this BitData untouched. The second form randomizes a bitfield within this BitData.

Example

```
bd = BitData(32, 0xffffffff)
randombd = bd.Randomize()
print("Original BitData   : %s"%bd)
print("Randomized BitData : %s"%randombd)
bd.Randomize(4, 8)
print("Randomized bitfield: %s"%bd)
```

The output looks like this:

```
Original BitData   : [32b] 0xFFFFFFFF
Randomized BitData : [32b] 0x36E4A66F
Randomized bitfield: [32b] 0xFFFFF83F
```

4.4.4.31 Read()

Description: Return a collection of bits from this BitData as a new BitData object.

Syntax: Read(indices)

Read(bitOffset, bitSize)

Parameters: indices An IndexList object containing a collection of bit indices.

bitOffset First bit in the bitfield to retrieve.

bitSize 32-bit count of bits in the bitfield to retrieve.

Returns: Return a BitData containing the requested bits.

Remarks: This method is the same as the Copy() command (the form that extracts bits from this BitData).

Example

```
bd = BitData(32, 0xffffffff)
randombd = bd.Randomize()
print("Original BitData   : %s"%bd)
print("Randomized BitData : %s"%randombd)
```

The output looks like this:

```
Original BitData   : [32b] 0xFFFFFFFF
Randomized BitData : [32b] 0x36E4A66F
```

4.4.4.32 ReadByteArray()

Description: Return a collection of bits from this BitData as an array of unsigned 8-bit values.

Syntax: ReadByteArray(bitOffset, byteCount)

Parameters: bitOffset First bit in the bitfield to retrieve.

byteSize Number of bytes (8-bit chunks) to retrieve.

Returns: Return a System.Byte[] array containing the data.

Remarks: The returned data is in a .Net array. The example shows one way to convert this returned array into python list. This method is used for serializing the bits to a .Net stream, which requires the data to be in a byte array.

Example

```
bd = BitData(32, 0x123456789)
data = bd.ReadByteArray(0, 4)
dataList = []
for b in data:
    dataList.append(b)

print("raw data: %s"%dataList)
print("In hex form:")
for b in dataList:
    print(hex(b)),
```

The output looks like this:

```
raw data: [137, 103, 69, 35]
In hex form:
0x89 0x67 0x45 0x23
```

4.4.4.33 ReadUInt32()

Description: Return a 32-bit integer from the specified offset into this BitData.

Syntax: ReadUInt32(bitOffset)

Parameters: bitOffset First bit in the 32-bit bitfield to retrieve.

Returns: Return an integer containing the extracted 32 bits.

Remarks: This method is the same as the Extract32()

Example

```
bd = BitData("0xf12345678f")
print("Extracted value = 0x%x"%bd.ReadUInt32(4))
```

The output looks like this:

```
Extracted value = 0x12345678
```

4.4.4.34 ReadUInt64()

Description: Return a 64-bit integer from the specified offset into this BitData.

Syntax: ReadUInt64(bitOffset)

Parameters: bitOffset First bit in the 64-bit bitfield to retrieve.

Returns: Return an integer containing the extracted 64 bits.

Remarks: This method is the same as the Extract64() method

Example

```
bd = BitData("0xffee1122334455667788aabb")
print("Extracted value = 0x%x"%bd.ReadUInt64(0))
```

The output looks like this:

```
Extracted value = 0x334455667788aabb
```

4.4.4.35 Reverse()

Description: Retrieve a copy of this BitData with the bits reversed or reverse the bits within a bitfield of this BitData.

Syntax: Reverse()

Reverse(o4Offset, o4Size)

Parameters: o4Offset First bit in the bitfield to reverse.

o4Size Number of bits in the bitfield to reverse.

Returns: The first form returns a new BtiData object with the bits of this BitData in reversed order. The second form always returns None.

Remarks: The first form leaves this BitData unchanged.The second form changes this BitData.

Example

```
bd = BitData(16, "0x5555")
print("Original BitData : %s"%bd)
bdReversed = bd.Reverse()
print("Reversed Copy    : %s"%bdReversed)
bd.Reverse(0, 8)
print("Reversed bitfield: %s"%bd)
```

The output looks like this:

```
Original BitData : [16b] 0x5555
Reversed Copy    : [16b] 0xAAAA
Reversed bitfield: [16b] 0x55AA
```

4.4.4.36 Rotate()

Description: Retrieve a copy of this BitData with the bits rotated by a specified amount or rotate a bitfield within this BitData by a specified amount.

Syntax: Rotate(count, bitOffset = None, bitSize = None)

Parameters: count Number of times to rotate the bits.

bitOffset First bit in the bitfield to rotate

bitSize Number of bits in the bitfield to rotate.

Returns: If bitOffset or bitSize is None, a copy of this BitData is returned with the bit rotated. Otherwise, this BitData is modified.

Remarks: If the count is positive, the bits are rotated to the left. If the count is negative, the bits are rotated to the right. Bits rotated off of one side of the BitData are rotated onto the other.

Example

```
bd = BitData(16, "0xf123")
print("Original BitData      : %s"%bd)
bdRotated = bd.Rotate(4)
print("Rotated BitData left  : %s"%bdRotated)
bdRotated2 = bd.Rotate(-4)
print("Rotated BitData right : %s"%bdRotated2)
bd.Rotate(4, 4, 8)
print("Rotated bitfield      : %s"%bd)
```

The output looks like this:

```
Original BitData      : [16b] 0xF123
Rotated BitData left  : [16b] 0x123F
Rotated BitData right : [16b] 0x3F12
```

```
Rotated bitfield      : [16b] 0xF213
```

4.4.4.37 SetBit()

Description: Set the specified bit in this BitData to the specified value.

Syntax: SetBit(o4Offset, o4Bit)

Parameters: o4Offset Bit offset in this BitData to the bit to change.

o4Bit Value to write to the bit (0 or non-zero).

Returns: None.

Remarks: None.

Example

```
bd  = BitData(16, "0xf111")
bd.SetBit(3, 1)
bd.SetBit(12, 0)
print(bd)
```

The output looks like this:

```
[16b] 0xE119
```

4.4.4.38 SetBitOff()

Description: Clear the specified bit in this BitData.

Syntax: SetBitOff(o4Offset)

Parameters: o4Offset Bit offset in this BitData to the bit to change.

Returns: None.

Remarks: None.

Example

```
bd  = BitData(16, "0xf111")
bd.SetBitOff(12)
print(bd)
```

The output looks like this:

```
[16b] 0xE111
```

4.4.4.39 SetBitOn()

Description: Set the specified bit in this BitData.

Syntax: SetBitOn(o4Offset)

Parameters: o4Offset Bit offset in this BitData to the bit to change.

Returns: None.

Remarks: None.

Example

```
bd  = BitData(16, "0xf111")
bd.SetBitOn(3)
print(bd)
```

The output looks like this:

```
[16b] 0xF119
```

4.4.4.40 ShiftLeft()

Description: Shift the specified bitfield in this BitData left the specified number of times.

Syntax: ShiftLeft(o4Offset, o4Size, o4count)

Parameters: o4Offset Bit offset to the first bit in the bitfield to shift.

o4Size Number of bits in the bitfield to shift.

o4Count Number of times to shift the bits left.

Returns: None.

Remarks: The bits shifted in from the right are always 0.

Example

```
bd  = BitData(16, "0xffff")
bd.ShiftLeft(0, 16, 4)
print(bd)
```

The output looks like this:

```
[16b] 0xFFF0
```

4.4.4.41 ShiftRight()

Description: Shift the specified bitfield in this BitData right the specified number of times.

Syntax: ShiftRight(o4Offset, o4Size, o4count)

Parameters: o4Offset Bit offset to the first bit in the bitfield to shift.

o4Size Number of bits in the bitfield to shift.

o4Count Number of times to shift the bits left.

Returns: None.

Remarks: The bits shifted in from the left are always 0.

Example

```
bd  = BitData(16, "0xffff")
bd.ShiftRight(0, 16, 4)
print(bd)
```

The output looks like this:

```
[16b] 0x0FFF
```

4.4.4.42 ShiftRightGroup()

Description: Shift the bits of this BitData into the bits in the specified list of BitData objects.

Syntax: ShiftRightGroup(acBitData)

Parameters: acBitData An array of BitData objects to affect.

Returns: None.

Remarks: The intent of this specialized method is to treat all the BitData object in the array as a single collection of bits. The bits in this BitData are shifted into the collection of bits from the left. The bits in this BitData object remain unaffected.

> **Note**
> The first entry in the list is treated as the least significant component. This means the bits from this BitData are pushed into the last entry in the list and bits fall off the first entry in the list.

Example

```
bdList = [BitData(32, 0x11111111), BitData(32, 0x22222222), BitData(32, 0x33333
bd = BitData(16, 0x4444)
bd.ShiftRightGroup(bdList)
print(bdList)
```

The output looks like this:

```
[[32b] 0x22221111, [32b] 0x33332222, [32b] 0x44443333]
```

4.4.4.43 Subtract()

Description: Subtract two values represented by ranges of bits in two BitData objects and deposit the result in the specified bitfield in this BitData.

Syntax: Subtract(o4DstOffset, o4DstSize, cbdSrcA, o4SrcAOffset, o4SrcASize, cbdSrcB, o4SrcBOffset, o4SrcBSize)

Parameters: o4DstOffset Bit offset in this BitData where the result will be stored.

o4DstSize Size of the bitfield in this BitData where the result will be stored.

cbdSrcA First BitData from which to get the first minuend.

o4SrcAOffset Bit offset into first BitData from where to get the minuend.

o4SrcSize Size of the bitfield in the first BitData from where to get the minuend.

cbdSrcB Second BitData from which to get the second minuend.

o4SrcBOffset Bit offset into the second BitData from where to get the minuend.

o4SrcBSize Size of the bitfield in the second BitData from where to get the minuend.

Returns: None.

Remarks: The result is padded or truncated to the destination bitfield.

Example

```
bd  = BitData(32, "0xff0f")
bdA = BitData(32, "0x0300")
bdB = BitData(32, "0x0002")
bd.Subtract(4, 4, bdA, 8, 4, bdB, 0, 4)
print(bd)
```

The output looks like this:

```
[32b] 0x0000FF1F
```

4.4.4.44 ToBinary()

Description: Convert the BitData to a formatted binary string.

Syntax: string ToBinary()

Parameters: None.

Returns: Return a string containing the formatted BitData.

Remarks: None.

Example

```
bd = BitData(32, 10)
print("BitData in binary: %s"%bd.ToBinary())
```

The output looks like this:

```
BitData in binary: 0y00000000000000000000000000001010
```

4.4.4.45 ToDecimal()

Description: Convert the BitData to a formatted decimal string.

Syntax: string ToDecimal()

Parameters: None.

Returns: Return a string containing the formatted BitData.

Remarks: None.

Example

```
bd = BitData(32, 10)
print("BitData in decimal: %s"%bd.ToDecimal())
```

The output looks like this:

```
BitData in decimal: 10
```

4.4.4.46 ToHex()

Description: Convert the BitData to a formatted hexadecimal string.

Syntax: string ToHex()

Parameters: None.

Returns: Return a string containing the formatted BitData.

Remarks: This method differs from ToString() (section 4.4.1.51) in that the number of bits is not prefixed to the output.

Example

```
bd = BitData(32, 10)
```

```
print("BitData in hex: %s"%bd.ToHex())
```

The output looks like this:

```
BitData in hex: 0x0000000A
```

4.4.4.47 ToString()

Description: Convert the BitData to a formatted hexadecimal string with leading bit count.

Syntax: string BitData.ToString()

Parameters: None.

Returns: Return a string containing the formatted BitData.

Remarks: This method is called automatically by the str() python command and does not need to be called directly. Note that the %s formatter in the print statement uses the str() command internally to get the object in string form.

To get a formatted hex string without the number of bits prepended to the output, use the ToHex() method

Example

```
bd = BitData(32, 10)
print("Formatted BitData: %s"%bd)
```

The output looks like this:

```
Formatted BitData: [32b] 0x0000000A
```

4.4.4.48 ToUInt32()

Description: Return this BitData object as a 32-bit integer.

Syntax: ToUInt32()

Parameters: None.

Returns: Return an integer containing this BitData value as a 32-bit integer.

Remarks: This method is the same as the Extract32() method except the entire BitData contributes to the value. If the value in this BitData cannot fit in 32 bits, an exception is thrown.

Example

```
bd = BitData("0x12345678")
print("32-bit value = 0x%X"%bd.ToUInt32())
```

The output looks like this:

```
32-bit value = 0x12345678
```

4.4.4.49 ToUInt64()

Description: Return this BitData object as a 64-bit integer.

Syntax: ToUInt64()

Parameters: None.

Returns: Return an integer containing this BitData value as a 64-bit integer.

Remarks: This method is the same as the Extract32() method except the entire BitData contributes to the value. If the value in this BitData cannot fit in 64 bits, an exception is thrown.

Example

```
bd = BitData("0x123456789abcdef")
print("64-bit value = 0x%X"%bd.ToUInt64())
```

The output looks like this:

```
64-bit value = 0x123456789ABCDEF
```

4.4.4.50 TrimCapacity()

Description: Reduce the internal storage size of this BitData to the minimum needed to contain the current value.

Syntax: TrimCapacity()

Parameters: None.

Returns: None.

Remarks: The minimum capacity for a non-empty BitData is 32 bits (a result of how the bits are stored internally).

Example

```
bd = BitData(16)
print("Original capacity: %d"%bd.BitCapacity)
bd.BitCapacity = 128
print("Updated capacity : %d"%bd.BitCapacity)
bd.TrimCapacity()
```

```
print("Trimmed capacity : %d"%bd.BitCapacity)
```

The output looks like this:

```
Original capacity: 32
Updated capacity : 128
Trimmed capacity : 32
```

4.4.4.51 Update()

Description: Perform a bitwise update of a destination bitfield, with a value and a mask.

Syntax: Update(o4DstOffset, o4DstSize,bitDataUpdate, o4SrcOffset)

Update(o4DstOffset, o4DstSize,bitDataUpdate, o4SrcAOffset, bitDataMask, o4SrcBOffset)

Parameters: o4DstOffset Bit offset in this BitData where the result will be stored.

o4DstSize Size of the bitfield in this BitData where the result will be stored.

bitDataUpdate Value to write to the bitfield.

o4SrcOffset Offset to first bit in the value from which to write.

o4SrcAOffset Offset to first bit in the value from which to write.

bitDataMask The mask to apply to the value before writing.

o4SrcBOffset Offset to first bit in the mask to use.

Returns: None.

Remarks: The operation performed is this = (this & ~mask) | (value & mask). Basically, use the mask to make a hole in this BitData and insert the new data into that hole.

The first form is the same as using the Deposit() method as no mask is provided. Using a mask allows for masking across many bits; using the Deposit() method would require a individual method call for each unique section of bits. The example shows two sections of the data being affected in a single operation.

Example

```
bd  = BitData(32, "0xffff")
bdValue = BitData(32, "0x3002")
bdMask  = BitData(32, "0xf00f")
bd.Update(0, 16, bdValue, 0, bdMask, 0)
print(bd)
```

```
```

The output looks like this:

```
[32b] 0x00003FF2
```

4.4.4.52 ValueCompare()

Description: Compare the value stored in this BitData to the value stored in a specified BitData.

Syntax: ValueCompare(cBitData)

Parameters: cBitData The BitData to compare to.

Returns: Return an integer result:
-1 means 'A < B'

0 means 'A == B'

+1 means 'A > B'

Remarks: This method is the same as the CompareTo() method.

Example

```
bd1 = BitData(32, 0x100)
bd2 = BitData(16, 0x200)
result = bd1.ValueCompare(bd2)
print("Result of comparing %s to %s = %d"%(bd1, bd2, result))
```

The output looks like this:

```
Result of comparing [32b] 0x00000100 to [16b] 0x0200 = -1
```

4.4.4.53 ValueEquals()

Description: Compare the value stored in this BitData to the value stored in a specified BitData for equality.

Syntax: ValueEquals(cBitData)

Parameters: cBitData The BitData to compare to.

Returns: Return True if the this BitData does not equal the specified BitData.

Remarks: This method is the same as calling the CompareTo() method and checking for a 0 return.

Example

```
bd1 = BitData(32, 0x100)
```

```
bd2 = BitData(16, 0x100)
result = bd1.ValueEquals(bd2)
print("Result of comparing %s to %s = %s"%(bd1, bd2, result))
```

The output looks like this:

```
Result of comparing [32b] 0x00000100 to [16b] 0x0100 = True
```

4.4.4.54 Write()

Description: Write bits from another BitData into this BitData.

Syntax: Write(data, indices)

Write(data, bitOffset)

Write(data, bitDstOffset, bitSize)

Parameters: data A source BitData object.

indices An IndexList object containing a collection of bit indices into this BitData where to write to.

bitOffset Bit offset to where in this BitData to write the source bits.

bitDstOffset Bit offset to where in this BitData to write the source bits.

bitSize Number of bits to write from the source BitData.

Returns: None.

Remarks: Some or all of the bits from the source BitData object are copied into this BitData object. The bits are always copied from the source BitData object starting at bit offset 0.

The Write() method is the same as the form of the Copy() method that copies from a second BitData object.

Example

```
bd = BitData(16, "0xf00f")
bd2 = BitData(16, "0x1234")
print("Original BitData          : %s"%bd)
bd.Write(bd2, 4, 8)
print("After Writing to bits 4:11: %s"%bd)
```

The output looks like this:

```
Original BitData          : [16b] 0xF00F
After Writing to bits 4:11: [16b] 0xF34F
```

4.4.4.55 WriteByteArray() ()

Description: Write an array of bytes into this BitData starting at the specified bit offset.

Syntax: WriteByteArray(data, bitOffset)

Parameters: data A .Net array of bytes (System.Byte[]).

bitOffset First bit in the bitfield to write to.

Returns: None.

Remarks: This method is used for serializing the bits from a .Net stream, where the data comes in as a byte array. If this BitData is not large enough to hold all the data, the remaining data is ignored.

Example

```
bd = BitData(32, 10)
bd = BitData(32, 0x123456789)
data = bd.ReadByteArray(0, 4)
bd2 = BitData(32)
bd2.WriteByteArray(data, 0)
print(bd2)
```

The output looks like this:

```
[32b] 0x23456789
```

4.4.4.56 WriteUInt32()

Description: Write a 32-bit integer into this BitData at the specified offset.

Syntax: WriteUInt32(data, bitOffset)

Parameters: None.

Returns: None.

Remarks: This method is the same as the Deposit32() method except the parameters are in the opposite order.

Example

```
bd = BitData("0xf00000000f")
bd.WriteUInt32(0x12345678, 4)
print(bd)
```

The output looks like this:

```
[40b] 0xF12345678F
```

4.4.4.57 WriteUInt64()

Description: Write a 64-bit integer into this BitData at the specified offset.

Syntax: WriteUInt64(data, bitOffset)

Parameters: data The data to write.

bitOffset First bit in the 64-bit bitfield to write to.

Returns: None.

Remarks: None.

Example

```
bd = BitData("0xf0000000000000000f")
bd.WriteUInt64(0x1122334455667788, 4)
print(bd)
```

The output looks like this:

```
[72b] 0xF1122334455667788F
```

4.5 itpii.node.Node class

A Node class represents one node in the device tree (see the Logical Device Tree section in the Appendix). An instance of the Node class contains properties specific to the type of node being represented, although there are common properties for all nodes (such as name and type).

There are specific node classes for each device node exposed by the ITP Commands Library. All of these are derived from the Node base class.

- NodeRoot
- NodeDomain
- NodePackage
- NodeDie
- NodeCore
- NodeUncore
- NodeChipset
- NodeBox

- NodeThread
- NodeDebugPort

An instance of a Node class can contain zero or more properties that represent collections of different types of children nodes. These properties are represented as lists of the Node class instances.

All of the collections of nodes are instances of the itpii.NodeContainer class

4.5.1 Attributes

These attributes are common to all Node instances.

Table 4.8. itpii.node.Node attributes

Attribute	Type	Description
children	NodeContainer	If this node has any direct children, this is a list of those children Nodes. If the node has no children, this list is empty.
debugPortNode	NodeDebugPort	The debug port that contains this node.
hooks	NodeHook	Collection of Dfx hooks available on this node.
isvalid	Bool	True if this node is considered valid. If the device tree is reconfigured, all existing nodes are marked as invalid.
isenabled	Bool	True if this node is enabled.
name	string	Name of the node. For example, a core might be named "Nehalem".
parent	Node	The parent node to this node. If this node has no parent then the node is a root node.
type	string	The type of node. See the Notes part of section 4.5.2.3 for a list of the types that can appear.
state	State	This is not used at this time other than to hold a collection of architectural registers. This attribute will eventually hold all state access for the platform.

Attribute	Type	Description
stepping	string	The stepping of the device. Some node types typically do not have a stepping, in which case this attribute returns None.

4.5.2 Functions

4.5.2.1 getByName()

Description: Search child nodes (and their children recursively) for the first node with the specified name.

Syntax: Node Node.getByName(match)

Parameters: match: String to match against the device node name. See Remarks.

Returns: A single node if found; otherwise, returns None.

Remarks: The name can include wildcards ('?' for single character, '*' for zero or more characters), in which case getByName() returns the first match.

Example

```
domain = itp.domains[0]
somedevice = domain.getByName("NHMUC")
if somedevice != None:
print("device %s has type %s"%(somedevice.Name, somedevice.Type))
```

4.5.2.2 getAllByName()

Description: Search child nodes (and their children recursively) for all nodes whose names match.

Syntax: Node[] Node.getAllByName(match)

Parameters: match: String to match against the device node name. See Remarks.

Returns: A list of nodes if any matches are found; otherwise, returns an empty list.

Remarks: The name can include wildcards ('?' for single character, '*' for zero or more characters).

Example

```
domain = itp.domains[0]
somedevices = []
somedevices = domain.getAllByName("NHM*")
if somedevices != None:
    for device in somedevices:
        print("device %s has type %s"%(device.Name, device.Type))
```

4.5.2.3 getByType()

Description: Search child nodes (and their children recursively) for the first node of the specified type.

Syntax: Node Node.getByType(match)

Parameters: match: String to match against the device node types. See Remarks.

Returns: A single node if found; otherwise, returns None.

Remarks: Supported types that can be searched for (the ItpII.nodetypes list contains the latest supported types):

Table 4.9. itpii.node search locations

Type	Can be found searching from
"package"	domain
"die"	domain or package
"core"	domain, package, or die
"uncore"	domain, package, or die
"chipset"	domain, package, or die
"thread"	domain, package, die, or core
"box"	domain, package, die, core, chipset, or uncore **Note** A "box" is a general term for some more specific functionality. The name of the "box" specifies what the box functionality is for. For example, a box named "PCU" would provide access to a Power Control Unit.

Example

```
domain = itp.domains[0]
somecore = domain.getByType("core")
if somecore != None:
    print("core ID = %s"%somecore.coreID)
```

4.5.2.4 getAllByType()

Description: Search child nodes (and their children recursively) for all nodes of a specific type.

Syntax: Node[] Node.getAllByType(match)

Parameters: match: String to match against the device node types. See Remarks.

Returns: A list of nodes if any matches are found; otherwise, returns an empty list.

Remarks: See Notes in getByType() section for the supported types that can be searched for.

Example

```
domain = itp.domains[0]
somecores = []
somecores = domain.getAllByType("core")
if somecores != None:
    for core in somecores:
        print("core ID = %s"%core.coreID)
```

4.5.2.5 info()

Description: Display the properties of the node in a formatted way. What is displayed is dependent on the type of node. The display could include a break-down into sub-components.

Syntax: Node.info()

Parameters: indent -- (optional) The number of spaces to indent any display of text.

detailsOn -- (optional) Controls how much information to show per node.

True: display the device tree with multiple pieces of information per node (list takes many lines).

False: display the device tree in a minimal form, showing only the names and types of each node.

Returns: None.

Remarks: The parameters are normally used by the NodeContainer.infoall() method

Example

```
domain = itp.domains[0]
somecore = domain.getByType("core")
```

4.6 itpii.NodeState class

The NodeState class represents the various properties or state of a device node in a hierarchical tree. An instance of the NodeState class appears on a device node object as the state attribute. For example, itp.threads[0].state or itp.uncores[0].state. This potentially

very large state tree can be navigated using standard python dot notation; for example, itp.threads[0].state.archreg.rax.eax.

4.6.1 Attributes

Table 4.10. itpii.NodeState attributes

Attribute	Type	Description
value	itpii.datatypes .BitData	Returns the value of the state node as a BitData. Can also be assigned to so as to change the value of the state node.
regs	itpii.NodeState .NodeState	An alias for the archreg state node. Maintained for backward compatibility. Only device nodes that have architectural state will have the regs attributes.

4.6.2 Functions

4.6.2.1 find()

Description: Retrieve the specified node by searching the current node and all child nodes.

Syntax: NodeState.find(name)

Parameters: name -- Find the specified node by searching the current node and all child nodes.

Returns: The requested state node if found; otherwise, returns None (no node by that name found).

Remarks: This method differs from the search() method by returning the actual node object and is limited to returning a single node.

Example

```
def example():
    itp.halt()
    try:
        node = itp.threads[0].state.regs.find("eax")
        print("eax = %s"%node)

        t0 = itp.threads[0]
        ctr = 1; token = 'mm'
        node = t0.state.find('%s%d'%(token, ctr+1))
        print("%s = %s"%(node.name, node.value))
    finally:
        itp.go()
```

```
example()
```

The output looks like this:

```
          [HSW_C0_T0] Halt Command break at 0xF000:000000000000FFF0
          [HSW_C0_T1] Halt Command break at 0xF000:000000000000FFF0
          [HSW_C1_T0] Halt Command break at 0xF000:000000000000FFF0
          [HSW_C1_T1] Halt Command break at 0xF000:000000000000FFF0
          [HSW_C2_T0] Halt Command break at 0xF000:000000000000FFF0
          [HSW_C2_T1] Halt Command break at 0xF000:000000000000FFF0
          [HSW_C3_T0] Halt Command break at 0xF000:000000000000FFF0
          [HSW_C3_T1] Halt Command break at 0xF000:000000000000FFF0
eax = [32b] 0xC070EAEA
mm2 = [64b] 0x0000000000000000
```

4.6.2.2 get()

Description: Return a copy of the state tree starting at this node. This copy can be modified as desired without affecting the DAL state. To commit the changes to the DAL, call the put() method on this node with the modified copy of the state tree.

Syntax: NodeState.get()

Parameters: None.

Returns: A NodeState object specifying the root of the copied tree. The tree's root corresponds to the state node the get() method was called on.

Remarks: Note: Calling the get() method retrieves the state from the device if it has not already been read in.

Example

```
itp.halt()
regState = itp.threads[0].state.archreg.get()
regState.rax = 0xffffffffffffffff
regState.rbx.ebx = 0x11111111
itp.threads[0].state.archreg.put(regState)
```

4.6.2.3 put()

Description: Commit the specified state tree to the DAL's state. The state tree presumably came from the get() method on this node.

Syntax: NodeState.put(stateTree)

Parameters: stateTree -- Root of the state tree to commit to the DAL's state.

Returns: None.

Remarks: None.

Example

```
itp.halt()
regState = itp.threads[0].state.archreg.get()
regState.rax = 0xffffffffffffffff
regState.rbx.ebx = 0x11111111
itp.threads[0].state.archreg.put(regState)
```

4.6.2.4 save()

Description: Search the state tree from this node onward for the specified node or nodes and return a list of all matching nodes.

Syntax: NodeState.save(filename, filetype=None)

Parameters: filename -- Name of the file to save to. Must be a string that points to a valid path for the file. If the filename ends with a valid extension (those listed below), then that is the format that will be used. If it cannot be determined from the filename, the filetype must be explicitly specified.

filetype -- Type of file to save to. Can be unspecified in favor of using a valid extension in the filename. If not, or the extension cannot be determined from the filename, the type must be specified as a string that matches one of the following:

"txt" -- Text file, same in format as the textual representation seen at the command line.

"vcd" -- VCD file (Value Change Dump).

Returns: None.

Remarks: This example shows three ways to save the rax state. The first saves to hello.txt by explicitly providing the file type. The second saves to hello.vcd, with the implied file type of "vcd". The third gets a local copy of the rax state and saves that to hellolocal.txt, with the implied file type of "txt".

> **Note**
> register access, even through the state tree, requires the processor to be halted.

Example

```
def example():
    itp.halt()
    try:
        itp.threads[0].state.rax.save("hello", "txt")
        itp.threads[0].state.rax.save("hello.vcd")
        itp.threads[0].state.rax.get().save("hellolocal.txt")
    finally:
        itp.go()

example()
```

4.6.2.5 search()

Description: Search the state tree from this node onward for the specified node or nodes and return a list of the names of all matching nodes.

Syntax: NodeState.search(query, maxResults=sys.maxint)

Parameters: query -- the query to find descendant node paths. Supports the following wildcards:

? -- match any single character

* -- match 0 or more characters

maxResults -- an integer value specifying how many matches to gather; set to 1 for a "shallow" search or to a higher value for a "deep" search; currently defaults to sys.maxint; will print a warning if the maxResults is actually hit.

Returns: List of strings (relative paths to the matching nodes)

Remarks: A "shallow" search does not mean limited to the immediate children of the current state node where the search begins. In this case, it means stop after the first matching node is found. If the node does not exist, the entire state tree from the current node down is still searched.

Use the find() method to retrieve the actual node object for a specified search.

Example

```
def example():
    itp.halt()
    regs = itp.threads[0].state.archreg.search("rax")
    print("Path to rax register = %s"%regs)
    regs = itp.threads[0].state.archreg.search("r?x")
    print("Paths to all 64-bit registers = %s"%regs)
    regs = itp.threads[0].state.archreg.search("*index")
    print("Paths to all registers end in 'index' = %s"%regs)
    itp.go()

example()
```

The output looks like this:

```
                [HSW_C0_T0] Halt Command break at 0x0:0000000000000000
        [HSW_C0_T1] Halt Command break at 0x0:0000000000000000
        [HSW_C1_T0] Halt Command break at 0x0:0000000000000000
        [HSW_C1_T1] Halt Command break at 0x0:0000000000000000
        [HSW_C2_T0] Halt Command break at 0x0:0000000000000000
        [HSW_C2_T1] Halt Command break at 0x0:0000000000000000
        [HSW_C3_T0] Halt Command break at 0x0:0000000000000000
        [HSW_C3_T1] Halt Command break at 0x0:0000000000000000
Path to rax register = [u'archreg.rax']
```

```
    Paths to all 64-bit registers = [u'archreg.rax', u'archreg.rbx',
    u'archreg.rcx', u'archreg.rdx', u'archreg.fscp64.fscp_cr_io_smi_backup_rcx'
    Paths to all registers end in 'index' =
    [u'archreg.fscp32.fscp_cr_tagec_ts_emon_index',
    u'archreg.fscp32.fscp_cr_tagec_ts_emon_index.tagec_emon_index']
```

4.7 itpii.NodeContainer class

The NodeContainer class acts like a list of Node objects but provides a number of helper functions. The find() and findall() functions search the items in the node container for a match by name; the search is not recursive. The getByName(), getAllByName(), getByType() and getAllByType() are the same functions as described in the Node class except their searches cover all the elements in the NodeContainer instance recursively. The info() method shows information about the top-level nodes in the container, without recursively descending through all child nodes. The infoall() function iterates through all of the nodes in the container recursively, calling the info() function on each node.

In addition, a NodeContainer instance can be indexed with array notation using a number or a string. The number is just a straight index into the list, for example, "domains[1]" to return the second domain (such index values start at 0). If the index is a string, then the top-level children of the container are searched by name or alias in a case-insensitive manner.

4.7.1 Functions

4.7.1.1 clone()

Description: Create a clone of this container. Does not perform a deep copy of all child nodes and only creates references to the child nodes.

Syntax: NodeContainer.clone()

Parameters: None.

Returns: None.

Remarks: None.

Example

```
domain0_threads = itp.threads.clone()
```

4.7.1.2 find()

Description: Search the list of top-level nodes in this container by name for a specific node.

Syntax: Node NodeContainer.find(match)

Parameters: match: String to match against the node names.

Returns: A single node if found; otherwise, returns None.

Remarks: The match string can contain wildcards ('*' means match 0 or more characters and '?' means match any single character), in which case the search will find the first matching node.

The search is case-insensitive.

The find() method is the same as indexing the node container with a string except that wildcards are supported by the find() method.

Example

```
singledomain = itp.domains.find("someDomainName")
singledomain = itp.domains["someDomainName"]   ## Alternative
```

4.7.1.3 findAll()

Description: Search the list of top-level nodes in this container by name for one or more nodes.

Syntax: NodeContainer NodeContainer.findall(match)

Parameters: match: String to match against the node names.

Returns: A list of nodes if any matches are found; otherwise, returns an empty list.

Remarks: The match string can contain wildcards ('*' means match 0 or more characters and '?' means match any single character), in which case the search will find all matching nodes.

Example

```
somedomains = itp.domains.findall("some*")
```

4.7.1.4 getByName()

Description: Search child nodes (and their children recursively) for the first node with the specified name.

Syntax: Node NodeContainer.getByName(match)

Parameters: match: String to match against the device node name. See Remarks.

Returns: A single node if found; otherwise, returns None.

Remarks: The name can include wildcards ('?' for single character, '*' for zero or more characters), in which case getByName() returns the first match.

Example

```
somedevice = itp.domains.getByName("NHMUC")
if somedevice != None:
    print("device %s is of type %s"%(somedevice.name, somedevice.type))
```

4.7.1.5 getAllByName()

Description: Search child nodes (and their children recursively) for all nodes whose names match.

Syntax: NodeContainer NodeContainer.getAllByName(match)

Parameters: match: String to match against the device node name. See Remarks.

Returns: A list of nodes if any matches are found; otherwise, returns an empty list.

Remarks: The name can include wildcards ('?' for single character, '*' for zero or more characters).

Example

```
somedevices = itp.domains.getAllByName("NHM*")
for device in somedevices:
    print("device %s is of type %s"%(device.name, device.type))
```

4.7.1.6 getByType()

Description: Search child nodes (and their children recursively) for the first node of the specified type.

Syntax: Node NodeContainer.getByType(match)

Parameters: match: String to match against the device node types. See Remarks.

Returns: A single node if found; otherwise, returns None.

Remarks: Supported types that can be searched for (the ItpII.nodetypes list contains the latest supported types):

Table 4.11. itpii.NodeContainer search locations

Type	Can be found searching from
"package"	domain
"die"	domain or package
"core"	domain, package, or die
"uncore"	domain, package, or die
"chipset"	domain, package, or die
"thread"	domain, package, die, or core
"box"	domain, package, die, core, chipset, or uncore **Note** A "box" is a general term for some more specific

Type	Can be found searching from
	functionality. The name of the "box" specifies what the box functionality is for. For example, a box named "PCU" would provide access to a Power Control Unit.

Example

```
somecore = itp.domains.getByType("core")
if somecore != None:
    print("core ID = %s"%somecore.coreid)
```

4.7.1.7 getAllByType()

Description: Search child nodes (and their children recursively) for all nodes of a specific type.

Syntax: NodeContainer NodeContainer.getAllByType(match)

Parameters: match: String to match against the device node types. See Remarks.

Returns: A list of nodes if any matches are found; otherwise, returns an empty list.

Remarks: See Notes in getByType() section for the supported types that can be searched for.

Example

```
somecores = itp.domains[0].getAllByType("core")
for core in somecores:
    print("core ID = %s"%core.coreid)
```

4.7.1.8 info()

Description: Display the properties of the top level nodes in the container in a formatted way. What is displayed for each node is dependent on the type of node.

Syntax: NodeContainer.info()

Parameters: None.

Returns: None.

Remarks: None.

Example

```
itp.domains.info()
```

4.7.1.9 infoall()

Description: Display the properties of all nodes in the container in a formatted way, recursively following the hierarchy of every node and child node in the container. What is displayed for each node is dependent on the type of node.

Syntax: NodeContainer.infoall(brief=False)

Parameters: brief -- (optional) Controls how much information to show per node.

True: display the device tree in a minimal form, showing only the names and types of each node.

False: display the device tree with multiple pieces of information per node (list takes many lines).

Returns: None.

Remarks: None.

Example

```
itp.domains.infoall()
itp.domains.infoall(brief = True)
```

4.8 ITP Data Types

The following python classes have been created to help convert ITP scripts to python scripts that run in the Python CLI. These data types behave more or less like the corresponding python data types. For new python scripts, you are encouraged to use the python data types directly.

The Ordx and Intx types will truncate the values assigned to those data types. For example, assigning 0xffff to an Ord1 will result in a value of 0xff. In addition, incrementing and decrementing values of the Ordx and Intx data types will wrap within the range of that data type. For example, adding 1 to Ord1(0xff) will result in 0x00.

Table 4.12. itp data types

Data Type	Description
Ord1	8-bit (1 byte) unsigned number
Ord2	16-bit (2 byte) unsigned number
Ord4	32-bit (4 byte) unsigned number
Ord8	64-bit (8 byte) unsigned number
Int1	8-bit (1 byte) signed number
Int2	16-bit (2 byte) signed number

Data Type	Description
Int4	32-bit (4 byte) signed number
Int8	64-bit (8 byte) signed number
Real4	floating point number (real)
Real8	floating point number (double)
Real10	floating point number (longdouble)
Bool1	boolean, represents True or False
Pointer	Represents an ITP pointer (treated as a simple integer)
Handle	Represents an ITP handle (treated as an opaque type)
Char	A single 8-bit character
String	A null-terminated string up to 510 characters long

Remarks: After importing the itpii package, these data types are injected into the __main__ namespace so these types are available from the python CLI without requiring the use of the 'itpii' prefix. However, in a self-contained python module, it is necessary to export the data type from the itpii package to make use of them in that python module. For example, the following code shows how to get access to the Ord1 data type when executed in a python module file that is imported into python.

Example

```
from itpii import Ord1
x = Ord1(42)
print(x + 1)
```

5 DAL CLI Command Reference

5.1 Enumerations

5.1.1 AddressType

This collection of labels is used with the Address class constructor to explicitly specify the type of the address being constructed.

Note
The AddressType collection is pushed into the __main__ namespace by the python CLI.

Note
These labels are collected in the AddressType enumeration that is exposed directly from .Net. As such, they must be referenced with the AddressType label, for example, AddressType.physical.

Table 5.1. Address Type enumerations

Label	Value	Description
implied	0	The address type is implied. If the address was specified as a string, the address type is embedded in the string (for example, "0x1000P" means 0x1000 is a physical address. "0x8:1000" means the address is a two field virtual address). The DAL will interpret the address when the address is used.
linear	1	A linear address. A linear address is converted to a physical or virtual address by walking the paging tables set up in the CPU. A linear address is thread-specific, since each thread has its own paging tables.
physical	2	A physical address. A physical address has the same meaning regardless of the thread being accessed.
guestphysical	3	A physical address in a virtual or guest environment.

Label	Value	Description
twofieldvirtual	4	A two field virtual address in the form <seg>:<offset>.
threefieldvirtual	5	A three field virtu- al address in the form <ldt>:<seg>:<offset>
onefieldcodevirtual	6	A two field virtual address where the offset is explicit but the <seg> is assumed to be the cs register.
onefielddatavirtual	7	A two field virtual address where the offset is explicit but the <seg> is assumed to be the ds register.
onefieldstackvirtual	8	A two field virtual address where the offset is explicit but the <seg> is assumed to be the ss register.

5.1.2 OBSPins

This collection of labels is used with the methods that affect the observation (or obs) pins. The labels can be OR'ed together to create a composite value. The value for each label is provided only as reference.

> **Note**
> These labels are collected in the OBSPins enumeration that is exposed directly from .Net. As such, they must be referenced with the OBSPins label, for example, OBSPins.obsfn_a0. To combine the values, use the '|' operator (IronPython only supports the '|'): OBSPins.obsfn_a0 | OBSPins.obsdata_a0.

Table 5.2. OBSPins enumerations

Label	Value	Description
obsfn_a0	0x1	OBS Bank A function bit 0
obsfn_a1	0x2	OBS Bank A function bit 1
obsdata_a0	0x4	OBS Bank A data bit 0
obsdata_a1	0x8	OBS Bank A data bit 1
obsdata_a2	0x10	OBS Bank A data bit 2
obsdata_a3	0x20	OBS Bank A data bit 3
obsfn_b0	0x40	OBS Bank B function bit 0
obsfn_b1	0x80	OBS Bank B function bit 1
obsdata_b0	0x100	OBS Bank B data bit 0
obsdata_b1	0x200	OBS Bank B data bit 1

Label	Value	Description
obsdata_b2	0x400	OBS Bank B data bit 2
obsdata_b3	0x800	OBS Bank B data bit 3
obsfn_c0	0x1000	OBS Bank C function bit 0
obsfn_c1	0x2000	OBS Bank C function bit 1
obsdata_c0	0x4000	OBS Bank C data bit 0
obsdata_c1	0x8000	OBS Bank C data bit 1
obsdata_c2	0x10000	OBS Bank C data bit 2
obsdata_c3	0x20000	OBS Bank C data bit 3
obsfn_d0	0x40000	OBS Bank D function bit 0
obsfn_d1	0x80000	OBS Bank D function bit 1
obsdata_d0	0x100000	OBS Bank D data bit 0
obsdata_d1	0x200000	OBS Bank D data bit 1
obsdata_d2	0x400000	OBS Bank D data bit 2
obsdata_d3	0x800000	OBS Bank D data bit 3

5.1.3 PinOperation

This collection of labels is used with the methods that affect the observation and BPM pins. The labels are used individually and cannot be combined.

> **Note**
> These labels are collected in the PinOperation enumeration that is exposed directly from .Net. As such, they must be referenced with the PinOperation label, for example, PinOperation._assert.

Table 5.3. PinOperation enumerations

Label	Description
_assert	The BPM pin(s) indicated by parameter pin will be set to "low".
deassert	The BPM pin(s) indicated by parameter pin will be set to "high".
strobe	The BPM pin(s) indicated by parameter pin will be set to "low" and then to "high" in one operation.

5.2 Command Definitions

5.2.1 Global CLI Commands

5.2.1.1 allclkdis

Description Disable all clocks (stop forcing all clocks to run) on the specified device.

Signature: allclkdis(device)

Arguments: device -- Device ID or alias of the device to affect. Can also be a Node object.

Returns: None.

Remarks: Warning: On some processors, this command issues a few extra scans to the Power Control Unit (PCU) to let the core clocks turn off during the next sleep state transition.

Usage:

```
>>> itp.allclkdis(0)
```

5.2.1.2 allclken

Description Enable all clocks (force all clocks to run) on the specified device.

Signature: allclken(device)

Arguments: device -- Device ID or alias of the device to affect. Can also be a Node object.

Returns: None.

Remarks: Warning: On some processors, this command issues a few extra scans to the Power Control Unit (PCU) to let the core clocks turn off during the next sleep state transition.

Usage:

```
>>> itp.allclken(0)
```

5.2.1.3 authorize

Description Authorize all the IA32 devices in the target system. Provides limited access to authorized features. Credentials will be asked for if they are required and if the device(s) is/are not authorized already. If no device is specified (device is None), then any device that would normally throw a "no permissions" exception to the user will instead send a warning message, allowing the command to run to completion, potentially authorizing remaining devices.

Signature: authorize()

Arguments: None

Returns: Zero if authorize was successful; otherwise returns non-zero.

Usage:

```
>>> itp.authorize()
>>> itp.authorize(0) # only authorize device 0
                       Related commands:
supportsauthorization
requirescredentials
entercredentials
clearcredentials
deauthorize
isauthorized
```

5.2.1.4 autoconfig

Description Displays or changes a setting that controls which events will trigger an automatic configuration of the device list.

Signature: autoconfig(debugPort, configType = None)

Arguments: debugPort -- Device node representing the debug port to access. Can also be an ID (an index into the debug ports). Set to None to retrieve or change the global autoconfig setting.

configType -- A numerical value, BitData, or a string containing one or more of the following values that alter what triggers an automatic (re)configuration of the device list:

"off" (0x0000) Disable automatic configuration. "invoke" (0x0001) Initialization of the DAL (IGNORED). "power" (0x0002) Power restore event. "resettarget" (0x0004) Target reset event. "resettap" (0x0008) "Reset tap" command is executed (IGNORED FOR NOW) "fuses" (0x0010) Fuses are changed (IGNORED). "allevents" (0xffff) All possible events.

If a configType is not recognized, a CommandsErrorException is raised.

The "invoke" option is controlled by the AutoconfigOnTreeAccess option in the TopoConfig.xml file. autoconfig() will always report the initial state of "invoke" but cannot change the state.

Returns: None.

Usage:

```
>>> itp.autoconfig(0)
>>> itp.autoconfig(0, "resettarget | power")
>>> itp.autoconfig(0, 0x0004 + 0x0008)
>>> savedConfig = itp.autoconfigraw(0)
>>> itp.autoconfig(0, savedConfig)
```

5.2.1.5 autoconfigraw

Description Retrieve the setting that controls which events will trigger an automatic configuration of the device list.

Signature: autoconfigraw(debugPort = None)

Arguments: debugPort -- Device node representing the debug port to access. Can also be an ID (an index into the debug ports). If None, the global autoconfig setting is returned.

Returns: Return a BitData object containining the current setting as a numerical value.

Usage:

```
>>? itp.autoconfigraw()
>>? itp.autoconfigraw(0)
```

5.2.1.6 autoscandr

Description Perform a combined IR/DR scan to the specified device, passing in the specified instruction for the IR scan and returning the appropriate amount of data by auto-discovering the length of the data to read. Also returns the quality of the scan chain.

Signature: autoscandr(device, instruction, maxBitLength = None)

Arguments: device -- Device ID or alias of the device. Can also be a Node object.

instruction -- The instruction to scan into the device (an 8-bit handle).

maxBitLength -- Specifies the maximum bit length to assume for the scan chain. This number of bits are scanned in and a sentinel handle is looked for. If maxBitLength is None, assume 32,768 bits.

PLEASE NOTE: If the TAP instruction you are testing is very large, it will be important to declare a maxBitLength that is longer than the TAP register being tested. The default of 32768 may not be large enough for some very long registers. Using a maxBitLength that is not large enough will return a value of "([0b] empty, u'OPEN')", even if the TAP register actually has continuity.

Returns: A tuple containing a BitData object which contain the bits that were read back and a string indicating the quality of the scan chain.

The continuity string can contain one or more of the following, delimited by commas:

GOOD (0)

OPEN (1)

SHORT (2)

INVERTED (4)

CORRUPTION (8)

Usage:

```
Example 1:
  >>> (returnData, continuity) = itp.autoscandr(0, 2)
  >>> returnData, continuity
Example 2:
  >>> (returnData, continuity) = itp.autoscandr(0, 2, 65536)
  >>> returnData, continuity
```

5.2.1.7 autoscandrraw

Description Perform a combined IR/DR scan to the specified device, passing in the specified instruction for the IR scan and returning the appropriate amount of data by auto-discovering the length of the data to read.

Signature: autoscandrraw(device, instruction, maxBitLength = None)

Arguments: device -- Device ID or alias of the device. Can also be a Node object.

instruction -- The instruction to scan into the device (an 8-bit handle).

maxBitLength -- Specifies the maximum bit length to assume for the scan chain. This number of bits are scanned in and a sentinel handle is looked for. If maxBitLength is None, assume 32,768 bits.

Returns: A BitData object containing the bits that were read back.

Usage:

```
Example 1:
  >>> returnData = itp.autoscandrraw(0, 2)
  >>> returnData
Example 2:
  >>> returnData = itp.autoscandrraw(0, 2, 65536)
  >>> returnData
```

5.2.1.8 br

Description View breakpoints, create breakpoints, or change an existing breakpoint. If the breakpoint does not exist, it is created (if possible). If an aspect cannot be changed, an error is thrown. If the address is None, then output a list of existing breakpoints (or the specified breakpoint, if the breakpointID is not None).

Signature: br(breakpointID = None, address = None, breakType = None, dbreg = None, enable = None)

Arguments: breakpointID -- ID of the breakpoint to change or create. Can be None, in which case a new ID is assigned. The BreakpointID can be a string or a number starting from 1. address -- Address to change to. This parameter is required. breakType -- A string containing the type of the break point to create. Set to None to leave this parameter alone. Can be one of the following: "sw" -- Software breakpoint "exe" -- Hardware execution breakpoint "exe global" -- Hardware execution breakpoint (Global is across all tasks) "exe local" -- Hardware execution breakpoint (Local is for current task) "acc" -- Hardware memory access breakpoint "acc global" -- Hardware memory access breakpoint (Global is across all tasks) "acc local" -- Hardware memory access breakpoint (Local is for current task)> "wr" -- Hardware memory write breakpoint "wr global" -- Hardware memory write breakpoint (Global is across all tasks) "wr local" -- Hardware memory write breakpoint (Local is for current task) "io" -- Hardware I/O access breakpoint dbreg -- Which debug register to use for the breakpoint information. Set to None to leave this parameter alone. enable -- True or false to enable or disable the breakpoint. Set to None to leave this parameter alone.

Returns: None.

Usage:

```
>>> itp.br(None, "0x1000L")
>>> itp.br(1, "0x1100L")
>>> itp.br(None, "0x110L", "io")
>>> itp.br(None, "0x1200L", "exe global")
>>> itp.br(None, "0x1300L", "wr global", dbreg = 3)
>>> itp.br(None, "0x1400L", "acc global", enable = False)
>>> itp.br("MyBrkpoint", "0x1500L", None, None, None)
```

5.2.1.9 brchange

Description Change one or more aspects of an existing breakpoint. If the breakpoint does exist, an error is thrown. If an aspect cannot be changed, an error is thrown.

Signature: brchange(breakpointID, address, enable = None, breakType = None, dbreg = None)

Arguments: breakpointID -- ID of the breakpoint to change. The BreakpointID can be a string or a number starting from 1. address -- Address to change to. This parameter is required. enable -- True or false to enable or disable the breakpoint. Set to None to leave this parameter alone. breakType -- A string containing the type of the break point to create. Set to None to leave this parameter alone. Can be one of the following: "sw" -- Software breakpoint "exe" -- Hardware execution breakpoint "exe global" -- Hardware execution breakpoint (Global is across all tasks) "exe local" -- Hardware execution breakpoint (Local is for current task) "acc" -- Hardware memory access breakpoint "acc global" -- Hardware memory access breakpoint (Global is across all tasks) "acc local" -- Hardware memory access breakpoint (Local is for current task)> "wr" -- Hardware memory write break-

point "wr global" -- Hardware memory write breakpoint (Global is across all tasks) "wr local" -- Hardware memory write breakpoint (Local is for current task) "io" -- Hardware I/O access breakpoint dbreg -- Which debug register to use for the breakpoint information. Set to None to leave this parameter alone.

Returns: None.

Usage:

```
>>> itp.brchange(1, "0x10000L")
>>> itp.brchange(1, "0x10000L", breakType = "wr global")
>>> itp.brchange(1, "0x10000L", enable = False)
>>> itp.brchange(1, "0x10000L", breakType = "exe local", dbreg = 3)
>>> itp.brchange("MyBrkpoint", "0x1250L", None, "wr global", None)
```

5.2.1.10 brdisable

Description Disable one or more breakpoints.

Signature: brdisable(breakpointID, *breakpointIDs)

Arguments: breakpointID -- ID of the first breakpoint to disable. The BreakpointID can be a string or a number starting from 1. breakpointIDs -- IDs of additional breakpoints to disable. The BreakpointID can be a string or a number starting from 1.

Returns: None.

Usage:

```
>>> itp.brdisable(1)          # disable breakpoint 1
>>> itp.brdisable(1, 4, 5, 6) # disable breakpoints 1, 4, 5, and 6
>>> itp.brdisable("brkpoint1")  # disable breakpoint "brkpoint1"
>>> itp.brdisable("BP1", "BP2", "BP3") #disable breakpoint "BP1","BP2","BP3"
```

5.2.1.11 breakdetectionmethod

Description Configures the DAL break detection method.

Signature: breakdetectionmethod(debugPort, value = None)

Arguments: debugPort -- ID, alias or debug port node for which to get status.

value -- The break detection method to use. Valid values are: Pin and Tap. if value is None, system will check and return current status.

Returns: The current setting of the break detection.

Remarks: In normal operations, PIN based break detection is used to identify silicon that has entered probe mode. TAP based break detection an alternative method that is used through polling when PIN based is not available.

Usage:

```
>>? itp.breakdetectionmethod(0)
>>? itp.breakdetectionmethod(0, "Pin")
>>? itp.breakdetectionmethod(0, "Tap")
```

5.2.1.12 brenable

Description Enable one or more breakpoints.

Signature: brenable(breakpointID, *breakpointIDs)

Arguments: breakpointID -- ID of the first breakpoint to enable. The BreakpointID can be a string or a number starting from 1. breakpointIDs -- IDs of additional breakpoints to enable. The BreakpointID can be a string or a number starting from 1.

Returns: None.

Usage:

```
>>> itp.brenable(1)          # enable breakpoint 1
>>> itp.brenable(1, 4, 5, 6) # enable breakpoints 1, 4, 5, and 6
>>> itp.brenable("brkpoint1")  # enable breakpoint "brkpoint1"
>>> itp.brenable("BP1", "BP2", "BP3") #enable breakpoint "BP1","BP2","BP3"
```

5.2.1.13 brget

Description Retrieve a single breakpoint or all breakpoints.

Signature: brget(breakpointID = None)

Arguments: breakpointID -- ID of the breakpoint to retrieve. The BreakpointID can be a string or a number starting from 1. If this is None, then retrieve all breakpoints. If this is a NodeThread object then retrieve all breakpoints on that thread.

Returns: A formatted string containing a list of all breakpoints. Can be empty if there are no breakpoints.

Usage:

```
>>> itp.brget(1)
>>> itp.brget("brkpoint1")    # display breakpoint "brkpoint1"
```

5.2.1.14 brnew

Description Create a new breakpoint across all threads, using one or more aspects, including initially disabled.

Signature: brnew(address, breakType = None, dbreg = None, enable = None)

Arguments: address -- Address to break at. Must be specified. This is a string. breakType -- A string containing the type of the break point to create. If None

then defaults to 'exe global' type. Can be one of the following: "sw" -- Software breakpoint "exe" -- Hardware execution breakpoint "exe global" -- Hardware execution breakpoint (Global is across all tasks) "exe local" -- Hardware execution breakpoint (Local is for current task) "acc" -- Hardware memory access breakpoint "acc global" -- Hardware memory access breakpoint (Global is across all tasks) "acc local" -- Hardware memory access breakpoint (Local is for current task) "wr" -- Hardware memory write breakpoint "wr global" -- Hardware memory write breakpoint (Global is across all tasks) "wr local" -- Hardware memory write breakpoint (Local is for current task) "io" -- Hardware I/O access breakpoint dbreg -- Which debug register to use for the breakpoint information (0 - 3). If None then no dbreg override is used. enable -- Initial enable state for the new breakpoint (True or False). Set to None defaults to True (enabled).

Returns: None.

Usage:

```
>>> itp.brnew("0x1000L")
>>> itp.brnew("0x100L", "io")
>>> itp.brnew("0x1200L", "exe global")
>>> itp.brnew("0x1200L", "exe global", dbreg = 3)
>>> itp.brnew("0x1200L", "exe global", dbreg = 3, enable = False)
>>> itp.brnew("0x1100L", None, None, None, 3)
>>> itp.threads[0].brnew("0x1100L", breakpointID = 3)
```

5.2.1.15 brremove

Description Remove some or all breakpoints.

Signature: brremove(*breakpointIDs)

Arguments: breakpointIDs -- IDs of zero or more breakpoints to enable. The BreakpointID can be a string or a number starting from 1. If no breakpoint IDs are specified, remove all breakpoints.

Returns: None.

Usage:

```
>>> itp.brremove() # remove all breakpoints
>>> itp.brremove(1, 3, 4) # remove breakpoints 1, 3, and 4
>>> itp.brremove("brkpoint1")  # remove breakpoint "brkpoint1"
>>> itp.brremove("BP1", "BP2", "BP3") #remove breakpoint "BP1","BP2","BP3"
```

5.2.1.16 c0hldst

Description Modify the C0HLDST data register on the specified device.

Signature: c0hldst(device, dataValue)

Arguments: device -- Device ID or alias of the device for which to get the tap port ID. Can also be a Node object.

dataValue -- 32-bit value (or BitData object containinga 32-bit value) to be shifted into the C0HLDST TAP data register. Refer to the processor specification document on the exact number of bits and bit positions of the data register

Returns: None.

Usage:

```
>>> itp.c0hldst(0, 0x1)
```

5.2.1.17 calibrate

Description Calibrate HSDP Interface for AET. If there is no progress made in the calibration for the last 30 s it will stop the calibration process

Signature: calibrate(device, portType = "obs")

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

portType -- Port to be calibrated. Can be obs, jtag, i2c

Returns: True, if Calibration is successful.

Usage:

```
>>> itp.calibrate(0)
>>> itp.calibrate(0, "obs")
>>> itp.calibrate(itp.threads[0])
>>> itp.calibrate(itp.domains[0].packages[0], "obs")
>>> itp.calibrate("IVT0_C0_T0", "obs")
```

5.2.1.18 cbotapconfig

Description Retrieve or change the value of the CBO_TAPCONFIG data register on the specified device. Note that device must be a core/thread, not Uncore device.

Signature: cbotapconfig(device, newValue = None)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object. Must be a core/thread device, not Uncore device.

newValue -- The 32-bit value (or BitData object containing the 32-bit value) to write to the register.

Returns: A BitData object containing the 32-bit value from the specified register.

Usage:

```
>>> itp.cbotapconfig()  (***DEPRECATED*** please use
```

```
 '<devicenode>.state.tap.<*cbo_tapconfig*>'
>>> itp.cbotapconfig(0, 0xefefefef)
```

5.2.1.19 cbotapstatus

Description Display the TAP status bits from the specified uncore device as a formatted string.

Signature: cbotapstatus(device)

Arguments: device -- Device ID or alias of the device with an associated uncore to access. Can also be a Node object.

Returns: None.

Usage:

```
>>> itp.cbotapstatus(1)
```

5.2.1.20 cbotapstatusraw

Description Retrieve the Cache Box (CBO) TAP status using the cbotapstatus TAP command.

Signature: cbotapstatusraw(device)

Arguments: device -- Device ID or alias of the device with an associated uncore to access. Can also be a Node object.

Returns: A BitData object containing the status bits.

Usage:

```
>>> itp.cbotapstatusraw(1)
```

5.2.1.21 clearcredentials

Description Clear the credentials associated with authorize command.

Signature: clearcredentials(user=None)

Arguments: user -- optional string to only remove certain user credentials

Returns: None

Usage:

```
>>> itp.clearcredentials()
                        Related commands:
supportsauthorization
requirescredentials
entercredentials
authorize
```

```
deauthorize
isauthorized
```

5.2.1.22 clockctrl

Description Read or change the CLOCKCTRL data register which controls the specified clock domain.

Signature: clockctrl(device, clockLocation = None, newValue = None)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

clockLocation -- A string specifying the location (system agent, core, cache) where the clock resides. If this value is None, defaults to the system agent (uncore) clock. Can be one of the following: "saclks" -- (0) System Agent (Uncore) "coreclks" -- (1) Processor Core "cboclks" -- (2) Processor Cache

newValue -- If not None, specifies a 32-bit value (or a BitData object containing a 32-bit value) to write to the register.

Returns: A BitData object containing a 32-bit value. Returns None if writing to the register (newValue is not None). See Also:

itp.uncores[n].clockctrl() itp.threads[n].clockctrl()

Usage:

```
>>> itp.clockctrl(0)
>>> itp.clockctrl(0, "saclks", 0x1)
>>> itp.clockctrl(itp.threads[0], "coreclks")
>>> itp.clockctrl(itp.threads[0], "coreclks", 0x1)
```

5.2.1.23 crb

Description Display/write a CPU core control register bus register using a 32-bit value across all cores.

Signature: crb(address, newValue = None)

Arguments: address -- The register number/address to access.

newValue -- If not None, specifies a 32-bit value to write to the register.

Returns: None.

Usage:

```
>>> itp.crb(0x4010)
>>> itp.crb(0x4010, 0x2110)
```

5.2.1.24 crb64

Description Display/write a CPU core control register bus register using a 64-bit value across all cores.

Signature: crb64(address, newValue = None)

Arguments: address -- The register number/address to access.

newValue -- If not None, specifies a 64-bit value to write to the register.

Returns: A 64-bit value from the control register bus register if reading. Returns None if writing to the register (newValue is not None).

Usage:

```
>>> itp.crb(0x4010)
>>> itp.crb(0x4010, 0x21102312341)
```

5.2.1.25 deauthorize

Description Deauthorize all the IA32 devices in the target system.

Signature: deauthorize(device=None)

Arguments: None

Returns: Zero if authorize was successful; otherwise returns non-zero.

Usage:

```
>>> itp.deauthorize()
                        Related commands:
supportsauthorization
requirescredentials
entercredentials
authorize
clearcredentials
isauthorized
```

5.2.1.26 debugport

Description Retrieve the debug port ID for the specified device.

Signature: debugport(device)

Arguments: device -- An ID, alias, or device node representing the device for which to get the ID of the associated debug port.

Returns: ID of the debug port.

Usage:

```
>>? itp.debugport(itp.threads[0]) # Get debug port for thread 0
```

5.2.1.27 display

Description Display the contents of the value associated with the specified state navigation language (SNL) query as queried against the specified device.

Signature: display(device, snlQuery)

Arguments: device -- ID or alias of the device to get status for. Can also be a Node object.

snlQuery -- The state navigation language (SNL) query.

Returns: None.

Remarks: Use this version of the display command to display the value of a State Navigation Language (SNL) query. Although the thread-specific display command (section 0) can do the same thing, this version of the display command can use any kind of device node, not just the NodeThread type.

Usage:

```
>>> itp.display(itp.threads[0], "eax")
>>> itp.display(0, "tap.pcodewakeupreq")
```

5.2.1.28 drive_obs

Description Drive the selected pin(s) in the OBS banks (in ITP hardware) to pulse, assert or de-assert.

Signature: drive_obs(action, debugPort, pins)

Arguments: action -- A value from the itp.PinOperation collection specifying the operation to apply to the pins.

debugPort -- Device node representing the debug port to access. Can also be an ID (an index into the debug ports).

pins -- One or more values from the itp.OBSPins collection specifying which pins to affect.

Note: In IronPython, the '+' is not supported by enumerations, so use the OR function: '|'.

Returns: None.

Remarks: The length of the strobe pulse is approximately 3 clocks when the ITP hardware is used to strobe the pin(s). Note: In IronPython, the '+' is not supported by enumerations, so use the OR function: '|' to combine the OBSPins flags.

Usage:

```
>>? itp.drive_obs(itp.PinOperation.strobe, 0, itp.OBSPins.obsfn_a1 |
 itp.OBSPins.obsdata_a1)
>>? itp.drive_obs(itp.PinOperation._assert, 0, itp.OBSPins.obsfn_b1)
```

5.2.1.29 drscan

Description Read the data registers of devices in the target boundary scan chain.

Signature: drscan(device, bitCount, readData = None, writeData = None)

Arguments: device -- Device ID or alias of the device. Can also be a Node object.

bitCount -- The number of bits to scan from the data register of the designated device as selected by the current instruction register handle.

readData -- List to be filled in with the scanned data read in. If this list is not empty, the type of the first element is examined to get an idea of the size of the data to return. If the type of the first element has the 'byteSize' attribute, that is used to determine how to fill the array. Otherwise, the data is always returned as a byte array.

writeData -- An optional array of integers or a BitData object containing the bits to be written to the device. The integer values are assumed to be packed bytes, where a 32-bit integer holds 4 bytes. Use the ITP data types for more precise control over the data in each element of the array.

Returns: A BitData object containing the bits that were read back. Can be ignored if the readData parameter is provided.

Remarks: If the writeValue is an array of values and the number and size of values are not sufficient to match the specified bitCount, a CommandsErrorException is raised.

If the writeData is an array of values and there are more values than needed for the specified bitCount, the remaining values are silently ignored.

WARNING: Manual scans mode should be enabled before using the drscan() command. When enabled, manual scans ensures the tap chain is left alone until manual scans are disabled. Use the itp.cv.manualscans control variable.

Usage:

```
Example 1:
  >>> returnData = []
  >>> itp.cv.manualscans = 1
  >>> itp.drscan(0, 32, returnData)
  >>> itp.cv.manualscans = 0
Example 2:
  >>> writeData = [Ord1(0)] * 8   # 8 zeroes
  >>> itp.cv.manualscans = 1
  >>> itp.drscan(0, 32, None, writeData)
  >>> itp.cv.manualscans = 0
Example 3:
  >>> itp.cv.manualscans = 1
  >>> bitData = itp.drscan(0, 32)
```

```
  >>> itp.cv.manualscans = 0
  >>> bitData
Example 4:
  >>> itp.cv.manualscans = 1
  >>> writeData = [Ord8()] * 3
  >>> writeData[0] = Ord8(0x0007F68AF825D240)
  >>> writeData[1] = Ord8(0x8484848484800000)
  >>> writeData[2] = Ord8(0x0000000000000004)
  >>> bitData = itp.drscan(0, 133, None, writeData)
  >>> itp.cv.manualscans = 0
```

5.2.1.30 entercredentials

Description Allow for secure user credential entry for authorization commands

Signature: entercredentials()

Arguments: None

Returns: True if credentials were successfully entered.

Usage:

```
>>> itp.entercredentials()
                          Related commands:
supportsauthorization
requirescredentials
authorize
clearcredentials
deauthorize
isauthorized
```

5.2.1.31 forcememoryscandelay

Description Configures the DAL to force a memory scan delay instead of triggered scans. If this is set to true, it will force memory scan delays for all memory access operations, instead of a triggered scan that would normally be done in order to verify the completion of the memory scan.

Signature: forcememoryscandelay(debugPort, enabled = None) if enabled is None, system will check and return current status.

Arguments: debugPort -- ID, alias or debug port node for which to get status.

enabled -- If memory scans should be forced to have a delay instead of triggered scans.

Returns: If enabled is None, checks and returns if the memory scan delays are forced on the specified debug port.

Usage:

```
>>? itp.forcememoryscandelay(0)
>>? itp.forcememoryscandelay(0, 1)
>>? itp.forcememoryscandelay(0, 0)
```

5.2.1.32 forcereconfig

Description Perform an immediate check for any change to the devicelist. If a change is detected in the devicelist, the new device list is reconfigured and the change reported; otherwise, nothing happens.

Signature: forcereconfig(*args)

Arguments: filename -- filename of the xml file defining the new tap configuration.

debugPort -- an integer representing a debug port ID, a device node (of or derived from Node), or the debug port node itself (of type NodeDebug-Port).

scanChain -- an integer representing a scan chain ID. This integer should only be a 0 or a 1 because there are only two scan chains per debug port.

device -- first entry of tap info tuple, device type string

stepping -- second entry of tap info tuple, stepping string

irLen -- optional third entry of tap info tuple, an integer to specify IR length.

Returns: None.

Remarks: forcereconfig():

no argument, normal case of forcing a reconfiguration.

forcereconfig(filename):

one argument which specifies a XML file defining the new tap configuration.

forcereconfig(debugPort, scanChain,(device, stepping),(device, stepping, irLen), ...):

force a reconfiguration for new tap configuration on a specified JTAG chain each following tuple must specify tap info including device, stepping and optional irLen.

forcereconfig(debugPort, scanChain, devList):

force a reconfiguration on a specified JTAG chain (specified by debugPort and scanChain), devList must be a list of tuples where each tuple must specify tap info as described above.

Usage:

```
>>? itp.forcereconfig()
>>? itp.forcereconfig(xmlFile)
>>? itp.forcereconfig(0, 0, ('HSW', 'C0'))
>>? itp.forcereconfig(0, 1, ('HSW', 'C0', 8), ('HSW', 'C0'))
```

```
>>? devlist = [('HSW', 'C0'), ('HSW', 'C0', 8)]
>>? itp.forcereconfig(0, 0, devlist)
```

5.2.1.33 fuseinfo

Description Get information on the fuses for a specified device.

Signature: fuseinfo(device)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

Returns: Dictionary whose keys are the fuse block names, and whose values are lists of fuse names. If a particular device does not support fuses, the dictionary will be empty.

Usage:

```
>>> info = itp.fuseinfo(0)
>>> info.keys()
['PhysicalByRow', 'Physical']
>>> info = itp.fuseinfo(0)
>>> for blockName in info.keys():
...     print("%s %d" % (blockName, len(info[blockName])))
...
PhysicalByRow 192
Physical 1250
>>>
```

Remarks: Use the fuseinfo command to obtain a data structure which can then be programmatically or manually traversed and whose contents can be used in the other fuse commands.

5.2.1.34 fuseoverride

Description Override (write) the logical fuse values to the specified block and device.

Signature: fuseoverride(device, blockName, fuseNameList, fuseDataList)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

blockName -- A string of the fuse block to access.

fuseNameList -- A list of strings of the fuses to access.

fuseDataList -- A list of BitData objects which correspond to the names specified in the fuseNameList.

Returns: None.

Usage:

```
>>> fuseNames = ["row0"]
>>> fuseData = [BitData("[64b]0")]
```

```
>>> itp.fuseoverride(0, "PhysicalByRow", fuseNames, fuseData)
```

Remarks: Use the fuseoverride command to temporarily change the physical fuse cell values that a device contains. Be aware that not all devices will support this type of access.

5.2.1.35 fusepostprogram

Description Post-Program (write) the physical fuse values to the specified block and

Signature: fusepostprogram(device, blockName, fuseNameList, fuseDataList)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

blockName -- A string of the fuse block to access.

fuseNameList -- A list of strings of the fuses to access.

fuseDataList -- A list of BitData objects which correspond to the names specified in the fuseNameList.

Returns: None.

Usage:

```
>>> fuseNames = ["row0"]
>>> fuseData = [BitData("[64b]0")]
>>> itp.fusepreprogram(0, "PhysicalByRow", fuseNames, fuseData)
```

Remarks: Use the fusepostprogram command after calling fuseprogram for supported devices.

5.2.1.36 fusepreprogram

Description Pre-Program (write) the physical fuse values to the specified block and

Signature: fusepreprogram(device, blockName, fuseNameList, fuseDataList)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

blockName -- A string of the fuse block to access.

fuseNameList -- A list of strings of the fuses to access.

fuseDataList -- A list of BitData objects which correspond to the names specified in the fuseNameList.

Returns: None.

Usage:

```
>>> fuseNames = ["row0"]
```

```
>>> fuseData = [BitData("[64b]0")]
>>> itp.fusepreprogram(0, "PhysicalByRow", fuseNames, fuseData)
```

Remarks: Use the fusepreprogram command before calling fuseprogram for supported devices.

5.2.1.37 fuseprogram

Description Program (write) the physical fuse values to the specified block and device.

Signature: fuseprogram(device, blockName, fuseNameList, fuseDataList)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

blockName -- A string of the fuse block to access.

fuseNameList -- A list of strings of the fuses to access.

fuseDataList -- A list of BitData objects which correspond to the names specified in the fuseNameList.

Returns: None.

Usage:

```
>>> fuseNames = ["row0"]
>>> fuseData = [BitData("[64b]0")]
>>> itp.fuseprogram(0, "PhysicalByRow", fuseNames, fuseData)
```

Remarks: Use the fuseprogram command to permanently change the physical fuse cell values that a device contains. Be aware that not all devices will support this type of access. Also, the fuseprogram is considered "single-pass", meaning: it is up to the user to manually run multiple times, at different platform voltages (if that is what is required).

5.2.1.38 fuseread

Description Read the physical fuse values from the specified block and device.

Signature: fuseread(device, blockName, fuseNameList)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

blockName -- A string of the fuse block to access.

fuseNameList -- A list of strings of the fuses to access.

Returns: List of BitData types, corresponding to the order of the fuseNameList passed in.

Usage:

```
>>> itp.fuseread(0, "PhysicalByRow", ["row0"])
[[64b] 0x0000000000000000]
>>>
```

Remarks: Use the fuseread command to access the physical fuse cell values that a device contains. Be aware that not all devices will support this type of access.

5.2.1.39 fusereadoverride

Description Read the logical fuse values from the specified block and device

Signature: fusereadoverride(device, blockName, fuseNameList)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

blockName -- A string of the fuse block to access.

fuseNameList -- A list of strings of the fuses to access.

Returns: List of BitData types, corresponding to the order of the fuseNameList passed in.

Usage:

```
>>> itp.fusereadoverride(0, "PhysicalByRow", ["row0"])
[[64b] 0x0000000000000000]
>>>
```

Remarks: Use the fusereadoverride command to access the logical fuse values that a device contains. Be aware that not all devices will support this type of access.

5.2.1.40 getDeviceState

Description Return the device state for the specified device.

Signature: getDeviceState(device)

Arguments: device -- device ID, alias, or node.

Returns: Return an IStateNav object containing the device state.

Usage:

```
>>> itp.getDeviceState(0)
```

5.2.1.41 get_obs

Description Retrieve the current status of all OBS/MBP/BPM pins for the specified debug port. It is important to note that by reading a pin's status, you will

clear its toggled flag, therefore this function will clear all of the toggled flags. Also, the initial state of the occurred pins might have stale data. It is recommended that you call this function to clear the occurred states before trying to check for useful data. This function is used to display whether pins were toggled since the last call of this function.

Signature: get_obs(debugPort)

Arguments: debugPort -- Device node representing the debug port to access. Can also be an ID (an index into the debug ports).

Returns: BitData containing he state of all OBS pins in a 64-bit value. (Bits 63-32 = Obs occurred, Bits 31-0 = Obs status.)

Usage:

```
>>? itp.display_obs(0)      # display (and clear) pin status for debug port
 0
>>? status = itp.get_obs(0) # get status for debug port 0
>>? status                  # display as 64-bit value
```

5.2.1.42 go

Description Resume execution on all processors.

Signature: go(address = None, goTilType = None, *args)

Arguments: address -- address to break on, if specified. Otherwise, processors are just resumed.

goTilType -- value from GoTilType enumeration specifying the type of break. Default is 'exe'. If goTilType is given, then the address to be specified.

*args -- additional address/GoTilType pairs with GoTilType being optional.

Returns: None.

Remarks: Use the go command to start a processor executing. There are two fundamental operations of the go command, go and go "til addr". The go command begins execution at the current execution point using all enabled breakpoints.

The go "til addr" command begins execution at the current execution point temporarily setting aside any existing hardware breakpoint specifications and using the "addr" address or addresses as temporary hardware breakpoints.

Any enabled software breakpoints remain in effect. When a break in execution occurs for any reason, the "til addr" breakpoints are removed and previously existing enabled hardware breakpoints are reinstated.

The go "til addr" command, like the go command, starts and stops one or all processors depending on the setting of the breakall control variable. Using a specific thread causes the "til addr" breakpoints to be set using the hardware resources of the specified thread. However, the global go command (itp.go()) causes the specified breakpoints to be set in the debug registers of each non-running processor defined in the boundary scan chain.

The number of breakpoints that can be specified in a go "til addr" command is limited by the number of breakpoint registers implemented in the processor. See the br() command (section 5.4) for more information on breakpoints. The breakall control variable determines whether the go command applies to a single processor, all processors, or all processors connected to a single debug port on targets with multiple debug ports.

If the go command is applied to a processor that is already running, a warning will be displayed. An error will be displayed if too many breakpoints are specified on a go "til addr".

Usage:

```
>>> itp.go()
>>> itp.go("0x100")
>>> itp.go("0x100", itp.GoTilType.exe)
>>> itp.go("0x100", itp.GoTilType.exe, "0x80", itp.GoTilType.io)
>>> itp.go("0x100", itp.GoTilType.exe, "0x200")
>>> itp.go("0x100", "0x200")
```

5.2.1.43 halt

Description Halt execution on all processors.

Signature: halt()

Arguments: None.

Returns: None.

Remarks: The halt command stops all processors. The breakall control variable determines whether the halt command applies to a single processor or all processors.

If the halt command is issued but the processor is in a mode from which it cannot be halted, an error message is returned indicating the reason -- if a reason can be determined. If the reason cannot be determined, a generic

error message will be displayed.

The go command is used to instruct processors to resume execution.

Usage:

```
>>> itp.halt()
```

5.2.1.44 holdhook

Description Hold one of the eight ITP hook pins in the specified state.

Signature: holdhook(debugPort, hook, assertHook)

Arguments: debugPort -- ID, alias or debug port node for which to affect. If this is None, then assert/deassert the hook across all debug ports.

hook -- A value of 0 through 7 that specifies a hook to access.

assertHook -- True (non-zero) means sets the hook LOW; False (zero) means set the hook HIGH.

Returns: None.

In a typical configuration, hook 0 is the power-good signal and hook 6 is the reset occurred signal. These are considered "input" pins and generally shouldn't be asserted/deasserted. However, not all hook configurations are the same so the holdhook() command allows any hook value. The user is expected to understand the debug port's hook configuration before using this command.

NOTE: The HOOK pins are active-low signals. This command operates in terms of the _logical_ value, not the _electrical_ value. Passing in 1 or True means a logical assertion, which is electrically LOW. Passing in a 0 or False means a logical non-assertion, which means the signal will float to whatever state (HIGH or LOW) the target naturally outputs.

Usage:

```
>>? itp.holdhook(None, 1, True) # set hook 1 LOW on all debug ports
>>? itp.holdhook(0, 1, False) # set hook 1 HIGH on debug port 0
>>? itp.holdhook(0, 1, 1) # set hook 1 LOW on all debug ports
>>? itp.holdhook(0, 1, 0) # set hook 1 HIGH on all debug ports
```

5.2.1.45 hookstatus

Description Read the status of any of the 8 ITP hook pins.

Signature: hookstatus(debugPort, hook)

Arguments: debugPort -- ID, alias or debug port node for which to affect.

hook -- A value of 0 through 7 that specifies a hook to access.

Returns: The HOOK pins are active-low signals. This command returns the _logical_ value of the hook, not the _electrical_ value of the hook.

True - if the hook logically asserted (electrically LOW). False - if the hook is logically deasserted (electrically HIGH).

Usage:

```
>>? itp.hookstatus(0, 0) # get hook 0 status from debug port 0
```

5.2.1.46 hwstatus

Description Display detailed status information about the ITP hardware represented by the specified debug port.

Signature: hwstatus(debugPort)

Arguments: debugPort -- ID, alias or debug port node for which to get status.

Returns: None.

Usage:

```
>>> itp.hwstatus(0) # status of debug port 0
```

5.2.1.47 i2cscan

Description Perform an I2C scan.

Signature: i2cscan(debugPortNumber, addressData, writeData, readLength=0, hiSpeed=False)

Arguments: debugPortNumber -- The debug port to use the I2C master on.

addressData -- The 7- or 10-bit BitData for the I2C slave address.

writeData -- The BitData object containing the bits to write; BitSize should be a factor of 8; set to None if no bits are to be written.

readLength -- The number of bits to read; should be 0 or factor of 8

hiSpeed -- Boolean switch for high speed mode.

Returns: None (if readLength 0) or BitData containing the read data

Usage:

```
>>? for x in range(0x12, 0xee, 2):
...     if x in [0x18, 0x50, 0x6E, 0xC2]: continue
...     try: itp.i2cscan(0, BitData(7,x), BitData(8,0x00))
...     except: pass
...     else: print hex(x)
...
0x2e
0x3e
0x44
0x88
0x8c
0xae
0xbe
0xc4
```

5.2.1.48 idcode

Description Return the processor boundary scan idcode for the specified device.

Signature: idcode(device)

Arguments: device -- device ID, alias, or node.

Returns: Return the idcode as a string.

Remarks: Use the idcode command to display the processor boundary scan idcode for a device. The returned value is always in hexadecimal regardless of the setting of the base control variable.

Note: The idcode is fetched from the hardware every time this method is called. A cached version of the idcode can be obtained for threads from the device attribute on the NodeThread class (itp.threads[0].device.idcode) or from the device list for any device (itp.devicelist[0].idcode).

Usage:

```
>>> itp.idcode(0)
```

5.2.1.49 ignorepowergood

Description Retrieve or change the setting to ignore powergood to allow scans through XDP.

Signature: ignorepowergood(debugPort, doIgnore = None)

Arguments: debugPort -- ID, alias or debug port node for which to get status.

doIgnore -- True if to ignore power good; otherwise False.

Returns: If doIgnore is None, return the current setting.

Remarks: Specifically, this command tells the XDP hardware pod to ignore the state of the power good signal.

Usage:

```
>>> itp.ignorepowergood(0)
>>> itp.ignorepowergood(0, False)
```

5.2.1.50 ignoreresetdetection

Description Retrieve or change the setting to ignore reset events in the ITP hardware. Specifically, this command tells the XDP hardware pod to ignore the reset events from the target.

Signature: ignoreresetdetection(debugPort, doIgnore = None)

Arguments: debugPort -- ID, alias or debug port node for which to get status.
doIgnore -- True if to ignore reset events; otherwise False.

Returns: If doIgnore is None, return the current setting.

Usage:

```
>>> itp.ignoreresetdetection(0)
>>> itp.ignoreresetdetection(0, False)
```

5.2.1.51 interestingdevices

Description Get/Set a list of interesting devices. These devices will dictate how features within the DAL operate. One such example is dictating which break messages should appear when a halt is executed. The default interestingdevices value is impacted by the global control variable savestateondemand. When savestateondemand is False, the default interestingdevices is set to “all”. When savestateondemand is True, the default interestingdevices is set to “P0” representing thread 0 alias. If interestingdevices is set to a value, the default behavior will be disabled unless setting interestingdevices to "default".

Signature: interestingdevices(*args)

Arguments: args -- variable arguments, depending how arguments are passed in, the command will operate accordingly. The following cases are supported: 1. interestingdevices(): no argument, Gets the current configuration. 2. interestingdevices(None): Dictates no devices are interesting. 3. interestingdevices("none"): Dictates no devices are interesting. 4. interestingdevices(""): Dictates no devices are interesting. 5. interestingdevices("all"): Dictates all devices are interesting. 6. interestingdevices(-1): Dictates all devices are interesting. 7. interestingdevices("default"): Resets interestingdevices to initial default behavior that switches between "all" and "P0" when control variable savestateondemand changes. 7. interestingdevices(Device, ...): One or more ID or alias of the device to configure. Can also be a LNode object. Invalid values are acceptable because not all devices are known when the part is locked. Invalid values will not cause anything to happen when the interesting devices are taken into account.

Returns: Current configuration if nothing is passed in.

Usage:

```
>>? itp.interestingdevices()
>>? itp.interestingdevices(0)
>>? itp.interestingdevices(-1)
>>? itp.interestingdevices(None)
>>? itp.interestingdevices("")
>>? itp.interestingdevices("all")
>>? itp.interestingdevices("P0", itp.threads[0], foobar)
```

5.2.1.52 irdrscan

Description Perform a combined IR/DR scan to the specified device, passing in the specified instruction for the IR scan and the specified bit count for the DR scan.

Signature: irdrscan(device, instruction, bitCount, readData = None, writeData = None)

Arguments: device -- Device ID or alias of the device. Can also be a Node object.

instruction -- The instruction to scan into the device (an 8-bit handle).

bitCount -- The number of bits to scan from the data register of the designated device as selected by the current instruction register handle.

readData -- List to be filled in with the scanned data read in. If this list is not empty, the type of the first element is examined to get an idea of the size of the data to return. If the type of the first element has the 'byteSize' attribute, that is used to determine how to fill the array. Otherwise, the data is always returned as a byte array.

writeData -- An optional array of integer values or a BitData object containing the bits to be written to the device. The integer values are assumed to be packed bytes, where a 32-bit integer holds 4 bytes. Use the ITP data types for more precise control over the data in each element of the array.

Returns: A BitData object containing the bits that were read back. Can be ignored if the readData parameter is provided.

Remarks: This method combines an IR scan with a DR scan so as to eliminate any possibility of something else interfering with the scan.

The bitCount parameter determines the number of bits that will be scanned. The writeData value must contain at least as many bits as are to be scanned.

If the writeData is an array of values and the number and size of values are not sufficient to match the specified bitCount, a CommandsErrorException is raised.

If the writeData is an array of values and there are more values than needed for the specified bitCount, the remaining values are silently ignored.

Usage:

```
Example 1:
  >>> returnData = []
  >>> itp.irdrscan(0, 2, 32, returnData)
  >>> returnData
Example 2:
  >>> writeData = [Ord1(0xff)] * 8 # eight 0xff bytes
```

```
  >>> bitData = itp.irdrscan(0, 2, 32, None, writeData)
  >>> bitData
Example 3:
  >>> bitData = itp.irdrscan(0, 2, 32)
  >>> bitData
Example 4:
  >>> writeData = [Ord8()] * 3
  >>> writeData[0] = Ord8(0x0007F68AF825D240)
  >>> writeData[1] = Ord8(0x8484848484800000)
  >>> writeData[2] = Ord8(0x0000000000000004)
  >>> bitData = itp.irdrscan(0, 0x7c, 133, None, writeData)
```

5.2.1.53 irdrscanauto

Description Perform a combined IR/DR scan to the specified device, passing in the specified instruction for the IR scan and the specified bit count for the DR scan.

Signature: irdrscanauto(device, instruction, writeData = None)

Arguments: device -- Device ID or alias of the device. Can also be a Node object.

instruction -- The instruction to scan into the device (an 8-bit handle).

writeData -- An optional array of integer values or a BitData object containing the bits to be written to the device. The integer values are assumed to be packed bytes, where a 32-bit integer holds 4 bytes. Use the ITP data types for more precise control over the data in each element of the array.

Returns: A BitData object containing the bits that were read back.

Remarks: If the writeData is an array of values and the number and size of values are not sufficient to match the specified bitCount, ValueError is raised.

If the writeData is an array of values and there are more values than needed for the specified bitCount, the remaining values are silently ignored.

Usage:

```
Example 1:
  >>> bitData = itp.irdrscanauto(0, 2)
  >>> bitData
Example 2:
  >>> writeData = [Ord1(0xff)] * 8 # eight 0xff bytes
  >>> bitData = itp.irdrscanauto(0, 2, writeData)
  >>> bitData
```

5.2.1.54 irdrscanreplace

Description Perform a combined IR/DR scan to the specified device, passing in the specified instruction for the IR scan and the specified bit count for the DR scan.

Signature: irdrscanreplace(device, instruction, bitCount, writeData = None)

Arguments: device -- Device ID or alias of the device. Can also be a Node object.

instruction -- The instruction to scan into the device (an 8-bit handle).

bitCount -- The number of bits to scan from the data register of the designated device as selected by the current instruction register handle.

writeData -- An optional array of integer values or a BitData object containing the bits to be written to the device. The integer values are assumed to be packed bytes, where a 32-bit integer holds 4 bytes. Use the ITP data types for more precise control over the data in each element of the array.

Returns: A BitData object containing the bits that were read back.

Remarks: If the writeData is an array of values and the number and size of values are not sufficient to match the specified bitCount, ValueError is raised.

If the writeData is an array of values and there are more values than needed for the specified bitCount, the remaining values are silently ignored.

Usage:

```
Example 1:
  >>> bitData = itp.irdrscanreplace(0, 2, 32)
  >>> bitData
Example 2:
  >>> writeData = [Ord1(0xff)] * 8 # eight 0xff bytes
  >>> bitData = itp.irdrscanreplace(0, 2, 32, writeData)
  >>> bitData
```

5.2.1.55 irdrscanrmw

Description Perform a combined IR/DR scan to the specified device, which reads the original contents of a DR, modifies it with a masked value, and then writes the updated value back into the DR.

Signature: irdrscanrmw(device, instruction, writeData, maskData)

Arguments: device -- Device ID or alias of the device. Can also be a Node object.

instruction -- The instruction to scan into the device (an 8-bit handle).

writeData -- A BitData reflecting a masked value to be written into a DR.

maskData -- A BitData reflecting the masked of bits to modify in the original value scanned out.

Remarks: The value written into the DR amounts to (value_read mask) | (writeData mask).

Usage:

```
Example 1:
  >>> itp.irdrscanrmw(0, 2, BitData(32, 0x00000004), BitData(32,
   0x0000000F))
```

5.2.1.56 irdrscanverify

Description Perform a combined IR/DR scan to the specified device, passing in the specified instruction for the IR scan and the specified bit count for the DR scan.

Signature: irdrscanverify(device, instruction, bitCount, writeData = None)

Arguments: device -- Device ID or alias of the device. Can also be a Node object.

instruction -- The instruction to scan into the device (an 8-bit handle).

bitCount -- The number of bits to scan from the data register of the designated device as selected by the current instruction register handle.

writeData -- An optional array of integer values or a BitData object containing the bits to be written to the device. The integer values are assumed to be packed bytes, where a 32-bit integer holds 4 bytes. Use the ITP data types for more precise control over the data in each element of the array.

Returns: A BitData object containing the bits that were read back.

Remarks: If the writeData is an array of values and the number and size of values are not sufficient to match the specified bitCount, ValueError is raised.

If the writeData is an array of values and there are more values than needed for the specified bitCount, the remaining values are silently ignored.

Usage:

```
Example 1:
  >>> bitData = itp.irdrscanverify(0, 2, 32)
  >>> bitData
Example 2:
  >>> writeData = [Ord1(0xff)] * 8 # eight 0xff bytes
  >>> bitData = itp.irdrscanverify(0, 2, 32, writeData)
  >>> bitData
```

5.2.1.57 irscan

Description Write the designated instruction handle into the instruction register of the specified device.

Signature: irscan(device, instruction)

Arguments: device -- Device ID or alias of the device. Can also be a Node object.

instruction -- The instruction to scan into the device.

Returns: None.

Remarks: Use the irscan command with the drscan command to read from or write to the data register of a device on the target system boundary scan chain. The irscan command writes the designated instruction value into the instruction register of the specified device.

Note: Perform an irscan followed as soon as possible by a drscan to avoid interference between the two steps.

Warning: Manual scans mode should be enabled before using the irscan() command, especially if a drscan() command is to follow. See the manualscans control variable for more details.

Usage:

```
>>> itp.cv.manualscans = 1
>>> itp.irscan(0, 2)
>>> itp.cv.manualscans = 0
```

5.2.1.58 isauthorized

Description Check if the device is authorized, or all devices able to be authorized are authorized.

Signature: isauthorized(device=None)

Arguments: None

Returns: True if device is authorized; False if not

Usage:

```
>>> itp.isauthorized()
                        Related commands:
supportsauthorization
requirescredentials
entercredentials
authorize
clearcredentials
deauthorize
```

5.2.1.59 islocked

Description Check if the device is locked, or any devices able to be unlocked are locked.

Signature: islocked(device=None, level="red")

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

level -- Specifies the lock type as a string.

Returns: True if device is locked; False if not

Usage:

```
>>> itp.islocked()
True
>>> itp.unlock()
>>> itp.islocked()
False
>>> itp.islocked(0)
False
>>> itp.islocked(0, "orange")
True
>>> itp.islocked(0, "red")
False
```

5.2.1.60 isunlocked

Description Check if the device is unlocked, or all devices able to be unlocked are

Signature: isunlocked(device=None, level="red")

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

level -- Specifies the lock type as a string.

Returns: True if device is unlocked; False if not

Usage:

```
>>> itp.isunlocked()
False
>>> itp.unlock()
>>> itp.isunlocked()
True
>>> itp.isunlocked(0)
True
>>> itp.isunlocked(0, "orange")
False
>>> itp.isunlocked(0, "red")
True
```

5.2.1.61 jtagonlymode

Description Configures the DAL to run using JTAG only commands.

Signature: jtagonlymode(debugPort, enabled = None) if enabled is None, system will check and return current status.

Arguments: debugPort -- ID, alias or debug port node for which to get status. enabled -- Optional to configure system to JTAG only mode or not.

Returns: If enabled is None, checks and returns the status of JTAG Only mode.

Remarks: When running using JTAG Only Mode enabled, the PREQ, PRDY, DBR and RESET pins are considered off, and PowerGood is considered on. We also enable TAP based break detection, and and start to poll for probe mode entry. Triggered scans are disabled and memory scan delays are put into place.

Usage:

```
>>? itp.jtagonlymode(0)
>>? itp.jtagonlymode(0, 0)
>>? itp.jtagonlymode(0, 1)
```

5.2.1.62 jtagonlymodestatus

Description Displays the configuration of the DAL to see if it is running using JTAG only commands, and outputs any remaining items that need to be configured.

Signature: jtagonlymodestatus(debugPort)

Arguments: debugPort -- ID, alias or debug port node for which to get status.

Returns: None.

Usage:

```
>>? itp.jtagonlymodestatus(0)
```

5.2.1.63 load

Description Load an object-module file into the target system memory and load the debugging symbols into the host memory across all processor threads. Set the instruction pointer on the thread to point to the entry of the file.

Signature: load(filename, operation = None, at = None, offset = None, raw = None, loadkey = None)

Arguments: filename -- Name and path of the file to load.

operation -- One or more values from the itp.LoadOperations collection modifying the load behavior. The following options are possible:

itp.LoadOperations.nocode Do not load the code, only the symbols. Cannot be combined with any other operation.

itp.LoadOperations.nosymbol Do not load debug symbols. Cannot be combined with 'nocode'.

itp.LoadOperations.allregs Load the registers of all processors during the load, even if the load targets once processor.

itp.LoadOperations.nopsp Do not create a Program Segment Prefix (PSP) during the load. Valid only if loading a Microsoft binary that already has a PSP.

at -- A string or Address object that specifies the address in the target system to which to load the file.

offset -- A string or integer that specifies the offset, in bytes, to the OMF's relocatable addresses where a file's code, data, and symbolics will be loaded in memory.

raw -- True if to load the file without debug symbols and without performing any relocation.

loadkey -- String used to make unique each symbol load of each file so the files can be loaded cumulatively (normally each load removes any previous symbols).

Returns: None.

Usage:

```
>>? itp.load("myfile.exe")
>>? itp.load("myfile.exe", itp.LoadOperations.nocode)
>>? itp.load("myfile.exe", itp.LoadOperations.nosymbols +
 itp.LoadOperations.allregs)
>>? itp.load("myfile.exe", at = "0x300P")
>>? itp.load("myfile.exe", itp.LoadOperations.nocode, offset = 0x300)
>>? itp.load("myfile.exe", at = "0x40200L", loadkey = "TEST1")
>>? itp.load("myfile.bin", at = "0x300P", raw = True)
                          See Also:
- itp.threads[0].load()
- itp.threads[0].memload()
```

5.2.1.64 lock

Description Lock devices in the target system.

Signature: lock(device=None)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object. If None, then all devices are locked.

Returns: None

Usage:

```
>>> itp.lock()
>>> itp.lock(0)
```

5.2.1.65 lrtapconfig

Description Retrieve or change the value of the LR_TAPCONFIG data register on the specified device. Note that device must be an uncore, not a core/thread device.

Signature: lrtapconfig(device, newValue = None)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object. Must be an uncore, not a core/thread device.

newValue -- The 32-bit value (or BitData object containing the 32-bit value) to write to the register.

Returns: A BitData object containing the 32-bit value from the specified register. See Also:

itp.uncores[n].lrtapconfig()

Usage:

```
>>> itp.lrtapconfig(0)
>>> itp.lrtapconfig(0, 0xefefefef)
```

5.2.1.66 memoryscandelay

Description Configures the memory scan delay for the DAL when running in JTAG only mode.

Signature: memoryscandelay(debugPort, delay = None)

Arguments: debugPort -- ID, alias or debug port node for which to get status.

delay -- The delay in milliseconds for memory scan access.

Returns: If delay is None, checks and returns the current value of the memory scan delay.

Remarks: This is used for memory access commands that are not using a triggered scan in order to provide the silicon enough time to access the memory and return prior to continuing with other operations.

Usage:

```
>>? itp.memoryscandelay(0)
>>? itp.memoryscandelay(0, 100)
```

5.2.1.67 msgclear

Description Stop and clear the message in the ITP hardware.

Signature: msgclear(msgHandle)

Arguments: msgHandle -- Handle returned from msgopen().

Returns: True if the transaction successfully cleared; otherwise, returns False.

Usage:

```
>>> itp.msgclear(msgHandle)
```

5.2.1.68 msgclose

Description Terminate the construction of the message.

Signature: msgclose(msgHandle)

Arguments: msgHandle -- Handle returned from msgopen().

Returns: True if the scan successfully closed; otherwise, returns False.

Remarks: Once the message is successfully closed, no further transactions can be added.

Usage:

```
>>> itp.msgclose(msgHandle)
```

5.2.1.69 msgdata

Description Retrieve the return data of a message previously scanned to the target

Signature: msgdata(msgHandle, returnData)

Arguments: msgHandle -- Handle returned from msgopen().

returnData -- An instance of a list through which the data is returned. If this list is not empty, the type of the first element is examined to get an idea of the size of the data to return. If the type of the first element has the 'byteSize' attribute, that is used to determine how to fill the array. Otherwise, the data is always returned as a byte array.

Returns: True if the data was successfully retrieved; otherwise, returns False.

Remarks: The values returned in the returnData parameter contain the bits of data that were returned by the last scan.

Usage:

```
>>> data = None
>>> returnData = [int(),] * 8
>>> if msgdata(msgHandle, returnData):
>>>     print(returnData)
```

5.2.1.70 msgdelete

Description Delete the message and releases the message handle.

Signature: msgdelete(msgHandle)

Arguments: msgHandle -- Handle returned from msgopen(). After this call, the handle is no longer valid.

Returns: True if the scan successfully deleted; otherwise, returns False.

Remarks: Whether the scan was successfully deleted or not, the handle is no longer valid.

Usage:

```
>>> itp.msgdelete(msgHandle)
```

5.2.1.71 msgdr

Description Record a DR scan to the message.

Signature: msgdr(msgHandle, bitCount, writeValue = None, readwrite = None, scanChain = None, scanTrigger = None, stopState = None, broadcastMode = None)

Arguments: msgHandle -- Handle returned from msgopen().

bitCount -- Number of bits to write out as the DR scan.

writeValue -- A single 32-bit handle, an array of integer values, or a BitData object containing the bits to write. If None, then all zeroes are scanned and all subsequent parameters are ignored.

The integer values are assumed to be packed bytes, where a 32-bit integer holds 4 bytes. Use the ITP data types for more precise control over the data in each element of the array.

readwrite -- 0 = write-only, 1 = readwrite, 2 = read-only. If None, assumes write-only.

scanChain -- Specify which scan chain to use. The default is 0.

scanTrigger -- A handle from 0 to 7 specifying the scan trigger to use. If None, assumes 0 (trigger immediately).

stopState -- The TAP state in which to stop at the end of the scan. The default is 0 (RTI).

broadcastMode -- Specify whether to repeat the same data on all JTAG devices (The default is true (non-zero)): - True (non-zero) indicates the data is repeated for all JTAG devices specified in device-array. The writeValue contains bitCount bits of data to write to each device. - False (0) indicates unique data is specified for each JTAG device. The writeValue must contain data for each device. The writeValue must contain at least (bitCount * number-of-devices) bits.

Returns: True if the transaction successfully added; otherwise, returns False.

Remarks: If the writeValue is an array of values and the number and size of values are not sufficient to match the specified bitCount, a CommandsErrorException is raised.

If the writeValue is an array of values and there are more values than needed for the specified bitCount, the remaining values are silently ignored.

Usage:

```
>>> data = itp.msgdr(msgHandle, 32)
>>> data = itp.msgdr(msgHandle, 32, (0xff, 0xff, 0xff, 0xff), 1)
```

```
>>> data = itp.msgdr(msgHandle, 32, BitData(32, 0xffffffff), 1)
>>> data = itp.msgdr(msgHandle, 32, BitData(32, 0xffffffff), 1, 0)
>>> data = itp.msgdr(msgHandle, 32, BitData(32, 0xffffffff), 1, 0, 1)
>>> data = itp.msgdr(msgHandle, 32, BitData(32, 0xffffffff), 1, 0, 1, 0)
>>> data = itp.msgdr(msgHandle, 32, BitData(32, 0xffffffff), 0, 0, 0, True)
```

5.2.1.72 msgir

Description Record an IR scan to the message.

Signature: msgir(msgHandle, writeValue, bitCount = None, readwrite = None, scanChain = None, scanTrigger = None, stopState = None, broadcastMode = None)

Arguments: msgHandle -- Handle provided by msgopen() or msgload().

bitCount -- Number of bits to write out as the IR scan. Can be None.

writeValue -- A single 32-bit handle, an array of 32-bit values, or a BitData object containing the bits to write.

readwrite -- 0 = write-only, 1 = readwrite, 2 = read-only. If None, assumes write-only.

scanChain -- Specify which scan chain to use. The default is 0.

scanTrigger -- A handle from 0 to 7 specifying the scan trigger to use. If None, assumes 0 (trigger immediately).

stopState -- The TAP state in which to stop at the end of the scan. The default is 0 (RTI).

broadcastMode -- Specify whether to repeat the same data on all JTAG devices (The default is true (non-zero)):

- True (non-zero) indicates the data is repeated for all JTAG devices specified in device-array. The writeValue contains bitCount bits of data to write to each device.

- False (0) indicates unique data is specified for each JTAG device. The writeValue must contain data for each device. The writeValue must contain at least (bitCount * number-of-devices) bits.

Returns: True if the transaction successfully added; otherwise, returns False.

Remarks: Use the msgir command to add instruction register (IR) scans to the open message. The command returns an error if the message has been closed. If the list of devices specified by msgopen spans more than one scan chain, a scan will be added for each scan chain.

Usage:

```
>>> itp.msgir(msgHandle, 2)
```

```
>>> itp.msgir(msgHandle, (0x00, 0x02))
>>> itp.msgir(msgHandle, BitData(8, 2))
>>> itp.msgir(msgHandle, 2, 1)
>>> itp.msgir(msgHandle, 2, 1, 0)
>>> itp.msgir(msgHandle, 2, 1, 0, 0)
>>> itp.msgir(msgHandle, 2, 1, 0, 0, 1)
>>> itp.msgir(msgHandle, 2, 1, 0, 0, 1, 0)
>>> itp.msgir(msgHandle, 2, 1, 0, 0, 1, 0, True)
```

5.2.1.73 msgload

Description Load a previously recorded message from a binary file into a new message targeted for one or more multiple devices. File must be composed entirely of JTAG scans.

Signature: msgload(msgHandle, filename, devices)

Arguments: msghandle -- An instance of Handle(). The handle created by msgload() is stored in this class instance. filename -- Path to the file to load. devices -- A single device ID or an array of IDs of devices that are to receive the scan.

Returns: True if the transaction successfully loaded; otherwise, returns False.

Usage:

```
>>> msgHandle = itpii.Handle()
>>> if itp.msgload(msgHandle, "filename.bin", 0):
>>>    data = []
>>>    if (itp.msgscan(msgHandle, data)):
>>>        data
>>> msgHandle = itpii.Handle()
>>> if itp.msgload(itp.msgload(msgHandle, "filename.bin", (0, 1, 2, 3)):
>>>    ...
```

5.2.1.74 msgopen

Description Create a new message for one or more devices.

Signature: msgopen(msgHandle, devices)

Arguments: msgHandle -- An instance of itpii.Handle(). The handle created by msgopen() is stored in this class instance.

devices -- A single device or an array of devices that are to receive the scan. Each device is either an ID, alias, or a device node. If this is None, the devices must be specified in the msgscan() commands.

Returns: True if the scan was successfully opened; otherwise, returns False.

Remarks: Use the msgopen command to create a message identified by msgHandle. The specified msgHandle must exist or an error will be reported. If the msgHandle points to a message that is already open even if it has been closed, an error will be reported.

Usage:

```
>>> msgHandle = itpii.Handle()
>>> if itp.msgopen(msgHandle, itp.threads[0]):
>>>     itp.msgir(msgHandle, 2)
>>>     itp.msgdr(msgHandle, 32)
>>>     itp.msgclose(msgHandle)
>>>     data = []
>>>     itp.msgscan(msgHandle, data)
>>>     itp.msgdelete(msgHandle)
>>>     msgHandle = None
```

5.2.1.75 msgreturndatasize

Description Retrieve the size (in bytes) of the return data that the message will

Signature: msgreturndatasize(msgHandle, returnData)

Arguments: msgHandle -- Handle provided by msgopen() or msgload().

returnData -- A list through with the data is returned. The first entry in the list is the returned size.

devices -- A single device ID, alias or node or a list of device IDs, aliases or nodes. If this is None, then use the devices specified in the msgopen() command.

Returns: True if the data size was successfully retrieved; otherwise, returns False.

Remarks: The handle returned is size in bytes of the data expected to be returned by the message scan.

Use the msgreturndatasize command to find out the size of the return data in number of bytes. The command returns an error if called before the message has been closed.

Usage:

```
>>> dataSize = None
>>> returnData = []
>>> if itp.msgreturndatasize(msgHandle, returnData):
>>>     dataSize = returnData[0]
>>> print("Data size = %s"%dataSize)
```

5.2.1.76 msgsave

Description Save the message into the binary file. Message must be composed

Signature: msgsave(msgHandle, filename)

Arguments: msgHandle -- Handle provided by msgopen() or msgload(). filename -- Path to the file to save to. The file will be overwritten if it exists.

Returns: True if the transaction successfully added; otherwise, returns False.

Usage:

```
>>> itp.msgsave(msgHandle, "filename.bin")
```

5.2.1.77 msgscan

Description Send a closed message to a specified ITP hardware and scans it to all

Signature: msgscan(msgHandle, data = None, device = None)

Arguments: msgHandle -- Handle provided by msgopen() or msgload().

data -- An instance of a list for return data

If this parameter is a list and the list is not empty, the type of the first element is examined to get an idea of the size of the data to return. If the type of the first element has the 'byteSize' attribute, that is used to determine how to fill the array. Otherwise, the data is always returned as a byte array.

device -- a single device ID or array of device IDs or a single device object or array of device objects to scan to. Note: If devices were specified in the msgopen() function, the devices do not need to be specified here (and are ignored).

Returns: True if the data was successfully retrieved; otherwise, returns False.

Remarks: The values returned in triggerOrData contain the bits of data that were returned by the last scan. Use the msgscan command to send a message to the ITP hardware. The scans in this message will be scanned to all the devices specified in the msgopen command during which all the other devices in the scan chain will be put in bypass mode (for more information about the JTAG standard, refer to IEEE 1149.2 standard).

Note: The message is automatically closed if not previously closed when the msgscan command is called. This behavior is different from ITP, which would return an error if the message was not closed.

Note: This command will always wait for the scan to complete before returning. This is different behavior than ITP, which would return immediately if no return data was requested (the Commands Library implementation always returns a BitData object, even if it is empty).

Usage:

```
Example 1:
>>> returnData = [int(0),] * 1
>>> if itp.msgscan(msgHandle, returnData):
>>>     print("Data = %s"%returnData)
```

5.2.1.78 msgtimedelay

Description Record a scan that causes the hardware to pause for a given amount of

Signature: msgtimedelay(msgHandle, timedelay, scanTrigger = None)

Arguments: msgHandle -- Handle provided by msgopen() or msgload().

timedelay -- Number of nanoseconds to delay.

scanTrigger -- A handle from 0 to 7 specifying the scan trigger to use.

Returns: True if the transaction successfully added; otherwise, returns False.

Remarks: The time value has a resolution of 10 nanoseconds (internally, the value is divided by 10 before using).

Usage:

```
>>> itp.msgtimedelay(msgHandle, 500)
>>> itp.msgtimedelay(msgHandle, 55, 0)
```

5.2.1.79 msr

Description Return or change a value across all processors' model-specific register

Signature: msr(registerIndex, newValue = None)

Arguments: registerIndex -- A valid register address (an index).

Returns: None.

Remarks: This command provides access to a processor's model-specific registers (MSR) via register index instead of register name. The parameter is the register's architectural address found in the debug handbook for the processor under test.

The msr command utilizes the "mov" and "readmsr" instructions submitted to the Probe Instruction Register (PIR) and requires the use of valid indices as well.

Note: MSR access requires the target processor to be halted.

Usage:

```
>>> itp.msr(186)                # view the msr 186 from all threads
>>> itp.msr(186, 0xff)          # change the msr 186 across all threads
```

5.2.1.80 pcudata

Description Read/write a value from a pcudata bus register for an uncore device. If the specified device is a NodeThread or NodeCore, the associated NodeUncore device is used. All other device types raise an exception.

Signature: pcudata(device, register, newValue = None)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

register -- A valid control bus register number.

newValue -- If not None, specifies a 32-bit value (or a BitData object containing a 32-bit value) to write to the register.

Returns: A BitData object containing a 32-bit value. Returns None if writing to the register (newValue is not None). See Also:

itp.uncores[n].pcudata()

Usage:

```
>>> itp.pcudata(0, 0x100)
>>> itp.pcudata(0, 0x100, 0x1234)
```

5.2.1.81 perfreport

Description Run a performance check in the DAL, and print the results.

Signature: perfreport()

Arguments: None.

Returns: A report to help diagnose performance problems with the setup of the host and target system.

Remarks: Items marked with "[WARNING]" could have a performance impact on the system and should be considered for possible remediation.

Usage:

```
Example 1:
  >>> itp.perfreport()
  CLI In-Process        : False
  Logging               : disabled
  DAL Process Bitsize   : 64
  DAL Version           : 1.9.3120.400
  DAL Build Type        : Release
  Processor             : Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
  Processor Max Clock   : 3.40GHz
  Host Thread Count     : 8
  Operating System      : Microsoft Windows 7 Enterprise 64-bit
  Host Name             : VTGDP-SW31K3
  Free Physical Memory  : 10.6GB
  Total Physical Memory : 15.9GB
  Processor Utilization : 4.2%
  Disk Utilization      : 0.0%
  Mem Write Throughput  : 5.76GB/s
  Mem Read Throughput   : 7.7GB/s
  Target Connection     : Intel XDP3-USB
  Driver Version        : 4294967295
  JTAG TCLK Rate        : 4.17MHz
  Minscan Write Latency : 905us
  Minscan Read Latency  : 923us
  JTAG Write Throughput : 4.03Mtcks/sec
  JTAG Read Throughput  : 2.9Mtcks/sec [WARNING]
```

5.2.1.82 pins

Description Display the current electrical-level values of the target processor pins.

Signature: pins(device, packageName = None)

Arguments: device -- ID or alias of the device to get status for. Can also be a Node object.

packageName -- Optional name of a TAP package.

Returns: None.

Remarks: Use the pins command to display the current electrical-level values of the specified device's processor pins. That is to say, if a pin is electrically low, the pin value is displayed as 0, if a pin is electrically high, the pin value is displayed as 1.

Warning: On newer Intel processors, accessing the pins can disrupt communications with other devices; the target may have to be reset after accessing the pins.

This command does not work on Sandy Bridge devices.

Usage:

```
>>? itp.pins()
```

5.2.1.83 pinsnotwired

Description Configures which pins on a debug port are not wired.

Signature: pinsnotwired(debugPort, pinsNotWired = None) if pinsNotWired is None, system will check and return current status.

Arguments: debugPort -- ID, alias or debug port node for which to get status.

pinsNotWired -- The pins that are not wired. Multiples can be added with pipes or commas. - Valid values are: Prdy, Preq, Dbr, Reset, PwrGood - Specify "None" to indicate all known pins are wired. Specify "All" to indicate all known pins are not wired. "None" and "All" must be specified by themselves.

Returns: The pins that are not wired.

Usage:

```
>>? itp.pinsnotwired(0)
>>? itp.pinsnotwired(0, "prdy, preq")
>>? itp.pinsnotwired(0, "prdy | preq")
```

5.2.1.84 polling

Description Retrieve or change the state of polling of target system events on one or all debug ports.

Signature: polling(debugPort = None, enable = None)

Arguments: debugPort -- ID, alias or debug port node for which to affect. If this is None, then the polling status of all debug ports is read/written.

enable -- True if to enable polling (the default condition) or False to disable polling (the DAL does not respond to events). If this is None, then the current polling state is returned.

Returns: True if polling is enabled; otherwise False if polling is disabled. None is returned if changing the polling state.

Remarks: Although the DAL does not actually "poll" for events, the DAL can be told not t o respond to target events. By default, "polling" is enabled, that is, the DAL responds to target events such as target reset and power loss/restore.

If "polling"is disabled, the DAL no longer responds to such events but the events are queued up so when "polling" is enabled, the events are responded to.

Usage:

```
>>> if itp.polling(): print("Polling enabled")
>>> else: print("Polling disabled")
>>> itp.polling(1) # is polling enabled on debug port 1
>>> itp.polling(None, False) # disable polling on all debug ports
>>> itp.polling(1, True) # enable polling on debug port 1
>>> itp.polling(enable=True) # enable polling on all debug ports
```

5.2.1.85 pollinginterval

Description Configures the polling interval for the DAL when running in JTAG only mode.

Signature: pollinginterval(debugPort, interval = None)

Arguments: debugPort -- ID, alias or debug port node for which to get status.

interval -- The interval in milliseconds for tap based polling in run control.

Returns: The currently configured polling interval.

Remarks: When operating with TAP based break detection, this is the interval of how often to check the TAP to see if we have entered probe mode.

If interval is None, system will check and return the currently configured polling interval.

Usage:

```
>>? itp.pollinginterval(0)
>>? itp.pollinginterval(0, 100)
```

5.2.1.86 probemode

Description Enter or exit probe mode via the PROBEMODE TAP instruction on the specified multi-core processor device.

Signature: probemode(device, dataValue)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

dataValue -- 32-bit value (or BitData object containing a 32-bit value) to be shifted into the PROBEMODE TAP data register. Each bit set indicates the corresponding core on the device should enter probe mode; otherwise, if the bit is clear then the core will exit probe mode.

Returns: None. See Also:

itp.uncores[n].probemode() itp.threads[n].probemode()

Usage:

```
>>> # Enter probe mode on cores 0 and 1 on processor associated with thread
 0
>>> # associated with uncore 1.
>>> itp.probemode(1, 0x3)
>>> # Enter probe mode on core 0 and exit probe mode on core 1 associated
>>> # with uncore 1.
>>> itp.probemode(1, 0x1)
>>> # Enter probe mode on all cores associated with uncore 0.
>>> itp.probemode(0, -1)
>>> uncore = itp.domains.getByType("uncore")
>>> itp.probemode(uncore, -1)
```

5.2.1.87 pulsehook

Description Pulse any of the 8 ITP hook pins that are not input pins.

Signature: pulsehook(debugPort, hook, assertHook, count)

Arguments: debugPort -- ID, alias or debug port node for which to affect. If this is None, then assert/deassert the hook across all debug ports.

hook -- A value in the range 1, 2, 3, 4, 5, and 7 that specifies the output hook to hold.

assertHook -- True (non-zero) if to assert the hook; False (zero) if to deassert.

count -- An unsigned integer value that specifies the number of 10ns increments to pulse the hook

Returns: None.

Remarks: The pulsehook command asserts or deasserts any of the 8 ITP hook pins that are not input pins on one or all debug ports for a specified length of time. Note that if a particular pin is already asserted/deasserted, then this will result in a wait of the specified length, then transition to the deasserted/asserted state.

Usage:

```
>>? itp.pulsehook(0, 1, True, 16700000) # assert hook 1 on debug port 0
```

5.2.1.88 pulsepwrgood

Description Cycle power on the target.

Signature: pulsepwrgood(debugPort=None, timeOut=10000, sleep=6000, count=16700000, hookPin=1)

Arguments: debugPort -- ID, alias or debug port node for which to affect. If this is none then the debug port of the first uncore will used. If the port can not be autodetermined and exception will be thrown.

timeOut -- Amount of time to wait for the target to power down before aborting. Units (milliseconds).

sleep -- Amount of time to wait after power loss, before attempting to power target system back on. Units (milliseconds)

count -- The pulse width of the pin assertion used to power the target system back. Units (10 nano-seconds)

hookPin -- The HOOK Pin to use as power control for the platform (ie the power button)

Returns: None.

Remarks: This is the equivalent of holding the power button down for a moment then releasing. This command is only supported on target systems that route pwrgood# to one or more XDP debug ports. It is the user's responsibility to determine if a platform is configured to support this feature.

Usage:

```
>>? itp.pulsepwrgood()  # auto-determine debug port
>>? itp.pulsepwrgood(0) # debug port 0
>>? itp.pulsepwrgood(0, 10000, 6000, 16700000, 1) # debug port 0, timeOut
 10000, sleep 6000, count16700000, hookPin 1
```

5.2.1.89 rawdrscan

Description Read or write to the data registers of devices in the target boundary scan

Signature: rawdrscan(debugPort, scanChain, bitCount, readData = None, writeData = None)

Arguments: debugPort -- Device node representing the debug port to access. Can also be a zero-based ID (an index into the debug ports).

scanChain -- The zero based number of the scan chain on the debug port to send the scan to. This number should only be a 0 or a 1 because there are only two scan chains per debug port.

bitCount -- The number of bits to scan from the data register of the designated device as selected by the current instruction register handle.

readData -- List to be filled in with the scanned data read in. If this list is not empty, the type of the first element is examined to get an idea of the size of the data to return. If the type of the first element has the 'byteSize' attribute, that is used to determine how to fill the array. Otherwise, the data is always returned as a byte array.

writeData -- An optional array of integers or a BitData object containing the bits to be written to the device. The integer values are assumed to be packed bytes, where a 32-bit integer holds 4 bytes. Use the ITP data types for more precise control over the data in each element of the array.

Returns: A BitData object containing the bits that were read back. Can be ignored if the readData parameter is provided.

Remarks: If the writeValue is an array of values and the number and size of values are not sufficient to match the specified bitCount, a ValueError is raised.

If the writeData is an array of values and there are more values than needed for the specified bitCount, the remaining values are silently ignored.

WARNING: Manual scans mode should be enabled before using the rawdrscan() command. When enabled, manual scans ensures the tap chain is left alone until manual scans are disabled. Use the itp.cv.manualscans control variable.

Usage:

```
Example 1:
  >>> returnData = []
  >>> itp.cv.manualscans = 1
  >>> itp.rawdrscan(0, 0, 32, returnData)
  >>> itp.cv.manualscans = 0
Example 2:
  >>> writeData = [Ord1(0)] * 8   # 8 zeroes
  >>> itp.cv.manualscans = 1
  >>> itp.rawdrscan(0, 0, 32, None, writeData)
```

```
  >>> itp.cv.manualscans = 0
Example 3:
  >>> itp.cv.manualscans = 1
  >>> bitData = itp.rawdrscan(0, 0, 32)
  >>> itp.cv.manualscans = 0
  >>> bitData
Example 4:
  >>> itp.cv.manualscans = 1
  >>> writeData = [Ord8()] * 3
  >>> writeData[0] = Ord8(0x0007F68AF825D240)
  >>> writeData[1] = Ord8(0x8484848484800000)
  >>> writeData[2] = Ord8(0x0000000000000004)
  >>> bitData = itp.rawdrscan(0, 0, 133, None, writeData)
  >>> itp.cv.manualscans = 0
```

5.2.1.90 rawirdrscan

Description Write to the instruction registers of the devices on the specified JTAG scan write and read the data registers that the instruction register write enabled on the scan chain.

Signature: rawirdrscan(debugPort, scanChain, irBitCount, irWriteData, drBitCount, drWriteData = None, readData = None)

Arguments: debugPort -- Device node representing the debug port to access. Can also be a zero-based ID (an index into the debug ports).

scanChain -- The zero based number of the scan chain on the debug port to send the scan to. This number should only be a 0 or a 1 because there are only two scan chains per debug port.

irBitCount -- The number of bits to scan from the instruction registers on the selected JTAG scan chain.

irWriteData -- An array of integers or a BitData object containing the bits to be written to the instruction registers on the scan chain. The integer values are assumed to be packed bytes, where a 32-bit integer holds 4 bytes. Use the ITP data types for more precise control over the data in each element of the array.

drBitCount -- The number of bits to scan from the data registers on the selected JTAG scan chain.

drWriteData -- An optional array of integers or a BitData object containing the bits to be written to the data registers on the scan chain. The integer values are assumed to be packed bytes, where a 32-bit integer holds 4 bytes. Use the ITP data types for more precise control over the data in each element of the array.

readData -- List to be filled in with the scanned data read in. If this list is not empty, the type of the first element is examined to get an idea of the size of the data to return. If the type of the first element has the 'byteSize'

attribute, that is used to determine how to fill the array. Otherwise, the data is always returned as a byte array.

Returns: A BitData object containing the bits that were read back. Can be ignored if the readData parameter is provided.

Remarks: If the writeValue is an array of values and the number and size of values are not sufficient to match the specified bitCount, a ValueError is raised.

If the writeData is an array of values and there are more values than needed for the specified bitCount, the remaining values are silently ignored.

WARNING: Manual scans mode should be enabled before using the rawirdrscan() command. When enabled, manual scans ensures the tap chain is left alone until manual scans are disabled. Use the itp.cv.manualscans control variable.

Usage:

```
Example 1:
  >>> readData = []
  >>> itp.cv.manualscans = 1
  >>> irWriteData = [2, 0]
  >>> itp.rawirdrscan(0, 0, 9, irWriteData, 32, None, readData)
  >>> itp.cv.manualscans = 0
Example 2:
  >>> writeData = [Ord1(0)] * 8   # 8 zeroes
  >>> itp.cv.manualscans = 1
  >>> irWriteData = [2, 0]
  >>> itp.rawirdrscan(0, 0, 9, irWriteData, 32, writeData, None)
  >>> itp.cv.manualscans = 0
Example 3:
  >>> itp.cv.manualscans = 1
  >>> irWriteData = [2, 0]
  >>> bitData = itp.rawirdrscan(0, 0, 9, irWriteData, 32 )
  >>> itp.cv.manualscans = 0
  >>> bitData
Example 4:
  >>> itp.cv.manualscans = 1
  >>> irWriteData = [2, 0]
  >>> writeData = [Ord8()] * 3
  >>> readData = []
  >>> writeData[0] = Ord8(0x0007F68AF825D240)
  >>> writeData[1] = Ord8(0x8484848484800000)
  >>> writeData[2] = Ord8(0x0000000000000004)
  >>> bitData = itp.rawirdrscan(0, 0, 9, irWriteData, 32, writeData,
   readData)
  >>> itp.cv.manualscans = 0
```

5.2.1.91 rawirscan

Description Read or write to the instruction registers of devices in the target boundary

Signature: rawirscan(debugPort, scanChain, bitCount, readData = None, writeData = None)

Arguments: debugPort -- Device node representing the debug port to access. Can also be a zero-based ID (an index into the debug ports).

scanChain -- The zero based number of the scan chain on the debug port to send the scan to. This number should only be a 0 or a 1 because there are only two scan chains per debug port.

bitCount -- The number of bits to scan from the instruction registers on the selected JTAG scan chain.

readData -- List to be filled in with the scanned data read in. If this list is not empty, the type of the first element is examined to get an idea of the size of the data to return. If the type of the first element has the 'byteSize' attribute, that is used to determine how to fill the array. Otherwise, the data is always returned as a byte array.

writeData -- An optional array of integers or a BitData object containing the bits to be written to the device. The integer values are assumed to be packed bytes, where a 32-bit integer holds 4 bytes. Use the ITP data types for more precise control over the data in each element of the array.

Returns: A BitData object containing the bits that were read back. Can be ignored if the readData parameter is provided.

Remarks: If the writeValue is an array of values and the number and size of values are not sufficient to match the specified bitCount, a ValueError is raised.

If the writeData is an array of values and there are more values than needed for the specified bitCount, the remaining values are silently ignored.

WARNING: Manual scans mode should be enabled before using the rawirscan() command. When enabled, manual scans ensures the tap chain is left alone until manual scans are disabled. Use the itp.cv.manualscans control variable.

Usage:

```
Example 1:
  >>> returnData = []
  >>> itp.cv.manualscans = 1
  >>> itp.rawirscan(0, 0, 32, returnData)
  >>> itp.cv.manualscans = 0
Example 2:
  >>> writeData = [Ord1(0)] * 8   # 8 zeroes
  >>> itp.cv.manualscans = 1
  >>> itp.rawirscan(0, 0, 32, None, writeData)
  >>> itp.cv.manualscans = 0
Example 3:
  >>> itp.cv.manualscans = 1
  >>> bitData = itp.rawirscan(0, 0, 32)
  >>> itp.cv.manualscans = 0
  >>> bitData
Example 4:
  >>> itp.cv.manualscans = 1
```

```
>>> writeData = [Ord8()] * 3
>>> writeData[0] = Ord8(0x0007F68AF825D240)
>>> writeData[1] = Ord8(0x8484848484800000)
>>> writeData[2] = Ord8(0x0000000000000004)
>>> bitData = itp.rawirscan(0, 0, 133, None, writeData)
>>> itp.cv.manualscans = 0
```

5.2.1.92 requirescredentials

Description Determine if credentials are required for authorization commands

Signature: requirescredentials()

Arguments: None

Returns: True if credentials are required. Note this is dependent on the authorization state of the device(s) being operated on. That is, if the device(s) are already authorized, then False will be returned.

Usage:

```
>>> itp.requirescredentials()
                        Related commands:
supportsauthorization
entercredentials
authorize
clearcredentials
deauthorize
isauthorized
```

5.2.1.93 reset

Description Reset specified target system functions, on one debug port. If no debug port is specified then will default to first available port.

Signature: reset(command, debugPort = None)

Arguments: command -- One of the following strings:

"target" - Reset the target and the target processor.

"tap" - Reset the target's Test Access Port by asserting/ deasserting the TRST signal.

debugPort -- Optional debug port to reset.

Remarks: Use the reset command to reset the target system via the DBR# signal on the debug port or to reset the boundary scan TAP controller via the TRST# and TMS signals.

Use the optional debug port parameter on Multiple Debug Port (MDP) configurations to specify on which debug port assert the DBR# signal. If the debug port is not specified on a MDP configuration, then the DBR# signal will be asserted on all the debug ports.

Additionally, use the optional debug port parameter on Multiple Debug Port (MDP) configurations to specify on which debug port assert the TRST# signal. If the debug port is not specified on a MDP configuration, then the TRST# signal will be asserted on all the debug ports.

To reset the Python CLI to its invocation state, exit the software and re-invoke the Python CLI. Doing so does not change the state of the target system.

Usage:

```
>>? itp.reset("target")
>>? itp.reset("tap")
>>? itp.reset("target", 0)
>>? itp.reset("tap", 0)
```

5.2.1.94 resettap

Description Reset the tap of the specified debug port or all debug ports.

Signature: resettap(debugPort = None) This is a short form for itp.reset("tap", debugPort)

Arguments: debugPort -- Optional debug port to reset (default is all debug ports).

Returns: None.

Usage:

```
>>? itp.resettap()
>>? itp.resettap(0) # reset debug port 0 taps
```

5.2.1.95 resettarget

Description Reset the target system.

Signature: resettarget(debugPort = None) This is a short form for itp.reset("target")

Arguments: debugPort -- Optional debug port to reset.

Returns: None.

Remarks: If there are multiple debug ports connected to the DAL from a single target, the resettarget() command will raise an error unless a debug port is explicitly specified.

if no debug port is specified hen only the first port will be reset.

This is a short form for itp.reset("target")

Usage:

```
>>? itp.resettarget()
>>? itp.resettarget(0)
```

5.2.1.96 status

Description Display the status of ITP, the target system, and the specified device.

Signature: status(device, packageName = None)

Arguments: device -- Optional ID or alias of the device to get status for. Can also be a Node object. If device is None, show status for all devices.

packageName -- Optional name of a TAP package.

Returns: None.

Remarks: Use the status command to display the state of all devices on the target system or just the specified device. Critical pin states and running status are displayed. This command can be entered at any time. Optionally, the status command with the DID parameter can be used to display status information for a particular device id.

Usage:

```
>>? itp.status(0)
>>? itp.status(itp.threads[0], "package1")
```

5.2.1.97 supportsauthorization

Description Determine if authorization is supported

Signature: supportsauthorization(device=None)

Arguments: None

Returns: True if authorization is supported.

Usage:

```
>>> itp.supportsauthorization()
True
>>> itp.supportsauthorization(0) # does device ID 0 support authorization?
True
                        Related commands:
requirescredentials
entercredentials
authorize
clearcredentials
deauthorize
isauthorized
```

5.2.1.98 tapport

Description Retrieve the TAP port ID for either a processor or a device.

Signature: tapport(device)

Arguments: device -- Device ID or alias of the device for which to get the tap port ID. Can also be a Node object.

Returns: Return TAP port ID of the specified device as a number.

Remarks: Use the tapport command to display the ID of the TAP port that a device is associated with. A TAP port is set of pins on a physical device. Every device in a scan chain has 1 TAP port; multi-threaded devices share a TAP port. If an invalid device number is entered, an error message is displayed to the screen.

Usage:

```
>>? itp.tapport(0)
>>? itp.tapport(itp.threads[0])
>>? itp.tapport('SNB_C0_T1')
```

5.2.1.99 tapresetidcodes

Description Display the idcodes of all the devices connected to debug port(s).

Signature: tapresetidcodes()

Arguments: None.

Returns: None.

Remarks: It is not uncommon for the number of TAPs displayed by this command to be less than what appears in the device tree due to how the scan chain may be configured between scans.

Usage:

```
>>? itp.tapresetidcodes()
```

5.2.1.100 tapresetidcodescount

Description Display the number of idcodes of all the devices connected to debug port(s).

Signature: tapresetidcodescount()

Arguments: None.

Returns: None.

Remarks: It is not uncommon for the number of TAPs displayed by this command to be less than what appears in the device tree due to how the scan chain may be configured between scans.

Usage:

```
>>? itp.tapresetidcodescount()
```

5.2.1.101 tapstatus

Description Display the TAP status bits from internal TAP state and from various units sent to the TAP associated with this thread.

Signature: tapstatus(device)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

Returns: None.

Remarks: To get the raw bits, use the tapstatusraw() function.

The display shows the logical bit indices if there is more than one bit or the first bit is not 0. The physical mapping into the parent bits is always shown. The following example shows the tapstatus register with two fields. The PMRDY field has an implied logical mapping of [0:0] and a physical mapping to bit 88 in the tapstatus register. The PhySlice field has an explicit logical mapping of [7:0] and a physical mapping to bits 34 through 27.

tapstatus(109:0) = [109:0] 0x00000000000000878000000000000 PMRDY = [88] 0x0 PhySlice(7:0) = [34:27] 0x00

Usage:

```
>>> itp.tapstatus(0)
```

5.2.1.102 tapstatusraw

Description Retrieve the TAP status bits from internal TAP state and from various units sent to the TAP associated with this thread.

Signature: tapstatusraw(device)

Arguments: device -- Device ID or alias of the device for which to get the tap port ID. Can also be a Node object.

Returns: A BitData object containing the status bits.

Remarks: To see the bits in a formatted display, use the tapstatus() function.

Usage:

```
>>> itp.tapstatusraw(0)
```

5.2.1.103 transportdisable

Description Disable all configurations, and then disconnect the transport.

Signature: transportdisable(debugPort)

Arguments: debugPort -- ID, alias or debug port node or transport id that we are attempting to connect.

Returns: Nothing

Remarks: Specifically, this command disables the requested transport.

Usage:

```
>>> itp.transportdisable(0)
```

5.2.1.104 transportdisconnect

Description Graceful disconnect that maintains transport configuration but tristates JTAG, auto-reconnection, recovery and releases arbitration.

Signature: transportdisconnect(debugPort)

Arguments: debugPort -- ID, alias or debug port node or transport id that we are attempting to connect.

Returns: Nothing

Remarks: Specifically, this command disconnects the requested transport.

Usage:

```
>>> itp.transportdisconnect(0)
```

5.2.1.105 transportenable

Description Connect and if needed re-configure the transport, where connect arbitrates, auto-reconnection ecovery, and enables JTAG-IO.

Signature: transportenable(debugPort)

Arguments: debugPort -- ID, alias or debug port node or transport id that we are attempting to connect.

Returns: Nothing

Remarks: Specifically, this command enables the requested transport.

Usage:

```
>>> itp.transportenable(0)
```

5.2.1.106 uallclkdis

Description Disable all clocks (stop forcing all clocks to run) on the specified uncore.

Signature: uallclkdis(device) - deprecated. Use 'clockctrl' command instead.

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

Returns: None. See Also:

itp.uncores[n].uallclkdis()

Usage:

```
>>> itp.uallclkdis(0)
```

5.2.1.107 uallclken

Description Enable all clocks (force all clocks to run) on the specified uncore.

Signature: uallclken(device) - deprecated. Use 'clockctrl' command instead.

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

Returns: None. See Also:

itp.uncores[n].uallclken()

Usage:

```
>>> itp.uallclken(0)
```

5.2.1.108 ucrb

Description Read/write an uncore control register bus register using 32-bit values.

Signature: ucrb(device, address, newValue = None)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

address -- 16-bit address or offset to access.

newValue -- If not None, specifies a 32-bit value (or a BitData object containing a 32-bit value) to write to the register.

Returns: A BitData object containing a 32-bit value from the uncore control register bus register if reading. Returns None if writing to the register (newValue is not None). See Also:

itp.uncores[n].ucrb() itp.threads[n].ucrb()

Usage:

```
>>> itp.ucrb(0, 0x4000)
>>> itp.ucrb(0, 0x4000, 0x1234)
```

5.2.1.109 ucrb64

Description Read/write an uncore control register bus register using 64-bit values.

Signature: ucrb64(device, address, newValue = None)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

address -- 16-bit address or offset to access.

newValue -- If not None, specifies a 64-bit value (or a BitData object containing a 64-bit value) to write to the register.

Returns: A BitData object containing a 64-bit value from the uncore control register bus register if reading. Returns None if writing to the register (newValue is not None). See Also:

itp.uncores[n].ucrb64() itp.threads[n].ucrb64()

Usage:

```
>>> itp.ucrb64(0, 0x4000)
>>> itp.ucrb64(0, 0x4000, 0x1234)
```

5.2.1.110 unloadsymbols

Description Remove all symbols from the symbol table for all threads.

Signature: unloadsymbols()

Arguments: None.

Returns: None.

Usage:

```
>>? itp.unloadsymbols()
```

5.2.1.111 unlock

Description Unlock all the IA32 devices in the target system. Credentials will be asked for if not previously supplied and if device is locked.

Signature: unlock(device=None, level=None)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object. level -- Specifies the unlock level as a string.

Returns: None.

Usage:

```
>>> itp.unlock()
>>> itp.unlock(0)
>>> itp.unlock(0, "orange")
>>> itp.unlock(0, "red")
```

```
>>> itp.unlock(level="orange")
>>> itp.unlock(level="red")
>>> itp.unlock(None, "orange")
>>> itp.unlock(None, "red")
```

5.2.1.112 unlockerflush

Description Flush the credentials (username and password) that are stored by the unlocker.

Signature: unlockerflush(device=None)

Arguments: device -- Device ID or alias of the device to access. Can also be a Node object.

Returns: None.

Usage:

```
>>> itp.unlockerflush()
>>> itp.unlockerflush(0)
```

5.2.1.113 ureg64_raw

Description Read/write the 64-bit value from a register on the specified Uncore device (or the uncore device associated with the specified core or thread device), utilizing a full list of parameters, including uArch port, required for register request packet building.

Signature: ureg64_raw(device, portID, registerType, bar, deviceNumber, function, address, iaScope, newValue = None)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

portID -- uArch parameter that specifies the message channel destination port.

registerType -- A string specifying the register type. Can be one of "cr", "cfg", "io", or "mem".

bar -- Base Access Register (BAR) parameter. If registerType = "mem", then bar is treated as a bus ID.

deviceNumber -- Register device attribute. If registerType = "cr", then deviceNumber is treated as a core ID.

function -- Register function attribute. If registerType = "cr", then function is treated as a thread ID.

address -- 16-bit address or offset to access.

iaScope -- The IA scope parameter (typically set to 0).

newValue -- If not None, specifies a 64-bit value to write to the register.

Returns: A 64-bit value from the uncore register bus register if reading. Returns None if writing to the register (newValue is not None).

5.2.1.114 ureg64_raw_withsubportid

Description Read/write the 64-bit value from a register on the specified Uncore device (or the uncore device associated with the specified core or thread device), utilizing a full list of parameters, including uArch port, required for register request packet building.

Signature: ureg64_raw_withsubportid(portID, registerType, bar, deviceNumber, function, address, iaScope, subPortID, newValue = None)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device. portID -- uArch parameter that specifies the message channel destination port. registerType -- A string specifying the register type. Can be one of "cr", "cfg", "io", or "mem". bar -- Base Access Register (BAR) parameter. If registerType = "mem", then bar is treated as a bus ID. deviceNumber -- Register device attribute. If registerType = "cr", then deviceNumber is treated as a core ID. function -- Register function attribute. If registerType = "cr", then function is treated as a thread ID. address -- 16-bit address or offset to access. iaScope -- The IA scope parameter (typically set to 0). subPortID -- If not None, specifies the message channel sub port. newValue -- If not None, specifies a 64-bit value (or a BitData object containing a 64-bit value) to write to the register.

Returns: A 64-bit value from the uncore register bus register if reading. Returns None if writing to the register (newValue is not None).

Usage:

```
>>> itp.uncores[0].ureg64_raw_withsubportid(0, "cr", 0, 0, 0, 0x4000, 0,
 0xa)
>>> itp.uncores[0].ureg64_raw_withsubportid(0, "cr", 0, 0, 0, 0x4000, 0,
 0xa, 0x2128)
```

5.2.1.115 ureg_raw

Description Read/write the 32-bit value from a register on the specified Uncore device (or the uncore device associated with the specified core or thread device), utilizing a full list of parameters, including uArch port, required for register request packet building.

Signature: ureg_raw(device, portID, registerType, bar, deviceNumber, function, address, iaScope, newValue = None)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

portID -- uArch parameter that specifies the message channel destination port.

registerType -- A string specifying the register type. Can be one of "cr", "cfg", "io", or "mem".

bar -- Base Access Register (BAR) parameter. If registerType = "mem", then bar is treated as a bus ID.

deviceNumber -- Register device attribute. If registerType = "cr", then deviceNumber is treated as a core ID.

function -- Register function attribute. If registerType = "cr", then function is treated as a thread ID.

address -- 16-bit address or offset to access.

iaScope -- The IA scope parameter (typically set to 0).

newValue -- If not None, specifies a 32-bit value to write to the register.

Returns: A 32-bit value from the uncore register bus register if reading. Returns None if writing to the register (newValue is not None).

5.2.1.116 ureg_raw_withsubportid

Description Read/write the 32-bit value from a register on the specified Uncore device (or the uncore device associated with the specified core or thread device), utilizing a full list of parameters, including uArch port, required for register request packet building.

Signature: ureg_raw_withsubportid(portID, registerType, bar, deviceNumber, function, address, iaScope, subPortID, newValue = None)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device. portID -- uArch parameter that specifies the message channel destination port. registerType -- A string specifying the register type. Can be one of "cr", "cfg", "io", or "mem". bar -- Base Access Register (BAR) parameter. If registerType = "mem", then bar is treated as a bus ID. deviceNumber -- Register device attribute. If registerType = "cr", then deviceNumber is treated as a core ID. function -- Register function attribute. If registerType = "cr", then function is treated as a thread ID. address -- 16-bit address or offset to access. iaScope -- The IA scope parameter (typically set to 0). subPortID -- If not None, specifies the message channel sub port. newValue -- If not None, specifies a 32-bit value (or a BitData object containing a 32-bit value) to write to the register.

Returns: A BitData object containing a 32-bit value from the uncore register bus register if reading. Returns None if writing to the register (newValue is not None).

Usage:

```
>>> itp.uncores[0].ureg_raw_withsubportid(0, "cr", 0, 0, 0, 0x4000, 0, 0xa)
>>> itp.uncores[0].ureg_raw_withsubportid(0, "cr", 0, 0, 0, 0x4000, 0, 0xa,
 0x2128)
```

5.2.1.117 utapstatus

Description Display the TAP status bits from the specified uncore device as a formatted string.

Signature: utapstatus(device)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

Returns: None.

Remarks: To get the raw bits, use the utapstatusraw() function.

The display shows the logical bit indices if there is more than one bit or the first bit is not 0. The physical mapping into the parent bits is always shown. The following example shows the tapstatus register with two fields. The PMRDY field has an implied logical mapping of [0:0] and a physical mapping to bit 88 in the tapstatus register. The PhySlice field has an explicit logical mapping of [7:0] and a physical mapping to bits 34 through 27.

tapstatus(109:0) = [109:0] 0x0000000000000878000000000000 PMRDY = [88] 0x0 PhySlice(7:0) = [34:27] 0x00 See Also:

itp.uncores[n].utapstatus() itp.threads[n].utapstatus()

Usage:

```
>>> itp.utapstatus(0)
```

5.2.1.118 utapstatusraw

Description Retrieve the TAP status bits from the specified uncore device.

Signature: utapstatusraw(device)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

Returns: A BitData object containing the status bits.

Remarks: To see the bits in a formatted display, use the utapstatus() function. See Also:

itp.uncores[n].utapstatusraw() itp.threads[n].utapstatusraw()

Usage:

```
>>> itp.utapstatusraw(0)
```

5.2.1.119 vcudata

Description Retrieve or change a value from a register in the Validation Control Unit NodeCore, the associated NodeUncore device is used. All other device types throw an exception.ion.

Signature: vcudata(device, register, newValue = None)

Arguments: device -- An uncore, core, or thread device, identified by device ID, name, or an actual object. If core or thread, then uses the associated uncore device.

register -- A valid VCU register number.

newValue -- If not None, specifies a 32-bit value (or a BitData object containing a 32-bit value) to write to the register.

Returns: A BitData object containing a 32-bit value. Returns None if writing to the register (newValue is not None). See Also:

itp.uncores[n].vcudata()

Usage:

```
>>> itp.vcudata(0, 0x100)
>>> itp.vcudata(0, 0x100, 0x1234)
```

5.2.1.120 vmcsinfo

Description Display miscellaneous VMCS information for all threads.

Signature: vmcsinfo(infoKind)

Arguments: infoKind -- A string specifying the kind of vmcs information to return. Can be one of the following: "vmcsrevision" - revision number of the VMCS supported by the processor. "vmcssize" - size of the VMCS supported by the processor. "vmxmode" - current VMX mode. "ilp" - Is processor the Initiating Logical Processor. "secenter" - Has SEnter been executed on the processor. "enterac" - Has EnterAC been executed on the processor. "enteraccs" - Has EnterACCS been executed on the processor.

Returns: None.

Usage:

```
>>> itp.vmcsinfo("vmcsrevision")
>>> itp.vmcsinfo("vmcssize")
```

Remarks: The output will contain the following based on the kind of information requested:

"vmcsrevision": Revision number of the VMCS supported by the processor. "vmcssize" : Size of the VMCS supported by the processor. "vmxmode" : The current VMX mode: 0 = VMX off 1 = Root mode 2 = Monitor mode (deprecated) 3 = Guest mode 11 = Root parallel SMM handler 13 = Guest parallel SMM handler "ilp" : 1 if the processor is the Initiating Logical Processor. "secenter" : 1 if SEnter has been executed on the processor. "enterac" : 1 if EnterAC has been executed on the processor. "enteraccs" : 1 if EnterACCS has been executed on the processor.

5.2.1.121 wait

Description Suspend script execution until a break has occurred on the target or the specified number of seconds has passed.

Signature: wait(seconds = None)

Arguments: seconds -- Number of seconds to wait. If set to None, waits forever.

Returns: If seconds are specified, return True if the wait returned before the timeout expired. If seconds are not specified, always returns None.

Remarks: Use the wait command following a go command to prevent the Python CLI from accepting input until after the specified number of seconds, or a break in execution, whichever occurs first.

The wait command is required immediately after a go command in a debug procedure or imported file if the commands that follow are not to be executed until execution breaks. If there is no wait command after a go command in a debug procedure or imported file and a command tries to access the target processor, the procedure or import file execution terminates.

Usage:

```
>>? itp.wait()  # wait forever
>>? if itp.wait(3): # wait 3 seconds
>>?     print("A break was encountered.");
>>? if itp.threads[0].cv.isrunning:
>>?     print("thread did not hit breakpoint.")
```

5.2.1.122 waitforevent

Description Suspend script execution until a the requested event has completed

Signature: waitforevent(eventType,seconds = None)

Arguments: eventType -- the event that requires to be monitored. The following events are supported: processorbreak, platformbreak, powerLoss, powerRestore, bclkLoss, bclkRestore, reconfigStart, reconfigComplete.

seconds -- Number of seconds to wait. If set to None, waits forever.

Returns: If seconds are specified, return True if the wait returned before the time-out expired. If seconds are not specified, always returns None.

Usage:

```
>>? itp.waitforevent(itp.EventType.reconfigComplete)  # wait for the
 reconfig event for the default timeout
>>? if itp.waitforevent(itp.EventType.powerRestore,3): # wait 3 seconds for
 the powerRestore event
>>?     print "Event Received"
>>? itp.pulsepwrgood()
>>? itp.waitforevent(itp.EventType.reconfigComplete |
 itp.EventType.powerRestore,20);
True # Value returned is true if both the conditions above are satisfied
>>?
```

5.2.2 Thread CLI Commands

5.2.2.1 alstall

Description Stall the allocator via JTAG by scanning in the Allocator Stall

Signature: alstall(tapData = None)

Arguments: tapData -- Value to be shifted into the ALSTALL TAP data register. Refer to the processor specification document on the exact number of bits and bit positions of the data register.

This value is needed only for processors whose ALSTALL TAP command has a data section; otherwise, the ALSTALL TAP command is assumed to be only an Instruction Register.

Returns: None.

Usage:

```
>>> itp.threads[0].alstall()
>>> itp.threads[0].alstall(0x1234)
```

5.2.2.2 alunstall

Description Un-stall the allocator via JTAG by scanning in the Allocator Unstall thread.

Signature: alunstall()

Arguments: None.

Returns: None.

Usage:

```
>>> itp.threads[0].alunstall()
```

5.2.2.3 asm

Description Display memory as IA32 disassembled instructions or assembles IA32

Signature: asm(address, *args)

Arguments: address -- Starting address to disassemble from or assemble to. Can also be '$', indicating the current instruction pointer for the thread. Can also be a symbol name or line number "#lineNumber".

*args -- If the first argument is a number or is a string that does not start with an alphabetic character, the argument is treated as a "to" parameter (if number, it is the number of instructions to work with, otherwise it is an end address). Otherwise, all arguments are treated as strings containing instructions to be assembled to memory.

Returns: None.

Remarks: The asmmode control variable override is used only if the address provided to this command is linear or physical; otherwise, the current addressing mode takes precedence and dictates the mode to assemble/disassemble.

Usage:

```
>>> itp.threads[0].asm("0x1000P")     # disassemble single instruction from
 address 0x1000
>>> itp.threads[0].asm("$")           # disassemble single instruction from
 current instruction pointer address.
>>> itp.threads[0].asm("0x1000P", 10) # disassemble 10 instructions starting
 from address 0x1000
>>> itp.threads[0].asm("0x1000P", "0x1010P")      # disassemble from address
 0x1000P through 0x1010P.
>>> itp.threads[0].asm("0x1000P", "nop", "jmp $") # assemble two
 instructions to address 0x1000
>>> itp.threads[0].asm("$", "clflush [0x150000]") # assemble an instruction
 using a memory reference
```

5.2.2.4 bits

Description Access the contents of a bit-field within a register, virtual register, or register component.

Signature: bits(register, offset, size, newValue = None)

Arguments: register -- A valid register name (with optional component path), an arbitrary integer, or a BitData object.

offset -- The bit index where the bit field begins. Must be less than the size of the register, integer, or BitData object.

size -- Size, in bits, of the bit field to extract. The size plus the offset must be less than the size of the register, integer, or BitData object.

newValue -- A 64-bit integer or another BitData object to be assigned to the bit field. If the value is greater than what can fit in the bitfield, the value is truncated to the bit-size during assignment (most significant bits are clipped).

Returns: If newValue is None, returns a BitData object containing the requested bits. If newValue is not None and register is an integer, then returns a BitData object containing the updated integer. Otherwise, the register or provided BitData object is changed in place and nothing is returned.

Remarks: Use the bits command to accesses the contents of a bit-field within a register or register component. When specifying a register, the contents of the register are changed when writing a new value.

The register component must be one of the already defined registers within the ITP otherwise an error is generated.

The processor must be halted to alter register values.

When used for assignment, bits will overwrite only the bit field indicated within the specified register (all other bits within the register are unaffected by the assignment). Any bits in an assigned expression that go beyond the size of the bit-field being assigned to are truncated.

Usage:

```
>>> itp.threads[0].bits("rax.ax", 8, 0, 0xff)
>>> newValue = itp.threads[0].bits(0x8000, 8, 4, 0xf)
>>> bitData = BitData(110) # 110 bits of all zero's.
>>> itp.threads[0].bits(bitData, 10, 8, 0xea)
>>> bitData # altered object
```

5.2.2.5 br

Description View breakpoints, create breakpoints, or change an existing breakpoint.

Signature: br(breakpointID = None, address = None, breakType = None, dbreg = None, enable = None)

Arguments: breakpointID -- ID of the breakpoint to change or create. Can be None, in which case a new ID is assigned. The BreakpointID can be a string or a number starting from 1.

address -- Address to change to. This parameter is required.

breakType -- A string containing the type of the break point to create. Set to None to leave this parameter alone. Can be one of the following:

"sw" -- Software breakpoint

"exe" -- Hardware execution breakpoint

"exe global" -- Hardware execution breakpoint (Global is across all tasks)

"exe local" -- Hardware execution breakpoint (Local is for current task)

"acc" -- Hardware memory access breakpoint

"acc global" -- Hardware memory access breakpoint (Global is across all tasks)

"acc local" -- Hardware memory access breakpoint (Local is for current task)

"wr" -- Hardware memory write breakpoint

"wr global" -- Hardware memory write breakpoint (Global is across all tasks)

"wr local" -- Hardware memory write breakpoint (Local is for current task)

"io" -- Hardware I/O access breakpoint

dbreg -- Which debug register to use for the breakpoint information. Set to None to leave this parameter alone.

enable -- True or False to enable or disable the breakpoint. Set to None to leave this parameter alone.

Returns: None.

Remarks: If the breakpoint does not exist, it is created (if possible). If an aspect cannot be changed, an error is thrown. If the address is None, then output a list of existing breakpoints (or the specified breakpoint, if the breakpointID is not None).

Usage:

```
>>> itp.threads[0].br(None, "0x1000")
>>> itp.threads[0].br(1, "0x1100")
>>> itp.threads[0].br(None, "0x80P", "io")
>>> itp.threads[0].br(None, "0x1200L", "exe global")
>>> itp.threads[0].br(None, "0x1300L", "wr global", dbreg = 3)
>>> itp.threads[0].br(None, "0x1400L", "acc global", enable = False)
>>> itp.threads[0].br("MyBrkpoint", "0x1500L", None, None, None)
```

5.2.2.6 brdisable

Description Disable all breakpoints on the thread processor.

Signature: brdisable()

Arguments: None.

Returns: None.

Usage:

```
>>> itp.threads[0].brdisable()
```

5.2.2.7 brenable

Description Enable all breakpoints on the thread processor.

Signature: brenable()

Arguments: None.

Returns: None.

Usage:

```
>>> itp.threads[0].brenable()
```

5.2.2.8 brex

Description Create, change, or view a new extended breakpoint of MSR or I/O types.

Signature: brex(breakpointID = None, address = None, addressMask = None, breakType = None, accessType = None, data = None, dataMask = None, enable = None)

Arguments: breakpointID -- ID of the breakpoint to change or create. Can be None, in which case a new ID is assigned.

address -- 32-bit MSR address or 16-bit physical I/O address.

addressMask -- 32-bit MSR address mask or 16-bit I/O address mask. This mask is applied to the address.

breakType -- A string containing the type of the break point to create. Set to None to leave this parameter alone. Can be one of the following: "port" - specifies the breakpoint as an I/O access breakpoint. "msr" - specifies the breakpoint as a MSR access breakpoint. Set to None to leave this parameter alone.

accessType -- A string specifying the kind of access that triggers the breakpoint. Can be one of the following: "rd" - specifies the breakpoint as a read access breakpoint. "wr" - specifies the breakpoint as a write access breakpoint. "rdwr" - specifies the breakpoint as a read/write access breakpoint. Set to None to leave this parameter alone.

data -- 64-bit MSR data value or 32-bit I/O data value that is contained at the specified address to break on. Set to None to leave this parameter alone.

dataMask -- 64-bit MSR data mask or 32-bit I/O data mask. This mask is applied to the data at the address. Set to None to leave this parameter alone.

enable -- True or false to enable or disable the breakpoint. Set to None to leave this parameter alone.

Returns: None.

Remarks: Create, change, or view a new extended breakpoint of MSR or I/O types on supported devices with data and data mask and with the specified breakpoint ID. If the breakpoint ID already exists, then an error is thrown. If the address is not specified, an error is thrown.

Usage:

```
>>> itp.threads[0].brex(None, 0x100, None, "msr", "rd", 0xdeadbeef)
>>> itp.threads[0].brex(1, 0x210, 0xFFFFFFF0, "msr", "rd", 0xdeadbeef)
>>> itp.threads[0].brex(None, 0x80, None, "port", "wr", 0xFEED)
>>> itp.threads[0].brex(None, 0x80, 0xFFF0, "port", "rd", 0x12340000,
 0xFFFF0000)
```

5.2.2.9 brget

Description Retrieve all breakpoints for the thread.

Signature: brget()

Arguments: None.

Returns: A formatted string containing a list of all breakpoints on the thread. Can be empty if there are no breakpoints.

Usage:

```
>>> itp.threads[0].brget()
```

5.2.2.10 brnew

Description Create a new breakpoint using one or more aspects, including initially disabled.

Signature: brnew(address, breakType = None, dbreg = None, enable = None, breakpointID = None)

Arguments: address -- Address to break at. Must be specified. This is a string. breakType -- A string containing the type of the break point to create. If None then defaults to 'exe global' type. Can be one of the following: "sw" -- Software breakpoint "exe" -- Hardware execution breakpoint "exe global" -- Hardware execution breakpoint (Global is across all tasks) "exe local" -- Hardware execution breakpoint (Local is for current task) "acc" -- Hardware memory access breakpoint "acc global" -- Hardware memory ac-

cess breakpoint (Global is across all tasks) "acc local" -- Hardware memory access breakpoint (Local is for current task) "wr" -- Hardware memory write breakpoint "wr global" -- Hardware memory write breakpoint (Global is across all tasks) "wr local" -- Hardware memory write breakpoint (Local is for current task) "io" -- Hardware I/O access breakpoint dbreg -- Which debug register to use for the breakpoint information (0 - 3). If None then no dbreg override is used. enable -- Initial enable state for the new breakpoint (True or False). Set to None defaults to True (enabled). breakpointID -- ID to use for the breakpoint. The BreakpointID can be a string or a number starting from 1. If this ID already exists, an error is raised. If None, a new breakpoint ID is created.

Returns: None.

Usage:

```
>>> itp.threads[0].brnew("0x1000P")
>>> itp.threads[0].brnew("0x100P", "io")
>>> itp.threads[0].brnew("0x1200L", "exe global")
>>> itp.threads[0].brnew("0x1200L", "exe global", dbreg = 4)
>>> itp.threads[0].brnew("0x1200L", "exe global", dbreg = 4, enable = False)
>>> # Create new breakpoint on thread 0 at address 0x1100P with specific
 breakpoint ID 3.
>>> itp.threads[0].brnew("0x1100P", None, None, None, 3)
>>> itp.threads[0].brnew("0x1100P", breakpointID = 3)
>>> # Create new breakpoint on thread 0 at address 0x1100P with specific
 breakpoint ID "MyBrkpoint".
>>> itp.threads[0].brnew("0x1100L", None, None, None, "MyBrkpoint")
```

5.2.2.11 brremove

Description Remove all breakpoints on the the processor.

Signature: brremove()

Arguments: None.

Returns: None.

Usage:

```
>>> itp.threads[0].brremove() # remove all breakpoints on thread 0
```

5.2.2.12 bstep

Description Perform a step branch operation.

Signature: bstep()

Arguments: None.

Returns: None.

Remarks: The bstep command is an alias for step(itp.StepType.branch, 1).

Usage:

```
>>> itp.threads[0].bstep()
```

5.2.2.13 c0hldst

Description Modify the C0HLDST data register on the specified device.

Signature: c0hldst(dataValue)

Arguments: dataValue -- 32-bit value (or BitData object containinga 32-bit value) to be shifted into the C0HLDST TAP data register. Refer to the processor specification document on the exact number of bits and bit positions of the data register

Returns: None.

Usage:

```
>>> itp.threads[0].c0hldst(0x1)
```

5.2.2.14 calls

Description Display stack frames that contain information about function calls from the call stack.

Signature: calls(noParams = None, frameCount = None)

Arguments: noParams -- If True, each stack frame is displayed without showing parameters to the function that is associated with each stack frame.

frameCount -- Number of frames to display. If this is None, then the entire call stack is displayed.

Returns: None.

Remarks: The processor must be halted first. If symbols have not been loaded, the calls() command will attempt to display the addresses on the stack; however, without the symbols file to interpret how the stack is used, this stack may not be accurate.

Usage:

```
>>? itp.threads[0].calls()
>>? itp.threads[0].calls(True)
>>? itp.threads[0].calls(None, 4)
```

5.2.2.15 cbotapconfig

Description Retrieve or change the value of the CBO_TAPCONFIG data register on the specified device. Note that device must be a core/thread, not Uncore device.

Signature: cbotapconfig(newValue)

Arguments: newValue -- The 32-bit value (or BitData object containing the 32-bit value) to write to the register.

Returns: A BitData object containing the 32-bit value from the specified register.

Usage:

```
>>> itp.threads[n].cbotapconfig()  (***DEPRECATED*** please use
 '<devicenode>.state.tap.<*cbo_tapconfig*>'
>>> itp.threads[n].cbotapconfig(0xefefefef)
```

5.2.2.16 cbotapstatus

Description Display the Cache Box (CBO) TAP status using the cbotapstatus TAP command."

Signature: cbotapstatus()

Arguments: None.

Returns: None.

Usage:

```
>>> itp.thread[0].cbotapstatus()
```

5.2.2.17 cbotapstatusraw

Description Retrieve the Cache Box (CBO) TAP status using the cbotapstatus TAP command.

Signature: cbotapstatusraw()

Arguments: None.

Returns: A BitData object containing the status bits.

Usage:

```
>>> itp.threads[0].cbotapstatusraw()
```

5.2.2.18 clockctrl

Description Read or change the CLOCKCTRL data register which controls the specified clock domain.

Signature: clockctrl(clockLocation = None, newValue = None)

Arguments: clockLocation -- A string specifying the location (system agent, core, cache) where the clock resides. If this value is None, defaults to the system agent (uncore) clock. Can be one of the following: "saclks" System Agent (Uncore) "coreclks" Processor Core "cboclks" Processor Cache

newValue -- If not None, specifies a 32-bit value (or a BitData object containing a 32-bit value) to write to the register.

Returns: A BitData object containing a 32-bit value. Returns None if writing to the register (newValue is not None).

Usage:

```
>>> itp.threads[0].clockctrl()
>>> itp.threads[0].clockctrl("saclks")
>>> itp.threads[0].clockctrl("coreclks", 1)
>>> itp.threads[0].clockctrl("cboclks", 0)
```

5.2.2.19 cpuid_eax

Description Return the processor identification and feature information stored in the EAX register, where the EAX register is initialized to 1.

Signature: cpuid_eax(eax = None, ecx = None)

Arguments: eax -- 32-bit value to initialize eax to before issuing CPUID instruction. Set to None to use 1.

ecx -- 32-bit value to initialize ecx to. Set to None to use 0. Ignored if eax is None.

Returns: A BitData object containing the value from the EAX register after issuing the CPUID instruction.

Remarks: Use the cpuid_eax() command to read values from the cpuid instruction. The eax value specifies the leaf from which to read the value returned in the eax value after executing the cpuid instruction.

Usage:

```
>>> itp.threads[0].cpuid_eax()
>>> itp.threads[0].cpuid_eax(0x2)
>>> itp.threads[0].cpuid_eax(0x2, 0x1)
```

5.2.2.20 cpuid_ebx

Description Return the processor identification and feature information stored in the EBX register, where the EAX register is initialized to 1.

Signature: cpuid_ebx(eax = None, ecx = None)

Arguments: eax -- 32-bit value to initialize eax to before issuing CPUID instruction. Set to None to use 1.

ecx -- 32-bit value to initialize ecx to. Set to None to use 0. Ignored if eax is None.

Returns: A BitData object containing the value from the EBX register after issuing the CPUID instruction.

Remarks: Use the cpuid_ebx() command to read values from the cpuid instruction. The eax value specifies the leaf from which to read the value returned in the ebx value after executing the cpuid instruction.

Usage:

```
>>> itp.threads[0].cpuid_ebx()
>>> itp.threads[0].cpuid_ebx(0x2)
>>> itp.threads[0].cpuid_ebx(0x2, 0x1)
```

5.2.2.21 cpuid_ecx

Description Return the processor identification and feature information stored in the ECX register, where the EAX register is initialized to 1.

Signature: cpuid_ecx(eax = None, ecx = None)

Arguments: eax -- 32-bit value to initialize eax to before issuing CPUID instruction. Set to None to use 1.

ecx -- 32-bit value to initialize ecx to. Set to None to use 0. Ignored if eax is None.

Returns: A BitData object containing the value from the ECX register after issuing the CPUID instruction.

Remarks: Use the cpuid_ecx() command to read values from the cpuid instruction. The eax value specifies the leaf from which to read the value returned in the ecx value after executing the cpuid instruction.

Usage:

```
>>> itp.threads[0].cpuid_ecx()
>>> itp.threads[0].cpuid_ecx(0x2)
>>> itp.threads[0].cpuid_ecx(0x2, 0x1)
```

5.2.2.22 cpuid_edx

Description Return the processor identification and feature information stored in the EDX register, where the EAX register is initialized to 1.

Signature: cpuid_edx(eax = None, ecx = None)

Arguments: eax -- 32-bit value to initialize eax to before issuing CPUID instruction. Set to None to use 1.

ecx -- 32-bit value to initialize ecx to. Set to None to use 0. Ignored if eax is None.

Returns: A BitData object containing the value from the EDX register after issuing the CPUID instruction.

Remarks: Use the cpuid_edx() command to read values from the cpuid instruction. The eax value specifies the leaf from which to read the value returned in the edx value after executing the cpuid instruction.

Usage:

```
>>> itp.threads[0].cpuid_edx()
>>> itp.threads[0].cpuid_edx(0x2)
>>> itp.threads[0].cpuid_edx(0x2, 0x1)
```

5.2.2.23 crb

Description Read/write a CPU core control register bus register using 32-bit values.

Signature: crb(address, newValue = None)

Arguments: address -- The register number/address to access. newValue -- If not None, specifies a 32-bit value to write to the register.

Returns: A 32-bit value from the control register bus register if reading. Returns None if writing to the register (newValue is not None).

Usage:

```
>>> itp.threads[0].crb(0x4010)
>>> itp.threads[0].crb(0x4010, 0x2110)
```

5.2.2.24 crb64

Description Read/write a CPU core control register bus register using 64-bit values.

Signature: crb64(address, newValue = None)

Arguments: address -- The register number/address to access. newValue -- If not None, specifies a 64-bit value to write to the register.

Returns: A 64-bit value from the control register bus register if reading. Returns None if writing to the register (newValue is not None).

Usage:

```
>>> itp.threads[0].crb64(0x4010)
>>> itp.threads[0].crb64(0x4010, 0x21102312341)
```

5.2.2.25 display

Description Display the contents of the specified architectural register and all its sub-components.

Signature: display(registerName)

Arguments: registerName -- Name of the register to display from the specified thread.

Returns: None.

Remarks: Use the display command to display the contents of a register. This command also enables the user to find out the components of a register if they have been forgotten. In addition to the value of the register itself, each

ITP-defined component of the specified register is display along with its value. If the register contains no defined components then only the value of the register is displayed.

Usage:

```
>>> itp.threads[0].display("eax")
>>> itp.threads[0].display("rflags")
```

5.2.2.26 dport

Description Retrieve or change the contents of a 32-bit IA32 I/O port.

Signature: dport(portNumber, newValue = None)

Arguments: portNumber -- A port number in the target processor I/O space. The port number value exists in the range 0x00 to 0xfffc.

newValue -- If specified, a 32-bit value to write to the port.

Returns: If reading the port, returns a BitData containing the 32-bit value from the port; otherwise, returns nothing.

Remarks: Use the dport command for 32-bit read and write access to the target system I/O space.

Note: Not all I/O ports support 32-bit access.

Usage:

```
>>> itp.threads[0].dport(0x80)
>>> itp.threads[0].dport(0x80, 0x12345678)
>>> itp.threads[0].dport(BitData(16,128))
>>> itp.threads[0].dport(BitData(16,128), 0xcode)
```

5.2.2.27 edbgread

Description Reads 4 or 8 bytes of data from a given EPC memory location.

Signature: edbgread(address)

Arguments: address -- A valid address, expressed as a string/unicode/Address

Returns: Return an BitData object containing the 4 or 8 bytes of data.

Usage:

```
>>> itp.threads[0].edbgread("0x7ac0be1100")
```

5.2.2.28 edbgwrite

Description Writes a byte (4/8) to the Secure enclave memory at the given address.

Signature: edbgwrite(address, bdWriteVal)

Arguments: address -- A valid address, expressed as a string/unicode/Address

bdWriteVal -- Value to be written into the secure enclave's memory

Returns: None

Usage:

```
>>> itp.threads[0].edbgwrite("0x7ac0be1100", BitData(64,
 0xDEADBEEFBAD4F00D))
```

5.2.2.29 eval

Description Evaluate an address or expression and return the results as a string.

Signature: eval(expression, convertToType = None, convertToSubtype = None)

Arguments: expression -- The expression to evaluate. If the expression is numerical, then it is converted to binary, decimal, and hexadecimal.

If the expression contains '$' then the expression is always treated as an address (the '$' represents the current instruction pointer).

Arithmetic operators and shift operators can be used.

The register symbols shown in "itp.threads[0].state.regs() can be used. (processor must be halted)

If symbols have been loaded symbols can be used. Local and Formal parameters in the current stack frame can be be used if on a live machine, the loaded symbols correspond to the image loaded in memory, and the processor is halted at a breakpoint. (Only supported in 32 bit mode) (processor must be halted)

:ModuleName - looks for the address of a module named "ModuleName" #52 - looks for the address of source line 52

If the expression is a string and cannot be converted to a number, then the bytes of the string are displayed. - Not Implemented

If the expression contains a floating point value, then the bytes of the floating point value are displayed.

convertToType -- A string that specifies what to convert the expression to. Can be None if the expression is to be treated just as as number or the parameter can be one of the following:

'line' Evaluate the expression as an address and display the result as a line number in the form: :module-name#line-number

'procedure' Evaluate the expression as an address and display it as a procedure name in the form: :module-name.procedure-name

'symbol' Evaluate the expression as an address and display it as the symbol associated with the address, in the form of a fully qualified reference

'physical' Assume the expression is an address and display it as a physical address

'guestphysical' Assume the expression is an address and display it as a guest physical address. This is a shortcut for convertToType = 'physical', convertToSubtype = 'gpa'.

'linear' Assume the expression is an address and display it as a linear address.

'virtual' Assume the expression is an address and display it as virtual address.

convertToSubtype -- A value that modifies what the expression is converted to. This is valid only if the convertToType is 'physical' or 'linear', implying the expression is an address. This parameter can be None if the sub-type is not used.

The following sub-types are allowed:

'gpa' Express the address as a guest physical address (only with 'physical').

'code' Express the address as a virtual address based on code segment (only with 'virtual'),

'data' Express the address as a virtual address based on data segment (only with 'virtual'),

'stack' Express the address as a virtual address based on stack segment (only with 'virtual'),

Returns: A string containing the results of the evaluation.

Usage:

```
>>? itp.threads[0].eval("1+1")
>>? itp.threads[0].eval("$")
>>? itp.threads[0].eval("$+10")
>>> itp.threads[0].eval("$", "line")
>>> itp.threads[0].eval("#29", "procedure")
>>> itp.threads[0].eval("0x20:0380", "procedure")
>>> itp.threads[0].eval("cs:0x100")
>>> itp.threads[0].eval("ldtr:cs:0x100")
>>> itp.threads[0].eval("MAIN")
>>> itp.threads[0].eval("0x020:104:0", "linear")
>>> itp.threads[0].eval("100L", "physical", "gpa")
>>> itp.threads[0].eval("100L", "guestphysical")
>>> itp.threads[0].eval("0x10:0", "physical")
>>> itp.threads[0].eval("1000L", "physical")
>>> itp.threads[0].eval("0x1000", "virtual")
```

```
>>> itp.threads[0].eval("arrayParam[3])
>>> itp.threads[0].eval("ptrToStructParam->field")
>>?
```

5.2.2.30 flush

Description NOT SUPPORTED Use wbinvd() or invd() instead.

Signature: flush()

Arguments: None.

Returns: None.

5.2.2.31 get_EPCM

Description Retrieve the Enclave Page Cache Map for the valid address in the EPC region.

Signature: get_EPCM(address)

Arguments: address -- A valid address, expressed as a string/unicode/Address.

Returns: Return an IStateNav object containing the EPCM structure.

Usage:

```
>>> itp.threads[0].get_EPCM("0x7ac0be1100")
```

5.2.2.32 get_SECS

Description Retrieve the Secure Enclave Control Structure for the valid address in the EPC

Signature: get_SECS(address)

Arguments: address -- A valid address, expressed as a string/unicode/Address

Returns: Return an IStateNav object containing the SECS structure.

Usage:

```
>>> itp.threads[0].get_SECS("0x7ac0be1100")
```

5.2.2.33 get_SSA

Description Retrieve the Secure Storage Area for the valid address in the EPC region.

Signature: get_SSA(address)

Arguments: address -- A valid address, expressed as a string/unicode/Address.

Returns: Return an IStateNav object containing the SSA structure.

Usage:

```
>>> itp.threads[0].get_SSA("0x1100:0x7ac0be1101")
```

5.2.2.34 get_TCS

Description Retrieve the Thread Control Structure for the valid address in the EPC region.

Signature: get_TCS(address)

Arguments: address -- A valid address, expressed as a string/unicode/Address.

Returns: Return an IStateNav object containing the TCS structure.

Usage:

```
>>> itp.threads[0].get_TCS("0x1100:0x7ac0be1101")
```

5.2.2.35 go

Description Resume execution on the specified processor.

Signature: go(address = None, goTilType = None, *args)

Arguments: address -- address to break on, if specified. Otherwise, processor is just resumed. goTilType -- value from GoTilType enumeration specifying the type of break. Default is 'exe'. If goTilType is given, then the address to be specified. *args -- additional address/GoTilType pairs with GoTilType being optional.

Returns: None.

Usage:

```
>>> thread.go()
>>> thread.go("0x100")
>>> thread.go("0x100", itp.GoTilType.exe)
>>> thread.go("0x100", itp.GoTilType.exe, "0x80", itp.GoTilType.io)
>>> thread.go("0x100", itp.GoTilType.exe, "0x200")
>>> thread.go("0x100", "0x200")
```

5.2.2.36 halt

Description Halt execution on the specified processor.

Signature: halt()

Arguments: None.

Returns: None.

Usage:

```
>>> itp.threads[0].halt()
```

5.2.2.37 idcode

Description Return the processor boundary scan idcode for the specified thread/processor.

Signature: idcode()

Arguments: None.

Returns: Return the idcode as a string.

Usage:

```
>>> itp.threads[0].idcode()
```

5.2.2.38 invd

Description Invalidate IA32 processor cache.

Signature: invd()

Arguments: None.

Returns: None.

Remarks: This command will cause the processor to execute a "INVD" assembly instruction. It invalidates the processor's internal caches. Data held in internal caches is not written back to main memory (see the wbinvd command). Refer to the Intel Architecture SW Developers Manual (Volume 2A) for more details on this instruction.

In a multiprocessor system, only the caches for the processor on which this command is executed are invalidated. The flush command can only be used while the target processor is not running.

Usage:

```
>>? itp.threads[0].invd()
```

5.2.2.39 istep

Description Perform a single step of machine-language instruction.

Signature: istep(steps = None)

Arguments: steps -- Number of steps to take before stopping. If not specified, defaults to 1 step.

Returns: None.

Remarks: The istep command uses bits in the processor's PSR register to control stepping. If this register is changed by the executing program, step breaks will occur or not occur as expected. Also, both interrupts and task switches modify the PSR and will affect step breaking.

This command is the equivalent to step(itp.StepType.into, steps).

Usage:

```
>>> itp.threads[0].istep() # step 1
>>> itp.threads[0].istep(3) # step 3
```

5.2.2.40 load

Description Load an object-module file into the target system memory and load the debugging symbols into the host memory. Set the instruction pointer on the thread to point to the entry of the file.

Signature: load(filename, operation = None, at = None, offset = None, raw = None, loadkey = None)

Arguments: filename -- Name and path of the file to load.

operation -- One or more values from the itp.LoadOperations collection modifying the load behavior. The following options are possible:

itp.LoadOperations.nocode Do not load the code, only the symbols. Cannot be combined with any other operation.

itp.LoadOperations.nosymbol Do not load debug symbols. Cannot be combined with 'nocode'.

itp.LoadOperations.allregs Load the registers of all processors during the load, even if the load targets once processor.

itp.LoadOperations.nopsp Do not create a Program Segment Prefix (PSP) during the load. Valid only if loading a Microsoft binary that already has a PSP.

at -- A string or Address object that specifies the address in the target system to which to load the file.

offset -- A string or integer that specifies the offset, in bytes, to the OMF's relocatable addresses where a file's code, data, and symbolics will be loaded in memory.

raw -- True if to load the file without debug symbols and without performing any relocation.

loadkey -- String used to make unique each symbol load of each file so the files can be loaded cumulatively (normally each load removes any previous symbols).

Returns: None.

Remarks: The upload command is used for code patching. The command saves small patches made in a loaded program. The patched program can then be reloaded without the need to rebuild each patch.

Use the upload command to transfer the contents of target memory to a host file. The information transferred is an unformatted raw-memory image. Therefore, no assumptions are made about the format of the data in the target memory. The uploader does not make any efforts to re-map addresses that are present in the uploaded memory. Absolute addresses contained within the block may be invalid when reloaded, if not reloaded at the same address.

NOTE: The .COM and .SYS filename extensions should be used for Microsoft* COM format as executable only.

The host file will be written in a raw memory dump. This is similar to the MS-DOS* .COM format, which has a limit of 65278 bytes.

If the file specified by filename already exists, upload will not overwrite the file nor prompt for an overwrite, but instead an error will be reported. To overwrite an existing file, you must specify the overwrite option. The overwrite option will cause the file to lose its original contents.

If an error occurs, the output file is deleted.

Usage:

```
>>? itp.threads[0].load("myfile.exe")
>>? itp.threads[0].load("myfile.exe", itp.LoadOperations.nocode)
>>? itp.threads[0].load("myfile.exe", itp.LoadOperations.nosymbols +
 itp.LoadOperations.allregs)
>>? itp.threads[0].load("myfile.exe", at = "0x300P")
>>? itp.threads[0].load("myfile.exe", itp.LoadOperations.nocode, offset =
 0x300)
>>? itp.threads[0].load("myfile.exe", at = "0x40200L", loadkey = "TEST1")
>>? itp.threads[0].load("myfile.bin", at = "0x300P", raw = True)
                         See Also:
- itp.load()
- itp.threads[0].memload()
```

5.2.2.41 loadthis

Description Backs up from address in target memory until the BIOS header is found to address and PDB filepath. Then loads the PDB file if not already loaded to

Signature: loadthis()

Returns: None.

Usage:

```
>>? itp.threads[0].loadthis()
```

5.2.2.42 mem

Description Read or write a single value to memory.

Signature: mem(address, dataSize, newValue = None)

Arguments: address -- A valid address, expressed as a string.

dataSize -- Size of the data to write, in bytes (1 = byte, 2 = word, 4 = dword, 8 = qword)

newValue -- If specified, the value to write to memory (as an integer or a BitData object); otherwise, memory is read from.

Returns: Return a BitData object containing the value from the memory if reading the memory; otherwise, returns nothing.

Usage:

```
>>> itp.threads[0].mem("0x100P", 2)
>>> itp.threads[0].mem("0x100P", 4, 0x12345678)
```

5.2.2.43 memblock

Description Read or write a single value to memory.

Signature: memblock(address, addressOrCount, dataSize, *args)

Arguments: address -- A valid address, expressed as a string.

addressOrCount -- If a string, then this is the "to" address marking the end of the memory block (inclusive); otherwise, this is the number of elements to read or write.

dataSize -- Size of each element to read or write, in bytes (1 = byte, 2 = word, 4 = dword, 8 = qword). A value of 0 means use an optimized read/write and assume the count or range is in bytes.

*args -- If not empty, specifies one or more values to write to memory. If the count or "to" address is greater than the number of values, the specified values form a pattern that is repeated through the memory block.

Returns: Return a BitData object containing the values from memory if reading the memory; otherwise, returns None.

Remarks: Using a dataSize of 0 will optimize the memory reads and writes and assume the access is in bytes. However, the optimized access may not work properly when accessing a block of memory-mapped I/O, where the system enforces a smaller size (for example, 32 bits).

Usage:

```
>>> itp.threads[0].memblock("0x100P", 4, 1)
>>> itp.threads[0].memblock("0x100P", "0x110P", 4)
>>> itp.threads[0].memblock("0x100P", 4, 1, 0x12, 0x34, 0x56, 0x78)
>>> itp.threads[0].memblock("0x100P", "0x110P", 4, 0x12345678)
```

5.2.2.44 memdump

Description Display the contents of a block of memory.

Signature: memdump(address, addressOrCount, dataSize)

Arguments: address -- A valid address, expressed as a string.

addressOrCount -- If a string, then this is the "to" address marking the end of the memory block (inclusive); otherwise, this is the number of elements to read.

dataSize -- Size of each element to read or write, in bytes (1 = byte, 2 = word, 4 = dword, 8 = qword). A value of 0 means use an optimized read and assume the count or range is in bytes.

Returns: None.

Using a dataSize of 0 will optimize the memory reads and format the data as bytes. However, the optimized reads may not work properly when reading a block of memory-mapped I/O, where the system enforces a smaller read size (for example, 32 bits).

Usage:

```
>>> itp.threads[0].memdump("0x100P", 4, 1)
>>> itp.threads[0].memdump("0x100P", "0x110P", 4)
```

5.2.2.45 memload

Description Fill a range of target memory with the contents of a host file, truncating or repeating the file contents as necessary.

Signature: memload(filename, address, addressOrCount = 0)

Arguments: filename -- Filename of the object file where the data comes from.

address -- Address of the first byte to write to.

addressOrCount -- If a string, then this is the "to" address marking the end of the memory block (inclusive). If a number, this is the length of the target memory range, in bytes. If not specified or 0, dictates to read the full contents of the file into memory.

Returns: None.

Remarks: A file smaller than the specified range has its contents repeated throughout the range.

Usage:

```
>>> # Note usage of "raw" string notation (leading 'r') where backslashes
>>> # are used in the filename:
>>> itp.threads[0].memload(r"c:\temp\file.bin","0x100P")
>>> itp.threads[0].memload( "c:/temp/file.bin","0x100P", 4)
>>> itp.threads[0].memload(r"c:\temp\file.bin","0x100P", "0x110P")
```

5.2.2.46 memsave

Description Save a range of target memory to a host file.

Signature: memsave(filename, address, addressOrCount = None, overwrite = False)

Arguments: filename -- Filename of the object file where the data is to be stored.

address -- Address of the first byte to read from.

addressOrCount -- If a string, then this is the "to" address marking the end of the memory block (inclusive). If a number, this is the length of the target memory range, in bytes.

overwrite -- True if to overwrite the file if it exists; otherwise, an error is raised.

Returns: None.

Usage:

```
>>> # Note usage of "raw" string notation (leading 'r') where backslashes
>>> # are used in the filename:
>>> itp.threads[0].memsave(r"c:\temp\file.bin","0x1000", "0x1100")
>>> itp.threads[0].memsave( "c:/temp/file.bin","0x100P", 4, True)
>>> itp.threads[0].memsave(r"c:\temp\file.bin","0x100P", "0x110P", True)
```

5.2.2.47 msr

Description Return or change a value from a processor's model-specific registers

Signature: msr(registerIndex, newValue = None)

Arguments: registerIndex -- A valid register address (an index). newValue -- If specified, the value to write to the msr, either an integer or a BitData object.

Returns: Return a BitData object containing the value from the msr if reading the msr; otherwise, returns nothing.

Usage:

```
>>> itp.threads[0].msr(186)
>>> itp.threads[0].msr(186, 42)
>>> itp.threads[0].msr(186, BitData(42))
```

5.2.2.48 port

Description Retrieve or change the contents of an 8-bit IA32 I/O port.

Signature: port(portNumber, newValue = None)

Arguments: portNumber -- A port number in the target processor I/O space. The port number value exists in the range 0x00 to 0xffff.

newValue -- If specified, an 8-bit value to write to the port.

Returns: If reading the port, returns a BitData object containing the 8-bit value from the port; otherwise, returns nothing.

Remarks: Use the port command for 8-bit read and write access to the target system I/O space.

Usage:

```
>>> itp.threads[0].port(0x80)
>>> itp.threads[0].port(0x80, 0x12)
>>> itp.threads[0].port(BitData(16,128))
>>> itp.threads[0].port(BitData(16,128), 0xcode)
```

5.2.2.49 probemode

Description Enter or exit probe mode via the PROBEMODE TAP instruction on the multi-core processor device associated with the specified thread.

Signature: probemode(dataValue)

Arguments: dataValue -- 32-bit value (or BitData object containinga 32-bit value) to be shifted into the PROBEMODE TAP data register. Each bit set indicates the corresponding core on the device should enter probe mode; otherwise, if the bit is clear then the core will exit probe mode.

Returns: None.

Usage:

```
>>> # Enter probe mode on cores 0 and 1 on processor associated with thread
 0.
>>> itp.threads[0].probemode(0x3)
>>> # Enter probe mode on core 0 and exit probe mode on core 1.
>>> itp.threads[0].probemode(0x1)
```

5.2.2.50 readpdr0

Description Retrieve the 64 bit probe data register (PDR) associated with thread 0.

Signature: readpdr0()

Arguments: None.

Returns: A BitData object containing a 64-bit value from the PDR on thread 0.

Usage:

```
>>> itp.threads[0].readpdr0()
```

5.2.2.51 readpdr1

Description Retrieve the upper 32 bits of the probe data register (PDR) associated with thread 1.

Signature: readpdr1()

Arguments: None.

Returns: A BitData object containing a 64-bit value from the PDR on thread 1.

Usage:

```
>>> itp.threads[0].readpdr1()
```

5.2.2.52 readpdrh

Description Retrieve the upper 32 bits of the probe data register (PDR) on the core associated with the thread.

Signature: readpdrh()

Arguments: None.

Returns: A BitData object containing a 32-bit value from the PDR.

Usage:

```
>>> itp.threads[0].readpdrh()
```

5.2.2.53 readpdrl

Description Retrieve the lower 32 bits of the probe data register (PDR) on the core associated with the thread.

Signature: readpdrl()

Arguments: None.

Returns: A BitData object containing a 32-bit value from the PDR.

Usage:

```
>>> itp.threads[0].readpdrl()
```

5.2.2.54 regs

Description Display a common set of architectural registers and their values.

Arguments: None.

Returns: None.

Remarks: Use the regs command to display the contents of the general registers and the extended flag bits.

Usage:

```
>>> itp.threads[0].regs()
```

5.2.2.55 set_debugoptin

Description Sets the Debug opt-in flag in the TCS structure, so that the enclave can be

Signature: set_debugoptin(address)

Arguments: address -- A valid address, expressed as a string/unicode/Address

Returns: None

Usage:

```
>>> itp.threads[0].set_debugoptin("0x7ac0be1101")
```

5.2.2.56 status

Description Display the status of ITP, the target system, and the device that contains the thread being used.

Signature: status(packageName = None)

Arguments: packageName -- Optional name of a TAP package.

Returns: None.

Usage:

```
>>> itp.threads[0].status()
>>> itp.threads[0].status("Package1")
```

5.2.2.57 step

Description Perform a single step of program execution.

Signature: step(stepType = None, steps = None)

Arguments: stepType -- A value from the itp.StepType enumeration specifying the type of step to take. If not specified, the 'itp.StepType.into' option is used.

steps -- Number of steps to take before stopping. If not specified, defaults to 1 step.

Returns: None.

Remarks: Use the step command to step your program at the current execution point, temporarily disabling all breakpoint specifications. The step command has options to take either an instruction step (itp.StepType.into),

a step out of the current function (itp.StepType.out), a step past the next function as a single instruction (itp.Steptype.out), or a branch step (itp.StepType.branch). Using the step command without options is the same as using the itp.StepType.into option.

Usage:

```
>>> itp.threads[0].step() # step 1
>>> itp.threads[0].step(itp.StepType.into)
>>> itp.threads[0].step(itp.StepType.over, 4) # step over statements 4 times
```

5.2.2.58 submitpir

Description Submit (execute) processor instructions to the processor in probe mode using the SUBPIR TAP command.

Signature: submitpir

Arguments: None.

Returns: None. Exceptions:

Usage:

```
>>> itp.threads[0].submitpir()
>>> itp.threads[0].submitpir()
```

5.2.2.59 tapstatus

Description Display the TAP status bits from internal TAP state and from various units sent to the TAP associated with this thread.

Signature: tapstatus()

Arguments: None.

Returns: None.

Remarks: To get the raw bits, use the tapstatus() function.

Usage:

```
>>> itp.threads[0].tapstatus()
```

5.2.2.60 tapstatusraw

Description Retrieve the TAP status bits from internal TAP state and from various units sent to the TAP associated with this thread.

Signature: tapstatusraw()

Arguments: None.

Returns: A BitData object containing the status bits.

Remarks: To see the bits in a formatted display, use the tapstatusformatted() function.

Usage:

```
>>> itp.threads[0].tapstatusraw()
```

5.2.2.61 uallclkdis

Description Disable all clocks (stop forcing all clocks to run) on uncore assocaited with the thread.

Signature: uallclkdis()

Arguments: device -- Device ID or alias of the device to affect. Can also be a Node object.

Returns: None.

Usage:

```
>>> itp.threads[0].uallclkdis()
```

5.2.2.62 uallclken

Description Enable all clocks (force all clocks to run) on the uncore associated with the thread.

Signature: uallclken()

Arguments: None.

Returns: None.

Usage:

```
>>> itp.threads[0].uallclken()
```

5.2.2.63 ucrb

Description Read/write an uncore control register bus register using 32-bit values.

Signature: ucrb(address, newValue = None)

Arguments: address -- Bits 13:0 contain the number of the uncore control bus register to access. Bits 16:15 contain the core ID. Bit 14 contains the thread ID. newValue -- If not None, specifies a 32-bit value to write to the register.

Returns: A 32-bit value from the uncore control register bus register if reading. Returns None if writing to the register (newValue is not None).

Usage:

```
>>> itp.threads[0].ucrb(0x4010)
>>> itp.threads[0].ucrb(0x4010, 0x2110)
```

5.2.2.64 ucrb64

Description Read/write an uncore control register bus register using 64-bit values.

Signature: ucrb64(address, newValue = None)

Arguments: address -- Bits 13:0 contain the number of the uncore control bus register to access. Bits 16:15 contain the core ID. Bit 14 contains the thread ID.
newValue -- If not None, specifies a 64-bit value to write to the register.

Returns: A 64-bit value from the uncore control register bus register if reading. Returns None if writing to the register (newValue is not None).

Usage:

```
>>> itp.threads[0].ucrb64(0x4010)
>>> itp.threads[0].ucrb64(0x4010, 0x2110231232)
```

5.2.2.65 unloadsymbols

Description Remove all symbols from the symbol table for the specified thread.

Signature: unloadsymbols()

Arguments: None.

Returns: None.

Usage:

```
>>? itp.threads[0].unloadsymbols()
```

5.2.2.66 upload

Description Save contents of selected portions of target memory to a host file.

Signature: upload(filename, address, addressOrCount = None, overwrite = None)

Arguments: filename -- Filename of the object file where the information is stored.

address -- Address of the first byte to read from.

addressOrCount -- If a string, then this is the "to" address marking the end of the memory block; otherwise, this is the number of elements to read.

overwrite -- True if to overwrite the file if it exists; otherwise, an error is raised.

Returns: None.

Usage:

```
>>> itp.threads[0].upload("filename.bin", "0x1000", "0x1100")
>>> itp.threads[0].upload("filename.bin", "0x1000", 256, True)
```

5.2.2.67 utapstatus

Description Display the TAP status bits from the associated uncore device as a formatted string.

Signature: utapstatus()

Arguments: None.

Returns: None.

Remarks: To get the raw bits, use the utapstatusraw() function.

The display shows the logical bit indices if there is more than one bit or the first bit is not 0. The physical mapping into the parent bits is always shown. The following example shows the tapstatus register with two fields. The PMRDY field has an implied logical mapping of [0:0] and a physical mapping to bit 88 in the tapstatus register. The PhySlice field has an explicit logical mapping of [7:0] and a physical mapping to bits 34 through 27.

tapstatus(109:0) = [109:0] 0x00000000000000878000000000000 PMRDY = [88] 0x0 PhySlice(7:0) = [34:27] 0x00

Usage:

```
>>> itp.threads[0].utapstatus(0)
```

5.2.2.68 utapstatusraw

Description Retrieve the TAP status bits from the associated uncore device.

Signature: utapstatusraw()

Arguments: None.

Returns: A BitData object containing the status bits.

Remarks: To see the bits in a formatted display, use the utapstatus() function.

Usage:

```
>>> itp.threads[0].utapstatusraw(0)
```

5.2.2.69 vmcscap

Description Retrieve VMCS capability information.

Signature: vmcscap(encoding)

Arguments: encoding -- The encoding to examine.

Returns: An array of BitData objects containing 64-bit values filled in with the capabilities based on the encoding.

Remarks: The capability data is returned only for components that have capability data associated with them. Requests for capability information from components that have no capability data associated with them return an error. The capability array will contain either one or two elements. If there are two elements, the 0th element specifies the 0-settings and the 1st element specifies the 1-settings. 0-settings indicate bits that can be set to 0 and similarly 1-settings indicate bits that can be set to 1.

Usage:

```
>>> itp.threads[0].vmcscap(1)
```

5.2.2.70 vmcsinfo

Description Display miscellaneous VMCS information for a specific thread.

Signature: vmcsinfo(infoKind)

Arguments: infoKind -- A string specifying the kind of vmcs information to return. Can be one of the following: "vmcsrevision" - revision number of the VMCS supported by the processor. "vmcssize" - size of the VMCS supported by the processor. "vmxmode" - current VMX mode. "ilp" - Is processor the Initiating Logical Processor. "secenter" - Has SEnter been executed on the processor. "enterac" - Has EnterAC been executed on the processor. "enteraccs" - Has EnterACCS been executed on the processor.

Returns: A BitData object containing a value based on the kind of information requested:

"vmcsrevision": Revision number of the VMCS supported by the processor. "vmcssize" : Size of the VMCS supported by the processor. "vmxmode" : The current VMX mode: 0 = VMX off 1 = Root mode 2 = Monitor mode (deprecated) 3 = Guest mode 11 = Root parallel SMM handler 13 = Guest parallel SMM handler "ilp" : 1 if the processor is the Initiating Logical Processor. "secenter" : 1 if SEnter has been executed on the processor. "enterac" : 1 if EnterAC has been executed on the processor. "enteraccs" : 1 if EnterACCS has been executed on the processor.

Usage:

```
>>> itp.threads[0].vmcsinfo("vmcsrevision")
>>> itp.threads[0].vmcsinfo("vmcssize")
```

5.2.2.71 vmcsptr

Description Retrieve the specified VMCS pointer.

Signature: vmcsptr(ptrKind, newValue = None)

Arguments: ptrKind -- A string specifying the type of pointer to view or set. Can be one of the following:

"wptr" - The working pointer.

"cptr" - The controlling pointer.

"smmcvp" - The SMM CVP pointer.

newValue -- A 64-bit value (or a BitData object containing a 64-bit value) to assign to the specified kind of vmcs pointer.

Returns: A BitData object containing the current 64-bit value associated with the specified vmcs pointer. If no virtual machine is active, returns None. If newValue is specified, None is returned.

Remarks: Working Pointer: The monitor or sub-monitor software establishes a working VMCS before a new VMX mode is entered. This working VMCS is referenced through the working pointer. The working VMCS is used during the VMLAUNCH and its value is saved into the controlling pointer after a successful VMLAUNCH.

Controlling Pointer: The controlling pointer is invalid in VMX root mode as the processor functionality in VMX root is almost the same as it is outside VMX operation. In all other modes, VMX operation is controlled by the controlling VMCS, which is referenced through the controlling pointer.

SMMVCP: The SMMCVP pointer is valid when a VMCS structure is present for a parallel SMM monitor and/or guest.

Usage:

```
>>> workingptr = itp.threads[0].vmcsptr("wptr")
>>> itp.threads[0].vmcsptr("wptr", itp.threads[0].vmcsinfo("cptr")
>>> itp.threads[0].vmcsptr("wptr", workingptr)
```

5.2.2.72 vmcsread

Description Read VMCS component data from the current working pointer on the specified encoding.

Signature: vmcsread(encoding, data, errorData)

Arguments: encoding -- A single 32-bit value or a list of 32-bit values specifying the encoding to read from.

data -- A list filled in with a BitData object containing a 64-bit value for each encoding provided.

errorData -- A list filled in with a BitData object containing a 32-bit error value for each encoding provided. A BitData containing 0 means no error.

Returns: None.

Remarks: The vmcsread command by default reads the component data corresponding to the working pointer. If a pointer other than the working pointer is to be used for the read operation, save the value of the working pointer, update the value of the working pointer with the required pointer, execute the vmcsread command and finally restore the value of the working pointer.

The error list will contain the result of the read operation for the corresponding encoding parameter. A zero value indicates success and a positive value indicates failure. For example, invalid field encodings specified in the vmcsread command returns an error.

VMCS Component Structure

The VMCS contains the processor state during VMX operation. The components of the VMCS can be grouped into the following classes:

1. Guest-state area

2. Host-state area

3. VM-execution control fields

4. VM-exit control fields

5. VM-entry control fields

6. VM-exit information area

Each of the classes contains fields that in turn contain data relevant to the class. Each field in the VMCS structure has an encoding that uniquely identifies the field. This encoding is specified during vmcsread operation to read the data contained by the field. Certain fields are supported only by some implementations. Attempts to read these fields will fail.

Error Codes

0x1FE1 = Invalid field encoding

0x1FE2 = Processor not halted

0x1FE3 = Processor not in LT mode

0x1FE4 = Invalid pointer type

0x1FE6 = Instruction failed

Usage:

```
Example 1 (single encoding):
    >>> encoding = 1
    >>> data = [int()] * 1
    >>> errorData = [int()] * 1
    >>> itp.threads[0].vmcsread(encoding, data, errorData)
```

```
    >>> print("Encoding = %d, value = %s, error = %s"%(encoding, data[0],
     errorData[0]))
Example 2 (multiple encodings):
    >>> encoding = (1, 2, 3)
    >>> data = [int()] * len(encoding)
    >>> errorData = [int()] * len(encoding)
    >>> itp.threads[0].vmcsread(encoding, data, errorData)
    >>> for encoding in encoding:
    >>>     for d in zip(data, errorData):
    >>>         print("Encoding = %d, value = %s, error = %s"%(encoding,
     d[0], d[1]))
```

5.2.2.73 vmcswrite

Description Write VMCS component data to the current working pointer on the specified encoding.

Signature: vmcswrite(encoding, data, errorData)

Arguments: encoding -- A single 32-bit value or a list of 32-bit values specifying the encoding to read from.

data -- A single 64-bit value (or a BitData object containing a 64-bit value) or an array of 64-bit values (or BitData objects containing 64-bit values) to write to an encoding.

errorData -- A list filled in with a BitData object containing a 32-bit error value for each encoding provided. A BitData containing 0 means no error.

Returns: None.

Remarks: The vmcswrite command by default updates the component(s) corresponding to the working pointer. If a pointer other than the working pointer is to be used in the update operation, save the value of the working pointer, update the value of the working pointer with the required pointer, execute the vmcswrite command and finally restore the value of the working pointer.

The error variable contains the result of the write operation for the corresponding encoding parameter. A zero value indicates success and a positive value indicates failure. For example, invalid field encodings specified in the vmcswrite command returns an error. The possible error values are specified in the error section below.

VMCS Component Structure

The VMCS contains the processor state during VMX operation. The components of the VMCS can be grouped into the following classes:

1. Guest-state area

2. Host-state area

3. VM-execution control fields

4. VM-exit control fields

5. VM-entry control fields

6. VM-exit information area

Each of the classes contains fields that in turn contain data relevant to the class. Each field in the VMCS structure has an encoding that uniquely identifies the field. This encoding is specified during vmcsread operation to read the data contained by the field. Certain fields are supported only by some implementations. Attempts to read these fields will fail.

Error Codes

0x1FE1 = Invalid field encoding

0x1FE2 = Processor not halted

0x1FE3 = Processor not in LT mode

0x1FE4 = Invalid pointer type

0x1FE6 = Instruction failed

Usage:

```
Example 1 (single encoding):
    >>> encoding = 1
    >>> data = 0x10000000
    >>> errorData = [int()] * 1
    >>> itp.threads[0].vmcswrite(encoding, data, errorData)
    >>> print("Error (0 means success) = %s"%errorData[0])
Example 2 (multiple encodings):
    >>> encoding = (1, 2, 3)
    >>> data = [0x10000000, 0x20000000, 0x30000000]
    >>> errorData = [int()] * len(encoding)
    >>> itp.threads[0].vmcswrite(encoding, data, errorData)
    >>> for d in zip(encoding, errorData):
    >>>     print("Encoding = %d, error = %s"%(d[0], d[1]))
```

5.2.2.74 wait

Description Suspend script execution until a break has occurred on the target for the specific thread or the specified number of seconds has passed.

Signature: wait(threadNode, seconds = None)

Arguments: seconds -- Number of seconds to wait. If set to None, waits forever.

Returns: If seconds are specified, return True if the wait returned before the time-out expired. If seconds are not specified, always returns None.

Usage:

```
>>? itp.threads[0].wait()  # wait forever
>>? if itp.threads[0].wait(3): # wait 3 seconds
>>?     print("A break was encountered.");
>>? if itp.threads[0].cv.isrunning:
>>?     print("thread did not hit breakpoint.")
```

5.2.2.75 wbinvd

Description Write back modified data from the IA32 caches to main memory and invalidates the caches.

Signature: wbinvd()

Arguments: None.

Returns: None.

Remarks: This ITP command causes the processor to execute a WBINVD assembly instruction.

It writes back all modified cache lines in all the processor's internal caches to main memory and invalidates (flushes) them. It also issues additional special-function bus cycles that direct external caches to also write back modified data and get invalidated.

Refer to the Intel Architecture SW Developers Manual (Volume 2B) for more details on this instruction.

Usage:

```
>>? itp.threads[0].wbinvd()
```

5.2.2.76 wport

Description Retrieve or change the contents of a 16-bit IA32 I/O port.

Signature: wport(portNumber, newValue = None)

Arguments: portNumber -- A port number in the target processor I/O space. The port number value exists in the range 0x00 to 0xfffe. newValue -- If specified, a 16-bit value to write to the port.

Returns: If reading the port, returns a BitData object containing the 16-bit value from the port; otherwise, returns nothing.

Remarks: Use the wport command for 16-bit read and write access to the target system I/O space. Note: Not all I/O ports support 16-bit access.

Usage:

```
>>> itp.threads[0].wport(0x80)
>>> itp.threads[0].wport(0x80, 0x1234)
>>> itp.threads[0].wport(BitData(16,128)
```

```
>>> itp.threads[0].wport(BitData(16,128), 0xcode)
```

5.2.2.77 writepir

Description Write processor instructions to the processor in probe mode using the WRiTEPIR TAP command.

Signature: writepir(value, *args)

Arguments: value -- Either a byte or a string. A byte is typically the first byte of a processor instruction. A string contains a processor instruction (for example, "mov eax,ebx"). *args -- Zero or more additional bytes. If 'value' is a string, then these remaining arguments are ignored.

Returns: None. Exceptions:

TypeError: the types in *args does not match the type in 'value'. ValueError: The value in 'value' or *args do not fit in a byte.

Usage:

```
>>> itp.threads[0].writepir(0x66, 0xb9, 0xff, 0xff)
>>> itp.threads[0].writepir("mov eax, ebx")
```

5.2.2.78 wrsubpir

Description Write and submit (execute) processor instructions to the processor in probe mode using the WRSUBP TAP command.

Signature: wrsubpir(value, *args)

Arguments: value -- Either a byte or a string. A byte is typically the first byte of a processor instruction. A string contains a processor instruction (for example, "mov eax,ebx"). *args -- Zero or more additional bytes. If 'value' is a string, then these remaining arguments are ignored.

Returns: None. Exceptions:

TypeError: the types in *args does not match the type in 'value'. ValueError: The value in 'value' or *args do not fit in a byte.

Usage:

```
>>> itp.threads[0].wrsubpir(0x66, 0xb9, 0xff, 0xff)
>>> itp.threads[0].wrsubpir("mov eax, ebx")
>>> itp.threads[0].wrsubpir("clflush [0x150000]")
```

5.2.2.79 xmregs

Description Display the contents of the Intel Extended Memory 64 Technology

Signature: xmregs()

Arguments: None.

Returns: None.

Remarks: Use the xmregs command to display the contents of the general 64-bit registers and the extended flag bits.

Usage:

```
>>> itp.threads[0].xmregs()
```

5.2.3 Group Commands

5.2.3.1 brdisable

Description Disable an existing breakpoint on all threads in the specified core group.

Signature: brdisable(address, breakType)

Arguments: address -- Targeted address for the break. Either a string or Address type. breakType -- Targeted break type. Must be a string matching one of the following: "sw" -- Software breakpoint "exe" -- Hardware execution breakpoint "exe global" -- Hardware execution breakpoint (Global is across all tasks) "exe local" -- Hardware execution breakpoint (Local is for current task) "acc" -- Hardware memory access breakpoint "acc global" -- Hardware memory access breakpoint (Global is across all tasks) "acc local" -- Hardware memory access breakpoint (Local is for current task) "wr" -- Hardware memory write breakpoint "wr global" -- Hardware memory write breakpoint (Global is across all tasks) "wr local" -- Hardware memory write breakpoint (Local is for current task) "io" -- Hardware I/O access breakpoint

Returns: None.

Usage:

```
>>> itp.groups[0].brdisable("0x100P", "io")
>>> itp.groups[0].brdisable("0x1200L", "exe global")
```

5.2.3.2 brenable

Description Enable an existing breakpoint on all threads in the specified core group.

Signature: brenable(address, breakType)

Arguments: address -- Targeted address for the break. Either a string or Address type. breakType -- Targeted break type. Must be a string matching one of the following: "sw" -- Software breakpoint "exe" -- Hardware execution breakpoint "exe global" -- Hardware execution breakpoint (Global is across all tasks) "exe local" -- Hardware execution breakpoint (Local is for current task) "acc" -- Hardware memory access breakpoint "acc global" -- Hardware memory access breakpoint (Global is across all tasks) "acc local" --

Hardware memory access breakpoint (Local is for current task) "wr" -- Hardware memory write breakpoint "wr global" -- Hardware memory write breakpoint (Global is across all tasks) "wr local" -- Hardware memory write breakpoint (Local is for current task) "io" -- Hardware I/O access breakpoint

Returns: None.

Usage:

```
>>> itp.groups[0].brenable("0x100P", "io")
>>> itp.groups[0].brenable("0x1200L", "exe global")
```

5.2.3.3 brget

Description Retrieve all breakpoints associated with the specified core group.

Signature: brget()

Arguments: None.

Returns: A "StringWrapper" (see datatypes module) of the breakpoints for all threads in the core group. The string will be empty if there are no breakpoints associated with the group.

Usage:

```
>>> itp.groups[0].brget()
```

5.2.3.4 brnew

Description Add a new breakpoint to all threads in the specified core group.

Signature: brnew(address, breakType)

Arguments: address -- Targeted address for the break. Either a string or Address type. breakType -- Targeted break type. Must be a string matching one of the following: "sw" -- Software breakpoint "exe" -- Hardware execution breakpoint "exe global" -- Hardware execution breakpoint (Global is across all tasks) "exe local" -- Hardware execution breakpoint (Local is for current task) "acc" -- Hardware memory access breakpoint "acc global" -- Hardware memory access breakpoint (Global is across all tasks) "acc local" -- Hardware memory access breakpoint (Local is for current task) "wr" -- Hardware memory write breakpoint "wr global" -- Hardware memory write breakpoint (Global is across all tasks) "wr local" -- Hardware memory write breakpoint (Local is for current task) "io" -- Hardware I/O access breakpoint

Returns: None.

Usage:

```
>>> itp.groups[0].brnew("0x100P", "io")
```

```
>>> itp.groups[0].brnew("0x1200L", "exe global")
```

5.2.3.5 brremove

Description Remove an existing breakpoint from all threads in the specified core group.

Signature: brremove(address, breakType)

Arguments: address -- Targeted address for the break. Either a string or Address type. breakType -- Targeted break type. Must be a string matching one of the following: "sw" -- Software breakpoint "exe" -- Hardware execution breakpoint "exe global" -- Hardware execution breakpoint (Global is across all tasks) "exe local" -- Hardware execution breakpoint (Local is for current task) "acc" -- Hardware memory access breakpoint "acc global" -- Hardware memory access breakpoint (Global is across all tasks) "acc local" -- Hardware memory access breakpoint (Local is for current task) "wr" -- Hardware memory write breakpoint "wr global" -- Hardware memory write breakpoint (Global is across all tasks) "wr local" -- Hardware memory write breakpoint (Local is for current task) "io" -- Hardware I/O access breakpoint

Returns: None.

Usage:

```
>>> itp.groups[0].brremove("0x100P", "io")
>>> itp.groups[0].brremove("0x1200L", "exe global")
```

5.2.3.6 go

Description Starts execution on a core group.

Signature: go()

Arguments: None.

Returns: None.

Usage:

```
>>> itp.groups[0].go()
```

5.2.3.7 halt

Description Halts execution on a core group.

Signature: halt()

Arguments: None.

Returns: None.

Usage:

```
>>? itp.groups[0].halt()
```

6 DAL CLI Control Variables

A control variable alters the behavior of the software stack in some fashion. ITP used control variables for a great many things, including manipulating hardware. The Python CLI supports a number of ITP control variables and some ITP control variables that manipulated hardware were converted to functions.

6.1 Global Control Variables

6.1.1 asmmode

Definition The IA32 instruction mode for the asm command.

Options: 'use16' (or 0)

'use32' (or 1)

'use64' (or 2)

Remarks: This setting is used only if the asm/dasm addresses are linear or physical. Use the asmmode control variable to display or set the mode for the asm command (section 5.54) when working with the IA32 instruction set. Entering the control variable without an option returns the current setting. When displayed, the values are 0 for use16, 1 for use32, and 2 for EM64T.

When set to 'use16' or 'use32', the Python CLI interprets the instruction mode as IA32 16-bit or 32-bit, respectively. When asmmode is set to 'use64', the Python CLI interprets the instructions using Intel Extended Memory 64 Technology. You can use the asmmode control variable to change the asm command's instruction mode for a given processor whether the processor is running or stopped.

Any setting of the asmmode control variable affects only the single thread specified. To affect all threads, use the itp.cv.asmmode control variable. Note: The asmmode control variable override is used only if the address provided to the asm() command is linear or physical; otherwise, the current addressing mode takes precedence and dictates the mode to assemble/disassemble.

Usage:

```
>>>itp.cv.asmmode
use16
>>>itp.cv.asmmode=0
>>>itp.cv.asmmode='use32'
```

6.1.2 autoauthorize

Definition Enable automatic authorize

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.autoauthorize = 1
>>> itp.autoauthorize
True
```

6.1.3 autounlock

Definition Enable/disable automatic unlock of each device in the target system during DAL invocation and when target power restore events are detected.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.autounlock=0
>>> itp.cv.autounlock
False
```

6.1.4 bootstall

Definition Display or change a setting that controls the bootstall (hook3) setting when a Powergood de-assertion event occurs.

Options: 0 (or 'disabled')

1 (or 'enabled')

Usage:

```
>>> itp.cv.bootstall=0
>>> itp.cv.bootstall
[HSW_UC0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T1] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T1] disabled (hardware ID: 0xdeadbeef)
```

6.1.5 boundaryscanvalidate

Definition Enable/disable verification of the idcodes of devices and sub-devices discovered by the DAL during Autoconfig/Reconfig.

Options: False (or 0)

True (or 1)

Usage:

```
>>>itp.cv.boundryscanvalidate
False
>>>itp.cv.boundryscanvalidate=0
>>>itp.cv.boundryscanvalidate=True
```

6.1.6 breakall

Definition Display or change the synchronous run control state in multiprocessor systems.

Options: 0 (or 'off')

1 (or 'global')

Remarks: Use the breakall control variable to display or change whether all target processors in a system or all processors connected to a debug port start and stop together in a multiprocessor system. Entering the control variable without an option returns the current setting. The default setting for breakall is 1 ('global').

If breakall is set to 1 or 'global', all processors in a multiprocessor system start and stop running at the same time in all cases except when performing a step operation. In this mode, if one processor breaks for a user halt or a breakpoint is reached all processors in the system will stop. When a processor is started, all processors will be started together.

If breakall is set to 0 or 'off', each processor in a multiprocessor system can be controlled independently of the others. A viewpoint override or the current viewpoint in which the go command is used and various control variables will determine the run/stop status for each processor.

Usage:

```
>>>itp.cv.breakall
Global
>>>itp.cv.breakall=0
>>>itp.cv.breakall='global'
```

6.1.7 brkstatus

Definition Display a code for the cause of the most recent break in execution. Read-only

Options: None.

Remarks: Use the brkstatus read-only control variable to display a numerical value representing the cause of the most recent target processor break in execution. The returned value is a 32-bit value. The upper word is the processor's ID in the boundary scan chain, and the lower word is the break status value. The following codes can be seen in the brkstatus control variable.

Note: Unlike ITP, the brkstatus value does not include the processor ID in the upper 16 bits of the returned value.

No break cause 0x0000

SMM entry 0x0003

SMM exit 0x0004

INIT 0x0005

RESET 0x0006

External halt trigger 0x0009

Halt command 0x000a

Software breakpoint 0x000c

Debug register break on I/O reads or writes (hw break 0) 0x000d

Debug register break on instruction execution only(hw break 0) 0x100d

Debug register break on data writes(hw break 0) 0x200d

Debug register break on data reads or writes(hw break 0) 0x300d

Debug register break on I/O reads or writes (hw break 1) 0x400d

Debug register break on instruction execution only(hw break 1) 0x500d

Debug register break on data writes(hw break 1) 0x600d

Debug register break on data reads or writes(hw break 1) 0x700d

Debug register break on I/O reads or writes (hw break 2) 0x800d

Debug register break on instruction execution only(hw break 2) 0x900d

Debug register break on data writes(hw break 2) 0xa00d

Debug register break on data reads or writes(hw break 2) 0xb00d

Debug register break on I/O reads or writes (hw break 0) 0xc00d

Debug register break on instruction execution only(hw break 3) 0xd00d

Debug register break on data writes(hw break3) 0xe00d

Debug register break on data reads or writes(hw break 3) 0xf00d

Branch step 0x000e

Unknown 0x000f

Write to the debug regs which were protected by the GD bit setting 0x0010

Other processor caused break upon startup IPI 0x0018

Single step 0x0019

LT SIPI 0x0020

Breakall 0x0022

CC1 entry / exit 0x0023

CC2 entry / exit 0x0024

CC3 entry / exit 0x0025

CC4 entry / exit 0x0026

CC5 entry / exit 0x0027

CC6 entry / exit 0x0028

C6 save / restore 0x0029

System cluster break 0x0031

Debug trap 0x0032

Fault 0x0033

Machine check break 0x0034

Error in shutdown 0x0035

Shutdown 0x0036

Sleep state 0x0037

Suspend instruction 0x0038

Sleep state and Halt instruction 0x0039

Sleep and error shutdown 0x003a

Sleep and shutdown 0x003b

Sleep state and SIPI loop 0x003c

Sleep state and MWAIT 0x003d

Sleep state and MWAIT_TC_FLUSHED 0x003e

Sleep state and SENTER 0x003f

Data breakpoint 0x007d

VMENTER 0x0082

VMEXIT 0x0083

VMLAUNCH 0x0085

VMLAUNCH from STM to OSV for SMM 0x0086

VMLAUNCH from SMM host to SMM guest 0x0087

VMLAUNCH from STM to OSV for service call 0x0088

VMLAUNCH from OSV host to OSV guest 0x0089

VMENTRY from STM to OSV for SMM 0x0090

VMENTRY from SMM host to SMM guest 0x0091

VMENTRY from STM to OSV for service call 0x0092

VMENTRY from OSV host to OSV guest 0x0093

Usage:

```
>>>itp.cv.brkstatus
[HSW_C0_T0] (0x000a) Halt Command break
[HSW_C0_T1] (0x000a) Halt Command break
[HSW_C1_T0] (0x000a) Halt Command break
[HSW_C1_T1] (0x000a) Halt Command break
[HSW_C2_T0] (0x000a) Halt Command break
[HSW_C2_T1] (0x000a) Halt Command break
[HSW_C3_T0] (0x000a) Halt Command break
[HSW_C3_T1] (0x000a) Halt Command break
>>> itp.threads[0].cv.brkstatus
(0x000a) Halt Command break
```

6.1.8 cause

Definition The cause of the most recent break in execution. Read-only

Options: None.

Remarks: Use the cause command when a break message has scrolled off the screen, or to give output from a user-defined function. The cause command displays the last break message for the thread.

Break information is returned for all variations of the step command (step into, step branch, step out, instruction step, and step over).

Usage:

```
>>> itp.cv.cause
[HSW_C0_T0] Halt Command break at 0xF000:000000000000FFF0
[HSW_C0_T1] Halt Command break at 0xF000:000000000000FFF0
[HSW_C1_T0] Halt Command break at 0xF000:000000000000FFF0
[HSW_C1_T1] Halt Command break at 0xF000:000000000000FFF0
[HSW_C2_T0] Halt Command break at 0xF000:000000000000FFF0
[HSW_C2_T1] Halt Command break at 0xF000:000000000000FFF0
[HSW_C3_T0] Halt Command break at 0xF000:000000000000FFF0
[HSW_C3_T1] Halt Command break at 0xF000:000000000000FFF0
>>> itp.threads[0].cv.cause
Halt Command break at 0xF000:000000000000FFF0
```

6.1.9 coregroups

Definition Display or change the coregroups used for 'global context' commands.

Options: 'default' (use highest priority coregroup available; this option cannot be combined with other options)

varied (depending on platform selected; please use the coregroup name strings associated with the coregroups defined by examining GroupDeviceRoot or 'itp.groups' from the CLI)

Remarks: On platforms where Heterogeneous Core Support (HCS) is enabled, this control variable will be used to specify which cores and threads the user desires to target with 'global context' commands. 'Global context' commands refers to those such as 'itp.halt()', as opposed to 'core context', 'thread context', or 'group context' commands such as 'itp.cores[0].halt()', 'itp.threads[0].halt()', and 'itp.groups[0].halt()'.

Please note that this control variable does not guarantee the desired coregroups will be targeted by the 'global context' commands. Only coregroups that are 'active' are targeted. 'active' groups are those that have at least one enabled thread and have been specified by the user. To determine which coregroups are 'active', please refer to the read-only 'coregroupsactive' control variable

Usage:

```
>>? itp.groups
[<NodeCoreGroup: GPC      (4 cores, 4 threads, 1 devicetype:  'CHT_AMT')>,
<NodeCoreGroup: GT       (1 core,  1 thread,  1 devicetype:  'CHT_GEN_GT_1_GUC')>,
<NodeCoreGroup: ISH      (1 core,  1 thread,  1 devicetype:  'CHT_ISH_LMT')>]
>>? itp.cv.coregroups
default
>>? itp.halt()
```

```
[AMT_M0_C0_T0]  Halt Command break at 0xF000:000000000000FFF0
[AMT_M0_C1_T0]  Halt Command break at 0xF000:000000000000FFF0
[AMT_M1_C0_T0]  Halt Command break at 0xF000:000000000000FFF0
[AMT_M1_C1_T0]  Halt Command break at 0xF000:000000000000FFF0
>>> itp.status()
Status for       : AMT_M0_C0_T0
Processor      : Halted
Processor mode : Real, Legacy
Status for       : AMT_M0_C1_T0
Processor      : Halted
Processor mode : Real, Legacy
Status for       : AMT_M1_C0_T0
Processor      : Halted
Processor mode : Real, Legacy
Status for       : AMT_M1_C1_T0
Processor      : Halted
Processor mode : Real, Legacy
Status for       : CHT_ISH_LMT_C0_T0
Processor      : Running
Processor mode : Unavailable while running
Status for       : CHT_GEN_GT_1_GUC_C0_T0
Processor      : Running
Processor mode : Unavailable while running
>>> itp.go()
>>? itp.cv.coregroups='GT'
>>? itp.halt()
[CHT_GEN_GT_1_GUC_C0_T0]  Halt Command break at 0xF000:000000000000FFF0
>>> itp.status()
Status for       : AMT_M0_C0_T0
Processor      : Running
Processor mode : Unavailable while running
Status for       : AMT_M0_C1_T0
Processor      : Running
Processor mode : Unavailable while running
Status for       : AMT_M1_C0_T0
Processor      : Running
Processor mode : Unavailable while running
Status for       : AMT_M1_C1_T0
Processor      : Running
Processor mode : Unavailable while running
Status for       : CHT_ISH_LMT_C0_T0
Processor      : Running
Processor mode : Unavailable while running
Status for       : CHT_GEN_GT_1_GUC_C0_T0
Processor      : Halted
Processor mode : Real
```

6.1.10 coregroupsactive

Definition Display the coregroups actually in use (please also refer to 'coregroups')

Options: None

Remarks: This is a read-only reflection of the 'coregroups' control variable, and will display the names of the coregroups among those specified that are 'active'. The 'active' coregroups are those whose cores and threads will actually be

used in 'global context' commands because they are available (i.e. in the device list) and have been specified by the user.

When 'coregroups' is set to the special value of 'default', the active coregroup name defined in the platform support tables will be displayed instead (e.g. 'GPC' instead of 'default').

Any coregroup name explicitly specified using 'coregroups' that is not available due to target system state (e.g. certain devices are not enabled) will not be displayed by 'coregroupsactive'. If no coregroups specified by 'coregroups' are available, then an empty string is returned and a warning 'message event is published.

Explicit 'coregroup' name strings can be found by examining GroupDeviceRoot or 'itp.groups' from the CLI)

Usage:

```
>>? itp.groups
[<NodeCoreGroup: GPC      (4 cores, 4 threads, 1 devicetype:  'CHT_AMT')>,
<NodeCoreGroup: GT       (1 core,  1 thread,  1 devicetype:  'CHT_GEN_GT_1_GUC')>,
<NodeCoreGroup: ISH      (1 core,  1 thread,  1 devicetype:  'CHT_ISH_LMT')>]
>>? itp.cv.coregroups
default
>>> itp.cv.coregroupsactive
GPC
```

6.1.11 debugprotect

Definition Enable/disable whether a break occurs when your program accesses a processor's debug registers.

Options: False (or 0)

True (or 1)

Remarks: Use the debugprotect control variable to display or change whether a break occurs when your program accesses a processor's debug registers. The default setting for debugprotect is False.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>>itp.cv.debugprotect
[HSW_C0_T0] False
[HSW_C0_T1] False
>>>itp.cv.debugprotect=1
>>>itp.thread[0].debugprotect=True
>>>itp.go();itp.halt()
[HSW_C0_T0]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C0_T1]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C1_T0]  Halt Command break at 0xF000:000000000000FFF0
```

```
[HSW_C1_T1]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C2_T0]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C2_T1]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C3_T0]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C3_T1]  Halt Command break at 0xF000:000000000000FFF0
>>>itp.thread[0].debugprotect
True
```

6.1.12 devicetreeindicator

Definition Indicator of number of times when device tree has been changed after re-configuration. Read-only

Options: None.

Usage:

```
>>>itp.cv.devicetreeindicator
0
```

6.1.13 disabletapstatuspollforprdyafterpir

Definition Disable Polling tapstatus for PRDY after PIR operation (will applicable only if PRDY pin is not wired or JTAG only mode is enabled)

Options: False (or 0)

True (or 1)

Notes:

Depending on the processor family, CPU and probe mode operation being done,

PIR execution may take longer time to complete due to power management, frequency throttling or, low processor speed.

As a result we might see functional issues providing incorrect results when PRDY pin is not wired or JTAG only mode is enabled

and this variable help to wait for PIR operaton completion by polling tapstatus for PRDY.

Usage:

```
>>> itp.cv.disabletapstatuspollforprdyafterpir
0
itp.cv.disabletapstatuspollforprdyafterpir=1
```

6.1.14 drprotect

Definition Enable/disable DAL taking exclusive control over the processor's debug registers.

Options: False (or 0)

True (or 1)

Usage:

```
>>>itp.cv.drprotect
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
>>> itp.threads[0].cv.drprotect=True
>>> itp.threads[0].cv.drprotect
True
```

6.1.15 earbreak

Definition Display or change a setting that controls the External Aligned Reset (EAR) setting when a Powergood de-assertion event occurs.

Options: 0 (or 'disabled')

1 (or 'enabled')

2 (or 'rearm')

3 (or 'step' - available only for supporting processors only)

Usage:

```
>>> itp.cv.earbreak=0
>>> itp.cv.earbreak
[HSW_UC0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T1] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T1] disabled (hardware ID: 0xdeadbeef)
```

6.1.16 enableoxmdebug

Definition Enable/disable subset of DFX registers which is made available on locked part.

Options: False (or 0)

True (or 1)

Remarks: Use the enableoxmdebug control variable to display or enable/disable if message channel registers are allowed on locked part. The default setting for enableoxmdebug is False.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>>itp.cv.enableoxmdebug
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
>>>itp.cv.enableoxmdebug=True
>>>itp.go();itp.halt
```

6.1.17 enclaveentrybreak

Definition Enable/disable breaking into probemode, when the processor enters secure enclave execution instructions.

Options: 'False' (or 0)

'True' (or 1)

Usage:

```
>>> itp.cv.enclaveentrybreak=0
>>> itp.cv.enclaveentrybreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.1.18 enclaveresumebreak

Definition Enable/disable breaking into probemode, when the processor resumes secure enclave execution instructions.

Options: 'False' (or 0)

'True' (or 1)

Usage:

```
>>> itp.cv.enclaveresumebreak=0
>>> itp.cv.enclaveresumebreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.1.19 enteracbreak

Definition Enable/disable breaking from processor execution on entry to authenticated code (AC).

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.enteracbreak=0
>>> itp.cv.enteracbreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.1.20 enterpmmethod

Definition Display or change setting that controls the method to enter probemode .

Options: '0' (or 'pin')

'1' (or 'uncore')

'2' (or 'core')

Usage:

```
>>> itp.cv.enterpmmethod
[HSW_C0_T0] pin
[HSW_C0_T1] pin
[HSW_C1_T0] pin
[HSW_C1_T1] pin
[HSW_C2_T0] pin
[HSW_C2_T1] pin
[HSW_C3_T0] pin
[HSW_C3_T1] pin
>>> itp.cv.enterpmmethod='core'
```

6.1.21 etrace

Definition Display or change a setting that controls the generation of execution history trace records.

Options: 0x0000 (or 'off'): Disable all forms of execution history generation and collection.

0x0001 (or 'btm'): The Branch Trace Messages will be generated on the front-side bus.

0x0002 (or 'lbr'): The Last Branch Record registers will be used to collect execution history.

0x0004 (or 'ler'): The Last Exception Record registers will be used to collect execution history.

0x0008 (or 'btb'): The Branch Trace Buffer will be used to collect execution history.

0x0010 (or 'rct'): ITP will used one of the other methods to seamlessly collect a specified amount

execution history. This is called Run Control Trace (rct).

0x0020 (or 'bts'): The Branch Trace Store will be used to collect execution history.

Usage:

```
>>> itp.cv.etrace
[HSW_C0_T0] off
[HSW_C0_T1] off
[HSW_C1_T0] off
[HSW_C1_T1] off
[HSW_C2_T0] off
[HSW_C2_T1] off
[HSW_C3_T0] off
[HSW_C3_T1] off
>>> itp.cv.etrace='btm'
```

6.1.22 exitacbreak

Definition Enable/disable breaking from processor execution on exit from authenticated code (AC).

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.exitacbreak=0
>>> itp.cv.exitacbreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.1.23 fastmemoryreadenable

Definition Enable or disable fast memory reads on supported processors.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.fastmemoryreadenable=False
>>> itp.cv.fastmemoryreadenable
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
```

6.1.24 fivrbreak

Definition Display or change a setting that controls the Fully Integrated Voltage Regulator (FIVR) setting when a Powergood de-assertion event occurs.

Options: 0 (or 'disabled')

1 (or 'enabled')

2 (or 'rearm')

Usage:

```
>>> itp.cv.fivrbreak=1
>>> itp.cv.fivrbreak
[HSW_UC0] enabled (hardware ID: 0xdeadbeef)
[HSW_C0_T0] enabled (hardware ID: 0xdeadbeef)
[HSW_C0_T1] enabled (hardware ID: 0xdeadbeef)
[HSW_C1_T0] enabled (hardware ID: 0xdeadbeef)
[HSW_C1_T1] enabled (hardware ID: 0xdeadbeef)
```

6.1.25 forcereconfigtimeout

Definition Display or force the timeout (in seconds) for Python forcereconfig command to be blocked. This is for bug verification only, users should use reconfigtimeout instead. By default, forcereconfig command will be blocked with a timeout equal to reconfigtimeout + 5 seconds.

Options: a number (32-bit)

Usage:

```
>>> itp.cv.forcereconfigtimeout
0
```

6.1.26 getsecbreak

Definition Display or change behavior controlling breaking from processor execution when the GETSEC instruction is executed.

Options: 0 (or 'off') = Disable all GETSEC redirections

1 (or 'start') = break at the start of GETSEC

2 (or 'cram') = break just prior to CRAM load

4 (or 'sexit') = break at completion of SEXIT

Usage:

```
>>> itp.cv.getsecbreak = 0
>>> itp.cv.getsecbreak
[HSW_C0_T0] off
[HSW_C0_T1] off
[HSW_C1_T0] off
[HSW_C1_T1] off
```

6.1.27 globalpath

Definition Display or change the globalpath list for directories containing program source files.

Options: semicolon-delimited list of paths

Remarks: Use the globalpath control variable to display or set a global path list for directories containing global source files for modules loaded for any thread device node. Do not use filenames as part of a pathname. Entering the control variable without an option or in an expression returns the current setting. To set the search path for an individual thread device node, use the spath control variable (section 6.3.18) associated with that thread node.

Use the '+=' to append a path. However, this is really nothing more than a string concatenation so you have to provide a leading semi-colon to keep the new path separate from the existing paths (as shown in the Example).

There is no way to remove a path from globalpath; you have to clear it and add the paths back, leaving out the path(s) you do not want.

Note: All paths are separated by semi-colons; which is different behavior than in ITP.

Usage:

```
>>> itp.cv.globalpath='C:\\source_files'
>>> itp.cv.globalpath+=';C:\\foo_files'
>>> itp.cv.globalpath+=';C:\\bar_files'
>>> itp.cv.globalpath
C:\source_files;C:\foo_files;C:\bar_files
>>> itp.cv.globalpath='C:\\foo_bar_files'
>>> itp.cv.globalpath
C:\foo_bar_files
```

6.1.28 halttimeout

Definition Display or change the timeout (in ms) for the halt command.

Options: a number (32-bit)

Usage:

```
>>> itp.cv.halttimeout
2000
```

6.1.29 ignorecachedcredentials

Definition Enable/disable ignoring cached credentials on the debug host

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.ignorecachedcredentials=0
>>> itp.cv.ignorecachedcredentials
False
```

6.1.30 initbreak

Definition Display or change what happens to processor execution when the INIT signal is asserted but not the RESET signal.

Options: 0 (or 'continue')

1 (or 'break')

2 (or 'restart')

Remarks: Use the initbreak control variable to display the setting or control processor execution when the INIT signal is asserted but not the RESET signal. If initbreak is set to break and the RESET signal remains unasserted while the INIT signal is asserted, the specified processor or processors break execution. 'Continue' (0) is the default setting. For the 'break' (1) setting, the specified processors halt and are initialized by the target system, and then the ITP continues execution. Thus hardware breakpoints are cleared, but software breakpoint specifications will be unchanged.

This control variable can be set at any time, but only takes effect for a currently running processor when execution is restarted on that processor.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>> itp.cv.initbreak
[HSW_C0_T0] continue
[HSW_C0_T1] continue
[HSW_C1_T0] continue
[HSW_C1_T1] continue
[HSW_C2_T0] continue
```

```
[HSW_C2_T1] continue
[HSW_C3_T0] continue
[HSW_C3_T1] continue
>>> itp.cv.initbreak=1
>>> itp.cv.initbreak
[HSW_C0_T0] break
[HSW_C0_T1] break
[HSW_C1_T0] break
[HSW_C1_T1] break
[HSW_C2_T0] break
[HSW_C2_T1] break
[HSW_C3_T0] break
[HSW_C3_T1] break
```

6.1.31 internalresetbreak

Definition Display or change a setting that controls processor execution when an INTERNALRESET event occurs.

Options: 0 (or 'continue')

1 (or 'break')

2 (or 'restart')

Usage:

```
>>> itp.threads[0].cv.internalresetbreak
continue
>>> itp.threads[0].cv.internalresetbreak=1
>>> itp.threads[0].cv.internalresetbreak
break
```

6.1.32 ishalted

Definition Determine if the thread is halted or not. Read-only

Options: None.

Remarks: Displays True if the thread is enabled and halted, False if the thread is in an execution state other than halted.

Usage:

```
>>> itp.threads[0].cv.ishalted
True
```

6.1.33 isrunning

Definition Determine if the thread is running or not. Read-only

Options: None.

Remarks: Displays True if the thread is running and False if the thread is halted or disabled

Usage:

```
>>> itp.threads[0].cv.isrunning
True
```

6.1.34 isstopped

Definition Determine if the thread is stopped or not. Read-only

Options: None.

Remarks: Displays True if the thread is enabled and stopped, False if the thread is in an execution state other than stopped. This generally means the processor is indeed put into probe mode but no arch state has been extracted.

Usage:

```
>>> itp.threads[0].cv.isstopped
True
```

6.1.35 itpversion

Definition Get the version of the DAL as a 64-bit number, with each 16 bit chunk representing a field in the version. Read-only

Options: None.

Remarks: The itpversion control variable returns the product version of the DAL formatted as a 64-bit number. The version breaks down like this:

Bits Description 63-48 Major version 47-32 Minor version 31-16 Build 15-0 Revision/color

For example, v1.9.258.600 is returned as 0x1000901020258, which breaks down as follows:

0x0001 == 1 0x0009 == 9 0x0102 == 258 0x0258 == 600

For a human-readable form of the version, use the itp.cv.version control variable.

Usage:

```
>>> itp.cv.itpversion
281513896182360
```

6.1.36 jobexecutiontimeout

Definition Display or change the timeout interval (in ms) for an execution of a job.

Options: a number (32-bit)

Usage:

```
>>> itp.cv.jobexecutiontimeout
30000
```

6.1.37 keepprobemoderedirectioncleared

Definition Always clears the probe mode redirection bit(i.e. ICECTLPMR.IR)cleared for a given thread.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.keepprobemoderedirectioncleared
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
>>> itp.cv.keepprobemoderedirectioncleared = False
>>> itp.cv.keepprobemoderedirectioncleared
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
```

6.1.38 keepprobemoderedirectionset

Definition Always keeps the probe mode redirection bit(i.e. ICECTLPMR.IR)set for a given thread.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.keepprobemoderedirectionset
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
```

```
[HSW_C3_T1] True
>>> itp.cv.keepprobemoderedirectionset = False
>>> itp.cv.keepprobemoderedirectionset
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
```

6.1.39 keepturbodisandfreezeiafrequserdefined

Definition Enable/Disable whether the TurboDis and FreezeIAFreq bits are set or cleared on probe mode exit when no breaks are set.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.keepturbodisandfreezeiafrequserdefined
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
>>> itp.cv.keepturbodisandfreezeiafrequserdefined = False
>>> itp.cv.keepturbodisandfreezeiafrequserdefined
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
```

6.1.40 machinecheckbreak

Definition Enable/disable break from processor execution on machine check.

Options: False (or 0)

True (or 1)

Remarks: Use the machinecheckbreak control variable to display the setting or enable breaking from processor execution on entry into machine check. If machinecheckbreak is set to false, entry into machine check does not affect ex-

ecution control. If machinecheckbreak is true, execution will break when the specified processor or processors enter machine check. The default setting for machinecheckbreak is false.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>> itp.cv.machinecheckbreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
>>> itp.cv.machinecheckbreak = 1
>>> itp.cv.machinecheckbreak
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
```

6.1.41 manualscans

Definition Enable/disable manual scans, which prevents pre-scan builder operations between commands that affect the scan channel. Used primarily to surround irscan() and drscan() commands.

Options: False (or 0)

True (or 1)

Remarks: Use the manualcans control variable to display or change whether the scan chain is left untouched between commands. If manual scans mode is enabled (True), the scan chain is not touched between commands. If manual scans mode is disabled (False), the scan chain is altered between commands.

Internally, every python CLI command is composed of a job. At the start of each job, a prerequisite task is executed to make sure the devices are available on the scan chain. At the end of each job, the post-requisite task is executed to make sure the scan chain is restored to its original state (typically, all devices but the uncore is removed from the chain).

When enabled, the manual scans mode prevents the post-requisite task from doing anything. Instead, the postrequisite, in effect, remembers what

it was going to do but does not actually do anything. When manual scans mode is disabled, all of the remembered post-requisite tasks are executed, restoring the scan chain.

There are only two python CLI commands at the moment that require use of the manualscans command: irscan() and drscan(). If an irscan() is followed by a drscan(), the manualscans mode should be enabled so the post-requisite task does not interfere with the irscan().

Usage:

```
>>> itp.cv.manualscans
False
>>> itp.cv.manualscans = 1
>>> itp.cv.manualscans
True
```

6.1.42 minstatelevel

Definition Display or change the amount of state to be saved and restored when entering and exiting probe mode.

Options: type itp.cv.minstates to show possible values.

Remarks: This control variable can be used at the global level or thread level.

Usage:

```
>>> itp.cv.minstatelevel
[HSW_C0_T0] Maximum
[HSW_C0_T1] Maximum
[HSW_C1_T0] Maximum
[HSW_C1_T1] Maximum
[HSW_C2_T0] Maximum
[HSW_C2_T1] Maximum
[HSW_C3_T0] Maximum
[HSW_C3_T1] Maximum
>>> itp.cv.minstatelevel = 'Minimum'
>>> itp.cv.minstatelevel
[HSW_C0_T0] Minimum
[HSW_C0_T1] Minimum
[HSW_C1_T0] Minimum
[HSW_C1_T1] Minimum
[HSW_C2_T0] Minimum
[HSW_C2_T1] Minimum
[HSW_C3_T0] Minimum
[HSW_C3_T1] Minimum
```

6.1.43 minstatelevelcurrent

Definition Display or increase the level of min state to be saved in the current probe mode session.

Options: type itp.cv.minstates to show possible values.

Remarks: This control variable can be used at the global level or thread level. A go/halt sequence must be completed for any changes to take effect.

Usage:

```
>>>itp.cv.minstatelevelcurrent
[HSW_C0_T0] Level2
[HSW_C0_T1] Level2
[HSW_C1_T0] Level2
[HSW_C1_T1] Level2
[HSW_C2_T0] Level2
[HSW_C2_T1] Level2
[HSW_C3_T0] Level2
[HSW_C3_T1] Level2
>>>itp.cv.minstatelevelcurrent='Minimum'
>>>itp.go();itp.halt()
>>> itp.cv.minstatelevelcurrent
[HSW_C0_T0] Level1
[HSW_C0_T1] Level0
[HSW_C1_T0] Level0
[HSW_C1_T1] Level0
[HSW_C2_T0] Level0
[HSW_C2_T1] Level0
[HSW_C3_T0] Level0
[HSW_C3_T1] Level0
```

6.1.44 minstates

Definition Displays the possible Min State level values by device type and stepping. Read-only.

Options: None.

Remarks: This control variable can be used at the global level or thread level.

Usage:

```
>>> itp.cv.minstates
[HSW_C0_T0]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C0_T1]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C1_T0]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C1_T1]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C2_T0]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
```

```
[HSW_C2_T1]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C3_T0]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C3_T1]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
```

6.1.45 num_activeprocessors

Definition The total number of active logical processors in the target system. Read-only

Options: None.

Usage:

```
>>> itp.cv.num_activeprocessors
8
```

6.1.46 num_halt_retries

Definition Display or change the number of retries for the halt command.

Options: a number (32-bit)

Notes:

With few tick-less OSes(such as Linux), the package and CPU cannot be woken up reliably with the Dfx

hooks we have in silicon today. As a result, probe mode entry request can fail at times depending on how

sleepy the OS gets to, and this variable will help retry the halt command when it fails on the first attempt.

Usage:

```
>>> itp.cv.num_halt_retries
5
```

6.1.47 num_i2c_devices

Definition The total number of I2C devices. Read-only

Options: None.

Usage:

```
>>> itp.cv.num_i2c_devices
0
```

6.1.48 num_jtag_devices

Definition The total number of JTAG devices. Read-only

Options: None.

Usage:

```
>>> itp.cv.num_jtag_devices
10
```

6.1.49 num_obs_devices

Definition The total number of OBS devices. Read-only.

Options: None.

Usage:

```
>>> itp.cv.num_obs_devices
0
```

6.1.50 num_processors

Definition The total number of threads in the system. Read-only.

Options: None.

Usage:

```
>>> itp.cv.num_processors
8
```

6.1.51 osvvmentrybreak

Definition Enable/disable breaking from processor execution when OSV host to OSV guest VMLAUNCH and VMRESUME instructions execute.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.osvvmentrybreak=0
>>> itp.cv.osvvmentrybreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
```

```
[HSW_C1_T1] False
```

6.1.52 postpackagewakeupwaittime

Definition Display or change the interval (in ms) for the post package wakeup wait time.

Options: a number (32-bit)

Notes:

With few tick-less OSes(such as Linux), the package and CPU cannot be woken up real-time with the Dfx

hooks we have in silicon today. We can only request the next sleep state to defer whenever the OS

wakes up on a tick and this variable will help to determine the wakeup wait time window so that debug tool

operations can succeed more reliably when the deep sleep states are in play.

Usage:

```
>>> itp.cv.postpackagewakeupwaittime
1000
```

6.1.53 reconfiginprogress

Definition Displays whether or not the reconfiguration is currently in progress. Read-only

Options: None.

Remarks: This read-only variable returns true when the reconfiguration is in progress and returns false once the reconfiguration is complete.

Usage:

```
>>> itp.cv.reconfiginprogress
True
```

6.1.54 reconfigtimeout

Definition Display or change the timeout interval (in ms) during reconfiguration.

Options: a number (32-bit)

Usage:

```
>>> itp.cv.reconfigtimeout
```

```
30000
```

6.1.55 reconfigwaittime

Definition Display or change the delay interval (in ms) before reconfiguration.

Options: a number (32-bit)

Usage:

```
>>> itp.cv.reconfigwaittime
1000
```

6.1.56 releaseallthreadswhilestepping

Definition Enable/disable releasing all threads while single stepping a particular thread.

Options: False (or 0)

True (or 1)

Remarks: By default (releaseallthreadswhilestepping=0), when you single step in ITP, only the current processor thread/core briefly exits probe mode, executes a single instruction and then re-enters probe mode while the other threads/cores still remain in probe mode at the same EIP whereever they were during the halt.

When enabled (releaseallthreadswhilestepping=1), ITP will release all threads/cores from probe mode along with the current thread. If itp.cv.breakall=1 all threads will re-enter probe mode (PM) when the current thread completes executing one instruction.

SOC platforms (especially with recent firmware) can have pending interrupts that require a non-currently stepping thread to service. Interrupts take a higher priority than probe mode re-entry. Because the non-currently stepping threads all sit in PM by default (releaseallthreadswhilestepping=0), a deadlock condition occurs on the currently stepping thread because the interrupt from the current thread never gets serviced by the other threads sitting in PM. Enabling the releaseallthreadswhilestepping control variable will help in this case, because all threads in the platform will start running when the current thread is single stepped and all pending interrupts will be serviced by the other running processors.

This control variable can be set at any time and will take effect on the next step operation.

Usage:

```
>>> itp.cv.releaseallthreadswhilestepping
0
```

```
itp.cv.releaseallthreadswhilestepping=1
```

6.1.57 resetbreak

Definition Display or change a setting that controls processor execution when a RESET event occurs.

Options: 0 (or 'continue')

1 (or 'break')

2 (or 'restart')

Remarks: Use the resetbreak control variable to display the setting or control processor execution when the RESET signal is asserted. If resetbreak is set to break and the RESET signal is asserted, the specified processor or processors break execution.

For the 'continue' setting, the specified processors halt and are reset by the target system, and then the ITP continues execution. The hardware breakpoints are dependent on the processor and may be cleared or left unchanged but software breakpoint specifications will be unchanged.

For the 'restart' setting, the specified processors halt and are reset by the target system, and then the ITP reloads the debug registers to previous values and restarts execution. Thus no breakpoints specifications are changed.

This control variable can be set at any time, but only takes effect for a currently running processor when execution is restarted on that processor.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>>itp.cv.resetbreak = 1/n    >>> itp.cv.resetbreak/n    [HSW_C0_T0] continue (ha
```

6.1.58 resettaponhaltfailure

Definition Enable/disable resetting the tap on a halt failure.

Options: False (or 0)

True (or 1)

Notes:

Depending on the processor family, CPU and other taps come into the scan chain and go away

due to power management and sleep states. With few tick-less OSes(such as Linux)

the package and CPU cannot be woken up reliably with the Dfx hooks we have in silicon today.

As a result we are noticing few tap synchronization issues and this variable will help by resetting

and recovering the tap from that situation on a halt failure.

Usage:

```
>>> itp.cv.resettaponhaltfailure
0
itp.cv.resettaponhaltfailure=1
```

6.1.59 rtestbreak

Definition Display or change a setting that controls the RTest Break setting when a Powergood de-assertion event occurs.

Options: 0 (or 'disabled')

1 (or 'enabled')

2 (or 'rearm')

Usage:

```
>>> itp.cv.rtestbreak=0
>>> itp.cv.rtestbreak
[HSW_UC0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T1] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T1] disabled (hardware ID: 0xdeadbeef)
```

6.1.60 savestateondemand

Definition Enable or disable the Save State On Demand (aka Demand Based Save State) feature.

Options: True or 1

False or 0

Remarks: This control variable can be used at the global level or thread level.

Usage:

```
>>> itp.cv.savestateondemand
[HSW_C0_T0] False
[HSW_C0_T1] False
```

```
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
>>> itp.cv.savestateondemand=1
>>> itp.cv.savestateondemand
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
```

6.1.61 sexitendbreak

Definition Enable/disable breaking from processor execution on entry to the end of SEXIT.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.sexitendbreak=0
>>> itp.cv.sexitendbreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.1.62 shiftverify

Definition Enable or disable verified reads on supported TAP registers.

Options: True or 1

False or 0

Remarks: In cases where a TAP register read may fail without detection from the DAL it can be useful to use a verified read. A verified read overshifts TDO with a unique key to determine that a value the size of the TAP register is read from TDO. If a value of the expected size is not read then an error is raised. If the read is successful then the value is returned as normal.

Usage:

```
>>>itp.cv.shiftverify
False
>>>itp.cv.shiftverify = True
>>>itp.cv.shiftverify
```

```
True
```

6.1.63 shutdownbreak

Definition Enable/disable whether to break from processor execution on shutdown signal.

Options: False (or 0)

True (or 1)

Remarks: Use the shutdownbreak control variable to display the setting or enable breaking from processor execution when entering shutdown. The break occurs before the special bus cycle that causes a system reset is issued. If shutdownbreak is set to False, entry into shutdown does not affect execution control.

This control variable can be set at any time, but only takes effect for a currently running processor when execution is restarted on that processor.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>> itp.cv.shutdownbreak = 1
>>> itp.cv.shutdownbreak
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
```

6.1.64 smmentrybreak

Definition Enable/disable whether to break from processor execution on entry into system management mode (SMM).

Options: False (or 0)

True (or 1)

Remarks: Use the smmentrybreak control variable to display the setting or enable breaking from processor execution on entry into system management mode (SMM). If smmentrybreak is set to False, entry into SMM does not affect execution control. If smmentrybreak is True, execution will break when the specified thread enters SMM. The default setting for smmentrybreak is False.

This control variable can be set at any time, but only takes effect for a currently running processor when execution is restarted on that processor.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>> itp.cv.smmentrybreak = 1
>>> itp.cv.smmentrybreak
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
```

6.1.65 smmexitbreak

Definition Enable/disable whether to break from processor execution on exit from system management mode (SMM).

Options: False (or 0)

True (or 1)

Remarks: Use the smmexitbreak control variable to display the setting or enable breaking from processor execution on exit from system management mode (SMM). If smmexitbreak is set to False, exit from SMM does not affect execution control. If smmexitbreak is True, execution will break when the specified thread exits SMM. The default setting for smmexitbreak is False.

This control variable can be set at any time, but only takes effect for a currently running processor when execution is restarted on that processor.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>> itp.cv.smmexitbreak = 1
>>> itp.cv.smmexitbreak
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
```

6.1.66 stepintoexception

Definition Enable/disable stepping into exception.

Options: False (or 0)

True (or 1)

Remarks: Use the stepintoexception control variable to display the setting or control the ability to step into an exception. By default, this setting is disabled, which means you cannot step into an exception.

This control variable can be set at any time.

Usage:

```
>>> itp.cv.stepintoexception=True
>>> itp.cv.stepintoexception
True
```

6.1.67 stmservicebreak

Definition Enable/disable breaking from processor execution when a parallel STM to OSV VM entry occurs.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.stmservicebreak=0
>>> itp.cv.stmservicebreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.1.68 targpower

Definition Display the status of target system power. Read-only

Options: None.

Remarks: This read-only variable returns the state of target system power by polling the power good pin on each debug port in the current domain. If at least one power good pin is off, then target system power is considered off. Note that on some systems, a debug port dedicated to the PCH may still have power even though the CPU debug port has no power. In this situation, power is considered off.

Usage:

```
>>> itp.cv.targpower
```

```
True
```

6.1.69 testmodebreak

Definition Display or change a setting that controls the Test Mode Break setting when a Powergood de-assertion event occurs.

Options: 0 (or 'disabled')

1 (or 'enabled')

2 (or 'rearm')

Usage:

```
>>> itp.cv.testmodebreak=0
>>> itp.cv.testmodebreak
[HSW_UC0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T1] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T1] disabled (hardware ID: 0xdeadbeef)
```

6.1.70 usemode

Definition Display the current IA32 instruction mode. Read-only

Options: None.

Remarks: The usemode can be one of the following values: - 'use16' (0) = 16-bit mode - 'use32' (1) = 32-bit mode - 'use64' (2) = 64-bit mode

Usage:

```
>>> itp.cv.usemode = 'use16'
>>> itp.cv.usemode
use16
```

6.1.71 version

Definition Display the DAL version in human-readable format.

Options: None.

Remarks: The version control variable returns the product version of the DAL formatted as a human-readable string; for example, 'DAL v1.9.258.600'. For a machine-readable form of the version, use the itp.cv.itpversion control variable

Usage:

```
>>> itp.cv.version
DAL Version 1.9.4040.600
```

6.1.72 vmclearbreak

Definition Enable/disable breaking from processor execution of the VMCLEAR instruction.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.vmclearbreak=False
>>> itp.cv.vmclearbreak
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
```

6.1.73 vmexitbreak

Definition Display or change what to do when breaking from processor execution of the VMEXIT instruction. Separate options with '|': "softint | extint".

Options: 'softint', 'extint', 'triplefault', 'initinstr', 'sipi', 'iosmi'

'othersmi', 'pendingint', 'vmxtimer', 'taskswitch', 'cpuid'

'getsec', 'hlt', 'invdinstr', 'invlpg', 'rdpmc', 'rdtsc', 'rsm'

'vmcall', 'vmclear', 'vmlaunch', 'vmptrld', 'vmptrst', 'vmread'

'vmresume', 'vmwrite', 'vmxoff', 'vmxon', 'craccess', 'draccess'

'ioinstruction', 'msrread', 'msrwrite', 'vmentryfailbadguest'

'vmentryfailmsrload', 'vmexitfailure', 'mwait', 'monitortrapflag'

'corruptvmcs', 'monitor', 'pause', 'vmentrymcheck', 'smiincstate'

'tprthreshold', 'apicaccess', 'leveltrigeoi', 'gdtridtraccess'

'ldtrtraccess', 'eptviolation', 'eptmisconfig', 'invlept', 'rdtscp'

'vmxtimer2', 'pndvirtnmi', 'off'

Remarks: If vmexitbreak is set to 'off', execution of these instructions does not affect execution control. If vmexitbreak is enabled, execution will break when the specified processor or processors execute a VMEXIT. The default setting for vmexitbreak is 'off'. Entering the control variable without an option displays the current setting.

Note: Not all VM-Exit reasons are available all on steppings of processors that support Vanderpool Technology.

Usage:

```
>>> itp.cv.vmexitbreak=False
>>> itp.cv.vmexitbreak
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
```

6.1.74 vmlaunchbreak

Definition Enable/disable breaking from processor execution of the VMLAUNCH and VMRESUME instructions.

Options: 'False' (or 0)

'True' (or 1)

Usage:

```
>>> itp.cv.vmlaunchbreak=0
>>> itp.cv.vmlaunchbreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.2 Thread Control Variables

6.2.1 asmmode

Definition The IA32 instruction mode for the asm command.

Options: 'use16' (or 0)

'use32' (or 1)

'use64' (or 2)

Remarks: This setting is used only if the asm/dasm addresses are linear or physical. Use the asmmode control variable to display or set the mode for the asm command (section 5.54) when working with the IA32 instruction set. Entering the control variable without an option returns the current setting. When displayed, the values are 0 for use16, 1 for use32, and 2 for EM64T.

When set to 'use16' or 'use32', the Python CLI interprets the instruction mode as IA32 16-bit or 32-bit, respectively. When asmmode is set to 'use64',

the Python CLI interprets the instructions using Intel Extended Memory 64 Technology. You can use the asmmode control variable to change the asm command's instruction mode for a given processor whether the processor is running or stopped.

Any setting of the asmmode control variable affects only the single thread specified. To affect all threads, use the itp.cv.asmmode control variable. Note: The asmmode control variable override is used only if the address provided to the asm() command is linear or physical; otherwise, the current addressing mode takes precedence and dictates the mode to assemble/disassemble.

Usage:

```
>>>itp.cv.asmmode
use16
>>>itp.cv.asmmode=0
>>>itp.cv.asmmode='use32'
```

6.2.2 brkstatus

Definition Display a code for the cause of the most recent break in execution. Read-only

Options: None.

Remarks: Use the brkstatus read-only control variable to display a numerical value representing the cause of the most recent target processor break in execution. The returned value is a 32-bit value. The upper word is the processor's ID in the boundary scan chain, and the lower word is the break status value. The following codes can be seen in the brkstatus control variable.

Note: Unlike ITP, the brkstatus value does not include the processor ID in the upper 16 bits of the returned value.

No break cause 0x0000

SMM entry 0x0003

SMM exit 0x0004

INIT 0x0005

RESET 0x0006

External halt trigger 0x0009

Halt command 0x000a

Software breakpoint 0x000c

Debug register break on I/O reads or writes (hw break 0) 0x000d

Debug register break on instruction execution only(hw break 0) 0x100d

Debug register break on data writes(hw break 0) 0x200d

Debug register break on data reads or writes(hw break 0) 0x300d

Debug register break on I/O reads or writes (hw break 1) 0x400d

Debug register break on instruction execution only(hw break 1) 0x500d

Debug register break on data writes(hw break 1) 0x600d

Debug register break on data reads or writes(hw break 1) 0x700d

Debug register break on I/O reads or writes (hw break 2) 0x800d

Debug register break on instruction execution only(hw break 2) 0x900d

Debug register break on data writes(hw break 2) 0xa00d

Debug register break on data reads or writes(hw break 2) 0xb00d

Debug register break on I/O reads or writes (hw break 0) 0xc00d

Debug register break on instruction execution only(hw break 3) 0xd00d

Debug register break on data writes(hw break3) 0xe00d

Debug register break on data reads or writes(hw break 3) 0xf00d

Branch step 0x000e

Unknown 0x000f

Write to the debug regs which were protected by the GD bit setting 0x0010

Other processor caused break upon startup IPI 0x0018

Single step 0x0019

LT SIPI 0x0020

Breakall 0x0022

CC1 entry / exit 0x0023

CC2 entry / exit 0x0024

CC3 entry / exit 0x0025

CC4 entry / exit 0x0026

CC5 entry / exit 0x0027

CC6 entry / exit 0x0028

C6 save / restore 0x0029

System cluster break 0x0031

Debug trap 0x0032

Fault 0x0033

Machine check break 0x0034

Error in shutdown 0x0035

Shutdown 0x0036

Sleep state 0x0037

Suspend instruction 0x0038

Sleep state and Halt instruction 0x0039

Sleep and error shutdown 0x003a

Sleep and shutdown 0x003b

Sleep state and SIPI loop 0x003c

Sleep state and MWAIT 0x003d

Sleep state and MWAIT_TC_FLUSHED 0x003e

Sleep state and SENTER 0x003f

Data breakpoint 0x007d

VMENTER 0x0082

VMEXIT 0x0083

VMLAUNCH 0x0085

VMLAUNCH from STM to OSV for SMM 0x0086

VMLAUNCH from SMM host to SMM guest 0x0087

VMLAUNCH from STM to OSV for service call 0x0088

VMLAUNCH from OSV host to OSV guest 0x0089

VMENTRY from STM to OSV for SMM 0x0090

VMENTRY from SMM host to SMM guest 0x0091

VMENTRY from STM to OSV for service call 0x0092

VMENTRY from OSV host to OSV guest 0x0093

Usage:

```
>>>itp.cv.brkstatus
[HSW_C0_T0] (0x000a) Halt Command break
[HSW_C0_T1] (0x000a) Halt Command break
[HSW_C1_T0] (0x000a) Halt Command break
[HSW_C1_T1] (0x000a) Halt Command break
[HSW_C2_T0] (0x000a) Halt Command break
[HSW_C2_T1] (0x000a) Halt Command break
[HSW_C3_T0] (0x000a) Halt Command break
[HSW_C3_T1] (0x000a) Halt Command break
>>> itp.threads[0].cv.brkstatus
(0x000a) Halt Command break
```

6.2.3 cause

Definition The cause of the most recent break in execution. Read-only

Options: None.

Remarks: Use the cause command when a break message has scrolled off the screen, or to give output from a user-defined function. The cause command displays the last break message for the thread.

Break information is returned for all variations of the step command (step into, step branch, step out, instruction step, and step over).

Usage:

```
>>> itp.cv.cause
[HSW_C0_T0] Halt Command break at 0xF000:000000000000FFF0
[HSW_C0_T1] Halt Command break at 0xF000:000000000000FFF0
[HSW_C1_T0] Halt Command break at 0xF000:000000000000FFF0
[HSW_C1_T1] Halt Command break at 0xF000:000000000000FFF0
[HSW_C2_T0] Halt Command break at 0xF000:000000000000FFF0
[HSW_C2_T1] Halt Command break at 0xF000:000000000000FFF0
[HSW_C3_T0] Halt Command break at 0xF000:000000000000FFF0
[HSW_C3_T1] Halt Command break at 0xF000:000000000000FFF0
>>> itp.threads[0].cv.cause
Halt Command break at 0xF000:000000000000FFF0
```

6.2.4 cip

Definition Display or change the current instruction pointer for the thread.

Options: an address expressed as a string

Remarks: This control variable is new to the Python CLI and stands in for the '$' used in ITP (Python does not allow a '$' in its expressions). This control variable acts like an address, so you can do things like 'itp.threads[0].cv.cip += 10'.

Usage:

```
>>>itp.threads[0].cv.cip
0xF000:000000000000FFF0
>>> itp.threads[0].cv.cip += 10
>>> itp.threads[0].cv.cip
0xF000:000000000000FFFA
```

6.2.5 debugprotect

Definition Enable/disable whether a break occurs when your program accesses a processor's debug registers.

Options: False (or 0)

True (or 1)

Remarks: Use the debugprotect control variable to display or change whether a break occurs when your program accesses a processor's debug registers. The default setting for debugprotect is False.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>>itp.cv.debugprotect
[HSW_C0_T0] False
[HSW_C0_T1] False
>>>itp.cv.debugprotect=1
>>>itp.thread[0].debugprotect=True
>>>itp.go();itp.halt()
[HSW_C0_T0]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C0_T1]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C1_T0]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C1_T1]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C2_T0]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C2_T1]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C3_T0]  Halt Command break at 0xF000:000000000000FFF0
[HSW_C3_T1]  Halt Command break at 0xF000:000000000000FFF0
>>>itp.thread[0].debugprotect
True
```

6.2.6 drprotect

Definition Enable/disable DAL taking exclusive control over the processor's debug registers.

Options: False (or 0)

True (or 1)

Usage:

```
>>>itp.cv.drprotect
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
>>> itp.threads[0].cv.drprotect=True
>>> itp.threads[0].cv.drprotect
True
```

6.2.7 earbreak

Definition Display or change a setting that controls the External Aligned Reset (EAR) setting when a Powergood de-assertion event occurs.

Options: 0 (or 'disabled')

1 (or 'enabled')

2 (or 'rearm')

3 (or 'step' - available only for supporting processors only)

Usage:

```
>>> itp.cv.earbreak=0
>>> itp.cv.earbreak
[HSW_UC0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T1] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T1] disabled (hardware ID: 0xdeadbeef)
```

6.2.8 enableoxmdebug

Definition Enable/disable subset of DFX registers which is made available on locked part.

Options: False (or 0)

True (or 1)

Remarks: Use the enableoxmdebug control variable to display or enable/disable if message channel registers are allowed on locked part. The default setting for enableoxmdebug is False.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>>itp.cv.enableoxmdebug
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
>>>itp.cv.enableoxmdebug=True
>>>itp.go();itp.halt
```

6.2.9 enclaveentrybreak

Definition Enable/disable breaking into probemode, when the processor enters secure enclave execution instructions.

Options: 'False' (or 0)

'True' (or 1)

Usage:

```
>>> itp.cv.enclaveentrybreak=0
>>> itp.cv.enclaveentrybreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.2.10 enclaveresumebreak

Definition Enable/disable breaking into probemode, when the processor resumes secure enclave execution instructions.

Options: 'False' (or 0)

'True' (or 1)

Usage:

```
>>> itp.cv.enclaveresumebreak=0
>>> itp.cv.enclaveresumebreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.2.11 enteracbreak

Definition Enable/disable breaking from processor execution on entry to authenticated code (AC).

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.enteracbreak=0
>>> itp.cv.enteracbreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.2.12 enterpmmethod

Definition Display or change setting that controls the method to enter probemode .

Options: '0' (or 'pin')

'1' (or 'uncore')

'2' (or 'core')

Usage:

```
>>> itp.cv.enterpmmethod
[HSW_C0_T0] pin
[HSW_C0_T1] pin
[HSW_C1_T0] pin
[HSW_C1_T1] pin
[HSW_C2_T0] pin
[HSW_C2_T1] pin
[HSW_C3_T0] pin
[HSW_C3_T1] pin
>>> itp.cv.enterpmmethod='core'
```

6.2.13 etrace

Definition Display or change a setting that controls the generation of execution history trace records.

Options: 0x0000 (or 'off'): Disable all forms of execution history generation and collection.

0x0001 (or 'btm'): The Branch Trace Messages will be generated on the front-side bus.

0x0002 (or 'lbr'): The Last Branch Record registers will be used to collect execution history.

0x0004 (or 'ler'): The Last Exception Record registers will be used to collect execution history.

0x0008 (or 'btb'): The Branch Trace Buffer will be used to collect execution history.

0x0010 (or 'rct'): ITP will used one of the other methods to seamlessly collect a specified amount

execution history. This is called Run Control Trace (rct).

0x0020 (or 'bts'): The Branch Trace Store will be used to collect execution history.

Usage:

```
>>> itp.cv.etrace
[HSW_C0_T0] off
[HSW_C0_T1] off
[HSW_C1_T0] off
[HSW_C1_T1] off
[HSW_C2_T0] off
[HSW_C2_T1] off
[HSW_C3_T0] off
[HSW_C3_T1] off
>>> itp.cv.etrace='btm'
```

6.2.14 exitacbreak

Definition Enable/disable breaking from processor execution on exit from authenticated code (AC).

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.exitacbreak=0
>>> itp.cv.exitacbreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.2.15 fivrbreak

Definition Display or change a setting that controls the Fully Integrated Voltage Regulator (FIVR) setting when a Powergood de-assertion event occurs.

Options: 0 (or 'disabled')

1 (or 'enabled')

2 (or 'rearm')

Usage:

```
>>> itp.cv.fivrbreak=1
>>> itp.cv.fivrbreak
[HSW_UC0] enabled (hardware ID: 0xdeadbeef)
[HSW_C0_T0] enabled (hardware ID: 0xdeadbeef)
[HSW_C0_T1] enabled (hardware ID: 0xdeadbeef)
[HSW_C1_T0] enabled (hardware ID: 0xdeadbeef)
[HSW_C1_T1] enabled (hardware ID: 0xdeadbeef)
```

6.2.16 getsecbreak

Definition Display or change behavior controlling breaking from processor execution when the GETSEC instruction is executed.

Options: 0 (or 'off') = Disable all GETSEC redirections

1 (or 'start') = break at the start of GETSEC

2 (or 'cram') = break just prior to CRAM load

4 (or 'sexit') = break at completion of SEXIT

Usage:

```
>>> itp.cv.getsecbreak = 0
>>> itp.cv.getsecbreak
[HSW_C0_T0] off
[HSW_C0_T1] off
[HSW_C1_T0] off
[HSW_C1_T1] off
```

6.2.17 initbreak

Definition Display or change what happens to processor execution when the INIT signal is asserted but not the RESET signal.

Options: 0 (or 'continue')

1 (or 'break')

2 (or 'restart')

Remarks: Use the initbreak control variable to display the setting or control processor execution when the INIT signal is asserted but not the RESET signal. If initbreak is set to break and the RESET signal remains unasserted while the INIT signal is asserted, the specified processor or processors break execution. 'Continue' (0) is the default setting. For the 'break' (1) setting, the specified processors halt and are initialized by the target system, and then the ITP continues execution. Thus hardware breakpoints are cleared, but software breakpoint specifications will be unchanged.

This control variable can be set at any time, but only takes effect for a currently running processor when execution is restarted on that processor.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>> itp.cv.initbreak
[HSW_C0_T0] continue
[HSW_C0_T1] continue
[HSW_C1_T0] continue
[HSW_C1_T1] continue
[HSW_C2_T0] continue
[HSW_C2_T1] continue
[HSW_C3_T0] continue
[HSW_C3_T1] continue
>>> itp.cv.initbreak=1
>>> itp.cv.initbreak
[HSW_C0_T0] break
[HSW_C0_T1] break
[HSW_C1_T0] break
[HSW_C1_T1] break
[HSW_C2_T0] break
[HSW_C2_T1] break
[HSW_C3_T0] break
[HSW_C3_T1] break
```

6.2.18 internalresetbreak

Definition Display or change a setting that controls processor execution when an INTERNALRESET event occurs.

Options: 0 (or 'continue')

1 (or 'break')

2 (or 'restart')

Usage:

```
>>> itp.threads[0].cv.internalresetbreak
continue
>>> itp.threads[0].cv.internalresetbreak=1
>>> itp.threads[0].cv.internalresetbreak
break
```

6.2.19 isenabled

Definition Determine if the thread is enabled or disabled. Read-only

Options: None.

Remarks: Displays True if the thread is enabled and False if the thread is disabled.

Usage:

```
>>> itp.threads[0].cv.isenabled
True
```

6.2.20 ishalted

Definition Determine if the thread is halted or not. Read-only

Options: None.

Remarks: Displays True if the thread is enabled and halted, False if the thread is in an execution state other than halted.

Usage:

```
>>> itp.threads[0].cv.ishalted
True
```

6.2.21 isrunning

Definition Determine if the thread is running or not. Read-only

Options: None.

Remarks: Displays True if the thread is running and False if the thread is halted or disabled

Usage:

```
>>> itp.threads[0].cv.isrunning
True
```

6.2.22 isstopped

Definition Determine if the thread is stopped or not. Read-only

Options: None.

Remarks: Displays True if the thread is enabled and stopped, False if the thread is in an execution state other than stopped. This generally means the processor is indeed put into probe mode but no arch state has been extracted.

Usage:

```
>>> itp.threads[0].cv.isstopped
True
```

6.2.23 keepprobemoderedirectioncleared

Definition Always clears the probe mode redirection bit(i.e. ICECTLPMR.IR)cleared for a given thread.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.keepprobemoderedirectioncleared
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
>>> itp.cv.keepprobemoderedirectioncleared = False
>>> itp.cv.keepprobemoderedirectioncleared
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
```

6.2.24 keepprobemoderedirectionset

Definition Always keeps the probe mode redirection bit(i.e. ICECTLPMR.IR)set for a given thread.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.keepprobemoderedirectionset
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
>>> itp.cv.keepprobemoderedirectionset = False
>>> itp.cv.keepprobemoderedirectionset
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
```

6.2.25 machinecheckbreak

Definition Enable/disable break from processor execution on machine check.

Options: False (or 0)

True (or 1)

Remarks: Use the machinecheckbreak control variable to display the setting or enable breaking from processor execution on entry into machine check. If machinecheckbreak is set to false, entry into machine check does not affect execution control. If machinecheckbreak is true, execution will break when the specified processor or processors enter machine check. The default setting for machinecheckbreak is false.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>> itp.cv.machinecheckbreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
>>> itp.cv.machinecheckbreak = 1
>>> itp.cv.machinecheckbreak
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
```

6.2.26 minstatelevel

Definition Display or change the amount of state to be saved and restored when entering and exiting probe mode.

Options: type itp.cv.minstates to show possible values.

Remarks: This control variable can be used at the global level or thread level.

Usage:

```
>>> itp.cv.minstatelevel
[HSW_C0_T0] Maximum
[HSW_C0_T1] Maximum
[HSW_C1_T0] Maximum
```

```
[HSW_C1_T1] Maximum
[HSW_C2_T0] Maximum
[HSW_C2_T1] Maximum
[HSW_C3_T0] Maximum
[HSW_C3_T1] Maximum
>>> itp.cv.minstatelevel = 'Minimum'
>>> itp.cv.minstatelevel
[HSW_C0_T0] Minimum
[HSW_C0_T1] Minimum
[HSW_C1_T0] Minimum
[HSW_C1_T1] Minimum
[HSW_C2_T0] Minimum
[HSW_C2_T1] Minimum
[HSW_C3_T0] Minimum
[HSW_C3_T1] Minimum
```

6.2.27 minstatelevelcurrent

Definition Display or increase the level of min state to be saved in the current probe mode session.

Options: type itp.cv.minstates to show possible values.

Remarks: This control variable can be used at the global level or thread level. A go/halt sequence must be completed for any changes to take effect.

Usage:

```
>>>itp.cv.minstatelevelcurrent
[HSW_C0_T0] Level2
[HSW_C0_T1] Level2
[HSW_C1_T0] Level2
[HSW_C1_T1] Level2
[HSW_C2_T0] Level2
[HSW_C2_T1] Level2
[HSW_C3_T0] Level2
[HSW_C3_T1] Level2
>>>itp.cv.minstatelevelcurrent='Minimum'
>>>itp.go();itp.halt()
>>> itp.cv.minstatelevelcurrent
[HSW_C0_T0] Level1
[HSW_C0_T1] Level0
[HSW_C1_T0] Level0
[HSW_C1_T1] Level0
[HSW_C2_T0] Level0
[HSW_C2_T1] Level0
[HSW_C3_T0] Level0
[HSW_C3_T1] Level0
```

6.2.28 minstates

Definition Displays the possible Min State level values by device type and stepping. Read-only.

Options: None.

Remarks: This control variable can be used at the global level or thread level.

Usage:

```
>>> itp.cv.minstates
[HSW_C0_T0]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C0_T1]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C1_T0]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C1_T1]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C2_T0]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C2_T1]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C3_T0]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
[HSW_C3_T1]
Level0, 0, none, min, minimum
Level1, 1, MSR, PIR, archregLvl1, resetbreak, breakreason
Level2, 2, MSRLvl2, archreg, regs, memory, io, break, full, max, maximum
```

6.2.29 osvvmentrybreak

Definition Enable/disable breaking from processor execution when OSV host to OSV guest VMLAUNCH and VMRESUME instructions execute.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.osvvmentrybreak=0
>>> itp.cv.osvvmentrybreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.2.30 resetbreak

Definition Display or change a setting that controls processor execution when a RESET event occurs.

Options: 0 (or 'continue')

1 (or 'break')

2 (or 'restart')

Remarks: Use the resetbreak control variable to display the setting or control processor execution when the RESET signal is asserted. If resetbreak is set to break and the RESET signal is asserted, the specified processor or processors break execution.

For the 'continue' setting, the specified processors halt and are reset by the target system, and then the ITP continues execution. The hardware breakpoints are dependent on the processor and may be cleared or left unchanged but software breakpoint specifications will be unchanged.

For the 'restart' setting, the specified processors halt and are reset by the target system, and then the ITP reloads the debug registers to previous values and restarts execution. Thus no breakpoints specifications are changed.

This control variable can be set at any time, but only takes effect for a currently running processor when execution is restarted on that processor.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>>itp.cv.resetbreak = 1/n    >>> itp.cv.resetbreak/n    [HSW_C0_T0] continue (ha
```

6.2.31 rtestbreak

Definition Display or change a setting that controls the RTest Break setting when a Powergood de-assertion event occurs.

Options: 0 (or 'disabled')

1 (or 'enabled')

2 (or 'rearm')

Usage:

```
>>> itp.cv.rtestbreak=0
>>> itp.cv.rtestbreak
[HSW_UC0] disabled (hardware ID: 0xdeadbeef)
```

```
[HSW_C0_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T1] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T1] disabled (hardware ID: 0xdeadbeef)
```

6.2.32 savestateondemand

Definition Enable or disable the Save State On Demand (aka Demand Based Save State) feature.

Options: True or 1

False or 0

Remarks: This control variable can be used at the global level or thread level.

Usage:

```
>>> itp.cv.savestateondemand
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
>>> itp.cv.savestateondemand=1
>>> itp.cv.savestateondemand
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
```

6.2.33 sexitendbreak

Definition Enable/disable breaking from processor execution on entry to the end of SEXIT.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.sexitendbreak=0
>>> itp.cv.sexitendbreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.2.34 shutdownbreak

Definition Enable/disable whether to break from processor execution on shutdown signal.

Options: False (or 0)

True (or 1)

Remarks: Use the shutdownbreak control variable to display the setting or enable breaking from processor execution when entering shutdown. The break occurs before the special bus cycle that causes a system reset is issued. If shutdownbreak is set to False, entry into shutdown does not affect execution control.

This control variable can be set at any time, but only takes effect for a currently running processor when execution is restarted on that processor.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>> itp.cv.shutdownbreak = 1
>>> itp.cv.shutdownbreak
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
```

6.2.35 smmentrybreak

Definition Enable/disable whether to break from processor execution on entry into system management mode (SMM).

Options: False (or 0)

True (or 1)

Remarks: Use the smmentrybreak control variable to display the setting or enable breaking from processor execution on entry into system management mode (SMM). If smmentrybreak is set to False, entry into SMM does not affect execution control. If smmentrybreak is True, execution will break when the specified thread enters SMM. The default setting for smmentrybreak is False.

This control variable can be set at any time, but only takes effect for a currently running processor when execution is restarted on that processor.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>> itp.cv.smmentrybreak = 1
>>> itp.cv.smmentrybreak
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
```

6.2.36 smmexitbreak

Definition Enable/disable whether to break from processor execution on exit from system management mode (SMM).

Options: False (or 0)

True (or 1)

Remarks: Use the smmexitbreak control variable to display the setting or enable breaking from processor execution on exit from system management mode (SMM). If smmexitbreak is set to False, exit from SMM does not affect execution control. If smmexitbreak is True, execution will break when the specified thread exits SMM. The default setting for smmexitbreak is False.

This control variable can be set at any time, but only takes effect for a currently running processor when execution is restarted on that processor.

Note: This execution break will be enabled in the processor after the next 'go' operation. If the processor is running when this is set a 'halt;go' cycle is required to enable this break.

Usage:

```
>>> itp.cv.smmexitbreak = 1
>>> itp.cv.smmexitbreak
[HSW_C0_T0] True
[HSW_C0_T1] True
[HSW_C1_T0] True
[HSW_C1_T1] True
[HSW_C2_T0] True
[HSW_C2_T1] True
[HSW_C3_T0] True
[HSW_C3_T1] True
```

6.2.37 spath

Definition Display or change the current path list for directories containing source display files for the thread.

Options: semicolon-delimited list of paths

Remarks: Use the spath control variable to display or set a path list for directories containing source files for modules loaded on a specific thread device node. Do not use filenames as part of a pathname. Entering the control variable without an option or in an expression returns the current setting. To set the search path for all threads, use the globalpath control variable.

Use the '+=' to append a path. However, this is really nothing more than a string concatenation so you have to provide a leading semi-colon to keep the new path separate from the existing paths.

There is no way to remove a path from spath; you have to clear it and add the paths back, leaving out the path(s) you do not want. n Usage: >>> itp.cv.spath='C:\\source_files' >>> itp.cv.spath+=';C:\\foo_files' >>> itp.cv.spath+=';C:\\bar_files' >>> itp.cv.spath C:\source_files;C:\foo_files;C:\bar_files >>> itp.cv.spath='C:\\foo_bar_files' >>> itp.cv.spath C:\foo_bar_files

6.2.38 stepintoexception

Definition Enable/disable stepping into exception.

Options: False (or 0)

True (or 1)

Remarks: Use the stepintoexception control variable to display the setting or control the ability to step into an exception. By default, this setting is disabled, which means you cannot step into an exception.

This control variable can be set at any time.

Usage:

```
>>> itp.cv.stepintoexception=True
>>> itp.cv.stepintoexception
True
```

6.2.39 stmservicebreak

Definition Enable/disable breaking from processor execution when a parallel STM to OSV VM entry occurs.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.stmservicebreak=0
>>> itp.cv.stmservicebreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.2.40 targpower

Definition Display the status of target system power. Read-only

Options: None.

Remarks: This read-only variable returns the state of target system power by polling the power good pin on each debug port in the current domain. If at least one power good pin is off, then target system power is considered off. Note that on some systems, a debug port dedicated to the PCH may still have power even though the CPU debug port has no power. In this situation, power is considered off.

Usage:

```
>>> itp.cv.targpower
True
```

6.2.41 testmodebreak

Definition Display or change a setting that controls the Test Mode Break setting when a Powergood de-assertion event occurs.

Options: 0 (or 'disabled')

1 (or 'enabled')

2 (or 'rearm')

Usage:

```
>>> itp.cv.testmodebreak=0
>>> itp.cv.testmodebreak
[HSW_UC0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C0_T1] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T0] disabled (hardware ID: 0xdeadbeef)
[HSW_C1_T1] disabled (hardware ID: 0xdeadbeef)
```

6.2.42 usemode

Definition Display the current IA32 instruction mode. Read-only

Options: None.

Remarks: The usemode can be one of the following values: - 'use16' (0) = 16-bit mode - 'use32' (1) = 32-bit mode - 'use64' (2) = 64-bit mode

Usage:

```
>>> itp.cv.usemode = 'use16'
>>> itp.cv.usemode
use16
```

6.2.43 vmclearbreak

Definition Enable/disable breaking from processor execution of the VMCLEAR instruction.

Options: False (or 0)

True (or 1)

Usage:

```
>>> itp.cv.vmclearbreak=False
>>> itp.cv.vmclearbreak
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
```

6.2.44 vmexitbreak

Definition Display or change what to do when breaking from processor execution of the VMEXIT instruction. Separate options with '|': "softint | extint".

Options: 'softint', 'extint', 'triplefault', 'initinstr', 'sipi', 'iosmi'

'othersmi', 'pendingint', 'vmxtimer', 'taskswitch', 'cpuid'

'getsec', 'hlt', 'invdinstr', 'invlpg', 'rdpmc', 'rdtsc', 'rsm'

'vmcall', 'vmclear', 'vmlaunch', 'vmptrld', 'vmptrst', 'vmread'

'vmresume', 'vmwrite', 'vmxoff', 'vmxon', 'craccess', 'draccess'

'ioinstruction', 'msrread', 'msrwrite', 'vmentryfailbadguest'

'vmentryfailmsrload', 'vmexitfailure', 'mwait', 'monitortrapflag'

'corruptvmcs', 'monitor', 'pause', 'vmentrymcheck', 'smiincstate'

'tprthreshold', 'apicaccess', 'leveltrigeoi', 'gdtridtraccess'

'ldtrtraccess', 'eptviolation', 'eptmisconfig', 'invlept', 'rdtscp'

'vmxtimer2', 'pndvirtnmi', 'off'

Remarks: If vmexitbreak is set to 'off', execution of these instructions does not affect execution control. If vmexitbreak is enabled, execution will break when the specified processor or processors execute a VMEXIT. The default setting for vmexitbreak is 'off'. Entering the control variable without an option displays the current setting.

Note: Not all VM-Exit reasons are available all on steppings of processors that support Vanderpool Technology.

Usage:

```
>>> itp.cv.vmexitbreak=False
>>> itp.cv.vmexitbreak
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
[HSW_C2_T0] False
[HSW_C2_T1] False
[HSW_C3_T0] False
[HSW_C3_T1] False
```

6.2.45 vmlaunchbreak

Definition Enable/disable breaking from processor execution of the VMLAUNCH and VMRESUME instructions.

Options: 'False' (or 0)

'True' (or 1)

Usage:

```
>>> itp.cv.vmlaunchbreak=0
>>> itp.cv.vmlaunchbreak
[HSW_C0_T0] False
[HSW_C0_T1] False
[HSW_C1_T0] False
[HSW_C1_T1] False
```

6.3 Group Control Variables

6.3.1 ishalted

Definition Boolean value to determine if all cores/threads in the group are halted (True) or not (False). Read-Only

Options: None.

Usage:

```
>>> itp.groups[0].cv.ishalted
```

```
[AMT_M0_C0_T0] False
[AMT_M0_C1_T0] False
[AMT_M1_C0_T0] False
[AMT_M1_C1_T0] False
```

6.3.2 isrunning

Definition Boolean value to determine if all cores/threads in the group are running (True) or not (False). Read-Only

Options: None.

Usage:

```
>>> itp.groups[0].cv.isrunning
[AMT_M0_C0_T0] True
[AMT_M0_C1_T0] True
[AMT_M1_C0_T0] True
[AMT_M1_C1_T0] True
```

6.3.3 targpower

Definition Display the status of target system power. Read-only

Options: None.

Remarks: This read-only variable returns the state of power good in the current core group by polling the power good signal from the current JTAG chain.

Usage:

```
>>> itp.groups[0].cv.targpower
True
```

7 DAL CLI Enhancements

The Intel DAL CLI Enhancements module is needed only if the Python console you are using does not provide the functionality. Because of the way such enhancement modules work, only one set of enhancements should be included into a Python console at a time.

7.1 Enhancement Commands

7.1.1 allowCommandCompletion

Description Allow the command completion to be included in the enhancements when the enhancements are enabled with the enable() function. The default is to allow command completion.

Signature: itpii.enhancements.allowCommandCompletion(enable = True)

Arguments: enable - Set to True (the default) to allow the command completion to be enabled.

Returns: None.

Remarks: The expected use of the allowCommandCompletion() function is to disable command completion before enabling the enhancements.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.allowCommandCompletion(False)  ## do not allow
 command completion
>>> itpii.enhancements.enable()
```

7.1.2 allowCommandHistory

Description Allow the preservation of command history across invocations of the Python CLI to be included in the enhancements when the enhancements are enabled with the enable() function. The default is to allow preservation of command history.

Signature: itpii.enhancements.allowCommandHistory(enable = True)

Arguments: enable - Set to True, the default, to allow the preservation of command history to be enabled.

Returns: None.

Remarks: While command history is enabled, all commands are saved to a history file when the Python CLI exits. When the Python CLI is next started and the enhancements module is enabled, the history file is read in and stored in the in-memory buffer of the history, making the previous session's history available to the current Python session.

The expected use of the allowCommandHistory() function is to disable preservation of command history before enabling the enhancements.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.allowCommandHistory(False)  ## do not allow saving
 command history
>>> itpii.enhancements.enable()
```

7.1.3 allowCustomPrompt

Description The custom prompt shows the current aggregated running state of the threads on the target system. The prompt is '>>?' when all threads are running. The prompt is '>>>' when at least one thread is halted. If the custom prompt is not allowed, the Python prompt of '>>>' is left alone, allowing other prompt customizers to operate.

Signature: itpii.enhancements.allowCustomPrompt(enable = True)

Arguments: enabled -- True to allow exception override; otherwise, False. Default is true.

Returns: None.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.allowCustomPrompt(False)  ## do not allow a custom
 prompt
>>> itpii.enhancements.enable()
```

7.1.4 allowExceptionOverride

Description Allow exception override.

Signature: itpii.enhancements.allowExceptionOverride(enable = True)

Arguments: enable: Set to True, the default to allow the exceptions to be overridden

Returns: None.

Remarks: Override the standard exception handler to print only the exception type and message. Set the tracebacklimit attribute to a positive integer to enable the stack trace once again. The tracebacklimit specifies the number of stack entries to display from the stack trace. The normal Python limit is 1000. Exceptions from .Net display the exception type, message, and the first line of the trackback to provide some context to the message.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.allowExceptionOverride(False)
>>> itpii.enhancements.enable()
```

7.1.5 clear

Description Clears the console screen.

Signature: itpii.enhancements.clear()

Arguments: None.

Returns: None.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.clear()
```

7.1.6 disable

Description Disable the enhancements in the Python console if they have not already been disabled. Once disabled, none of the enhancements will operate.

Signature: itpii.enhancements.disable()

Arguments: None.

Returns: None.

Remarks: Once disabled, the Python console will have no enhancements.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.disable()
```

7.1.7 disableCommandCompletion

Description Disable command completion after enhancements have been enabled.

Signature: itpii.enhancements.disableCommandCompletion()

Arguments: None.

Returns: None.

Remarks: This method is called after the enable() function to disable the previously enabled command completion enhancement.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.enable()
>>> itpii.enhancements.disableCommandCompletion()  ## disable command
 completion
```

7.1.8 disableCommandHistory

Description Disable preservation of command history after enhancements have been enabled. When the command history option is disabled, any current history is written to the history file.

Signature: itpii.enhancements.disableCommandHistory()

Arguments: None.

Returns: None.

Remarks: This method is called after the enable() function to disable the previously enabled command completion enhancement.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.enable()
>>> itpii.enhancements.disableCommandHistory()  ## disable saving command
 history
```

7.1.9 disableCustomPrompt

Description Disable the custom prompt after enhancements have been enabled.

Signature: itpii.enhancements.disableCustomPrompt()

Arguments: None.

Returns: None.

Remarks: This method is called after the enable() function to disable the previously enabled custom prompt enhancement.

7.1.10 disableExceptionOverride

Description Disable exception overrides after enhancements have been enabled.

Signature: itpii.enhancements.disableExceptionOverride()

Arguments: None.

Returns: None.

Remarks: This method is called after the enable() function to disable the previously enabled exception overrides enhancement.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.enable()
>>> itpii.enhancements.disableExceptionOverrides()  ## disable exception
 overrides
```

7.1.11 enable

Description Enable the enhancements in the Python console if they have not already been

Signature: itpii.enhancements.enable()

Arguments: None.

Returns: None.

Remarks: Enabling these enhancements will remove any previous enhancements.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.enable()
```

7.1.12 enableCommandCompletion

Description Enable command completion after enhancements have been enabled.

Signature: itpii.enhancements.enableCommandCompletion()

Arguments: None.

Returns: None.

Remarks: This method is called after the enable() function to enable the previously disabled command completion enhancement.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.allowCommandCompletion(False)## do not allow command
 completion
>>> itpii.enhancements.enable()
>>> itpii.enhancements.enableCommandCompletion()    ## enable command
 completion
```

7.1.13 enableCommandHistory

Description Enable saving command history after enhancements have been enabled. When the command history option is enabled, the last history file is read in and put into the history buffer.

Signature: itpii.enhancements.enableCommandHistory()

Arguments: None.

Returns: None.

Remarks: This method is called after the enable() function to enable the previously disabled command history enhancement.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.allowCommandHistory(False)  ## do not allow saving
 command history
>>> itpii.enhancements.enable()
```

7.1.14 enableCustomPrompt

Description Enable a custom prompt after enhancements have been enabled.

Signature: itpii.enhancements.enableCustomPrompt()

Arguments: None.

Returns: None

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.allowCustomPrompt(False)  ## do not allow custom
 prompt
>>> itpii.enhancements.enable()
>>> itpii.enhancements.enableExceptionOveride()  ## enable custom prompt
```

7.1.15 enableExceptionOverride

Description Enable exception overrides after enhancements have been enabled.

Signature: itpii.enhancements.enableExceptionOverride()

Arguments: None.

Returns: None.

Remarks: This method is called after the enable() function to enable the previously disabled exception override enhancement.

Usage:

```
>>> import itpii.enhancements
>>> itpii.enhancements.allowExceptionOverride(False)  ## do not allow
 exception overrides
>>> itpii.enhancements.enable()
>>> itpii.enhancements.enableExceptionOveride()  ## enable exception
 overrides
```

7.1.16 eraseCommandHistory

Description Erase the command history stored in memory.

Signature: itpii.enhancements.commandhistory.eraseCommandHistory()

Arguments: None.

Returns: None.

Remarks: The command history is erased without warning or confirmation. Tread carefully.

Usage:

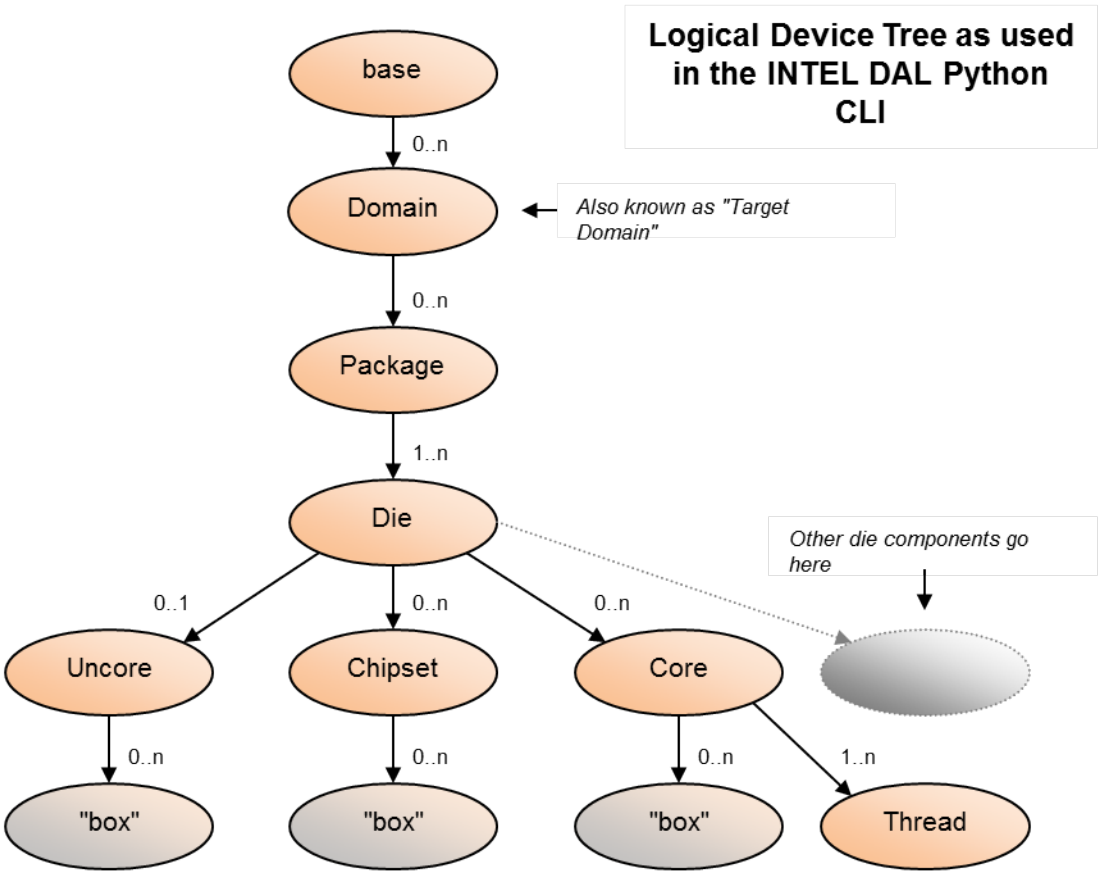
```
>>> itpii.enhancements.commandhistory.eraseCommandHistory()
```

Appendix A. Device Trees

The Intel DAL Python Module views the target or targets connected to a host as a collection of nodes in a tree, starting with some top node or root. For the Intel DAL Python module, the root node is the ItpII object returned from the itpii.baseaccess() factory function. The ItpII object corresponds to the "base" node in the diagram. The following diagram shows the relationships between the various nodes of the tree.

A.1 Logical Device Tree

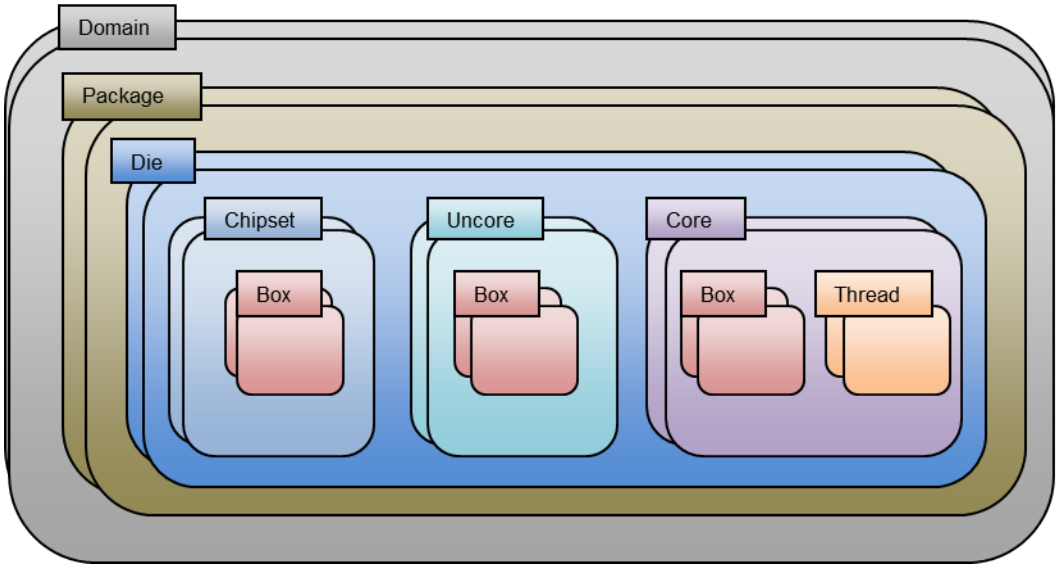
Figure A.1. Logical Device Tree



The term "box" is a stand-in for a collection of features specific to the device. For example, a chipset might have a Management Engine (ME) box and a QPI Link box. An Uncore might have a Power Control Unit (PCU) box.

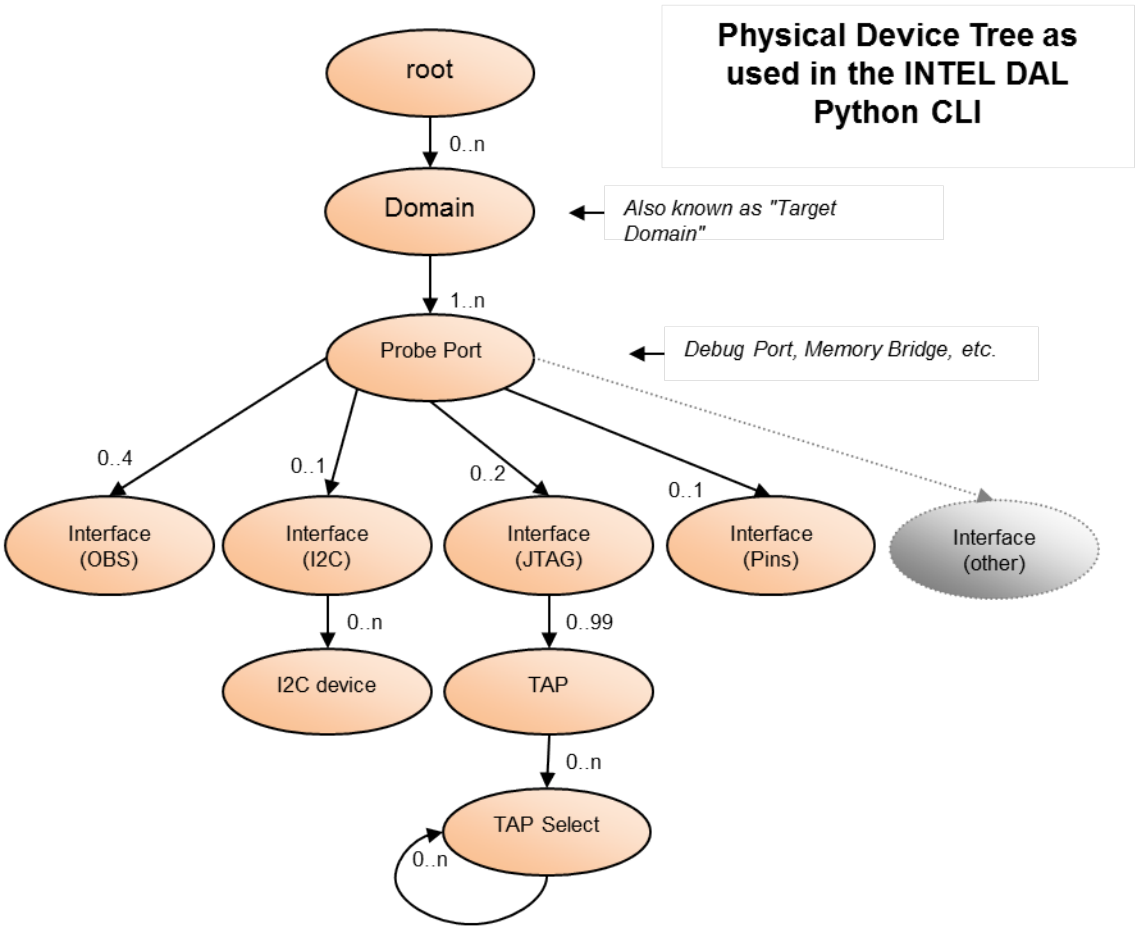
Figure A.2. Alternative View

The term "box" is a stand-in for a collection of features specific to the device. For example, a chipset might have a Management Engine (ME) box and a QPI Link box. An Uncore might have a Power Control Unit (PCU) box.



A.2 Physical Device Tree

Figure A.3. Physical Device Tree



The physical device tree is available through a logical node (typically the package, die, uncore, core, thread nodes) through two methods. The first method gets the probe port physical node, given the interface type (OBS, I2C, JTAG, Pins, etc.). The second method gets the physical device node for a given interface type (for example, return the tap select object representing the physical device on a JTAG interface).

Note

The physical device tree is available through a logical node (typically the package, die, uncore, core, thread nodes) through two methods. The first method gets the probe port physical node, given the interface type (OBS, I2C, JTAG, Pins, etc.). The second method gets the physical device node for a given interface type (for example, return the tap select object representing the physical device on a JTAG interface).

Key attributes on each node:

1. Logical device node

   a. getProbePortForInterface()
   b. getPNodeForInterface()
2. Physical Target Domain node
   a. debugPorts[]
3. Debug Port type of Probe Port node
   a. obs[]
   b. i2c[]
   c. jtag[]
   d. pins[]
4. I2C node
   a. devices[]
5. JTAG node
   a. taps[]
6. Tap node
   a. tapSelect[]
7. Tap Select node
   a. tapSelect[]

Appendix B. Examples

B.1 Memory Access Example

This example shows off several features of the Python CLI, with the primary focus on accessing memory from a single thread. The focus is on using the commands available to perform the work.

How to use:

1. Save the example code to the name demo.py in a folder.
2. Start the Python CLI.
3. Enter the following Python code on the command line:

```
import sys
sys.path.append(r"<PATH TO THE FOLDER CONTAINING DEMO.PY")
import demo
demo.runmemcopy()
```

Memory Access Example Code

```
# Module demo.py
import itpii
itp = itpii.baseaccess()

# Create global alias for thread 0
p0 = itp.threads[0]
byteWidth = 1 # 1 byte in size

def copymemory(thread, fromaddress, toaddress, length):
    # Copy a range of memory from one address to another.
    # Tolerates overlapping blocks.

    # Arguments:
    #  thread       thread object whose memory is to be accessed.
    #  fromaddress  Source address of the memory block
    #  toaddress    Destination address of the memory block
    #  length       Number of elements to copy.

    d = thread.memblock(fromaddress, length, byteWidth)
    thread.memblock(toaddress, length, byteWidth, d)

def clearmemory(thread, fromaddress, length):
    # Clear a range of memory.

    # Arguments:
    #  thread       thread object whose memory is to be accessed.
    #  fromaddress  Source address of the memory block
```

```
    #  length       Number of elements to erase.

    thread.memblock(fromaddress, length, byteWidth, 0)

def runmemcopy():
    #Simple demo of memory access with memory block read, dump, and write.

    # If thread 0 is running, halt it.
    if p0.cv.isrunning:
        print("Halting thread %s"%p0.name)
        p0.halt()

    # Source address is the current instruction pointer on thread 0.
    # This could also be expressed as "$" (a string with a dollar sign inside).
    fromaddress = p0.cip
    # The destination address, relatively safe on live hardware.
    toaddress = "0x1000P"
    # The number of elements to handle.  The size of each element is
    # controlled by the byteWidth variable.
    length = 64

    clearmemory(p0, toaddress, length)

    print("Source memory block:")
    p0.memdump(fromaddress, length, byteWidth)

    print("Destination memory block (before copy):")
    p0.memdump(toaddress, length, byteWidth)

    # Do the actual copy.
    copymemory(p0, fromaddress, toaddress, length)

    print("Destination memory block (after copy):")
    p0.memdump(toaddress, length, byteWidth)
    print("Resuming thread %s"%p0.name)

    # Resume thread 0.
    p0.go()
```

B.2 Event Handling Example

This example shows how to subscribe to all of the events exposed by the ITP Commands Library service.

How to use:

1. Save the example code to the name eventsdemo.py in a folder.
2. Start the Python CLI.
3. Enter the following Python code on the command line:

```
import sys
sys.path.append(r"<PATH TO THE FOLDER CONTAINING EVENTSDEMO.PY")
```

```
import eventsdemo
eventsdemo.subscribeToEvents()
eventsdemo.unsubscribeFromEvents()
```

Note

In this example, the event handlers show the event in a similar way to how the python CLI shows the events except the text is prefixed with "Event:", just to show the handler is getting called. In a more typical usage, a script might respond in some other way to the appropriate event.

Note

The only run control event received is "go". What would have been the "halt" run control event is actually a break event. If you need to recognize a "halt" event, examine the message in the break event for the string "Halt Command break".

```


# Get the Python CLI
import itpii
itp = itpii.baseaccess()

##############################################
# IMPORTS
##############################################
from itpii import (
     HardwareEvents,
     RunControlEvents,
     ReconfigEvents
)


##############################################
# Event Handlers
##############################################

def reportEvent(message):
    # This code will display the message.

    print("Event: %s"%message)


def reportHardwareEvents(args):
    # Callback to receive hardware events that are reported on the
    # command line.

    # Arguments:
    #  args -- A HardwareEventArgs class with two attributes:
    #          args.iEvent     - An integer specifying the type of event.
    #                            Can be one of the following values (also
    #                            defined in the HardwareEvents class):
    #                              0 = "\tNote:\tClock Loss occurred"
    #                              1 = "\tNote:\tClock Restore occurred"
    #                              2 = "\tNote:\tPower Loss occurred"
    #                              3 = "\tNote:\tPower Restore occurred"
    #                              4 = "\tNote:\tTarget reset has occurred"
    #                              5 = "A TAP.7 device reset occurred"
```

```
    #                               6 = "A TAP.7 device went offline"
    #          args.iDebugPort - The debug port on which the event occurred.


    eventType = args.iEvent
    if eventType == HardwareEvents.BCLKFailed:
        reportEvent("\tNote:\tClock Loss occurred")
    elif eventType == HardwareEvents.BCLKRestored:
        reportEvent("\tNote:\tClock Restore occurred")
    elif eventType == HardwareEvents.PowerFailed:
        reportEvent("\tNote:\tPower Loss occurred")
    elif eventType == HardwareEvents.PowerRestored:
        reportEvent("\tNote:\tPower Restore occurred")
    elif eventType == HardwareEvents.ResetOccurred:
        reportEvent("\tNote:\tTarget reset has occurred")
    elif eventType == HardwareEvents.Tap7ResetOccurred:
        reportEvent("Tap 7 reset occurred")
    elif eventType == HardwareEvents.Tap7DeviceOffline:
        reportEvent("Tap 7 device offline")



def reportRunControlEvents(args):
    # Callback to receive run control events that are reported on the
    # command line.

    # Arguments:
    #   args  A RunControlEventArgs class with four attributes:
    #          args.iEvent       An integer specifying the type of event.
    #                            Can be one of the following values (also
    #                            defined in the RunControlEvents class):
    #                              1 = Resumed
    #          args.iDebugPort  the debug port on which the event occurred.
    #          args.iScanChain  the scan chain on which the event occurred.
    #          args.iDeviceID   ID of the device on which the event occurred.


    iEvent = args.iEvent
    if iEvent == RunControlEvents.Halt:
        reportEvent("Note:\tHalt event received")
    elif iEvent == RunControlEvents.Go:
        reportEvent("Note:\tGo event received")


def reportBreakEvents(args):
    # Callback to receive break events that are reported on the
    # command line.

    # Arguments:
    #  args  A BreakEventArgs class with the following attributes:
    #          args.message  A string to be displayed, containing
    #                         information about the break.
    #

    reportEvent("\t" + args.message)


def reportErrorEvents(args):
    # Callback to receive error events that are reported on the
    # command line.
```

```
    #
    # Arguments:
    #  args  A ErrorEventArgs class with the following attributes:
    #          args.message  A string to be displayed, containing
    #                        information about the error.


    reportEvent("\t" + args.message)


def reportReconfigEvents(args):
    # Callback to receive reconfiguration events that are reported on the
    # command line.
    #
    #Arguments:
    #  args    A ReconfigEventArgs class with the following attributes:
    #          args.iEvent     - 0 = reconfig started, 1 = reconfig ended
    #          args.iDebugPort - Index of the debug port affected by the
    #                            reconfiguration.


    state = "finished"
    if args.iEvent == ReconfigEvents.Started:
        state = "started"

    if args.iDebugPort != -1:
        msg = "Note:\tReconfiguration on debug port %d has %s"%(args.iDebugPort, state)
    else:
        msg = "Note:\tConfiguration on all debug ports has %s"%(state)
    if args.deviceTreesChanged:
        msg += "\n\tWarn:\tDevice trees have changed.  Existing device nodes are now invalid."

    reportEvent("\t" + msg)


##############################################
# SETUP AND SHUTDOWN PROCEDURES
##############################################

def subscribeToEvents():
    itpii.subscribeToHardwareEvents(reportHardwareEvents)
    itpii.subscribeToRunControlEvents(reportRunControlEvents)
    itpii.subscribeToBreakEvents(reportBreakEvents)
    itpii.subscribeToErrorEvents(reportErrorEvents)
    itpii.subscribeToReconfigEvents(reportReconfigEvents)


def unsubscribeFromEvents():
    itpii.unsubscribeFromHardwareEvents(reportHardwareEvents)
    itpii.unsubscribeFromRunControlEvents(reportRunControlEvents)
    itpii.unsubscribeFromBreakEvents(reportBreakEvents)
    itpii.unsubscribeFromErrorEvents(reportErrorEvents)
    itpii.unsubscribeFromReconfigEvents(reportReconfigEvents)
```

B.3 Heterogeneous Run Control Example

The following example outputs show how heterogeneous core support can be used in the Python CLI. There is a by Global and by Group example.

How to use:

```
>>? itp.groups # view the current list of groups
[<NodeCoreGroup: GPC      (4 cores, 4 threads, 1 devicetype:  'AMT')>,
 <NodeCoreGroup: GT       (1 core,  1 thread,  1 devicetype:  'CHT_GEN_GT_1_GUC')>,
 <NodeCoreGroup: ISH      (1 core,  1 thread,  1 devicetype:  'CHT_ISH_LMT')>]
>>?
>>? itp.cv.coregroups # view the current "global context" group
GPC
>>?
>>? itp.threads # view the threads included in "GPC" group
[<NodeThread: AMT_M0_C0_T0 (core = 0, thread = 0, alias = 'P0', stepping = 'A0')>,
 <NodeThread: AMT_M0_C1_T0 (core = 1, thread = 0, alias = 'P1', stepping = 'A0')>,
 <NodeThread: AMT_M1_C0_T0 (core = 0, thread = 0, alias = 'P2', stepping = 'A0')>,
 <NodeThread: AMT_M1_C1_T0 (core = 1, thread = 0, alias = 'P3', stepping = 'A0')>]
>>?
>>? itp.allthreads # view all threads, regardless of current itp.cv.coregroups
[<NodeThread: AMT_M0_C0_T0 (core = 0, thread = 0, alias = 'P0', stepping = 'A0')>,
 <NodeThread: AMT_M0_C1_T0 (core = 1, thread = 0, alias = 'P1', stepping = 'A0')>,
 <NodeThread: AMT_M1_C0_T0 (core = 0, thread = 0, alias = 'P2', stepping = 'A0')>,
 <NodeThread: AMT_M1_C1_T0 (core = 1, thread = 0, alias = 'P3', stepping = 'A0')>,
 <NodeThread: CHT_ISH_LMT_C0_T0 (core = 0, thread = 0, alias = 'P4', stepping = 'A0')>,
 <NodeThread: CHT_GEN_GT_1_GUC_C0_T0 (core = 0, thread = 0, alias = 'P5', stepping = 'A0')>]
>>?
>>? itp.halt() # try a global context run control command
        [AMT_M0_C0_T0]  Halt Command break at 0xF000:000000000000FFF0
        [AMT_M0_C1_T0]  Halt Command break at 0xF000:000000000000FFF0
        [AMT_M1_C0_T0]  Halt Command break at 0xF000:000000000000FFF0
        [AMT_M1_C1_T0]  Halt Command break at 0xF000:000000000000FFF0
>>>
>>> itp.go()
>>?
>>? itp.cv.coregroups = "ish" # change to a different group, "ISH"
>>?
>>? itp.cv.coregroups # read it back, to confirm the change
ISH
>>?
>>? itp.threads # view the threads included in the "ISH" group
[<NodeThread: CHT_ISH_LMT_C0_T0 (core = 0, thread = 0, alias = 'P4', stepping = 'A0')>]
>>?
>>? itp.halt() # try a global context run control command
        [CHT_ISH_LMT_C0_T0]  Halt Command break at 0xF000:000000000000FFF0
>>?
)
```

Note

The following example changes control variable coregroups to help control the threads operated at a thread context.

```
>>? itp.groups # view the current list of groups
[<NodeCoreGroup: GPC      (4 cores, 4 threads, 1 devicetype:  'AMT')>,
 <NodeCoreGroup: GT       (1 core,  1 thread,  1 devicetype:  'CHT_GEN_GT_1_GUC')>,
 <NodeCoreGroup: ISH      (1 core,  1 thread,  1 devicetype:  'CHT_ISH_LMT')>]
>>?
>>? itp.cv.coregroups # view the current "global context" group
GPC
>>?
>>? itp.groups[0].threads # view the threads included in group[0] group (GPC)
[<NodeThread: AMT_M0_C0_T0 (core = 0, thread = 0, alias = 'P0', stepping = 'A0')>,
 <NodeThread: AMT_M0_C1_T0 (core = 1, thread = 0, alias = 'P1', stepping = 'A0')>,
 <NodeThread: AMT_M1_C0_T0 (core = 0, thread = 0, alias = 'P2', stepping = 'A0')>,
 <NodeThread: AMT_M1_C1_T0 (core = 1, thread = 0, alias = 'P3', stepping = 'A0')>]
>>?
>>? itp.groups[1].threads # view the threads included in group[1] group (GT)
[<NodeThread: CHT_GEN_GT_1_GUC_C0_T0 (core = 0, thread = 0, alias = 'P5', stepping = 'A0')>]
>>?
>>? itp.groups[2].threads # view the threads included in group[2] group (ISH)
 [<NodeThread: CHT_ISH_LMT_C0_T0 (core = 0, thread = 0, alias = 'P4', stepping = 'A0')>]
>>?
>>? itp.groups[0].halt() # try a group context run control command
        [AMT_M0_C0_T0]  Halt Command break at 0xF000:0000000000000FFF0
        [AMT_M0_C1_T0]  Halt Command break at 0xF000:0000000000000FFF0
        [AMT_M1_C0_T0]  Halt Command break at 0xF000:0000000000000FFF0
        [AMT_M1_C1_T0]  Halt Command break at 0xF000:0000000000000FFF0
>>>
>>> itp.groups[0].cv.isrunning # check if we are running
[AMT_M0_C0_T0] False
[AMT_M0_C1_T0] False
[AMT_M1_C0_T0] False
[AMT_M1_C1_T0] False
>>>
>>> itp.groups[0].cv.ishalted # check if we are halted
[AMT_M0_C0_T0] True
[AMT_M0_C1_T0] True
[AMT_M1_C0_T0] True
[AMT_M1_C1_T0] True
>>>
>>> itp.groups[0].brnew("0x80P","io")  # setup a breakpoint
>>>
>>> itp.groups[0].brget() # Make sure they are setup correctly
[AMT_M0_C0_T0] br["#001"] = 0x0000000000000080P io global dbreg = 0, enabled = True
[AMT_M0_C1_T0] br["#002"] = 0x0000000000000080P io global dbreg = 0, enabled = True
[AMT_M1_C0_T0] br["#003"] = 0x0000000000000080P io global dbreg = 0, enabled = True
[AMT_M1_C1_T0] br["#004"] = 0x0000000000000080P io global dbreg = 0, enabled = True
>>>
>>> itp.groups[0].go()  # see if we break after probe mode exit
>>?     [AMT_M0_C0_T0]  Debug register break on I/O reads or writes at 0xF000:0000000000000080
>>?
>>? itp.groups[2].halt() # try another group context run control command
        [CHT_ISH_LMT_C0_T0]  Halt Command break at 0xF000:000000000000FFF0
>>>
>>> itp.groups[2].go()
>>?
```

Appendix C. Definitions and Terms

Term	Definition
CLI	Acronym for Command Line Interface. A CLI is typically a window in which the user types commands and the results of those commands are displayed in the same window. The window is always pure text, although the text can be colored. The Windows Command Prompt and the Linux shell are examples of a CLI. Sometimes called a REPL (Read-Eval-Print-Loop).
Command Completion	The command completion feature is a way to explore an object-oriented software structure on a command line interface. Typing a word (or part of a word) then pressing the command completion character (typically the Tab key) displays a list of what can be typed next. Very useful for large, complex object hierarchies, especially if the hierarchies are dynamically created and cannot be committed to a static help file.
Commands Library	An intermediate layer between the Python CLI and the Intel DAL software stack. The Commands Library is written in .Net and presents almost all of the legacy ITP commands as static methods on various classes. The back-end of the library can be focused on legacy ITP or Intel DAL. The Commands Library can be used directly by other applications as well.
Control Variable	A configuration option that can affect a debug port, a device under test, or the behavior of the program itself. Legacy ITP used the term "control variable" for all these.Note: In Intel DAL, the term "control variable" is itself subject to change. Intel DAL uses the general term "Software Configuration" to refer to what control variables do and Intel DAL may choose to separate out the various kinds of controls that can be used into different categories and labels.
CPython	A version of the Python language implemented in the C language by the Python Software Foundation. This is the most widely recognized version of Python. When someone says "Python" with no other qualifiers, they are usually referring to CPython. Intel DAL is expected to use CPython v2.6, which can be obtained from the Python Software Foundation at http://python.org/.
IronPython	A Microsoft implementation of the Python language that is in the .Net runtime. The advantage of IronPython is that integrates seamlessly with any .Net assembly. IronPython 2.6 can be obtained from http://www.ironpython.net/.
ITP	(legacy ITP) In-Target Probe, a hardware-based solution for debugging target systems very close to the metal. In the context of this document, ITP refers to the software package that comes with the hardware and that implements the ITP scripting language. ITP runs only on Windows.
Intel DAL	The next generation ITP that is written entirely in C# (a .Net language). Intel DAL provides an entirely different way of interacting with a target system, using Python as a scripting language. Intel DAL is expected to run on multiple operating systems such as Windows, Linux, and MacOS.
namespace	A namespace is a way to group variables, functions, and types in such a way so as to keep them separate from all other variables, functions, and types with the same name in other namespaces. For example, a variable named 'foo' in namespace 'mystuff' is different from a variable named 'foo' in namespace 'otherstuff'.

Term	Definition
	In python or C#, dotted notation is used to access an attribute of a namespace. For example, 'mystuff.foo' and 'otherstuff.foo'.
Python.Net	An add-in for CPython that allows CPython-based programs to interact with the .Net runtime. Not quite as seamless as IronPython but pretty close. The original version of python.net can be obtained from http://pythonnet.sourceforge.net/. The Intel DAL team has modified python.net to fix some issues and extend some capabilities. This version of python.net will be provided with Intel DAL. In the meantime, send a request to Stephen Lepisto to get a copy of Intel's python.net.

Appendix D. A Brief Introduction to Python

Python is a language created over twenty years ago by Guido van Rossum, who designed and continues to guide Python's development to this day. Python is a powerful, flexible, and easy-to-use language that is often thought of as a scripting language, although it is far more than that.

D.1 Basic Programming Concepts

Python is a fully featured programming language and that means being able to express concepts such a conditional statements, loops, and functions. Python uses indentation to indicate when the body of a function, loop, of conditional statement starts and ends. Indentation is also used to identify class functions and attributes as belonging to that class. Indentation must be consistent and accurate. Once you get past this point, entering and reading Python becomes quite easy.

Tip

Always use spaces for indentation. Spaces are portable to any editor and always have the same meaning; tabs can mean 2, 4, or even 8 spaces, depending on the setting in the editor.

Tip

Never mix tabs and spaces in the same code. Just to reinforce the first tip.

D.2 Basic Types

Python has five basic types: integer, floating point, complex, boolean, and string.

Integers: For Python 2.x, the integer type is broken up into int and long, where int is a 32-bit value and long is an arbitrarily long integer (using however many bits it needs to be expressed; and yes, that can include hundreds or even thousands of bits). (In Python 3.x, the 32-bit integer was dropped in favor of a single integer type).

Floating Point: If Python sees a decimal point in a number, Python assumes it is a floating point number. It typically uses a double precision but it is dependent on the operating system the code is running on. The type name for a floating point number is float.

Complex Complex numbers are mathematically impossible and have thus proved to be extremely useful in a variety of situations. Python CLI doesn't make use of these. The type name for a complex number is complex.

Boolean A true/false value type. The constants True and False are used for the boolean type. All comparisons result in a boolean type. The type name for a boolean value is bool.

String A string is a sequence of characters. Python 2.x assumes strings are extended ASCII, one byte per character. Python 2.x also supports unicode strings but these have to be explicitly created with the unicode() type (for example, unicode("hello") creates a unicode string: u"hello"). (Python 3.x and IronPython treat all strings as unicode.) The type name for a string value is str. The type name for a unicode string is unicode.

Strings are surrounded by single quotes, double quotes, or triple quotes. Whatever the starting quote is, the string must end with the same character. triple quotes (three double quotes in row: """) indicate the start of some arbitrarily long string that can span multiple lines, with everything inside the triple quotes being inside the string, including single quotes, double quotes and newlines.

D.3 Modules

Modules are the primary building blocks of code in Python. Modules represent the contents of a single file. When the module is loaded, the contents of that file are stored under the namespace whose name is the same as the module (although this can be changed). Modules have no general format, although it is convention to put a string at the top of the file that describes the contents of the module.

To put this in different terms, the code in a module exists in its own little universe and has no knowledge of what is outside that universe. To use elements from other modules (universes), a module must pull in or import those elements to make use of them.

D.4 Loading a Module

The import statement is used to load a module or a portion of a module. The general syntax of the import statement looks like this:

```
import modulename [as alias]
from modulename import element
```

The first form loads the contents of the file modulename.py into memory and stores the contents of that module under the namespace modulename (if the as keyword is used, the namespace is exposed as whatever alias is set to). The types and functions in that module are then accessible by typing the namespace followed by a period followed by the type or function. For example, modulename.DataType or modulename.function().

The second form is used to bring in a type, function, or attribute from the module and put that type, function, or attribute into the current namespace. Be careful of this; although it

means you do not have to type modulename.element to use the element, importing the element could overwrite an element in the current namespace (or an element in the existing namespace could overwrite the imported element if that first element is defined after the import statement).

It is possible to use the form from modulename import *, which imports everything in the module into the current namespace. This is strongly discouraged by the Python community in general because of the extreme ease with which it is possible to have elements from a module conflict with elements in the current namespace.

D.5 Loop Types

Python supports two kinds of loops: for loops and while loops. For loops iterate of a range of things. While loops iterate until some condition is satisfied.

For Loops The general syntax for a Python for loop looks like this:

```
for var in sequence:
    statements
```

sequence is anything that acts like a container of items, such as a list, dictionary, set, or tuple. The variable var is set to each item in the sequence in turn and the statements are executed with that variable. The for loop is used to visit every item in a sequence in order.

The range type can be used to construct a sequence based on a range of numbers. For example, range (0, 100) (or just range(100)) creates a sequence of integers from 0 to 100, not including 100. The range class allows you to create for loops over a sequence of numbers. For example:

```
                [
for var in range(0, 10):
    print(var)
    Type help(range) for more details on the range class.
```

Type help(range) for more details on the range class.

While Loops The general syntax for a Python while loop looks like this:

```
while (condition):
    statements
```

The statements in the while loop are executed repeatedly until the condition changes from true to false. Read it as "while the condition is true, ex-

ecute the statements". The assumption is that at least one statement will cause the condition to change from true to false so the while loop will exit. Use the break statement to break out early from a while loop (or a for loop).

D.6 If Statements

The if statement is the fundamental building block for making decisions in Python. The general syntax looks like this:

```
if expr1:
    statements1
elif expr2:
    statements2
else:
    statements3
```

This is read as "if expr1 is true, execute statements1, else if expr2 is true, execute statements 2, else execute statements3". Either or both the elif and else parts can be left out if not needed.

D.7 Functions

A function is a collection of programming statements that accomplishes some task when executed or called. The general syntax looks like this:

```
def functionName1():
    statements1

def functionName2(arg1, arg2, arg3):
    statements2

def functionName3(arg1 = None, arg2 = 3, arg3 = "Hello"):
    statements3
```

functionName1 is a function that takes no arguments. functionName2 is a function that takes three arguments, all of which must be specified. functionName3 is a function that takes three arguments with default values provided if the arguments are not specified.

Calling a function always uses parenthesis like this:

```
functionName1()
functionName2(1, 2, "hello")
functionName3()
functionName3(1)
functionName3(1, 9)
```

```
functionName3(1, 9, "there")
```

The calls to the functionName3 function show how the default parameters come into play.

Function parameters can also be passed by keyword, where the name of the argument is explicitly assigned to when calling the function. For example:

```
functionName3(arg3 = "there", arg1 = 1, arg2 = 9)
```

The order of the arguments is not important if using keyword assignment. Keyword arguments are handled after so-called positional arguments are processed. For example:

```
functionName3(1, arg3 = "text")
```

In this case, arg1 gets the 1, arg2 uses the default of 3, and arg3 gets "text".

Functions can return values as well by using the return statement to return a value.

```
def functionName4():
    return "Hi"
```

Functions can also return multiple values using a tuple, which is an array of values. For example:

```
def functionName5():
    return (1, "Hello")
```

Multiple return values are handled with the following syntax:

```
(num, text) = functionName5()
```

D.8 Class Types

A class is defined with the following general syntax:

```
```

```
class ClassName(object):
    def __init__(self):
        pass # A "do nothing" command.  Replace with actual functionality

    def method1(self):
        pass

    def method2(self, arg, arg):
        pass
```

ClassName is the name of the class. The class is derived from object, the most basic type in Python. A class can derive from other classes as well but they should ultimately be derived from object. The __init__() function is the constructor. The self argument refers to the instance of the class being constructed. Within the class, all the methods that do something with the class must have a self argument as the first argument and then use the self argument to access the values in the specific instance of the class. For example:

```
class Device(object):
    def __init__(self, node):
        self._node = node # save the node for this instance

    def getID(self):
        return self._node.id # use the saved node in this instance to get the id

    def setID(self, newID):
        self._node.id = newID
```

To create an instance of a class, use the following notation:

```
inst = ClassName()
```

Here is an example of constructing an instance of the Device class example (where 'node' is defined someplace earlier):

```
dev = Device(node)
```

D.9 Types and Variables

For a lot of applications, what type a variable is in python is not as important as you might think. Python is pretty good at automatically converting among the basic types. However, it is useful to know about python's basic variable types.

Python uses a few basic data types (integers, booleans, strings, floating point, and complex numbers) and a rich selection of other types based on the object-oriented idea of class-

es. Everything in Python is a program object, from a single integer value to a user-defined class. Event types (class definitions) are program objects. The only exception is variables: variables are always references to objects. For example:

```
a = obj1
b = a
```

Variables a and b both refer to the same object, obj1. Now, change variable b:

```
b = 1
```

Variable b now holds a reference to the integer 1 and no longer references object obj1. Change b again:

```
b = int
```

Variable b now holds a reference to the integer type. What can you do with this? Well, the variable b is now essentially an alias for the int type. Using the int type, it is possible to convert a number stored as a string into an actual integer. For example:

```
n = int("5")
```

Variable n now holds 5 because int() creates or constructs an integer from the string "5". Since variable b holds a reference to the int type, you can do this:

```
c = b("5")
```

Variable c is set to the integer 5 in exactly the same way that n was set. Variable b is essentially an alias to the int type and does whatever the int type does until variable b is changed to refer to a different object.

Variables are not typed, only what the variable refers to is typed. The following four lines in sequence are legal Python:

```
n = "Hello"
```

```
n = 4.5
n = True
n = 1
```

The variable n holds, in turn, a string, a floating point, a boolean, and an integer. The variable n is just a reference, what changes is the type of whatever the variable n refers to. This means you can use variable n in whatever way you normally would with an object of the type variable n refers to. For example, integers can be added: 1 + 1. Since variable n is an integer (due to the last assignment n = 1), variable n can be added to: n + 1, and you will get the same result so long as n refers to the integer 1. Now change variable n to a string. The length of a string can be obtained with the len() function: len("Hello"). Since variable n refers to a string, len(n) will also return the length of the string. If variable n was an integer, len(n) would return an error because integer types do not support the len() function.

In summary, everything in Python are objects except variables. Variables are references to objects. Variables can be changed at any time to any object, regardless of the type of the object.

D.10 Code Organization

Python uses the concept of namespaces to organize all code. There is a main or global namespace. There is also the concept of current namespace, which just happens to be the namespace that contains the code currently being executed. A namespace is a way to group attributes, functions, and types so they do not conflict with other namespaces that have similarly named attributes, functions, and types.

Python code comes in modules, which corresponds to a file. Each module has its own namespace when loaded. Access to anything in a module is done through a "dotted" notation, for example, sys.path (the path attribute in the sys module). Another module might have a path attribute, for example, mymod.path. The path attribute in the mymod module is different from the path attribute in the sys module; by using the dotted notation, it becomes very clear which path attribute is being referred to. Modules and namespaces are very important in Python and allow the many thousands of Python applications to live nicely with each other in the same Python environment.

In Python, modules are loaded by importing them (specifically, the module is loaded and imported into the current namespace). Use the command import modulename to load a file called modulename.py into memory and stored under the namespace modulename. If you want only one particular attribute, function, or type from a module, use from modulename import item (where item is the name of an attribute, function, or type). The imported item is added directly to the current namespace and you access it without using modulename. Note, however, that as you use more and more from x import y statements, the current namespace gets more and more crowded and it gets easier for two names to collide. When that happens, the most recently imported name takes precedence, and the older name is lost. For example, using the earlier examples of mymod.path, the following lines creates a path attribute in the current namespace, then imports the path attribute from the sys module followed by the path attribute from the mymod module. Unfortunately, the path from the mymod module replaces the path attribute from the sys module in the current namespace, and both imports wipe out the original path attribute at the top:

```
path = "A path"
from sys import path
from mymod import path
```

This overwrite happens without any warnings. This is why it is much preferred in Python to import the entire module and use the module as a namespace. For example:

```
path = "A path"
import sys
import mymod
sys.path
mymod.path
path
```

D.11 Python Types

Python supports the object-oriented idea of classes. A class is formally known as a type definition. A class contains functions and attributes that are closely associated with each other and in that regard is similar to a module (which groups functions, attributes and types that are associated with each other). However, whereas a module can only be imported once, a class can have multiple instances of the class created and stored as individual programming objects. Classes also use a dotted notation to access the attributes and functions within that class. For example, sys.path.append("a path") calls the append() function on the object (specifically, an instance of the list class) referred to by the path variable in the sys module.

D.11.1 Exploring Python

Python is a dynamically typed language. This means the modules and classes in Python can be changed after they have been created. It also means an instance of a type can be changed after it has been created (for example, a new attribute can be added to an instance of a class and only that instance would have that new attribute). This is a very useful concept and provides a degree of freedom not possible in languages with much more strict rules about data types (such as C, C++, C#, Java, and Visual Basic). However, this freedom requires increased responsibility and discipline; otherwise the resulting code could be very difficult to maintain and fix. Because Python is a dynamically typed language, Python provides functions for exploring programming objects after they have been created.

D.11.2 dir() function

One of the two most useful functions provided by Python is the dir() function. This provides a list of all of the attributes and functions attached to a programming object. For example, dir(sys) shows all the attributes and functions in the sys module (yes, modules are programming objects too). dir(sys.path) shows all the attributes and functions on the path object in the sys module.

D.11.3 type() fuction

A third useful function is the type() function. This displays the underlying type of any object. For example:

```
(sys.path)
```

displays

```
<type 'list'>
```

indicating the path attribute on the sys module is an instance of the list class.

Together, dir(), help(), and type() allow you to explore most of Python. There are also extensive tutorials and help online for Python.See also:

Wikipedia article python syntax and semantics

Appendix E. Differences in ITP Scripting Language

Python is the scripting language used in Intel DAL. Python has all of the language functionality expressed in the legacy ITP scripting language; however, how that functionality is used is different. This section covers the primary differences between the two languages. This is not intended as a tutorial on Python.

In Python, there are no braces. Instead, blocks of code are indicated by consistent indentation. Also, in Python every control statement (such as for, if, while, def, etc.) ends with a colon to indicate the start of a control block. This is reflected in the examples that follow.

See also Appendix D: A Brief Introduction to Python.

E.1 Numbers

In ITP, it was possible to change the numerical base for all entered and displayed numbers and the default was base 16 (hexadecimal). In python, the default is base 10 and the base cannot be changed. What this means is that all non-decimal numbers in python must have an appropriate prefix. Like ITP, python supports base 2, base 10, and base 16.

```
// ITP Scripting
[P0]?base
hex
[P0]?11 // base 16 (default)
00000011
[P0]?0x11 // explicit base 16
00000011
[P0]?0n11 // explicit base 10
0000000b
[P0]?0y11 // explicit base 2
00000003
```

In Python:

```
# Python
>>> 11   # base 10 (default)
11
>>> 0x11 # explicit base 16
17
>>> 0b11 # explicit base 2
3
>>> hex(11)  # convert to hex (note: output is a string)
'0xb'
>>> bin(11)  # convert to binary (note: output is a string)
'0b1011'
```

E.2 Control Statements

ITP supported the following control statements:

```
if { } else { }
for { }
do { } while
while { }
```

There are similar control statements in python, although they are expressed a little differently:

```
if..elif..else
for
while
```

Note
See the next section for more detail on the for loop construct.

Here are some examples from both languages in a side-by-side comparision.

ITP	Python
`define ord4 n` `for (n = 0; n < 10; n += 2) {` `  // do something` `}`	`for n in range(0, 10, 2):` `  pass # do something`
`// Given: an array of 10 ord4s called ary.` `// Print contents of array.` `for (n = 0; n < 10; n++) {` `  printf("%d\n", ary[n])` `}`	`# Given: array of 10 values called ary` `# Print contents of array.` `for item in a:` `  print("%d"%(item))`
`if (x < y) {` `  // something` `} else {` `  if (x == z) {` `  // something more` `  } else {` `  // something else` `  }` `}`	`if x < y:` `  pass # something` `elif x == z:` `  pass # something more` `else:` `  pass # something else`
`define ord1 done = 0` `do {` `  // something`	`done = 0` `while not done:` `  # something`

ITP	Python
} while(!done)	

E.3 Control Loops

In ITP scripting, the for loop is similar to the C language version.

```
// ITP Scripting
define ord4 i;
for (i = 0; i < 0n100; i += 2)
{
    printf("%d\n", i)
}



## Python
for i in range(0, 100, 2):
    print("%d"%i)
```

E.4 Formated Printing

In the following example, the resetpower() procedure executes a Windows batch file, sleeps 3 seconds, then tests to see if the current processor is running. If the processor is running, all processors are halted.

```
// ITP Scripting
libcall("msvcrt", "system", "reset_system", ord4, string);

define proc resetpower()
{
    // Cycles power on attached power switch to reset power on platform.
    reset_system("powercycle_script.bat")
    sleep 3
    if (isrunning)
    {
        [all]halt
    }
}
```

Here is the equivalent procedure in Python. The first two lines are required to configure Intel DAL as well as make available commands such as sleep() and halt().

Note

These two lines are automatically executed if using the Python console that comes with Intel DAL.

```
## Python
```

```
import itpii
itp = itpii.baseaccess()

def resetpower():
    import os
    ## Cycles power on attached power switch to reset power on platform.
    os.system("powercycle_script.bat")
    itp.sleep(3)
    if p0.cv.isrunning:
        itp.halt()
```

E.5 Procedure Arguments

In ITP, arguments passed to procedures are handled through the argcount and argvector variables.

This example prints any number as a five digit decimal number with leading 0's.

```


// ITP Scripting
define proc print_number()
{
    if (argcount == 0)
    {
        printf("Usage: print_number(value)\n")
        printf("Error! No value specified.\n")
    } else {
        printf("%05d\n", argvector[0])
    }
}

print_number
Usage: print_number(value)
Error! No value specified.

print_number(73)
00073
```

Here is a Python version. Note how arguments are associated with an individual name (in this case, the variable "value", which is an arbitrary label). Arguments can even have default values that are used if an actual value is not specified:

```


## Python
def print_number(value = None):
    if value == None:
        print("Usage: print_number(value)")
        print("Error! No value specified.")
    else:
        print("%05d"%value)

print_number()  ## Note: Parentheses are required to call a command in Python.
Usage: print_number(value)
Error! No value specified.
```

```
print_number(73)
00073
```

E.6 Variable Number of Procedure Arguments

The example shows how to handle an arbitrary number of parameters passed into a procedure.

This ITP script computes the average of a variable number of values passed in.

```

// ITP Scripting
define proc average()
{
    define int1 a
    define int2 sum = 0
    define int2 result = 0
    if (argcount > 0)
    {
        for (a = 0; a < argcount; a++)
        {
            sum += argvector[a]
        }
        result = sum / argcount
    }
    return (result)
}

// Now execute the procedure with some values
average(0n1, 0n4, 0n6, 0n8, 0n10)
5
```

Here is the Python equivalent (yes, there are more compact ways to do this but the goal here is to provide a more-or-less direct comparison between the two languages).

```

## Python
def average(*argvector):
    argcount = 0
    if argvector: argcount = len(argvector)
    result = 0
    if argcount > 0:
        sum = 0
        for a in range(0, argcount):
            sum += int(argvector[a])
        result = sum / argcount
    return result

## Now execute the function with some values
average(1, 4, 6, 8, 10)
5
```

E.7 Calling Procedures

In legacy ITP, if a procedure did not take any parameters, the procedure must be called by specifying just the name of the procedure without parentheses. If a procedure required parameters then parentheses were required.

```
// ITP Scripting
define proc dosomething()
{
    printf("Hello from dosomething!\n")
}

dosomething
```

In Python, all procedures must include parentheses as part of the calling step whether or not they take parameters; the parentheses are how Python knows to call the procedure.

```
## Python
def dosomething():
    print("Hello from dosomething")

dosomething()
```

E.8 Calling DLLs

In ITP, the libcall() procedure is used to associate a user-defined procedure name with the name of a procedure in a DLL. Calling the procedure calls into the DLL.

This example shows how to call the system() function in the msvcrt DLL.

```
// ITP Scripting
libcall("msvcrt", "system", , ord4, string);

// Execute a script using the system() function in the Microsoft C Runtime DLL.
define ord4 retval = system("powercycle_script.bat")
```

In Python, the ctypes module is needed to call into DLLs.

Note

The ctypes module is not supported in IronPython 2.0 or earlier.

```
## Python
import ctypes
msvcrt = ctypes.cdll.LoadLibrary("msvcrt")
retval = msvcrt.system("powercycle_script.bat")
```

E.9 Includes

In ITP, the include command provides a way to load and execute a script file at the point where the include command is placed. The Python import command is roughly equivalent to the ITP include command.

Given the ITP script mycode.itp:

```
// ITP Scripting for mycode.itp file
define proc dowork()
{
    printf("Hello from dowork() in mycode.itp!\n")
}

printf("Hello from mycode.itp\n!")
```

Loading the mycode.itp script produces the following in legacy ITP:

```
// ITP Scripting
[p0]?include "mycode.itp"
Hello from myfunc.itp!
[p0]?dowork
Hello from dowork() in myfunc.itp!
```

Given the Python script mycode.py:

```
## Python for mycode.py file
def dowork():
    print("Hello from dowork() in mycode.py!")

print("Hello from mycode.py!")
```

Here is the Python equivalent.

```
## Python
>>> import mycode
Hello from mycode.py!
>>> mycode.dowork()    ## Note: The script is loaded into the mycode namespace
Hello from dowork() in mycode.py!
```

Appendix F. Python CLI Command Line Switches

The following command line switches are available on the pythonconsole.cmd file that launches the Python CLI. These command line switches can be specified in the shortcut, after the name of the script in the Shortcut's Target field.

Switch	Description
--mac python file	Execute the specified python module after the DAL is ready to go but before the python CLI prompt is shown. Up to four different files can be specified. The file can be located anywhere, just specify a full path. For example: `pythonconsole --mac c:\pythonscripts\reallycoolstuff.py`
--showexceptions	Display the full stack trace on all exceptions. This can also be accomplished by calling `itpii.enhancements.disableExceptionOverrides()` To hide stack traces again, either exit python CLI and restart without the --showexceptions switch or call `itpii.enhancements.enableExceptionOverrides()`

In addition to the above, the python CLI will look for two specific python modules to execute after the DAL is ready and before the --mac modules are loaded. These are the "auto-start" files.

> **Note**
> these "auto-start" files do not exist by default; you must create them from scratch if you need them.

File	Location
dalstartup.py	DAL install folder (typically, c:\Intel\DAL)
daluserstartup.py	The user's home folder (on Windows 7, this would be c:\users\user_login_name)

Here is an "auto-start" file that always sets the resetbreak control variable to break on reset. The first two lines are always needed if the script is going to access the python CLI commands.

```
import itpii
itp = itpii.baseaccess()
itp.cv.resetbreak = "break"
```

F.1 Debugging Errors in Auto-Start Files

For the auto-start files, all errors are ignored. So if the file is missing or has a syntax error, the error is ignored and the file does not get loaded. If an auto-start file is successfully loaded, a message is displayed:

```
Successfully imported "c:\users\someuser\daluserstartup.py".
```

To debug errors in auto-start files, open the python CLI normally then import the auto-start file. If the file is in the user folder, you need to first set up the sys.path variable to point to the user folder. For example:

```
>>? import sys
>>? sys.path.append("~")
>>? import daluserstartup
```

Now you can see any errors that can appear in the auto-start file.

For files loaded with the --mac command line switch, an error message is displayed if the module has an error in it. However, no stack trace is provided. In that case, manually import the module. The only caveat is you will need to set the sys.path variable to point to the folder that contains the module before importing the module. The --mac command switch knows how to make use of the full path to the file. For example, if the module is c:\pythonscripts, the following can be used to get a stack trace.

```
>>? import sys
>>? sys.path.append(r"C:\pythonscripts")
>>? import reallycoolstuff
>>? itp.print_last_exception()
```