

# MIPS Debug

## Low-Level Bring-Up Guide

# MIPS

---

## by Imagination

Copyright © Imagination Technologies Limited. All Rights Reserved.

This publication contains proprietary information which is subject to change without notice and is supplied 'as is' without warranty of any kind. Imagination Technologies, the Imagination logo, PowerVR, MIPS, Meta, Enigma and Codescape are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners.

Filename : MIPS\_Debug\_Low-Level\_Bring-Up\_Guide.docx  
Version : 1.3.266 External Issue Added JTAG characteristics..  
Issue Date : 30 Mar 2016  
Author : Imagination Technologies Limited  
Document No : MD01043

## Contents

<b>1. Introduction</b>	<b>7</b>
1.1. Licensing	7
1.2. Terminology	7
<b>2. Target connection details</b>	<b>8</b>
2.1. SP55E	8
2.1.1. Establishing host-debug adapter connection	8
2.1.2. SP55E connection to target JTAG	9
2.1.3. RJ45/Ethernet connection to host	9
2.1.4. USB connection to host	9
2.2. Connecting to MIPS development boards	10
2.2.1. Malta + coreFPGA6	10
2.2.2. SEAD3	12
2.3. Getting diagnostics information from the debug adapter and Codescape	12
2.3.1. Diagnostics in Codescape	12
2.3.2. Listing debug adapter transaction logs	12
2.4. Target Connection Schematic	14
2.4.1. SP55E	14
2.5. Notes for making your own JTAG cable	15
2.5.1. Board and cable impedance matching	15
<b>3. Using and configuring an SP55E debug adapter</b>	<b>16</b>
3.1. SP55E Overview	16
3.2. Power Requirements	16
3.3. Connectors	16
3.3.1. USB	16
3.3.2. RJ45/Ethernet	16
3.3.3. JTAG	16
3.4. External LEDs	16
3.5. SP55E interface specifications	17
3.6. SysProbe JTAG characteristics	17
3.6.1. Non-standard JTAG output configuration	17
3.6.2. Non-standard JTAG input configuration	18
3.6.3. JTAG signal timing	18
3.7. DC Characteristics	19
3.8. Opening a connection to an SP55E	20
3.8.1. Opening a connection to an SP55E with Codescape Console	20
3.9. Checking and reflashing SP55E firmware	20
3.9.1. Checking current firmware version	20
3.9.2. Reflashing the SP55E from Codescape Console	20
<b>4. Board and Core Definition files</b>	<b>22</b>
4.1. Overview	22
4.2. Working with Hardware Definition and Board files	22
4.2.1. Creating Hardware Definition files	22
4.2.2. Copying existing Hardware Definition files	23
4.2.3. Creating Board files	23
4.2.4. Selecting Hardware Definition and Board files	24
4.2.5. Modifying Hardware Definition and Board files	24
4.3. Modifying existing Board and Hardware files	24
4.3.1. Editing HSPs with a text editor	24
4.3.2. Using the Hardware Definition Editor	24
4.4. HSP file format	26
4.4.1. Format overview	26
4.5. Worked example of an XML file	28
<b>5. New Target Bring-up</b>	<b>31</b>
5.1. Introduction	31

5.2.	Stage 1 - Bypass Test .....	31
5.3.	Stage 2 - TAP Identification .....	32
5.4.	Stage 3 - basic debug operation.....	34
5.4.1.	MIPS.....	34
5.4.2.	Meta/UCC.....	37
5.5.	Stage 4 – Auto-detect with Codescape Console.....	38
5.5.1.	MIPS.....	38
5.5.2.	Meta/UCC.....	43
5.6.	Stage 5 – Auto-detect with Codescape .....	46
<b>6.</b>	<b>Low-level EJTAG Debug .....</b>	<b>48</b>
6.1.	Introduction .....	48
6.1.1.	Terminology.....	48
6.1.2.	Tools.....	49
6.2.	MIPS Processor Basics .....	49
6.2.1.	Execution Mode.....	49
6.2.2.	ISA Mode.....	50
6.2.3.	Execution Location .....	50
6.2.4.	Exception Cause .....	50
6.3.	Using NMI (Non Maskable Interrupt) .....	51
6.4.	EJTAG Debug Features .....	51
6.4.1.	EJTAG TAP Basics .....	52
6.4.2.	Boot Mode: EJTAGBOOT vs NORMALBOOT .....	52
6.4.3.	Basic Codescape Console Commands.....	52
6.4.4.	Advanced Codescape Console Commands .....	55
6.4.5.	Low level "scan" commands.....	55
6.4.6.	Diagnosing Cache Problems.....	59
6.5.	Debugging a Soft Hang .....	60
6.5.1.	Using PC Sample .....	60
6.5.2.	Using DINT (Debug Interrupt) .....	61
6.6.	Debugging a Hard Hang.....	61
6.6.1.	“Halt” Fails .....	61
6.6.2.	Halt Fails: CPU not taking DINT.....	62
6.6.3.	Halt Fails: CPU never accesses dmseg.....	62
6.6.4.	Halt Fails: CPU stops accessing dmseg .....	65
6.6.5.	Debug Through Reset.....	67
6.7.	Multi-Core Coherent Processing Systems (CPS).....	67
6.7.1.	Cluster Power Controller (CPC) .....	67
6.7.2.	CPC Reset .....	67
6.7.3.	Probe-Mode.....	68
6.7.4.	CPC DINT monitoring.....	68
<b>7.</b>	<b>OCI debugging with a DBU Debug Monitor.....</b>	<b>69</b>
7.1.	Introduction .....	69
7.2.	Debug Unit (DBU).....	69
7.3.	Debug Monitor .....	69
7.3.1.	Global Throttle .....	69
7.3.2.	Debug Monitor States.....	69
7.3.3.	Key Components.....	69
7.4.	Connecting.....	71
7.5.	Using the Debug Monitor via high level commands .....	72
7.5.1.	Debugging .....	72
7.5.2.	Exceptions .....	72
7.5.3.	Incorrect states.....	73
7.5.4.	Timeouts.....	73
7.6.	Low Level Usage .....	73
7.6.1.	Debug Mode .....	73
7.6.2.	Multiple VPs .....	74
7.6.3.	Monitor Commands .....	75
<b>8.</b>	<b>Advanced Debug Adapter settings .....</b>	<b>77</b>

8.1.	Global settings .....	77
8.2.	MIPS .....	80
8.3.	Meta .....	81
A.1.	Hardware Definition XML Elements.....	83
A.1.1.	Document .....	83
A.1.2.	Board .....	83
A.1.3.	SoC.....	84
A.1.4.	CoreInfo.....	85
A.1.5.	DAConfiguration .....	86
A.1.6.	Processor .....	86
A.1.7.	MemoryType .....	87
A.1.8.	MemoryBlock.....	88
A.1.9.	SharedMemory.....	89
A.1.10.	Module .....	90
A.1.11.	Register .....	90
A.1.12.	BitField.....	92
A.1.13.	BitFieldValue .....	93
A.1.14.	Settings.....	93
A.1.15.	Setting .....	94
A.1.16.	ProcessorLink.....	94
A.1.17.	SoCLink .....	94
A.1.18.	CoreID .....	95
A.2.	The Document Type Definition .....	95

## List of Figures

Figure 1 SP55E connection to target .....	8
Figure 2 Location of UART0 on Malta.....	11
Figure 3 Connecting up a Malta + coreFPGA6 development board.....	11
Figure 4 Connecting up a SEAD3 development board.....	12
Figure 5 SP55E target connection schematic.....	14
Figure 6 JTAG signal timing.....	18
Figure 7 CPS Resources (simplified).....	49

## Document History

Issue	Date	Changes/Comments
1.0.24	09 Apr 2014	First version.
1.0.61	11 Apr 2014	External Issue (Preliminary)
1.1.93	17 Jul 2015	Updated with new SP55E features and corrections
1.2.129	21 Aug 2015	New probe configuration options added.
1.3.253	08 Mar 2016	Updated for SysProbe SP55E. Removed DA-net. Added section OCI debug with debug monitor. Changed category to Public. Removed part number from document title.
1.3.257	09 Mar 2016	External Issue Added SP55E DC Characteristics.
1.3.266	30 Mar 2016	External Issue Added JTAG characteristics..

# 1. Introduction

This document provides connection information and detailed low-level, debugging assistance and techniques to help you diagnose common problems encountered when bringing up MIPS and Meta/UCC processor cores using Imagination's debugging hardware and software solutions.

It has the following sections:

## Target connection details

Connection details for popular development systems.

## Using and configuring an SP55E debug adapter

How to set up the SP55E debug adapter.

## Board and Core Definition files

How to write and customise Hardware Support Packages (HSPs) to provide target-specific information such as the layout of core memory and register definitions.

## New Target Bring-up

How to bring-up new designs either in silicon, FPGA, or emulation using Codescape Console and scripts for MIPS, Meta and UCC targets.

## Low-level EJTAG Debug

Information on EJTAG debug basics and a "How To" guide for Codescape Console commands and debugging system stalls.

## OCI debugging with a DBU Debug Monitor

How to debug an OCI compliant, 64-bit, multicore system containing a Debug Unit (DBU) using a debug monitor and Codescape Console.

## Advanced Debug Adapter settings

Advanced settings to control the debug adapter's behaviour.

## Hardware Definition Reference Documentation

Appendix describing the syntax of elements in XML hardware definition files.

## 1.1. Licensing

Licenses for open source components can be found on the probe's in-built webserver at: <http://img-sp00xxx/license> where xxx is the serial number of the probe as printed on side and base, eg: <http://img-sp00155/license>.

## 1.2. Terminology

**SysProbe SP55** - SysProbe and SP55 is the name of the master module that accommodates a sub-assembly PCB, such as SP55E, that provides a specific set of debug features. The name of the sub-assembly can be read on the end panel of the SysProbe.

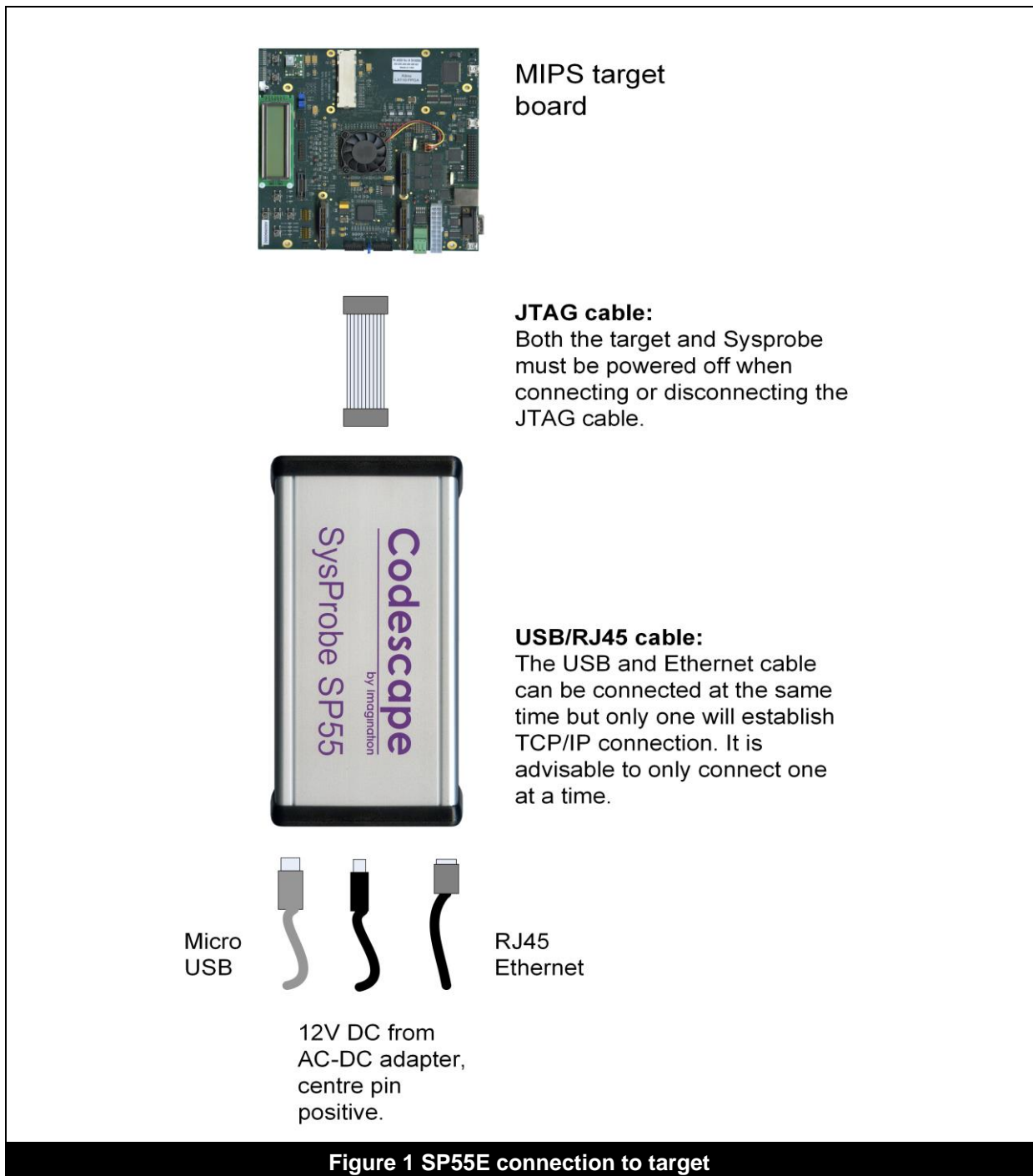
**SP55E** - SP55E is a network-capable probe from Imagination Technologies. It can be connected directly to a host PC via USB or Ethernet and connection to a MIPS targets via a JTAG.

### 'probe' vs 'debug adapter'

Throughout this document the terms 'probe' and 'debug adapter' are used interchangeably.

## 2. Target connection details

### 2.1. SP55E



The SP55E can connect to the debugger host PC either by USB (connected directly to the host PC) or via Ethernet.

#### 2.1.1. Establishing host-debug adapter connection

Host to SP55E communication is via TCP/IP either using USB cable to the host PC or RJ45 socket to Ethernet. Both USB and RJ45 can be connected at the same time, but only one will be used by the



SP55E. The first one to obtain an IP address will be used by the SP55E; it is therefore advisable to only connect one.

### 2.1.2. SP55E connection to target JTAG

The connection between the SP55E and the target is via ribbon cable. No other adapters are required. Both the target and the SP55E should be powered off when connecting the cable.

### 2.1.3. RJ45/Ethernet connection to host

When an Ethernet cable is connected to the SP55E, the probe tries to connect to DHCP and request an IP address. Default DNS name of the SP55E is `img-sp****` where `****` is the last five digits of the SP55E's serial number.

### 2.1.4. USB connection to host

#### USB IP address

The USB connection will present itself as a network adapter when connected to a Host PC running Windows, Apple OS or Linux. On connection, the Host PC will request an address from the SP55E. The SP55E will serve an IP address from the range 169.254.100.0 to 169.254.254.254. This will be the IP address of the USB port as seen from the host.

The address will be static, derived from the serial number of the SP55E.

The number is derived by the divisor and modulus of the last 4 digits of the serial number when divided by 100. For example an SP55E with a serial number of 02DALS32000378 would obtain an IP address of 169.254.3.78.

#### Troubleshooting USB-host connections

##### Ping the address

If Codescape cannot detect the SP55E, try pinging the IP derived from the probe's serial number.

##### Connection on Linux

Using older versions of some Linux distributions connectivity to the SP55E through USB isn't automatically setup and visible as an active network interface. In this case, the ping test will fail.

Check that the probe has been discovered using `lsusb` or checking `dmesg`. For example:

```
Host$ lsusb
Bus 002 Device 011: ID 0525:a4a2 Netchip Technology, Inc. Linux-USB Ethernet/RNDIS Gadget
```

Look for the interface in `ifconfig -a`. For example:

```
Host$ ifconfig -a
usb0      Link encap:Ethernet  HWaddr 76:1A:33:9D:CF:C9
          inet6 addr: fe80::741a:33ff:fe9d:cf9/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1494  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3269 (3.1 KiB)  TX bytes:332 (332.0 b)
```

If no IP address is assigned, the DHCP daemon may not be setup for this interface automatically. This can be configured using the network manager/network config that is

provided by your Linux distribution. Either you can set the IP address statically or start `avahi-autoipd` daemon on the interface. For example:

```
Host$ sudo avahi-autoipd -D usb0
```

With the `avahi autoipd` daemon running on the network interface, `ifconfig` should show the interface with an IP address. For example:

```
Host$ ifconfig
usb0      Link encap:Ethernet  HWaddr 76:1A:33:9D:CF:C9
          inet6 addr: fe80::741a:33ff:fe9d:cfc9/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1494  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3269 (3.1 KiB)  TX bytes:696 (696.0 b)
usb0:avahi Link encap:Ethernet  HWaddr 76:1A:33:9D:CF:C9
          inet addr:169.254.10.129  Bcast:169.254.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1494  Metric:1
```

*Note: If setting the IP address manually, it is important to ensure the scope is set to link-local.*

### Connection on Windows

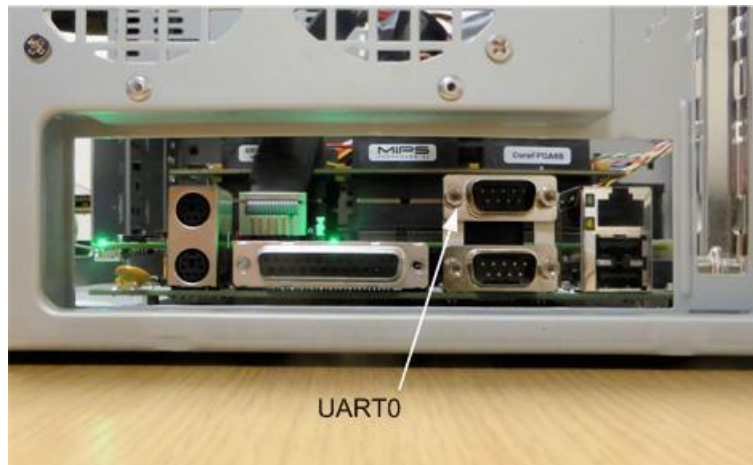
On Windows the probe will usually be identified as a RNDIS device. If connection has not worked, try updating your RNDIS driver. You can also try adding a new 'Remote RNDIS Compatible Device' from your Devices window.

## 2.2. Connecting to MIPS development boards

### 2.2.1. Malta + coreFPGA6

#### Yamon

The Malta board is supplied pre-flashed with YAMON. Please refer the YAMON documentation supplied (also available at [http://wiki.prplfoundation.org/wiki/MIPS\\_documentation](http://wiki.prplfoundation.org/wiki/MIPS_documentation) ). YAMON stdout and terminal interface by default uses UART0 as shown on the picture below. YAMON outputs at a UART speed of 38400 baud 8N1. When Linux is running this may be different depending on kernel config, if you connect a terminal to UART0 be aware that the terminal baud speed must match whatever baud rate is set by Linux.



**Figure 2 Location of UART0 on Malta**

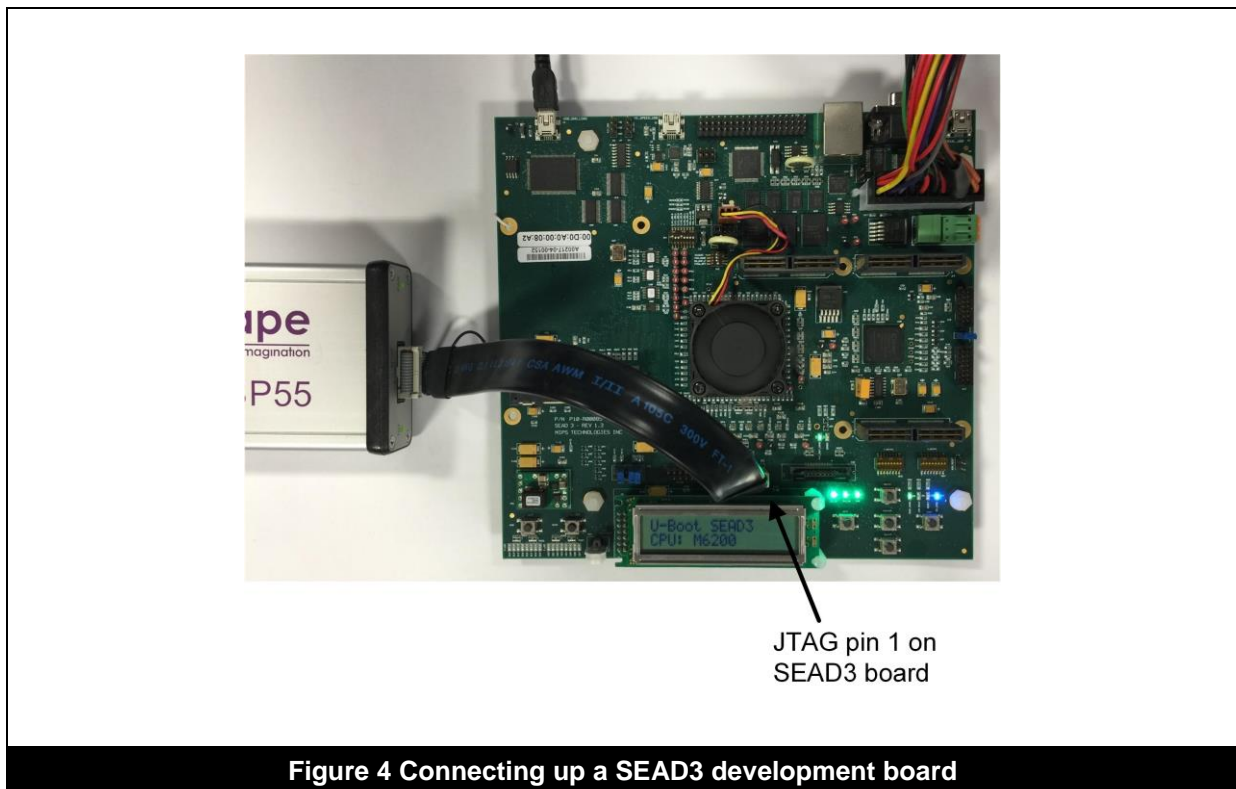
Make sure everything is powered off before connecting up.

The ribbon cable has a red line indicating the position of pin 1 and should be oriented as shown.



**Figure 3 Connecting up a Malta + coreFPGA6 development board**

## 2.2.2. SEAD3



## 2.3. Getting diagnostics information from the debug adapter and Codescape

### 2.3.1. Diagnostics in Codescape

In Codescape there is a diagnostics report available from the Target Diagnostics window (Help menu > Diagnostics). Select 'Codescape Debugger > Comms Log' from the expandable tree on the left side. In this report information is given about the debug adapter you are connected to and communications between the host PC, the debug adapter and the target.

If you are having difficulties connecting to the debug adapter or target you can use this information to diagnose the problem or you can copy and send the report using the 'Feature Request/Defect Reporting' option from the Help menu.

### 2.3.2. Listing debug adapter transaction logs

A log of debug adapter transactions can be generated from Codescape Console using the `logfile` command. This can display error and transaction logs for the connected debug adapter.

Using the command without parameters prints a list of the available logs. For example:

```
>>> logfile()
DA Info Log
DA Error Log
DA Verbose Log
DA JTAG Log
```

*Note: The logs available will depend upon the target and the type of probe (debug adapter) used.*

Using a log name as a parameter prints the contents of that log.

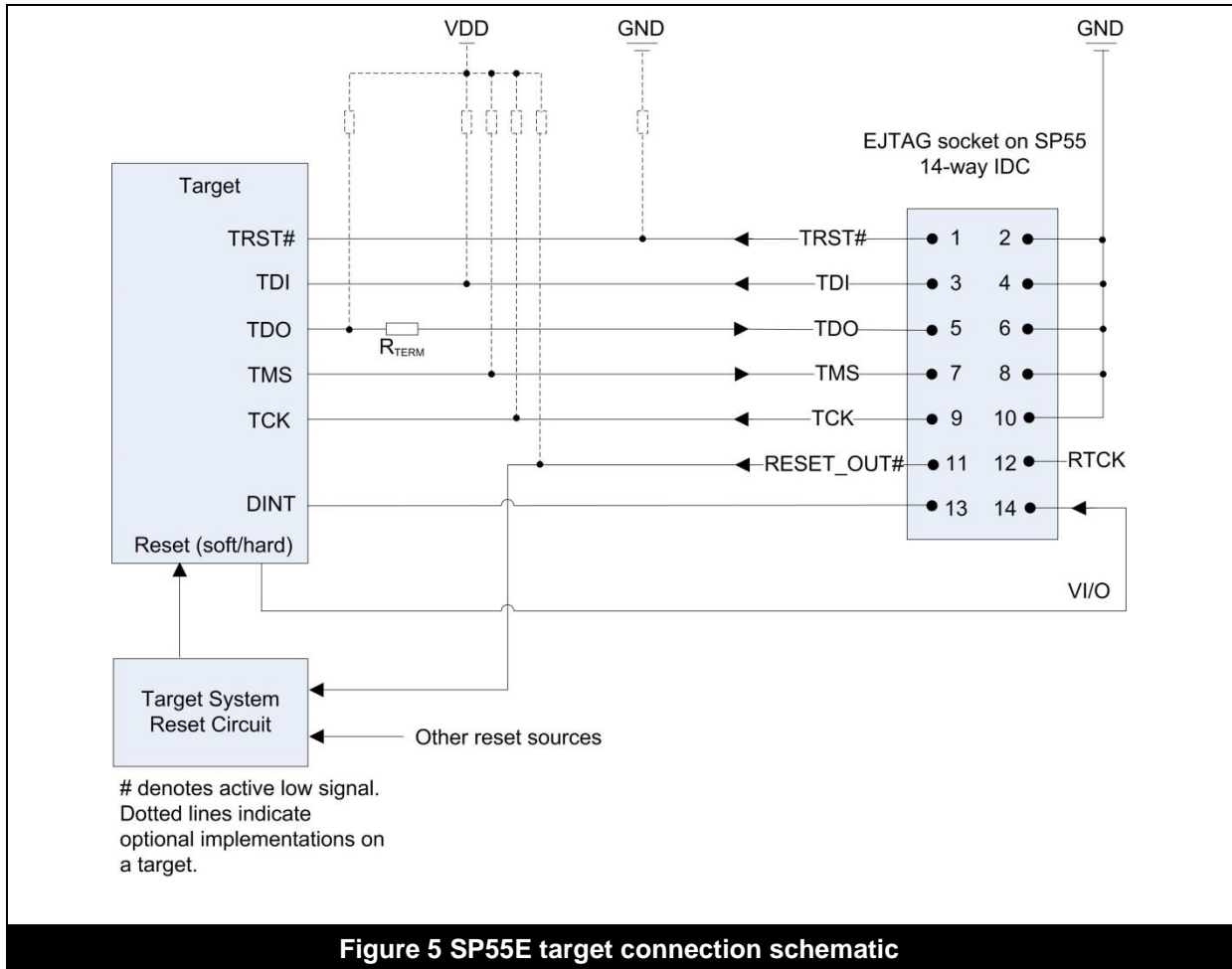
```

>>> logfile("DA Info Log")
    0.000:SoC X:Generic : <info> : main           : Initialising...
    0.000:SoC X:Generic : <info> : setup         :           Xilinx Configuration
OK, VHDL version = 1.C
    0.000:SoC X:Generic : <info> : board init  : Board Revision 3
    0.103:SoC X:Generic : <info> : main         : Dash ID - 01EGNT33000401
    0.104:SoC X:Generic : <info> : target_handler : Dash Initialised, waiting for
first host command to set operating mode.....
    0.105:SoC X:Generic : <info> : add_event    : add_event, add: id: 0, period:
1,
    30.823:SoC X:Generic : <info> : jtag_scan    : first command is JTAG Scan,
entering Passive Mode !!
    
```

## 2.4. Target Connection Schematic

### 2.4.1. SP55E

The diagram and table in this section show the pinouts for a target’s JTAG connector used with an SP55E debug adapter.



**Figure 5 SP55E target connection schematic**

The table below displays the pinouts that need to be implemented on the target’s EJTAG connector, to support the EJTAG protocol when used with an EJTAG adapter board.

Pin	Signal	Direction	Pin	Signal	Direction
1	TRST# - Test Reset Input	Input	2	GND - Ground	GND
3	TDI – Test Data Input	Input	4	GND – Ground	GND
5	TDO – Test Data Output	Output	6	GND – Ground	GND
7	TMS – Test Mode Select Input	Input	8	GND – Ground	GND
9	TCK – Test Clock Input	Input	10	GND – Ground	GND
11	RESET# - System Reset	Input	12	RTCK – not implemented	
13	DINT default low.		14	VI/O – Target reference voltage.	Output

**Pin 13 DINT**

This pin can be driven low or high from Codescape Console using the command `config("assert dint",1)` where 1 is to drive DINT high. A '0' will drive DINT low. It defaults to low.

**Pin 14 VIO**

This pin is an input circuit to the SP55E from the target. The normal requirement is 1.2mA at 1V2, rising to 3.3mA at 3V3.

## 2.5. Notes for making your own JTAG cable

### EMC Compliance

To fully comply with EC directive 2004/108/EC concerning emissions and immunity you must connect the flying ground lead (if fitted) on the supplied cable to a suitable secure earth point on your target system.

The JTAG debug output connector on the debug adapter is a 14-way IDC.

If you are making your own cable to connect to a custom installation, note that the JTAG interface has a theoretical maximum length of 300mm based on a 20MHz clock speed. As a guideline we recommend that the cable does not exceed 250mm in length and the on-board track length to the processor does not exceed 50mm where possible. You may be able to achieve longer cable lengths at slower clock speeds but performance is not guaranteed. All cables must be screened and earthed.

### 2.5.1. Board and cable impedance matching

Even though the JTAG signals are relatively slow some thought is needed when routing these signals for the target system PCB design.

The 14-way cable has a characteristic impedance of 65R, ideally the PCB traces for the JTAG signals should match this. If the impedances cannot be matched then the traces should be kept short, sub 5cm. All the traces need to be kept to similar lengths (within 1cm difference) to avoid skew.

Special care is needed on the TDO line (data out of the target SoC), the pad drive strength needs to be strong enough to deal with the relatively high capacitance of the cable + traces, but not too high so that it will generate very fast slew-rate edges. We recommend a value between 4mA and 12mA. Most output drivers will have a relatively low impedance, this needs matching to the PCB traces and cable with a source termination resistor ( $R_{\text{TERM}}$  on schematic), typical values will be in the range 15R - 33R.

## 3. Using and configuring an SP55E debug adapter

### 3.1. SP55E Overview

The SP55E is a high-speed debug adapter that uses JTAG and TCP/IP protocol to connect a host debugging PC to a target. Connection between debugging PC and an SP55E can be via Ethernet or a direct USB cable. Both connections use TCP/IP protocol. Although USB and Ethernet can be connected at the same time, only one of them will be used. When an SP55E is connected to a PC via USB, it presents as a client-mode network adapter.

### 3.2. Power Requirements

The SP55E is supplied with a 12V, 15W DC power supply (centre pin positive). The board can be run on a supply voltage from 5v to 12V. Wattage requirements will depend on the board activity.

### 3.3. Connectors

#### 3.3.1. USB

Connector type: microUSB  
Protocol: Client-mode, utilizing a network adapter for TCP/IP.

#### 3.3.2. RJ45/Ethernet

Connector type: RJ45  
Protocol: Ethernet TCP/IP

#### 3.3.3. JTAG

Connector type: 14-way IDC  
Protocol: JTAG compatible with targets complying with MIPS EJTAG architecture and the MIPS OCI architecture. See 'Figure 5 SP55E target connection schematic' on page 14 for more details. Note that an adapter is required to use the SP55E with Meta targets.

### 3.4. External LEDs

#### PWR LED

Shows a steady green light when the SP55E is powered up.

#### TGT LED

Shows a steady green light when the target is powered up and connected to the SP55E by the ribbon cable.

#### RJ45 Ethernet socket

The RJ45 socket has two built-in LEDs. If your SP55E is powered up and connected to a target, the LEDs give the following indications:

LED	Meaning
Steady green LNK	Ethernet link OK.
Flashing green ACT	Ethernet activity.



### 3.5. SP55E interface specifications

Feature	SP55E
<b>JTAG Interface</b>	
Supported JTAG IR width	1-2048
Supported JTAG DR width	1-2048
Target must support BYPASS scan	Yes
Additional register delays in scan-chain between TAP and core	0-2047 before TAP 0-2047 after TAP
TCK maximum frequency	31.25MHz currently
TCK edge on which JTAG outputs transition (TDO, TMS)	Configurable to be either
TCK edge on which JTAG input is registered (TDI)	Configurable to be either
Multiple TAPs on scan-chain	Yes
Multiple cores on single TAP	Yes
TRST required	No
Target TCK system	Must be simple clock buffer only; no PLL/DLL is permitted
Driver strength for target SoC TDO (for 31.25MHz TCK down 30cm shielded ribbon cable)	12mA. This output should be source-terminated for a 50R transmission-line
<b>Host interface</b>	
Protocol	1Gb Ethernet

### 3.6. SysProbe JTAG characteristics

The SysProbe JTAG outputs are driven by DDR output registers, (before going through voltage translation), yielding very low skew between TCK edges and TMS/TDO edges.

The TCK period must be long enough, after considering all skews, for all setup and hold time requirements to be met. For systems with high signal skew, the TCK period should be made longer.

The standard JTAG configuration is:

- SysProbe TMS/TDO to change on the falling-edge of TCK, ready for the target to register these signals on the following TCK rising-edge.
- SysProbe TDI to be registered on the rising-edge of TCK, after the target has produced edges on the falling-edge of TCK.

This configuration provides a half TCK-period for setup-times, and a half TCK-period for hold-times, and potentially providing margin for signal-skew between the probe and the target.

In many cases, more setup-time is required than hold-time, and so if a greater proportion of the TCK period is used for setup-time, a shorter TCK period may be used.

#### 3.6.1. Non-standard JTAG output configuration

The TCK edge on which the TMS/TDO outputs transition is programmable.

SysProbe can be configured to output TMS/TDO 8ns after the rising-edge of TCK, still providing some hold-time and margin for skew, but potentially meeting the target setup-time requirement with a shorter TCK period.

This feature is disabled by default.

### 3.6.2. Non-standard JTAG input configuration

The TCK edge on which the TDI input is registered is programmable.

As the target produces TDI edges on TCK in response to SysProbe generating edges, a delay is guaranteed between the SysProbe producing TCK and receiving TDI events as follows:

1. SysProbe generates TCK falling-edge
2. TCK edge propagates through voltage-translation buffer and cable
3. Target produces edge on TDI
4. TDI edge propagates through cable and voltage-translation buffer

This delay guarantees significant TDI hold-time, and so it's acceptable for SysProbe also to register TDI on the falling-edge. This configuration can potentially meet the SysProbe setup-time requirement with a shorter TCK period.

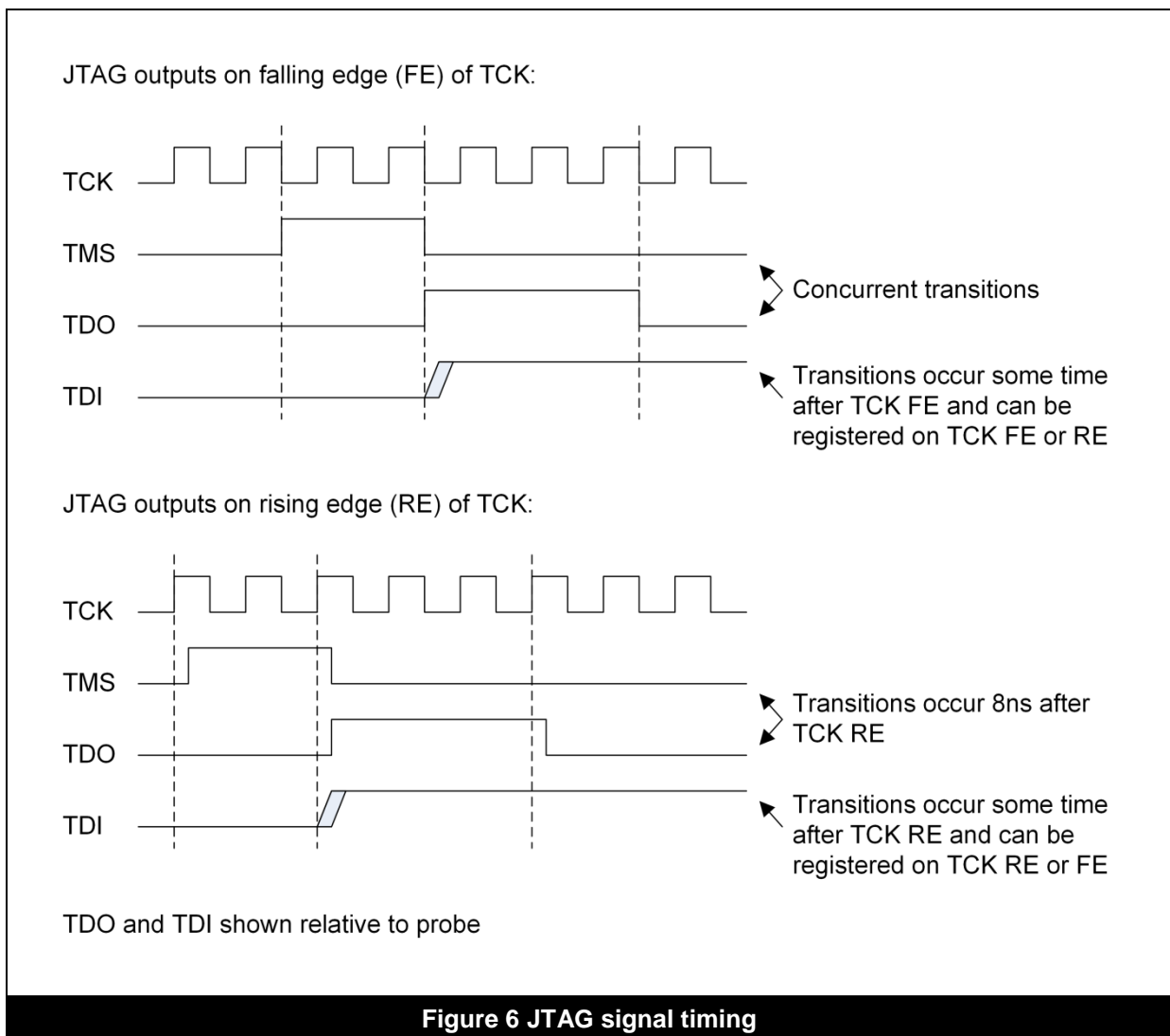
This feature is enabled by default.

### 3.6.3. JTAG signal timing

Timing diagrams are given for two cases:

1. JTAG outputs change on falling-edge of TCK (the JTAG standard)
2. JTAG outputs change on rising-edge of TCK

In each case, TCK is running at 31.25MHz.



### 3.7. DC Characteristics

Description	Condition	Symbol	Min	Max	Units
Input low voltage	$V_{io} = 0.8V$	VIL	-	0.24	V
	$V_{io}=1.1V$ to 1.95V		-	$0.35 \times V_{io}$	V
	$V_{io}=2.3V$ to 2.7V		-	0.7	V
	$V_{io}=3.0V$ to 3.3V		-	0.8	V
Input high voltage	$V_{io} = 0.8V$	VIH	0.56	0.8	V
	$V_{io}=1.1V$ to 1.95V		$0.65 \times V_{io}$	$V_{io}$	V
	$V_{io}=2.3V$ to 2.7V		1.6	$V_{io}$	V
	$V_{io}=3.0V$ to 3.3V		2	$V_{io}$	V
Target i/o voltage		$V_{io}$	0.8	3.3	V
Input current		I <sub>I</sub>	-	+/-5	uA
Input current, $V_{io}$	$V_{io}=0.8V-3.3V$	I <sub>Vio</sub>	-	3.4	uA
Output low voltage	$I_o=100uA$	VOL	-	0.1	V
	$V_{io}=0.8V-3.3V$				
	$I_o=3mA$		-	0.25	V
	$V_{io}=1.1V$				
	$I_o=6mA$		-	0.35	V
	$V_{io}=1.4V$				
	$I_o=8mA$		-	0.45	V
	$V_{io}=1.65V$				
	$I_o=9mA$		-	0.55	V
	$V_{io}=2.3V$				
	$I_o=12mA$		-	0.7	V
	$V_{io}=3.0V$				
Output high voltage	$I_o= -100uA$	VOH	$V_{io}-0.1$	-	V
	$V_{io}=0.8V-3.3V$				
	$I_o= -3mA$		0.85	-	V
	$V_{io}=1.1V$				
	$I_o= -6mA$		1.05	-	V
	$V_{io}=1.4V$				
	$I_o= -8mA$		1.2	-	V
	$V_{io}=1.65V$				
	$I_o= -9mA$		1.75	-	V
	$V_{io}=2.3V$				
	$I_o= -12mA$		2.3	-	V
	$V_{io}=3.0V$				

## 3.8. Opening a connection to an SP55E

### 3.8.1. Opening a connection to an SP55E with Codescape Console

Codescape Console is an interactive Python shell with built-in extensions for debugging (via a debug adapter) and control of a debug adapter. It can be used for reflashing, testing and for target bring-up scripts.

**Connect your SP55E to the target and Ethernet.**

1. See 'SP55E' on page 8.

**Start Codescape Console and connect to the target:**

1. Change directory to the Scripts directory below your Python location.
2. Start Codescape Console and connect to the probe:

```
C:\Python27\Scripts>CodescapeConsole sp####
Welcome to Codescape Console 8.3.0.30. Enter help()<enter> for help
<tab> completion has been enabled.
Identifier SysProbe 00078
Firmware 0.5.2.0
Location
Mode uncommitted
TCK Rate 31250kHz
```

where #### are the last 4 digits from the serial number on the SP55E

## 3.9. Checking and reflashing SP55E firmware

The firmware loaded on an SP55E can be reflashed via Codescape Console or Codescape Debugger.

From Codescape Debugger you can right-click on the target pain and select 'Reflash Firmware'.

Imagination Technologies technical support will notify customers when updated firmware is available.

The updates are sent as a single .fsh file.

*Note: More information on Codescape Console can be found in the Codescape Online Help.*

**Before reflashing:**

- You must be connected to the debug adapter via Codescape Console. See 'Opening a connection to an SP55E with Codescape Console' on page 20.

### 3.9.1. Checking current firmware version

The current firmware version can be shown with the `probe()` command:

```
>>> probe()
Identifier SysProbe 00078
Firmware 0.5.2.0
Location
Mode uncommitted
TCK Rate 31250kHz
```

### 3.9.2. Reflashing the SP55E from Codescape Console

**Checking available firmware**

Codescape Console has a command, `firmwarelist()`, that checks for compatible firmware available online. For example:

```
>>> firmwarelist()
```

```

0: 1.2.3 - SP55e - http://codescape-mips-
sdk.imgtec.com/components/probes/firmware/sp01020300.fsh
1: 1.2.1 - SP55e - http://codescape-mips-
sdk.imgtec.com/components/probes/firmware/sp01020100.fsh
2: 1.1.0 - SP55e - http://codescape-mips-
sdk.imgtec.com/components/probes/firmware/sp01010000.fsh
3: 1.0.0 - SP55e - http://codescape-mips-
sdk.imgtec.com/components/probes/firmware/sp01000000.fsh

```

### To reflash the SP55E

To reflash with the latest available flash image use the `firmwareupgrade()` command with no parameter:

```

>>> firmwareupgrade()
100% - Waiting for probe to restart
Identifier SysProbe 00155
Firmware 1.2.3.0
Location 192.168.154.55
Mode uncommitted
TCK Rate 31250kHz

```

Or, for a specific flash image:

```

>>> firmwareupgrade("http://codescape-mips-
sdk.imgtec.com/components/probes/firmware/sp01020100.fsh")

```

*Note: Reflashing is complete when the prompt is displayed again. Do not disconnect or power down the SP55E until the prompt appears.*

## 4. Board and Core Definition files

### 4.1. Overview

This section describes how to use Board and Hardware Definition files with Codescape Debugger. These files describe the physical architecture of a board and SoC.

Hardware Definition files provide information about an SoC. They contain lists of registers with the addresses, access rights to registers, and definitions of the ranges of memory areas. A number of Hardware Definition files are provided with the Codescape SDK and are located in the main Codescape SDK install area, in the `HardwareDefinition` directory.

*Note: Hardware Definition files can be created for a target from Codescape Console. See 'Using Codescape Console to create Hardware Definition files'.*

Board files specify memory area addresses, reset and connection scripts, and Hardware Definition files. The memory information in a Board file overrides memory information in Hardware Definition files. Board files can be created from within Codescape Debugger.

### 4.2. Working with Hardware Definition and Board files

#### 4.2.1. Creating Hardware Definition files

Hardware definition files can be created in the following ways:

- Copy an existing file and edit the copy
- Use Codescape Console's `makecorehd` command.

Board files are created from within Codescape Debugger.

#### Using Codescape Console to create Hardware Definition files

Codescape Console can be used to create a Hardware Definition file for a hardware target. This file will be created in a user area so it is editable.

##### Connect to the target

1. Connect your target board and probe to your PC.
2. Change directory to your Python Scripts directory.
3. Start Codescape Console and connect to the target using a command like:
- 4.

```
:\Python27\Scripts>CodescapeConsole #####
Welcome to Codescape Console 8.3.0.30. Enter help()<enter> for help
<tab> completion has been enabled.
Identifier SysProbe 00078
Firmware 0.5.2.0
Location
Mode uncommitted
TCK Rate 31250kHz
```

Where ##### is SP\*\*\*\*, danet\*\*\*\* depending on whether you are using an SP55E or DA-net probe, and \*\*\*\* is the last 4 digits of the probe's serial number.

##### Create the Hardware Definition files

1. Use the `makecorehd` command. For example:

```
makecorehd(output_name="testcore")
```

This command will create a file in the `{userhome}/imgtec/hwdefs` directory, `testcore.xml`. This file describes the registers in the core and memory information.

*Note:*

The file extension, `.xml`, is always added.

You can specify an output directory using `output_dir="<directory>"`.

#### 4.2.2. Copying existing Hardware Definition files

Each HSP consists of a pair of files, one `core_id` file and one XML file. However if you are creating a custom Hardware Definition file it may be advisable to just duplicate a supplied XML file, giving it a unique name. This ensures that the edited XML file is only used when explicitly selected.

The `core_id` file has a simple link to the corresponding XML file. This can be edited with a text editor to force loading of a different XML file.

The options for working with a customised HSP are:

- Edit the original XML file using the Hardware Definition Editor. It is advisable to take a backup of the original before starting editing.
- Create a new file, with a new filename, from the original XML file. You can then manually select this XML file from Codescape.
- Create a new file, with a new filename, from the original XML file, then edit the `core_id` file to select the new file when the core is detected.

#### Example: duplicating an XML file and reusing `core_id`

For example, the following two files are provided for 24k cores:

```
24Kc.core_id
```

```
24Kc.xml
```

The `core_id` file identifies which XML file should be used by this core.

```
<CoreID N="24Kc">
  <CoreIDValue>0x00019300</CoreIDValue>
  <src>24Kc.xml</src>
```

So we make a copy of `24kc.xml`, calling it `'24kc-new.xml'` and alter the `core_id` file:

```
<CoreID N="24Kc">
  <CoreIDValue>0x00019300</CoreIDValue>
  <src>24Kc-new.xml</src>
```

Now, when Codescape Debugger detects a core with an ID value of `0x00019300` it will load the `24kc-new.xml` file. The `24kc-new.xml` can be edited in any text editor, or using the Hardware Definition Editor from Codescape Debugger.

#### 4.2.3. Creating Board files

1. Start CS debug and connect to target
2. Right-click on target in Target Pane and select Target Debug Options.
3. Select 'Specify a specific Hardware Definition file', browse to and select the file you created earlier.
4. Click Edit. This opens the Edit Board File dialog where you can specify scripts and edit memory area information. The memory area information will be populated with information from the Hardware Definition file.

*Note: The SUM (Software User Manual) for your core or the SoC manufacturer's specifications will give the memory ranges and addresses.*

5. Click OK to save the changes and select 'Yes' when prompted to create a new board file.

#### 4.2.4. Selecting Hardware Definition and Board files

There are two mechanisms for selecting hardware definition files.

- Automatic detection
- Manual selection

The method used is controlled on a per-target basis from the 'Target Debug Options' dialog (right-click on the Target pane and select Target Debug Options).

*Note: Codescape Debugger remembers the Target Debug Options for each target. If you specify a Hardware Definition or Board file, that file will be used the next time you connect to that target.*

##### Automatic detection

When the debugger connects to a target through a debug adapter, the core ID is read by the debug adapter and reported to the debugger. Codescape uses this to search for a matching `.core_id` file. This file then indicates which XML file to load. The XML file contains the actual configuration data.

##### Manual selection

Manual selection is done from the Target Debug Options dialog and can be by explicitly selecting either a Hardware Definition or a Board file.

##### To manually select an HSP

1. Select 'Target Debug Options' from the Target menu. This opens the Target Debug Options Dialog. Check that the correct target is listed in the topmost drop-down list.
2. Select the 'Specify a specific Hardware Definition File' radio button, then click the browse button next to the field below the checkbox.
3. Browse to the Hardware Definition directory and select the XML file you want to use.
4. Click OK to save the selection.

#### 4.2.5. Modifying Hardware Definition and Board files

Two methods, edit XML, work from within Codescape

Hardware Definition Editor will open the board file (if using them). Need to manually open Hardware Definition file. If you have a board file specified then when you open the HD editor the board file is opened – need to explicitly open HD file.

### 4.3. Modifying existing Board and Hardware files

Editing and creating new Board and Hardware Definition files can be done in three ways:

- Directly working with the XML files with a text editor.
- Using the Hardware Definition Editor
- Via the Edit Board File dialog. This is accessed from the Target Debug Options dialog (right-click on Target pane > Target Debug Options > Create button). A Board File is created from this operation, containing a CoreID and general memory layout.

#### 4.3.1. Editing HSPs with a text editor

The HSPs installed with Codescape Debugger are installed in a non-editable location. If you want to edit the files, it is sensible to create a copy of the files in an editable location. See 'Copying existing Hardware Definition files' on page 23.

Check your edits result in valid XML. See 'HSP file format' on page 26 for information on the XML structure for HSPs.

#### 4.3.2. Using the Hardware Definition Editor

The Hardware Definition Editor can be opened from the Imagination Technologies Program Group or from Codescape Debugger (Tools menu > Open Hardware Definition Editor). If you open it from



Codescape, the editor will automatically load the HSP for the currently-selected target in the Target Pane.

*Note: If you have configured Codescape Debugger to use a Board file for your target, the Board file will be opened automatically by the Hardware Definition Editor. To edit the HSP file, you need to manually open the HSP file (File menu > Open).*

### Editing definitions

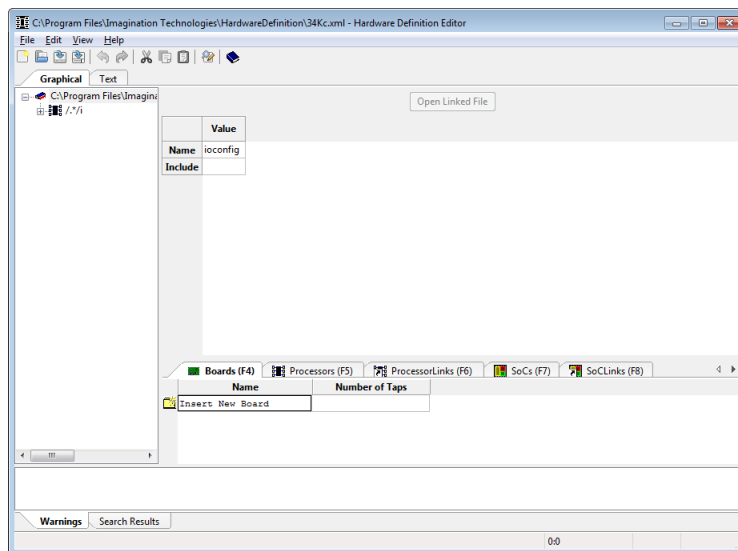
1. Start the Hardware Definition Editor (either from the Debugger Tools menu or via your Start menu).


*Note: If you start the editor from the debugger, it will automatically load the HSP that is in use. It would be good practise to make a backup of this file before editing it, and be aware that editing the file will affect your current debug session.*

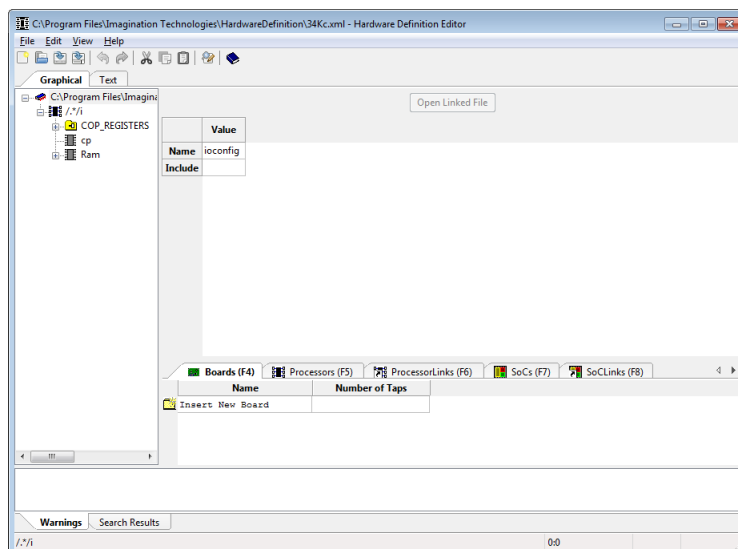
*If you edit an HSP that is in use, when you close the editor, the debugger will ask if you want to load the edited definition.*

2. Load the HSP you want to edit (File menu > Open).

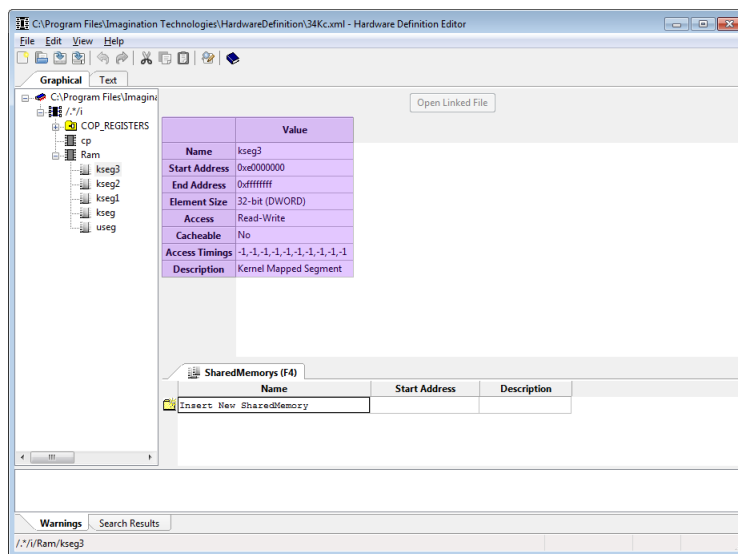
You will see something like this:



3. Click on the + next to the  icon to expand the tree.



- Note the tree entry titled 'Ram'. This contains the specification of the main memory segments.
- Expand the 'Ram' entry, and we see the kseg regions listed. Click on an entry and we can see the definition details for that segment.



All of the values in the highlighted area can be edited by clicking in the table.

*Note: Care must be taken when editing the addresses. Make sure you do not enter conflicting addresses (e.g. overlapping with another memory area).*

*Register details can be found under the 'COP\_REGISTERS' section of the tree. Editing should be done with extreme care.*

- Edit the values as required and save the file.

## 4.4. HSP file format

The purpose of the XML config file is to describe a processor's memory mapped registers. How they are laid out, grouped, and how the individual bits should be interpreted and displayed in Codescape. The XML config file also describes whether a register and the bits within the register are read/write, and how they should be written.

### 4.4.1. Format overview

This section gives a general overview of the structure of an HSP definition file.

#### Naming

All objects in the config file have a name. That name is used by Codescape to fill in a Symbol table so that all of the entries may be:

- Viewed in a Peripheral Region.
- Viewed in a Watch Region.
- Accessed from a Script using 'EvaluateExpression'.

The symbols are exactly the same as any symbols extracted from the debug information of a program file. As such they must follow the C naming convention. In particular this means:

- Only alphanumeric characters, and `_` can be used.
- The symbols are case sensitive.
- They cannot contain spaces.
- They must not be empty.
- They cannot begin with a number.

- They must be unique within their own scope  
 'Unique within their own scope' means that the names of all direct children of a parent must be different. For example a Module may not contain two Registers with the same name, and a Group may not contain a Module and a Register of the same name. However a Register named MyReg may contain a Format object (see below) named MyReg.

### Descriptions

Because of the restriction on names, all objects can also have a Description. This description has none of the restrictions associated with names.

The descriptions are used as tool-tips in Codescape, or can be permanently displayed using the 'Show Header->Description' menu option in the Peripheral region.

### Registers

A memory mapped register is represented using a Register object. It contains the following information:

- The address of the register
- The size of the register. (8, 16, 32 bits, 64 bit support is coming soon)
- It may contain one or more of three masks used to determine how the register should be read and written :
  - Read AND Mask - used to set any bits to zero after the register has been read, before it is displayed.
  - Write AND Mask - used to set any bits to zero before they are written. (For bits that must be written as zero)
  - Write OR Mask - used to set any bits to one before they are written. (For bits that must be written as one)
- It may contain the read/write property of the register, this can be any one of the following :
  - ReadWrite - the register can be read and written.
  - ReadOnly - the register can only be read, Codescape will not allow writes to this register.
  - WriteOnly - the register can only be written, Codescape will not allow reads from this register.
  - ReadOnce - the register cannot be written. Codescape will only allow reads when explicitly requested by the user.
  - WriteOnce - the register cannot be read. Codescape will only allow writes when explicitly requested by the user.
- It may contain either :
  - A Radix field, that determines in what radix the register should be displayed. (Binary, Octal, Decimal, or Hexadecimal)
 Or
  - A set of formatting objects that describes the bits and bitfields in the register. See below.

ReadOnce and WriteOnce can be used to describe registers that have side-effects, such as pipe read port, when the register is read it will take the next value in the pipe.

### Formatting Registers

Registers can either be displayed:

- In the whole form, in which case a radix property describes the radix in which the value of the register should be displayed. (Bin, Oct, Dec, Hex)
- Or they can be broken into bits and bitfields using any number of Format objects and Bit objects.

A **Format object** represents a bitfield of one or more bits, it contains the following information:

- A mask used to indicate the bits of interest to the bitfield.
- A shift value, used to shift the bits of interest before displaying. (Shift lefts are supported, but usually a shift right is used)
- Either:
  - A radix property, in the same sense as the Register radix property.
  - Or a set of Value objects that define a set of enumerated values.

A Bit object represents a bitfield of only one bit. It contains the following information:

- The zero indexed bit number to test.
- The value to display if the bit is True (1)
- The value to display if the bit is False (0)

### Grouping of Registers

Registers can be grouped in Modules. A Module is a collection of Registers and possibly other Modules.

Modules are contained in a Processor.

## 4.5. Worked example of an XML file

In this section we will take a small, contrived example. We want to describe a Processor called 'MyChip'.

The XML file starts with the opening tag, and a Processor tag:

```
<ioconfig>
  <p N="MyChip">
    </f>
  </ioconfig>
```

Now, let us suppose this chip has a set of DMA channels, we can group the registers for those channels together with a Module tag inside the Processor tag:

```
<m N="DMA">
  <d>DMA Channel Registers</d>
</m>
```

For the first channel, we group all the registers for just the first channel inside another Module (this goes inside the first Module tag before the Description):

```
<m N="DMA Channel 0">
  <d>DMA Channel 0 registers</d>
</m>
```

The Module DMA\_Channel\_0 has a 16 bit read only register called DMAC0 (this goes inside the Module tag before the Description):

```
<r N="DMAC0">
  <s>0x04800000</s>
  <sw />
  <ro />
  <d>Control Register For DMA Channel 0</d>
</r>
```

That register is made up of three parts:

Bits	Meaning
0	Power state
1-13	Bytes available in the channel
14-15	Mode

We break the register up into those parts using Formats. A Format is basically a bitfield within a register.

The Bytes Available bitfield is a numerical bitfield; that is, the bitfield should be displayed as a number. It looks like this:

```
<f N="BytesAvailable">
  <fm>0x3FFE</fm>
  <rs>1</rs>
  <D/>
  <d>bytes available in the channel</d>
</f>
```

The `<f>` tag determine what bits are relevant to the bitfield (1 thru 13 inclusive), and the `<ShiftR>` tag indicates that the value should be shifted to the right once before being displayed. The `<Dec/>` tag indicates that the bitfield should be displayed in Decimal.

The Mode bitfield is a two bit bitfield, where each possible value for the bitfield can have a different meaning. In this case 0x00 means forwards, 0x11 means backwards, and 0x02 means sideways. The bitfield looks like this:

```
<f N="Mode">
  <fm>0xC000</fm>
  <rs>14</ rs >
  <dv>Reserved</dv>
  <v N="Forwards">
    <x>0x00</x>
    <d>The mode of the DMA is forwards</d>
  </v>
  <v N="Backwards">
    <x>0x01</x>
    <d>The mode of the DMA is backwards</d>
  </v>
  <v N="Sideways">
    <x>0x02</x>
    <d>The mode of the DMA is sideways</d>
  </v>
  <d>Determines the mode of the channel</d>
</f>
```

Again, the `<f>` tag is used to mask out irrelevant bits, and then the `<rs>` tag indicates that the value of should be shifted to the right 14 places before being compared against any value. The `<v>` tag determines what should be displayed when none of the values match. Each `<v>` tag then states what should be displayed if the value in the register matches the `<x>` tag.

### What all this means to Codescape

Codescape will use the above information to display the value of the register DMAC0 in a human readable format. It will follow the following process (pseudocode):

```
value = Readl6(0x04800000)

print "PowerState = " + (value & 1) ? "On" : "Off";

bytesAvailable = (value & 0x3FFE) >> 1;
print "Bytes Available = " + decimal(bytesAvailable)

mode = (value & 0xC000) >> 14;
if mode == 0 :
    print "Mode = Forwards"
else if mode == 1 :
    print "Mode = Backwards"
else if mode == 2 :
    print "Mode = Sideways"
else :
    print "Mode = Reserved"
```

## 5. New Target Bring-up

### 5.1. Introduction

This section is intended to help user bring-up of new designs either in silicon, FPGA, or emulation. Simply plugging in a debug adapter into your new design, opening Codescape and expecting it to work is somewhat overoptimistic although it often does work. Following the steps detailed in this guide will help eliminate problems methodically.

Codescape Console is an interactive Python shell with built-in extensions for debugging via a debug adapter. It is a very good tool for low-level debug because only the commands that are submitted by the user are performed, whereas Codescape debugger has more intrusive target interaction. Previous users familiar with MIPS System Navigator Console (NavCon) will find that Codescape Console is very similar except it uses Python syntax. Command reference documentation is provided online with your installation of Codescape Console in the Documentation directory.

This guide covers MIPS, Meta and UCC targets; note that Meta and UCC debug is very similar and is treated as one in this document.

Target bring-up must be performed in the sequence as listed below, as each stage leaves the target in a particular state. Not following the sequence will result in different responses than the ones documented here.

*Note:*

*The instructions and examples in this chapter were made using a DA-Net probe from Imagination Technologies. If an SP55E probe is used, the procedure is the same but responses to Codescape Console commands may differ from the examples shown.*

*The responses to the commands will vary depending on the target you are using. In some cases the details of the responses have been abbreviated. More information may be shown, depending on your target.*

#### 1. Bypass test

This basic test stage confirms JTAG connectivity and determines scan chain layout for verification against the design.

#### 2. TAP identification

Attempt to identify all the TAPs on the JTAG chain.

#### 3. Perform basic debug operation

Perform a single debug operation using the probes JTAG scan mode, so all JTAG activity can be easily recorded and re-created in a simulator if required.

#### 4. Auto-detect with Codescape Console

Use the debug adapter to auto-detect the target and perform debug functions using Codescape Console.

#### 5. Auto-detect with Codescape Debugger

Use the debug adapter to auto-detect the target and perform debug using Codescape.

### 5.2. Stage 1 - Bypass Test

The bypass test will be done using the Codescape Console command `'jtagchain()'` This command the JTAG chain topology by performing a TAP reset and IR Scan. It will detect the number of taps and the length of each tap on the chain.

1. Connect the debug adapter to the target.





```

Mode      uncommitted
TCK Rate  156kHz [tap 0 of 1] >>> reset(probe)
Identifier DA-net 00238
Firmware  5.4.4.0
Mode      uncommitted
TCK Rate  20000kHz

```

Then the scanonly command:

```

[uncommitted] >>> scanonly()
Determine IR lengths on scan chain and validating number of taps...
[tap 0 of 1] >>>

```

If we check, the mode should now report 'scanonly'.

```

[tap 0 of 1] >>> probe()
Identifier DA-net 00238
Firmware  5.4.4.0
Mode      scanonly
TCK Rate  20000kHz

```

Now issue the `tapinfo()` command and check the results match with your design.

#### MIPS target:

```

[tap 0 of 1] >>> tapinfo()
TAP 0 is a MIPS32 TAP with JTAG ID of 0x00000001

```

#### Meta target with 1 TAP:

```

1 Taps discovered, topology [5]
TAP 0 is a IMG META/UCC TAP with JTAG ID of 0x1fa1166d

```

#### Turning on JTAG logging

If no MIPS or Meta TAPs are discovered or the topology looks wrong then a log of all issued JTAG scans can be created by turning on JTAG logging, this can then be turned into an RTL simulation to check against the design.

```

>>> logging(jtag,on)
jtag on
>>> tapinfo()
352.381:SoC X:Generic :<verbos>: jtag_scan          : JTAG Scan
352.425:SoC X:Generic :<verbos>: jtag_scan          : JTAG Scan
352.425:SoC X:Generic :<jtag> : fill_fifo         : TX Data: 0xffffffff
352.425:SoC X:Generic :<jtag> : fill_fifo         : TX Data: 0xffffffff
352.426:SoC X:Generic :<jtag> : fill_fifo         : TX Data: 0xffffffff
352.426:SoC X:Generic :<jtag> : fill_fifo         : TX Data: 0xffffffff
352.427:SoC 0:Generic :<jtag> : scan_command   : TAP 0: ir scan Scan 128 bits, Quantity 1
352.427:SoC X:Generic :<jtag> : empty_fifo     : RX Data Buffer words 0-3 0xffffffffel
0xffffffff 0xffffffff 0xffffffff
1 Taps discovered, topology [5]
352.513:SoC X:Generic :<verbos>: jtag_scan          : JTAG Scan
352.516:SoC X:Generic :<verbos>: jtag_scan          : JTAG Scan
352.513:SoC X:Generic :<jtag> : fill_fifo         : TX Data: 0x00000001
352.514:SoC 0:Generic :<jtag> : scan_command   : TAP 0: ir scan Scan 5 bits, Quantity 1
352.514:SoC X:Generic :<jtag> : empty_fifo     : RX Data Buffer words 0-3 0x08000000
0x20c62aa0 0xffffffff 0xffffffff
352.516:SoC X:Generic :<jtag> : fill_fifo         : TX Data: 0x00000000
352.517:SoC 0:Generic :<jtag> : scan_command   : TAP 0: dr_scan Scan 32 bits, Quantity 1

```

```

352.517:SoC X:Generic : <jtag> : empty_fifo : RX Data Buffer words 0-3 0x1fa1166d
0x20c626b8 0xffffffff 0xffffffff
352.561:SoC X:Generic :<verbo>: jtag_scan : JTAG Scan
352.564:SoC X:Generic :<verbo>: jtag_scan : JTAG Scan
352.562:SoC X:Generic : <jtag> : fill_fifo : TX Data: 0x0000001c
352.562:SoC 0:Generic : <jtag> : scan_command : TAP 0: ir_scan Scan 5 bits, Quantity 1
352.563:SoC X:Generic : <jtag> : empty_fifo : RX Data Buffer words 0-3 0x08000000
0x21124988 0xffffffff 0xffffffff
352.565:SoC X:Generic : <jtag> : fill_fifo : TX Data: 0x00000000
352.565:SoC 0:Generic : <jtag> : scan_command : TAP 0: dr_scan Scan 32 bits, Quantity 1
352.566:SoC X:Generic : <jtag> : empty_fifo : RX Data Buffer words 0-3 0x000529e7
0x21124940 0xffffffff 0xffffffff
TAP 0 is a IMG META/UCC TAP with JTAG ID of 0x1fa1166d

```

## 5.4. Stage 3 - basic debug operation

At this point we have to split the tests into Meta/UCC and MIPS specific sections as we are now going to attempt to communicate with the core(s) behind the TAP(s).

### 5.4.1. MIPS

Configure the TAP index of the core/VPE we want to get into debug mode using `configuretap(<tap_index>)` then issue the `enterdebug()` command:

```

>>> configuretap(0)
>>> enterdebug()
Pending Reset or Reset Occurred clearing Rocc
read of 0xff200200 00000000 nop
read of 0xff200204 0000000F sync
read of 0xff200208 1000FFFD b 0xff200200
read of 0xff20020c 00000000 nop
Second access seen to debug exception vector <done>

```

#### Possible reasons for failure:

Only one VPE can be in debug mode at any one time on a multi-VPE target. For example on a two VPE target if we do the following:

```

>>> enterdebug()
Pending Reset or Reset Occurred clearing Rocc
read of 0xff200200 00000000 nop
read of 0xff200204 0000000F sync
read of 0xff200208 1000FFFD b 0xff200200
read of 0xff20020c 00000000 nop
Second access seen to debug exception vector <done>

>>> configuretap(1)
>>>enterdebug()
RuntimeError: Timeout waiting to Enter Debug mode, ECR: 0x0000d000

```

Or we cannot get TAP1 into debug mode as VPE0 (TAP0) is in debug mode, this can be seen if we issue:

*Note: The commands in the 'for' loop must be indented by at least one space. The indentations must be the same for each line in the loop.*

```

>>> for tap in enumerate(jtagchain()):
...     configuretap(tap[0])
...     tapecr()
...
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv

```

```

0  2  0  0  0  0  0  0  0  1  0  0  1  1  0  0  0
1  0
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0  0  0  0  0  0  0  0  0  0  0  1  1  0  1  0
0  0
    
```

As shown above, Dm = 1 indicates that VPE0 is in debug mode so we cannot get VPE1 into debug mode. If we issue the following commands this will take TAP0 out of debug mode:

*Note: The commands in the 'for' loop must be indented*

```

>>> configuretap(0,jtagchain())
>>> dmseg(ExitDebug)
read of 0xff200200 4200001F      deret
No processor access to dmseg seen in last 10 reads of the EJTAG Control register <done>.
>>> for tap in enumerate(jtagchain()):
...     configuretap(tap[0])
...     tapecr()
...
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0  0  0  0  0  0  0  0  0  0  0  1  1  0  0  0
0  0
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0  2  0  0  0  0  0  0  1  0  0  1  1  0  0  0
1  0
    
```

Notice now that VPE1 has now gone into debug mode because EjtagBrk was pending from the previous `enterdebug()` call.

Another common problem with multi-VPE targets is if the VPE has no TCs bound to it then it cannot enter debug mode, this is indicated by the VPED bit in the ECR.

```

>>> enterdebug()
RuntimeError: VPE Disabled (No TC's Bound to it) when trying to Enter Debug Mode

>>> tapecr()
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0  0  0  1  0  0  0  0  0  0  0  1  1  0  1  0
0  0
    
```

Note that the DA-net does **not** actively drive the DINT signal. If the DINT signal in the system is floating or pulled high, the core may enter debug mode before the probe has had a chance to take control. This can be seen by checking the ECR; Dm will be active but ProbTrap and ProbEn are not set. Also, if we check the pc with `pcsamp()`, we may see it in the (non eJTAG) debug exception location of 0xbFC00480 (although this is dependent on what code is placed at this vector).

```

>>> tapecr()
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  0

>>> pcsamp()
PC New
0xbfc00480 1
    
```

For a more detailed discussion of issues relating to entering debug mode on a MIPS core please refer to 'Low-level EJTAG Debug, on page 48'.

Also note JTAG logging can be enabled to log all JTAG transactions when issuing these commands (to help create an RTL simulation for example) via the same mechanism as before:

```
>>> logging(jtag,on)
jtag on
```

### 5.4.2. Meta/UCC

Meta and UCC debug is considerably simpler than MIPS, there is no debug mode and no need to execute instructions on the processor to do anything, instead the debugger has direct access to memory and memory mapped registers and there are memory mapped 'ports' to allow access to non-mapped resources (registers, core memories, caches). So simply reading a memory mapped ID register is enough to prove basic debug into the core is working. For Meta the relevant ID register is at 0x04800000 and UCC at 0x04801FE0.

For this test use a normal python interpreter or run this as a script:

```
>>> from CSUtils import DATiny
>>> DATiny.UseTarget("DA-net XXX")
>>> DATiny.DAReset()
>>> settings = [
...     wiggle.JtagSettings(J_IMG = 0x3,
...                          J_IMG_F = 0x4,
...                          J_IMG_M = 0x5,
...                          IR_Length = 5,
...                          J_Atten = 0x12,
...                          J_Ctrl = 0),
...     wiggle.JtagSettings(J_IMG = 0x9,
...                          J_IMG_F = 0xa,
...                          J_IMG_M = 0xb,
...                          IR_Length = 5,
...                          J_Atten = 0x12,
...                          J_Ctrl = 0),
... ]
>>> tiny = wiggle.tiny.MetaDebugTiny(DATiny, settings)
>>> tiny.SetTarget(0)
>>> print hex(tiny.ReadMemory(0x04800000))
'0x2010032'
>>> tiny.SetTarget(1)
>>> print hex(tiny.ReadMemory(0x04801fe0))
'0xf010003'
```

*Note: The JTAG settings come from the file `img2_jtag_pack.vhd` in your SoC design. For this test we only need the `J_IMG` encoding and IR length to be correct, in the example below we have a Meta and a UCC on the same TAP (which is standard for a UCCP system).*

If this fails the most common failure is a timeout on the debug port ready bit. This is often caused by the core not being powered or its clocks being gated off. A log of the JTAG transactions can be generated by issuing the following commands:

before issuing `tiny.ReadMemory()`:

```
>>> DATiny.WriteDASetting("JTAG Logging",1)
```

after issuing `tiny.ReadMemory()`:

```
>>> DATiny.WriteDASetting("JTAG Logging",0)
>>> print DATiny.GetDiagnosticFile("DA JTAG Log")
```

## 5.5. Stage 4 – Auto-detect with Codescape Console

We first need to reset the debug adapter mode and turn on probe logging:

```
>>> reset(probe)
Identifier DA-net 00272
Firmware 5.2.2.3
Mode uncommitted
TCK Rate 5000kHz
[uncommitted] >>> logging(probe,1)
probe on
```

### 5.5.1. MIPS

Issue the auto-detect:

```
>>> autodetect()
6008.900:SoC X:Generic : <warn> : da_reset : DA Reset Issued - removing all core information and reverting
to uncommitted mode
6008.903:SoC X:Core 0 : <info> : da_read_config : first command is DA Read Config, Entering Classic Mode!!
6008.905:SoC X:Generic : <info> : determine_target_and_create : Discovering JTAG Chain
6008.911:SoC X:Generic : <info> : determine_target_and_create : JTAG Chain has 2 TAP(s)
6009.019:SoC 0:Generic : <info> : build_tables_from_pnp : JTAG PNP: 0x00000000
6009.020:SoC 0:Generic : <info> : build_tables_from_pnp : Target does not support IMG JTAG PnP
6009.021:SoC 0:Generic : <info> : is_mips32_target : Target looks like mips32!
6009.545:SoC 0:Core 0 : <info> : on_connect_target : Attempting to connect to the target...
6009.546:SoC 0:Core 0 : <info> : set_startup_options : Startup options (0x00423029):
6009.547:SoC 0:Core 0 : <info> : set_startup_options : Firmware: 5.2.2.3 : Mar 17 2014, 12:39:11
6009.548:SoC 0:Core 0 : <info> : set_startup_options : DIAGNOSTIC BUILD
6009.549:SoC 0:Core 0 : <info> : set_startup_options : Debug support enabled.
6009.550:SoC 0:Core 0 : <info> : set_startup_options : Halt after target reset.
6009.551:SoC 0:Core 0 : <info> : set_startup_options : Duplicate IP check enabled.
6009.552:SoC 0:Core 0 : <info> : set_startup_options : DHCP support enabled.
6009.554:SoC 0:Core 0 : <info> : set_startup_options : Process 'SWITCH' instructions in background polling.
6009.555:SoC 0:Core 0 : <info> : set_startup_options : JTAG clock frequency = 5MHz
6009.556:SoC 0:Core 0 : <info> : reset_dash_state : Resetting any stored state information.
6009.558:SoC 0:Core 0 : <info> : set_boot : Setting Normal Boot (on all TAPs) TAP 0
6009.559:SoC 0:Core 0 : <info> : set_boot : Setting Normal Boot (on all TAPs) TAP 1
6009.562:SoC 0:Core 0 : <info> : reset_target : Issuing Reset
6010.420:SoC 0:Core 0 : <info> : reset_target : CPC Probe Mode TAP reset
6010.421:SoC 0:Core 0 : <info> : reset_target : Pre Reset Delay
6010.923:SoC 0:Core 0 : <info> : reset_target : Post Reset Delay
6010.925:SoC 0:Core 0 : <info> : reset_target : Target Reset OK, doing bypass test to check if target
powered
6010.926:SoC 0:Core 0 : <warn> : target_reset_and_startup : Target Reset
6010.927:SoC 0:Core 0 : <info> : enable_debug_support : Enabling debug support...
6010.928:SoC 0:Core 0 : <info> : enable_debug_support : IMPCODE: 0xa1414800 , EJTAG Version 5, ASID Size 2, MIPS16e
Support, No DMA Support, MIPS32
6010.929:SoC 0:Core 0 : <info> : enable_debug_support : ECR: 0xc004c008, ISA:Mips32/64, Debug Mode Active
6010.931:SoC 0:Core 0 : <warn> : enterdebug : Pending Reset or reset occurred clearing Rocc
6010.945:SoC 0:Core 0 : <info> : cache_mips_info : DCR: 0x000703db, Endian Little Endian
6010.946:SoC 0:Core 0 : <info> : cache_mips_info : Enabling PC Sampling
6010.955:SoC 0:Core 0 : <info> : cache_mips_info : PRID: 0x0001a020, Company: Imagination Technologies MIPS, Cpu:
InterActiv UP, Revision 32
6010.961:SoC 0:Core 0 : <info> : cache_config : config0: 0x81840482
6010.967:SoC 0:Core 0 : <info> : cache_config : config1: 0xfea351df
6010.973:SoC 0:Core 0 : <info> : cache_config : config2: 0x80000447
6010.979:SoC 0:Core 0 : <info> : cache_config : config3: 0x82003e2d
6010.985:SoC 0:Core 0 : <info> : cache_config : config4: 0xc01c0000
6010.991:SoC 0:Core 0 : <info> : cache_config : config5: 0x10000000
6010.997:SoC 0:Core 0 : <warn> : cache_mips_info : MT-ASE detected, Core has 2 VPEs and 5 TCs
6011.026:SoC 0:Core 0 : <info> : dump_tc_status : TC 0 Running [on HW Thread (VPE) 0 which is Active]
6011.052:SoC 0:Core 0 : <info> : dump_tc_status : TC 1 Running and Halted [on HW Thread (VPE) 1 which is
Inactive (VPA in CP0 1.2 not set)]
6011.079:SoC 0:Core 0 : <info> : dump_tc_status : TC 2 Running and Halted [on HW Thread (VPE) 1 which is
Inactive (VPA in CP0 1.2 not set)]
6011.106:SoC 0:Core 0 : <info> : dump_tc_status : TC 3 Running and Halted [on HW Thread (VPE) 1 which is
Inactive (VPA in CP0 1.2 not set)]
6011.133:SoC 0:Core 0 : <info> : dump_tc_status : TC 4 Running and Halted [on HW Thread (VPE) 1 which is
Inactive (VPA in CP0 1.2 not set)]
6011.136:SoC 0:Core 0 : <info> : dump_tc_status : Note ALL VPEs except 0 disable via EVP bit in MVPConf (CP0
0.1)
6011.187:SoC 0:Core 0 : <info> : cache_mips_info : VPE1 ECR = 0xc004c008
6011.188:SoC 0:Core 0 : <warn> : enterdebug : Pending Reset or reset occurred clearing Rocc
6011.279:SoC 0:Core 0 : <info> : target_reset_and_startup : Debug support enabled.
Identifier DA-net 00272
Firmware 5.2.2.3
Mode autodetected
TCK Rate 5000kHz
>>>
```

This example is from a single-core Dual VPE target (interAptiv-UP).

Check everything in the log is as expected for your design, the main failure at this point would be a failure to enter debug mode. If this occurs repeat stage 3 to see if you can re-create the failure using the simple scan method.

In this example we can see that we have set 'Normal Boot' and reset the target, this causes Core0 VPE0 to run from the boot exception vector and not enter debug mode immediately (ETJAG Boot). If you do not want the debug adapter to reset the target on detection, the setting 'Reset on Connect' can be set to 0 before issuing the auto-detect:

```
>>> config("Reset on Connect",0)
>>> autodetect()
```

The reset type EJTAG boot vs Normal Boot is set via DAConfig by the option 'Halt After Target Reset'. This can also be set when issuing a reset command in Codescape Console, see **help(reset)** for details.

Examine the target registers:

```
>>> regs()
zero 00000000 at 00000000 v0 00000000 v1 80070000
a0 00000000 a1 00000000 a2 00000000 a3 800be248
t0 00000001 t1 800af7ec t2 800af80c t3 00000020
t4 00000001 t5 800af7c0 t6 00000000 t7 800af770
s0 800be288 s1 00000000 s2 00000001 s3 00000042
s4 0000001b s5 0000005b s6 ffffffff9b s7 800be818
t8 800af7fc t9 80033f30 k0 00000000 k1 80075ae8
gp 9fc10478 sp 800be230 s8 800be288 ra 8002b064

hi 00000000 lo 00000000 depc 8002b064 pc 8002b064
status 24002c01 cause 50808000 epc 80047288 badvaddr 00000000
index 00000000 random 0000000c entrylo0 00000000 entrylo1 00000000
context 00000000 pagemask 00000000 wired 00000000 count cb535e07
entryhi 00000000 compare 00000000 pridr 0001a020 errorepc ff200210
config 81840483 config1 fea351df config2 80000447 config3 82003e2d
lladdr 00000000 watchlo 00000000 watchhi 80000000 debug 40128020
taglo 00000000 data10 00000000 pagegrain 00000000 tracecontrol 00000000
```

Check they look sensible.

Stop all cores/VPEs:

```
>>> cmdall(halt)
c0v0: status=halted_by_probe pc=0x8002b064
0x8002b064 8fa4001c lw a0, 28(sp)
c0v1: status=stopped pc=0x80000e30
0x80000e30 000d7242 srl t6, t5, 0x9
```

At this point we will attempt to read memory for the first time as we try to disassemble the op-code at the PC.

If the PC is pointing at invalid memory you could potentially get an error. If your SoC is well designed hopefully you will get a bus error exception, but on many SoCs an access to an invalid address usually locks up the core and it can often only be recovered by a hard reset.

The example below is from a coreFPGA 5 system which has been reset with EJTAG Boot set. The memory controller on this FPGA locks up the bus if the DDR is accessed before it is set up. By enabling logging of debug mode ops we can see the failure point.

```

>>> reset (ejtagboot)
>>> config("Verbose Logging",1)
1
>>> config("Log Debug Instructions",1)
1
>>> logging (probe,1)
probe on
>>>
>>> word(0xA0000000)
331.607:SoC X:Core 0 :<verbo>: read_memory          : read mem:      Count: 1 Mem unit: 0, address: 0x80000000
331.608:SoC 0:Core 0 :<verbo>: halt                  : [thread 0]
331.609:SoC 0:Core 0 :<verbo>: push_default_regs       :
331.609:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200204 4081f800 mtc0 at,
c0_desave
331.610:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200208 3c01ff28 lui at, 0xff28
331.610:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff20020c ac284000 sw t0,
16384 (at)
331.610:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200210 ac294004 sw t1,
16388 (at)
331.611:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200214 4008c000 mfc0 t0, c0_depc
331.611:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200218 ac287ffc sw t0,
32764 (at)
331.612:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff20021c 0000000f sync
331.612:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200220 1000ffff b 0xff200204
331.613:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200224 00000000 nop
331.613:SoC 0:Core 0 :<verbo>: read_cp0_register       :
331.614:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200204 4008b800 mfc0 t0,
c0_debug
331.614:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200208 ac280000 sw t0, 0(at)
331.614:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff20020c 0000000f sync
331.615:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200210 1000ffff b 0xff200204
331.615:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200214 00000000 nop
331.616:SoC 0:Core 0 :<verbo>: write_cp0_register     :
331.616:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200204 3c084c12 lui t0, 0x4c12
331.617:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200208 350880a0 ori t0, t0,
0x80a0
331.617:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff20020c 4088b800 mtc0 t0,
c0_debug
331.617:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200210 1000ffff b 0xff200204
331.618:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200214 00000000 nop
331.618:SoC 0:Core 0 :<verbo>: read_eva_kernel       :
331.618:SoC 0:Core 0 :<verbo>: read_cp0_register     :
331.619:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200204 40082803 mfc0 t0, $5, 3
331.619:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200208 ac280000 sw t0, 0(at)
331.620:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff20020c 0000000f sync
331.620:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200210 1000ffff b 0xff200204
331.621:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200214 00000000 nop
331.621:SoC 0:Core 0 :<verbo>: check_eva_kernel_segments : Access is uncached, Access is unmapped
331.621:SoC 0:Core 0 :<verbo>: block_read32       :
331.622:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200204 3c088000 lui t0, 0xA000
331.622:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200208 35080000 ori t0, t0, 0x0
331.623:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff20020c 8d090000 lw t1, 0(t0)
331.623:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200210 ac290000 sw t1, 0(at)
331.623:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200214 0000000f sync
331.624:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200218 1000ffff b 0xff200204
331.624:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff20021c 00000000 nop
331.625:SoC 0:Core 0 :<verbo>: freeze             : [thread 0]
331.625:SoC 0:Core 0 :<verbo>: pop_default_regs   :
331.625:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200204 8c284000 lw t0,
16384 (at)
331.626:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200208 8c294004 lw t1,
16388 (at)
331.626:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff20020c 4001f800 mfc0 at,
c0_desave
331.626:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200210 1000ffff b 0xff200204
331.627:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200214 0000000f sync
331.627:SoC 0:Core 0 :<verbo>: execute             : eJTAG-DMSEG: exec 0xff200204 0000000f sync
332.628:SoC 0:Core 0 :<error>: execute             : [thread 0] Timeout waiting for PrACC
332.628:SoC 0:Core 0 :<info>: print_all_ehrs       : TAP 0, ECR = 0x0000c008
332.628:SoC 0:Core 0 :<info>: print_all_ehrs       : TAP 1, ECR = 0x0000c000
332.629:SoC X:Generic <except>: dispatch_cmd      : dbg::exception Timeout waiting for PrACC
status = 1
Error: Unable to read memory at 0x00:0xA0000000[1]. read_memory : CON: Command fatal error : Timeout waiting for PrACC

```

The example above shows that the target locks up at the two back-to-back sync instructions. This is because the core does not lock up until the next access (or sync) to the external bus after the access that caused the bus to hang.

Assuming your core/VPEs stopped normally (i.e. did not lock-up), try stepping a few instructions:

```

>>> step()
status=single_stepped pc=0x80033f30
0x80033f30 03e00008 jr ra
>>> step()
status=single_stepped pc=0x8000ab38
0x8000ab38 8fbf0024 lw ra, 36(sp)

```



```

>>> step()
status=single_stepped pc=0x8000ab3c
0x8000ab3c 03e00008 jr ra
>>> step()
status=single_stepped pc=0x80009d30
0x80009d30 1600ffff bnez s0, 0x80009d28
>>>
>>> step()
status=single_stepped pc=0x80009d28
0x80009d28 0c002ac5 jal 0x8000ab14
    
```

Now try a block read:

```

>>> word(0x80000000,count=256)
0x80000000 08011319 241a0000 a0000008 a000000c .....
$.
0x800003f0 a00003f0 a00003f4 a00003f8 a00003fc .....
>>>
    
```

Try a block write, start off in uncached space (kseg1):

```

>>> temp = word(0xA0000000,count=256)
>>> word(0xA0000000,temp,count=256)
0xa0000000 08011319 241a0000 a0000008 a000000c .....

0xa00003f0 a00003f0 a00003f4 a00003f8 a00003fc .....
    
```

There is a chance this could fail, the most obvious reason being the memory is not writable at this address (if so pick an address that is writable).

Other failure reasons could include:

By default, block writes use fast transfers (Fast Write), this requires loading a fast stub into memory (by default at 0x80000000), locking this into the first two cache lines and then restoring memory. There are various reasons why this could fail.

The first thing to try is to try the write again with fast transfers disabled (a reset may be needed to recover from the earlier failed write).

```

>>> config("Fast Writes",0)
0
>>> word(0xA0000000,temp,count=256)
    
```

If the write still fails then there is probably something wrong with the SoC design (at least something on the bus external to the MIPS core where the write transaction is issued). Enable 'Verbose Transactions' and 'Log Debug Instructions' configuration options and turn on probe logging, this may help pinpoint where the write failed.

If the write worked with fast transfers disabled, we need to check if the fast transfer stub load address is writeable. We need to be able to write the first 64-bytes at address 0x80000000. If this is not possible, the stub load address can be configured to a different address using the configuration option 'Fast Monitor Address' e.g.:

```

>>> config("Fast Monitor Address",0x90000000)
0x90000000
    
```

If 0x80000000 is writable, or you have moved the stub to a writeable address and Fast Writes are still failing, then try with the fast monitor not locked into the cache (we leave it in RAM, note that this is not safe on multi-core SoCs when the other cores are running).

We need to **run** and **halt** the core/VPE first to flush out the current fast monitor, then set the configuration option to do the write:

```
>>> go()
Running from 0x80009d28
status=running
>>> halt()
status=stopped pc=0x8000ab18
0x8000ab18 27bdffd8 addiu sp, sp, -40
>>> config("Lock Monitor in Cache",0)
0
>>> word(0xA0000000,temp,count=256)
```

If this still fails, the problem is most likely an issue with the bus fabric of the SoC.

If it now works then there could be an issue with the caches in the system (note that for systems with no I-cache the debug adapter will automatically switch to placing the debug stub in RAM).

So we will re-load the stub into cache and then dump the cache to see if we can see the monitor in the cache:

```
>>> go()
Running from 0x8000ab18
status=running
>>> halt()
status=stopped pc=0x8000cc90
0x8000cc90 27a40018 addiu a0, sp, 24
>>> config("Lock Monitor in Cache",1)
1
>>> word(0xA0000000,temp,count=256)
<snip>

>>> cachedump(instr,0,0x20)
Offset Set Way TagLo Word 0 Word 1 Word 2 Word 3 Word 4 Word 5 Word 6 Word 7
7
0000 000 0 0002a080 02028021 8e39d354 02602021 0320f809 02e02821 8ec3d350 3c048008
8c99d354 !...T.9.! `... !!(..P.....<T...
0000 000 1 100000a0 8d4b0000 ad0b0000 051f0000 25080004 1509ffffb 00000000 03e00408
00000000 ..K.....%.....
0000 000 2 000000a0 8d4b0000 ad0b0000 051f0000 25080004 1509ffffb 00000000 03e00408
00000000 ..K.....%.....
0000 000 3 0002c080 afb10024 0c00b75e afb00020 8fa30018 24020002 30630007 10620015
3c148007 $....^... ..$.c0..b....<
0020 001 0 000000a0 8d0b0000 ad4b0000 055f0000 25080004 1509ffffb 00000000 03e00408
0000000f .....K.....%.....
0020 001 1 0000c080 27bdffc8 afb00020 3c108007 96023b70 afb20028 24120001 afbf0034
afb40030 ...' .....<p;..(.....$4...0...
0020 001 2 100000a0 8d0b0000 ad4b0000 055f0000 25080004 1509ffffb 00000000 03e00408
0000000f .....K.....%.....
0020 001 3 0002c080 9282d7a8 1440001a 3c028007 3c128007 8e42d7e4 1040000b 3c118000
00008021 .....@.....<...<..B...@.....<!...
>>>
```

You should see the following line in one of the ways (which one is implementation specific) at offset 0:

```
000000a0 8d4b0000 ad0b0000 051f0000 25080004 1509ffff 00000000 03e00408 00000000 ..K.....%.....
```

You should see this line in one of the ways at offset 0x20:

```
000000a0 8d0b0000 ad4b0000 055f0000 25080004 1509ffff 00000000 03e00408 0000000f .....K..._...%.....
```

*Note: This assumes your monitor load address is 0x80000000. If it is different you would need to calculate the offset into the cache based on the address you used. If you look at the example dump data, you can see the monitor actual appears twice in this instance, but with a different tag value of 0x100000A0. This is because earlier the monitor was moved to address 0x90000000, which maps to the same line, but the physical address in the tag is different.*

The 0xA0 in the lower byte of the TAG shows the line is valid and locked, If you cannot see this data there may be a fault with your cache rams.

If you have got this far, it looks like basic target communications are OK and you can spend some time using Codescape Console to do some real debugging. You can load elf files, set breakpoints, and single step etc.

Codescape Console only performs the commands submitted by the user and is less intrusive than Codescape Debugger which performs additional memory reads to make debugging faster. This additional activity can cause failures when bringing-up new devices. Once you are happy, move on to 'Stage 5 – Auto-detect with Codescape' on page 46.

### 5.5.2. Meta/UCC

Issue the auto-detect:

```
>>> autodetect()
9701.621:SoC X:Core 0 :<verbos>: da_read_config          : read_config
9701.622:SoC X:Core 0 :<info> : da_read_config          : first command is DA Read Config, Entering Classic Mode!!
9701.622:SoC X:Generic :<info> : determine_target_and_create : Discovering JTAG Chain
9701.627:SoC X:Generic :<info> : determine_target_and_create : JTAG Chain has 1 TAP(s)
9701.728:SoC 0:Generic :<info> : build_tables_from_pnp      : JTAG PNP: 0x000529e7
9701.728:SoC 0:Generic :<info> : build_tables_from_pnp      : Target Supports IMG JTAG PnP ! , version 2 collecting SoC
information....
9701.728:SoC 0:Generic :<info> : build_tables_from_pnp      : Number of Cores: 5
9701.729:SoC 0:Generic :<info> : build_tables_from_pnp      : Additional latency in Debug Chains: 0 registers
9701.729:SoC 0:Generic :<info> : build_tables_from_pnp      : Attention Instruction Encoding: 00000012
9701.729:SoC 0:Generic :<info> : build_tables_from_pnp      : Core 1 --- Core ID(4): META2, Debug Rev: 2
9701.729:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG Instruction Encoding: 00000003
9701.729:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG F Instruction Encoding: 00000004
9701.730:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG M Instruction Encoding: 00000005
9701.730:SoC 0:Generic :<info> : build_tables_from_pnp      : Core 2 --- Core ID(2): MTX, Debug Rev: 2
9701.731:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG Instruction Encoding: 00000006
9701.731:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG F Instruction Encoding: 00000007
9701.731:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG M Instruction Encoding: 00000008
9701.731:SoC 0:Generic :<info> : build_tables_from_pnp      : Core 3 --- Core ID(3): UCC, Debug Rev: 2
9701.732:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG Instruction Encoding: 00000009
9701.732:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG F Instruction Encoding: 0000000a
9701.732:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG M Instruction Encoding: 0000000b
9701.733:SoC 0:Generic :<info> : build_tables_from_pnp      : Core 4 --- Core ID(2): MTX, Debug Rev: 2
9701.733:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG Instruction Encoding: 0000000c
9701.733:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG F Instruction Encoding: 0000000d
9701.733:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG M Instruction Encoding: 0000000e
9701.734:SoC 0:Generic :<info> : build_tables_from_pnp      : Core 5 --- Core ID(3): UCC, Debug Rev: 2
9701.734:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG Instruction Encoding: 0000000f
9701.734:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG F Instruction Encoding: 00000010
9701.734:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG M Instruction Encoding: 00000011
9701.735:SoC 0:Generic :<info> : build_tables_from_pnp      : Target supports JTAG Status Instruction
9701.735:SoC 0:Generic :<info> : build_tables_from_pnp      : J_IMG S Instruction Encoding: 0000001e
9701.835:SoC 0:Core 0 :<info> : create_core_object         : Creating Meta 2.1 Processor Object
9701.836:SoC 0:Core 0 :<info> : get_slave_setting          : Master Debug System Detected !
9701.836:SoC 0:Core 0 :<info> : get_slave_setting          : Master Debug System Detected !
9701.841:SoC 0:Core 0 :<info> : discover                    : Core Memory Information supplied by table data
9701.864:SoC 0:Core 0 :<info> : discover_size               : Icache size = 16K
9701.864:SoC 0:Core 0 :<info> : discover_size               : Dcache size = 16K
9702.365:SoC 0:Core 0 :<info> : on_connect_target          : Attempting to reset the target...
9702.365:SoC 0:Core 0 :<info> : set_startup_options         : Startup options (0x00423028):
9702.365:SoC 0:Core 0 :<info> : set_startup_options         : Firmware: 5.3.0.0 : Mar 19 2014, 00:33:18
9702.365:SoC 0:Core 0 :<info> : set_startup_options         : Debug support enabled.
9702.366:SoC 0:Core 0 :<info> : set_startup_options         : Do not halt after target reset.
```

```

9702.366:SoC 0:Core 0 : <info> : set_startup_options : DA is a master able to reset its target.
9702.366:SoC 0:Core 0 : <info> : set_startup_options : Duplicate IP check enabled.
9702.366:SoC 0:Core 0 : <info> : set_startup_options : DHCP support enabled.
9702.367:SoC 0:Core 0 : <info> : set_startup_options : Process 'SWITCH' instructions in background polling.
9702.367:SoC 0:Core 0 : <info> : set_startup_options : JTAG clock frequency = 5MHz
9702.367:SoC 0:Core 0 : <info> : target_reset_and_startup : Target is a Reset Master
9702.367:SoC 0:Core 0 : <info> : target_reset_and_startup : Target is a Debug Master
9702.368:SoC 0:Core 0 : <info> : reset_dash_state : Resetting any stored m_state information.
9702.370:SoC 0:Core 0 : <info> : reset_target : Issuing Reset
9703.722:SoC 0:Core 0 : <info> : reset_target : Target Reset OK, doing bypass test to check if target
powered
9703.722:SoC 0:Core 0 : <warn> : target_reset_and_startup : Target Reset
9703.722:SoC 0:Core 0 : <info> : enable_debug_support : Enabling debug support...
9703.723:SoC 0:Core 0 : <info> : get_JTAG_privilege : We have JTAG port privilege.
9703.726:SoC 0:Core 0 : <info> : enable_debug_support : Core version: major 2, minor 1, revision 3
9703.726:SoC 0:Core 0 : <info> : enable_debug_support : Core ID: 2t2d16 (harrier)
9703.728:SoC 0:Core 0 : <info> : set_vector_halt_triggers : Setting TOVECTINT_BHALT to MDBG port
9703.730:SoC 0:Core 0 : <info> : set_vector_halt_triggers : Setting TOVECTINT_IHALT to MDBG port
9703.732:SoC 0:Core 0 : <info> : set_vector_halt_triggers : Setting TOVECTINT_RHALT to MDBG port
9703.733:SoC 0:Core 0 : <info> : set_vector_halt_triggers : Setting T1VECTINT_BHALT to MDBG port
9703.735:SoC 0:Core 0 : <info> : set_vector_halt_triggers : Setting T1VECTINT_IHALT to MDBG port
9703.736:SoC 0:Core 0 : <info> : set_vector_halt_triggers : Setting T1VECTINT_RHALT to MDBG port
9703.737:SoC 0:Core 0 : <info> : read_thread_info : Thread 0 is DSP, PC=0x00000000, m_state is Stopped
9703.739:SoC 0:Core 0 : <info> : read_thread_info : Thread 1 is DSP, PC=0x00000000, m_state is Stopped
9703.767:SoC 0:Core 0 : <info> : target_reset_and_startup : Debug support enabled.
9703.767:SoC 0:Core 1 : <info> : create_core_object : Creating Mtx 1.2 Processor Object
9703.768:SoC 0:Core 1 : <info> : get_slave_setting : Master Debug System Detected !
9704.269:SoC 0:Core 1 : <info> : on_connect_target : Attempting to reset the target...
9704.269:SoC 0:Core 1 : <info> : set_startup_options : Startup options (0x00423028):
9704.270:SoC 0:Core 1 : <info> : set_startup_options : Firmware: 5.3.0.0 : Mar 19 2014, 00:33:18
9704.270:SoC 0:Core 1 : <info> : set_startup_options : Debug support enabled.
9704.270:SoC 0:Core 1 : <info> : set_startup_options : Do not halt after target reset.
9704.271:SoC 0:Core 1 : <info> : set_startup_options : DA is a master able to reset its target.
9704.271:SoC 0:Core 1 : <info> : set_startup_options : Duplicate IP check enabled.
9704.271:SoC 0:Core 1 : <info> : set_startup_options : DHCP support enabled.
9704.271:SoC 0:Core 1 : <info> : set_startup_options : Process 'SWITCH' instructions in background polling.
9704.272:SoC 0:Core 1 : <info> : set_startup_options : JTAG clock frequency = 5MHz
9704.272:SoC 0:Core 1 : <info> : target_reset_and_startup : Target is a Reset Slave
9704.272:SoC 0:Core 1 : <info> : target_reset_and_startup : Target is a Debug Master
9704.272:SoC 0:Core 1 : <info> : reset_dash_state : Resetting m_state
9704.273:SoC 0:Core 1 : <warn> : reset_target : Resetting Single MTX only
9704.283:SoC 0:Core 1 : <info> : reset_target : Target Reset OK, doing bypass test to check if target
powered
9704.283:SoC 0:Core 1 : <warn> : target_reset_and_startup : Target Reset
9704.283:SoC 0:Core 1 : <info> : enable_debug_support : enabling debug support...
9704.284:SoC 0:Core 1 : <info> : enable_debug_support : Core version: major 1, minor 2, revision 0, sub-identifier
0
9704.285:SoC 0:Core 1 : <info> : read_thread_info : Thread 0 is MTX, PC=0x00000000, m_state is Stopped
9704.285:SoC 0:Core 1 : <info> : discover : Identify dynamically Core Memory Region Units
9704.294:SoC 0:Core 1 : <verbo>: discover_coremem_range : valid memory at specifier 0x10
9704.294:SoC 0:Core 1 : <verbo>: discover_coremem_range : valid memory at specifier 0x11
9704.295:SoC 0:Core 1 : <verbo>: discover_coremem_range : valid memory at specifier 0x12
9704.296:SoC 0:Core 1 : <verbo>: discover_coremem_range : valid memory at specifier 0x13
9704.297:SoC 0:Core 1 : <verbo>: discover_coremem_range : valid memory at specifier 0x14
9704.315:SoC 0:Core 1 : <verbo>: discover_coremem_range : no memory at specifier 0x15 (0 != b0b1e06)
9704.333:SoC 0:Core 1 : <verbo>: discover_coremem_range : no memory at specifier 0x16 (0 != b0b1e07)
9704.352:SoC 0:Core 1 : <verbo>: discover_coremem_range : no memory at specifier 0x17 (0 != b0b1e08)
9704.364:SoC 0:Core 1 : <verbo>: discover_coremem_range : valid memory at specifier 0x18
9704.365:SoC 0:Core 1 : <verbo>: discover_coremem_range : valid memory at specifier 0x19
9704.365:SoC 0:Core 1 : <verbo>: discover_coremem_range : valid memory at specifier 0x1a
9704.366:SoC 0:Core 1 : <verbo>: discover_coremem_range : valid memory at specifier 0x1b
9704.367:SoC 0:Core 1 : <verbo>: discover_coremem_range : valid memory at specifier 0x1c
9704.385:SoC 0:Core 1 : <verbo>: discover_coremem_range : no memory at specifier 0x1d (0 != b0b1e06)
9704.404:SoC 0:Core 1 : <verbo>: discover_coremem_range : no memory at specifier 0x1e (0 != b0b1e07)
9704.422:SoC 0:Core 1 : <verbo>: discover_coremem_range : no memory at specifier 0x1f (0 != b0b1e08)
9704.655:SoC 0:Core 1 : <info> : print_layout :
5 Core Code Memories
9704.655:SoC 0:Core 1 : <info> : print_mem : RAM @ address 0x80000000, size 64 Kbytes.
9704.655:SoC 0:Core 1 : <info> : print_mem : RAM @ address 0x80010000, size 64 Kbytes.
9704.656:SoC 0:Core 1 : <info> : print_mem : RAM @ address 0x80020000, size 64 Kbytes.
9704.656:SoC 0:Core 1 : <info> : print_mem : RAM @ address 0x80030000, size 64 Kbytes.
9704.656:SoC 0:Core 1 : <info> : print_mem : RAM @ address 0x80040000, size 64 Kbytes.
9704.657:SoC 0:Core 1 : <info> : print_layout :
5 Core Data Memories
9704.657:SoC 0:Core 1 : <info> : print_mem : RAM @ address 0x82000000, size 64 Kbytes.
9704.657:SoC 0:Core 1 : <info> : print_mem : RAM @ address 0x82010000, size 64 Kbytes.
9704.657:SoC 0:Core 1 : <info> : print_mem : RAM @ address 0x82020000, size 64 Kbytes.
9704.658:SoC 0:Core 1 : <info> : print_mem : RAM @ address 0x82030000, size 64 Kbytes.
9704.658:SoC 0:Core 1 : <info> : print_mem : RAM @ address 0x82040000, size 64 Kbytes.
9704.666:SoC 0:Core 1 : <info> : target_reset_and_startup : Debug support enabled.
9704.667:SoC 0:Core 2 : <info> : create_core_object : Creating UCC-MCP Processor Object
9704.667:SoC 0:Core 2 : <info> : on_connect_target : Attempting to reset the target...
9704.668:SoC 0:Core 2 : <info> : set_startup_options : Startup options (0x00423028):
9704.668:SoC 0:Core 2 : <info> : set_startup_options : Firmware: 5.3.0.0 : Mar 19 2014, 00:33:18
9704.668:SoC 0:Core 2 : <info> : set_startup_options : Debug support enabled.
9704.668:SoC 0:Core 2 : <info> : set_startup_options : Do not halt after target reset.
9704.669:SoC 0:Core 2 : <info> : set_startup_options : Duplicate IP check enabled.
9704.669:SoC 0:Core 2 : <info> : set_startup_options : DHCP support enabled.
9704.669:SoC 0:Core 2 : <info> : set_startup_options : JTAG clock frequency = 5MHz
9704.670:SoC 0:Core 2 : <info> : reset_dash_state : Resetting state
9704.670:SoC 0:Core 2 : <warn> : reset_target : Resetting Single MCP only
9704.680:SoC 0:Core 2 : <info> : reset_target : Target Reset OK, doing bypass test to check if target
powered
9704.680:SoC 0:Core 2 : <warn> : target_reset_and_startup : Target Reset
9704.680:SoC 0:Core 2 : <info> : enable_debug_support : enabling debug support...
9704.681:SoC 0:Core 2 : <verbo>: read_id_reg : tcontext_ucc::read_id_reg - 0f010003
9704.681:SoC 0:Core 2 : <info> : enable_debug_support : MCP Core version: Group f, Id 1, Configuration 3
9704.681:SoC 0:Core 2 : <info> : enable_debug_support : UCC system version (assumption made on MCP id) : 310
9704.682:SoC 0:Core 2 : <info> : read_thread_info : UCC Thread PC=0x00000000, state is Stopped
9704.682:SoC 0:Core 2 : <warn> : target_reset_and_startup : Debug support enabled.
9704.682:SoC 0:Core 3 : <info> : create_core_object : Creating Mtx 1.2 Processor Object
9704.683:SoC 0:Core 3 : <info> : get_slave_setting : Master Debug System Detected !
9705.184:SoC 0:Core 3 : <info> : on_connect_target : Attempting to reset the target...
9705.185:SoC 0:Core 3 : <info> : set_startup_options : Startup options (0x00423028):

```

```

9705.185:SoC 0:Core 3 : <info> : set_startup_options : Firmware: 5.3.0.0 : Mar 19 2014, 00:33:18
9705.185:SoC 0:Core 3 : <info> : set_startup_options : Debug support enabled.
9705.185:SoC 0:Core 3 : <info> : set_startup_options : Do not halt after target reset.
9705.186:SoC 0:Core 3 : <info> : set_startup_options : DA is a master able to reset its target.
9705.186:SoC 0:Core 3 : <info> : set_startup_options : Duplicate IP check enabled.
9705.186:SoC 0:Core 3 : <info> : set_startup_options : DHCP support enabled.
9705.186:SoC 0:Core 3 : <info> : set_startup_options : Process 'SWITCH' instructions in background polling.
9705.187:SoC 0:Core 3 : <info> : set_startup_options : JTAG clock frequency = 5MHz
9705.187:SoC 0:Core 3 : <info> : target_reset_and_startup : Target is a Reset Slave
9705.187:SoC 0:Core 3 : <info> : target_reset_and_startup : Target is a Debug Master
9705.187:SoC 0:Core 3 : <info> : reset_dash_state : Resetting m_state
9705.188:SoC 0:Core 3 : <warn> : reset_target : Resetting Single MTX only
9705.199:SoC 0:Core 3 : <info> : reset_target : Target Reset OK, doing bypass test to check if target
powered
9705.199:SoC 0:Core 3 : <warn> : target_reset_and_startup : Target Reset
9705.199:SoC 0:Core 3 : <info> : enable_debug_support : enabling debug support...
9705.200:SoC 0:Core 3 : <info> : enable_debug_support : Core version: major 1, minor 2, revision 0, sub-identifier
0
9705.200:SoC 0:Core 3 : <info> : set_up_HWSTATMETA : HWSTATMETA - 0x80010000
9705.201:SoC 0:Core 3 : <info> : set_up_HWSTATMETA : now 0x00000000
9705.202:SoC 0:Core 3 : <info> : read_thread_info : Thread 0 is MTX, PC=0x00000000, m_state is Stopped
9705.202:SoC 0:Core 3 : <info> : discover : Identify dynamically Core Memory Region Units
9705.210:SoC 0:Core 3 : <verbo>: discover_coremem_range : valid memory at specifier 0x10
9705.211:SoC 0:Core 3 : <verbo>: discover_coremem_range : valid memory at specifier 0x11
9705.212:SoC 0:Core 3 : <verbo>: discover_coremem_range : valid memory at specifier 0x12
9705.212:SoC 0:Core 3 : <verbo>: discover_coremem_range : valid memory at specifier 0x13
9705.213:SoC 0:Core 3 : <verbo>: discover_coremem_range : valid memory at specifier 0x14
9705.232:SoC 0:Core 3 : <verbo>: discover_coremem_range : no memory at specifier 0x15 (0 != b0ble06)
9705.250:SoC 0:Core 3 : <verbo>: discover_coremem_range : no memory at specifier 0x16 (0 != b0ble07)
9705.269:SoC 0:Core 3 : <verbo>: discover_coremem_range : no memory at specifier 0x17 (0 != b0ble08)
9705.281:SoC 0:Core 3 : <verbo>: discover_coremem_range : valid memory at specifier 0x18
9705.281:SoC 0:Core 3 : <verbo>: discover_coremem_range : valid memory at specifier 0x19
9705.282:SoC 0:Core 3 : <verbo>: discover_coremem_range : valid memory at specifier 0x1a
9705.283:SoC 0:Core 3 : <verbo>: discover_coremem_range : valid memory at specifier 0x1b
9705.283:SoC 0:Core 3 : <verbo>: discover_coremem_range : valid memory at specifier 0x1c
9705.302:SoC 0:Core 3 : <verbo>: discover_coremem_range : no memory at specifier 0x1d (0 != b0ble06)
9705.320:SoC 0:Core 3 : <verbo>: discover_coremem_range : no memory at specifier 0x1e (0 != b0ble07)
9705.339:SoC 0:Core 3 : <verbo>: discover_coremem_range : no memory at specifier 0x1f (0 != b0ble08)
9705.570:SoC 0:Core 3 : <info> : print_layout : 5 Core Code Memories
9705.570:SoC 0:Core 3 : <info> : print_mem : RAM @ address 0x80000000, size 64 Kbytes.
9705.571:SoC 0:Core 3 : <info> : print_mem : RAM @ address 0x80010000, size 64 Kbytes.
9705.571:SoC 0:Core 3 : <info> : print_mem : RAM @ address 0x80020000, size 64 Kbytes.
9705.571:SoC 0:Core 3 : <info> : print_mem : RAM @ address 0x80030000, size 64 Kbytes.
9705.571:SoC 0:Core 3 : <info> : print_mem : RAM @ address 0x80040000, size 64 Kbytes.
9705.572:SoC 0:Core 3 : <info> : print_layout : 5 Core Data Memories
9705.572:SoC 0:Core 3 : <info> : print_mem : RAM @ address 0x82000000, size 64 Kbytes.
9705.572:SoC 0:Core 3 : <info> : print_mem : RAM @ address 0x82010000, size 64 Kbytes.
9705.572:SoC 0:Core 3 : <info> : print_mem : RAM @ address 0x82020000, size 64 Kbytes.
9705.572:SoC 0:Core 3 : <info> : print_mem : RAM @ address 0x82030000, size 64 Kbytes.
9705.572:SoC 0:Core 3 : <info> : print_mem : RAM @ address 0x82040000, size 64 Kbytes.
9705.582:SoC 0:Core 3 : <info> : target_reset_and_startup : Debug support enabled.
9705.582:SoC 0:Core 4 : <info> : create_core_object : Creating UCC-MCP Processor Object
9705.582:SoC 0:Core 4 : <info> : on_connect_target : Attempting to reset the target...
9705.583:SoC 0:Core 4 : <info> : set_startup_options : Startup options (0x00423028):
9705.583:SoC 0:Core 4 : <info> : set_startup_options : Firmware: 5.3.0.0 : Mar 19 2014, 00:33:18
9705.583:SoC 0:Core 4 : <info> : set_startup_options : Debug support enabled.
9705.584:SoC 0:Core 4 : <info> : set_startup_options : Do not halt after target reset.
9705.584:SoC 0:Core 4 : <info> : set_startup_options : Duplicate IP check enabled.
9705.584:SoC 0:Core 4 : <info> : set_startup_options : DHCP support enabled.
9705.584:SoC 0:Core 4 : <info> : set_startup_options : JTAG clock frequency = 5MHz
9705.585:SoC 0:Core 4 : <info> : reset_dash_state : Resetting state
9705.585:SoC 0:Core 4 : <warn> : reset_target : Resetting Single MCP only
9705.595:SoC 0:Core 4 : <info> : reset_target : Target Reset OK, doing bypass test to check if target
powered
9705.595:SoC 0:Core 4 : <warn> : target_reset_and_startup : Target Reset
9705.595:SoC 0:Core 4 : <info> : enable_debug_support : enabling debug support...
9705.596:SoC 0:Core 4 : <verbo>: read_id_reg : tcontext_ucc::read_id_reg - 0f010003
9705.596:SoC 0:Core 4 : <info> : enable_debug_support : MCP Core version: Group f, Id 1, Configuration 3
9705.596:SoC 0:Core 4 : <info> : enable_debug_support : UCC system version (assumption made on MCP id) : 310
9705.597:SoC 0:Core 4 : <info> : read_thread_info : UCC Thread PC=0x00000000, state is Stopped
9705.597:SoC 0:Core 4 : <warn> : target_reset_and_startup : Debug support enabled.
9705.699:SoC X:Core 0 : <verbo>: da_read_config : read_config
9705.800:SoC X:Core 1 : <verbo>: da_read_config : read_config
9706.095:SoC X:Core 2 : <verbo>: da_read_config : read_config
9706.097:SoC X:Core 3 : <verbo>: da_read_config : read_config
9706.098:SoC X:Core 4 : <verbo>: da_read_config : read_config
Identifier DA-net 00272
Firmware 5.3.0.0
Mode autodetected
TCK Rate 5000kHz
    
```

There is a wealth of information provide in this initial connection log, first start by checking the IMG JTAG PnP data to see that it matches your expectations and what has been specified in the img2\_jtag\_pack.vhd file for your design. Then for each core in the system look for the line:

```
9703.767:SoC 0:Core 0 : <info> : target_reset_and_startup : Debug support enabled.
```

If 'debug support enable' failed for any cores, the debug adapter will mark those cores as 'off-line' and no debug can be performed. This usually occurs because the core is powered off or it has its clock gated off.

For systems which have the IMG JTAG Control register, the debug adapter will set the 'force availability' bits. These signals should be wired to the SoC's clock/power controller and should be used to power-up and clock the core when asserted. Check your design in this area to ensure this functionality works.

If all your cores appeared OK, let's look at some registers (turn off probe logging for now):

```
>>> logging(probe,0)
probe off
>>> device(core0)
core0
>>> regs()
D0Re0 = b1f13933 D1Re0 = 01075e49 A0StP   = 0c36fc7b A1GbP   = 350066E9
D0Ar6 = a8388110 D1Ar5 = 26195e49 A0FrP   = a0b3b146 A1LbP   = 74181406
D0Ar4 = f86bd111 D1Ar3 = 81b2e7eb A0.2     = 85158cd9 A1.2     = 11623C52
D0Ar2 = c402e947 D1Ar1 = 95641aa3 A0.3     = 0b07b55c A1.3     = 5000BA31
D0FrT = 94481805 D1RtP = 5ab39050
D0.5   = 6a4e4175 D1.5   = e74e6143 TXRPT   = 00000000 TXBPOBITS = 00000000
D0.6   = 2981d833 D1.6   = 493c866f TXTIMER  = 4d011de8 TXTIMER1  = 4D011FDD
D0.7   = e66a5b01 D1.7   = 1de09c64
                                TXENABLE = 02010032
PC     = 00000000 PCX    = 00000000 TXSTATUS = 00000000 znoc
>>>
```

Now try and read some memory, we will use address 0x80000000 as this is core memory which most Meta cores have.

If your core does not have that address, then use an address with some RAM. If your core only has DDR and this requires setting up then skip the memory tests.

```
>>> word(0x80000000,count=256)
0x80000000 f6c6a88e 7a986886 4c068374 42de9471 .....h.zt..Lq..B
          <snip>
0x800003f0 3e6fc499 86bffb01 0414116b 29f4f8e1 ..o>....k.....)
```

Try a write:

```
>>> word(0x80000000,temp,count=256)
```

Common failures are caused by 'time out on debug port ready bit'. This means the debug port is no longer responding to memory requests, usually this is because the external bus has locked up for some reason preventing accesses from the debugger (and/or the core) completing.

If these basic tests are working move on to stage 5.

## 5.6. Stage 5 – Auto-detect with Codescape

Before connecting to your target with Codescape, you need an HSP (Hardware Support Package) for your board. This is an XML file describing the memory and register layout of your SoC/cores. Codescape provides default HSPs for all MIPS and META cores, but be aware that the default memory layout in these files allows accesses to the full address space of the cores.

Codescape can be intrusive in its caching of memory to optimize debug performance for the user. For example, if you stop or reset your target and the stack pointer is pointing at memory that is not configured, Codescape is likely to read a few KB before the stack pointer (i.e. from 0xFFFFxxxx) and

a few KB afterwards. On badly designed SoCs, memory reads to invalid addresses can often lock the external bus and lock up the core preventing debug requests from working. So the simple act of opening Codescape without the correct HSP can cause it to lock up, yet the core may exhibit correct behaviour in Codescape Console. This will only access memory locations that are explicitly requested.

For more information about working with HSPs see 'Board and Core Definition files on page 22'.

Once you have a good HSP, start Codescape and connect to your debug adapter. The first thing to do is check the correct HSP has loaded.

Go to Help > Diagnostics... then select 'Target > Scan Target Log'. Check that the filename and path to the XML file is as expected.

*Note: You can also check the tooltip for the target shown on the Target Pane.*

If everything is well you should be able to view memory and registers, load programs and do all the usual debug operations such as stepping/breakpoints/watchpoints etc.

Something to be aware of is that by default Codescape uses software breakpoints for single-stepping. This requires re-writing the program's instruction space, and so will not work on read-only memory like ROM or flash. To step code in read-only memory you need to switch to using 'Hardware Single Step'. This can be set in Debug > Target Debug Options.

If Codescape and the debug adapter are having problems communicating with your core, your core may be marked as offline. You may get a 'Disconnected – attempting to reconnect in Xs' message. This usually means that the debug adapter cannot talk to the target (not that Codescape cannot talk to the debug adapter).

To try and diagnose the problem, first look at the Communication Log between Codescape and the debug adapter. This can be found in Help > Diagnostics > Codescape Debugger > Comms Log.

Any errors in the Comms Log will be highlighted in red. Try to find the first failure and see what the command was (or the previous command).

If it was a memory access check that the address range is valid for your target. Try using Codescape Console to test accesses in that range in isolation and turn on debug features described in Stage 4 to help diagnose why the access is failing.

Also, while in the 'Target Diagnostics' dialog look at the 'DA Info Log' under 'Target' for clues to the problem. Generally you want to try to isolate the command or access that failed and then try to re-create it in Codescape Console.



## 6. Low-level EJTAG Debug

### 6.1. Introduction

This section is intended to help you take advantage of EJTAG debug capabilities while investigating problems encountered developing systems based on MIPS Technologies' cores. Much of the section will deal with low level EJTAG debug capabilities and the complexity of debugging multi-threaded, multi-core coherent processing system. Most of the EJTAG debug features covered also apply to more basic systems.

For Warrior cores, such as the I6400 that use the OCI debug system and a dedicated Debug Unit (DBU) refer to Section 7 OCI debugging with a DBU Debug Monitor.

#### 6.1.1. Terminology

An effort has been made to use terminology consistent with other MIPS documentation. Below is a brief list of terms that will be used throughout this application note.

##### **MT-ASE (Multithreading Application Specific Extension)**

MIPS ISA (Instruction set Architecture) specification for multi-threading.

##### **TC (Thread Context)**

Hardware resource to support non-privileged thread of execution. A simplified view of a TC is that it is a set of GPRs (General Purpose Registers) and a PC (Program Counter).

##### **VPE (Virtual Processing Element)**

State beyond a TC required for privileged execution. A simplified view of a VPE is that it is all of the state beyond a TC which an operating system expects of a MIPS processor.

##### **CPU (Central Processing Unit)**

What appears to software as an independent processor. This is a VPE with one or more TCs bound to it on a core implementing the MT-ASE.

##### **I\$, D\$, and L2\$**

Primary instruction and data caches and the unified level 2 cache

##### **CPS (Coherent Processing System)**

A cluster of cores and global logic capable of cache coherent operation.

##### **CM (Coherence Manager)**

Logic for maintaining cache coherence between cores in a coherent domain.

##### **CPC (Cluster Power Controller)**

Logic responsible for power-up, power-down, reset, and clock-off sequencing of individual cores in a CPS.

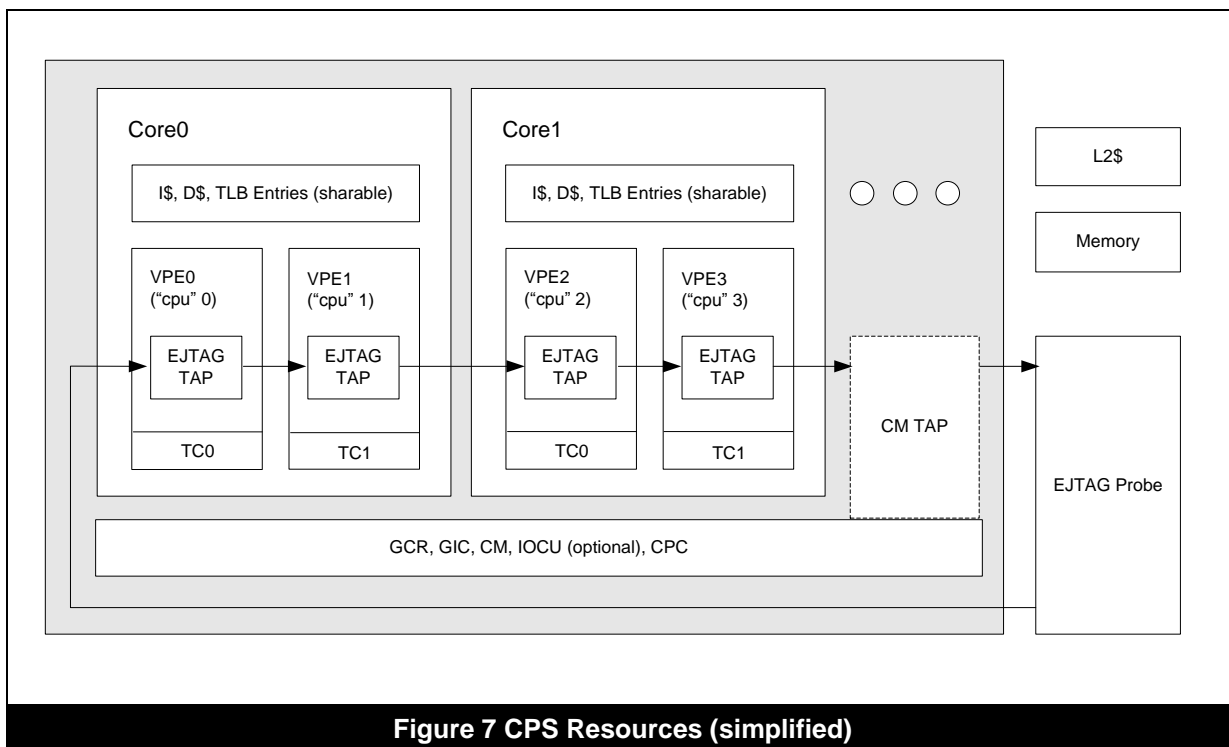
##### **GIC (Global Interrupt Controller)**

Interrupt routing block for global interrupts, core local interrupts, and yield qualifiers in a CPS implementing the MT-ASE.

##### **Core**

Processing element associated with a main execution pipeline.





**Figure 7 CPS Resources (simplified)**

The following terms will also be used as they relate to a debugger view of a target system.

- Device: Synonymous with “CPU” as described above.
- Halted: A device which, under debugger control, is not executing non-debug mode instructions.
- Running: A device which the debugger is not preventing from executing non-debug mode instructions.

*Note: The term “device” is not used to refer to a CM with a TAP in this note*

### 6.1.2. Tools

These notes were developed using the following hardware and software tools:

- DA-net with firmware 5.2.3.0
- Codescape Console 1.0

Information in this section and the accompanying files may require modification when used with other processors, boards, or tools. Device and tool behaviour may also change as new versions add or enhance features.

These instructions assume you are familiar with the basic operation of the listed tools, and that you have a functional development environment where you are able to build executables and use Codescape Console to control your target system. For additional information, refer to the documentation for each tool.

## 6.2. MIPS Processor Basics

Before describing EJTAG debug features it may be useful to review some MIPS ISA basics. (If you are familiar with MIPS execution modes, ISA modes, and exception program counters then you may want to skip this section.)

### 6.2.1. Execution Mode

There are four basic execution modes each with its own level of privilege. Highest privilege to lowest they are: debug-mode, kernel-mode, supervisor-mode, and user-mode. Kernel mode execution has a few “levels” which share the same privilege but cause significant changes to basic processor

operation. The following can be used to help you determine in which mode you are executing. (It can also help you understand what other modes you may have been in recently.)

#### **Debug mode**

CP0 Debug.DM == 1. This is the only mode able to access dseg (drseg and dmseg).

#### **Kernel mode Error level**

CP0 Status.ERL == 1 AND not in any mode above.

#### **Kernel mode Exception level**

CP0 Status.EXL == 1 AND not in any mode above.

#### **Kernel mode**

CP0 Status.KSU == 0 AND not any mode above.

#### **Supervisor mode (optional)**

CP0 Status.KSU == 1 AND not in any mode above.

#### **User mode**

Not any mode above.

### **6.2.2. ISA Mode**

If a processor implements the microMIPS or MIPS16e ISA modes then an ISA mode bit indicates that the alternate ISA (microMIPS or MIPS16e) is in effect.

### **6.2.3. Execution Location**

The MIPS32 and MIPS64 ISAs do not have an explicit “PC” (program counter) register<sup>2 3</sup>. Instead, there are several CP0 (coprocessor 0) registers which act as exception/restart location pointers. The ISA mode bit, on processors which implement the microMIPS or MIPS16e ISA modes, is accessible via the LSB of these location counters.

CP0 DEPC is set, on taking debug exception, to the address of the instruction killed by the debug exception. Executing “deret” (debug exception return) will lead to resumed non-debug-mode execution at the address contained in DEPC. When you halt a device using EJTAG debug tools the reported “PC” is CP0 DEPC (with the ISA mode bit stripped.)<sup>4</sup>

CP0 ErrorEPC is set on taking an error level exception (reset or NMI) to the virtual address of the instruction killed by the error level exception. ErrorEPC holds the address at which to resume non-error-level execution upon executing “eret” (exception return.) ErrorEPC is mainly used for error reporting. Boot code usually sets ErrorEPC to the location it wants to start non-error-level execution at and then executes “eret”.

CP0 EPC is set on taking a non-error level exception to the virtual address of the instruction killed by the non-errorlevel exception. EPC holds the address at which non-exception-level execution will be resumed upon executing “eret”.

CP0 TCRestart is set on halting a TC<sup>5 6</sup> to where execution will resume when the TC is issuable.

### **6.2.4. Exception Cause**

On taking a non-debug mode exception, a code indicating the cause of exception is stored in cp0 Cause.ExcCode.

On taking an exception while executing in debug mode, a code indicating the cause of exception is stored in cp0 Debug.DExcCode.

---

<sup>2</sup>The microMIPS ISA includes a software visible “PC” accessed via the ADDIUPC instruction.

<sup>3</sup>branch-and-link based instructions sequences can allow you to extract execution location if desired.

<sup>4</sup>Codescape Console writes of “PC” do affect the ISA mode.

<sup>5</sup>In this context “halted” is defined as cp0 TCHalt.Halt = 1 (not the “halted” debugger state.)

<sup>6</sup>TCRestart of non-halted TC is specified as UNSTABLE but likely contains a fetch location.

### 6.3. Using NMI (Non Maskable Interrupt)

Although not an EJTAG debug capability, it is worth mentioning that an NMI (Non-Maskable Interrupt) may provide a quick and simple way to investigate some problems. Assertion of a core's NMI input (SI\_NMI) will cause an error level exception. The ErrorEPC register will be loaded with a the exception address. It points to the instruction which was killed due to taking the NMI exception.

Very little core or system state (cache contents, memory controller setup, etc) is inherently disrupted on taking an NMI exception. However, it should be noted that a few bits of the cp0 Status register are modified (BEV=1/TS=0/SR=0/NMI=1/ERL=1), the last of which causes kuseg to become direct mapped. If your target provides a way to assert an NMI and your boot code can recognize an NMI and dump useful target state then this may be sufficient to debug some types of problems.

At present the DA-net does not provide a mechanism to assert an NMI exception.

### 6.4. EJTAG Debug Features

An EJTAG debug block is present in all MIPS cores. It contains support for things like hardware and software breakpoints, hardware single-step, and a JTAG based debug TAP for debug probe connection. Although EJTAG debug resources are often controlled via high level debugger commands, a quick overview of some underlying features is in order.

#### PCSAMPLE

A feature allowing for non-intrusive reading of recently completed instruction addresses. The PCSAMPLE TAP instruction selects the TAP data register "PCSAMPLE" which contains an execution address and a flag indicating whether or not a new instruction has completed since the last read of the PCSAMPLE TAP data register <sup>7 8</sup>.

#### EJTAG TAP

The optional JTAG TAP associated with an EJTAG debug block used for communications with an EJTAG probe and debugger.

#### ECR (EJTAG Control Register)

Shorthand for the EJTAG TAP data register CONTROL.

#### DINT (Debug Interrupt)

An interrupt which causes a debug exception and entry into debug mode <sup>9</sup>.

#### DRSEG (Debug Register Segment)

A memory overlay, present only while executing in debug mode, that allows access to registers controlling various EJTAG debug features.

#### DMSEG (Debug Memory Segment)

A memory overlay, present only while in debug mode and ECR.ProbEn is set, that an EJTAG probe emulates by satisfying processor accesses (fetches, loads, and stores.) The emulation is carried out via TAP data registers CONTROL, ADDRESS, and DATA.

#### Single-Step

A debug setting which will result in a debug exception after execution of a single <sup>10</sup> non-debug mode instruction has completed.

#### Hardware Breakpoint

A hardware resource capable of detecting execution or data access at virtual addresses.

<sup>7</sup>Other fields may be present if certain features are present and enabled. (MT-ASE adds TC field, and EVA adds K field.)

<sup>8</sup>The PCSAMPLE feature has been enhanced to allow sampling of the load/store addresses as well.

<sup>9</sup>There are 2 sources of DINT: the core interface signal SI\_DINT, and the ECR.EjtagBrk bit.

<sup>10</sup>Branches and their delay slots, if executed, execute atomically and result in 2 instructions completing in a single-step.

## Software Breakpoint

The instruction “sdbbp” which causes a debug exception on execution <sup>11</sup>. Debuggers will temporarily replace an instruction of your program with this instruction on setting a breakpoint in writeable memory.

### 6.4.1. EJTAG TAP Basics

Every TAP register access (also referred to as a “scan”) is a read-before-write operation. A TAP register access captures (reads) a register value from the target and then that value is serially shifted out to the tool as a new value is simultaneously shifted in. After all of the bits of the register have been shifted the input value is updated (written.)

There are two main paths through an EJTAG TAP state machine. One provides access to the single, 5-bit instruction register and the other provides access to the currently selected data register(s) <sup>12 13</sup>.

Every TAP instruction access should result in the 5 bit binary value “00001” being read. Most EJTAG TAP instructions’ sole purpose is to select which data register is accessed during a data scan. EJTAG TAP instructions not intended to select specific TAP data registers will select the BYPASS data register.

In a multi-device target system, the term “scan chain” is used to describe the serial (daisy-chained) set of TAPS which are read/written in a single scan.

#### ECR (EJTAG Control Register)

The ECR <sup>14</sup> is the primary control and status register for EJTAG TAP interaction. At a minimum, you should be familiar with the following fields:

ECR.Rocc(bit 31)	Indicates that a reset has occurred. (Only write as “0” when acknowledging a reset.)
ECR.PrAcc(bit 18)	Indicates that a processor access to dmseg is pending. (Only write as “0” when completing a dmseg access.)
ECR.ProbEn(bit 15)	Set to enable dmseg overlay while in debug mode.
ECR.ProbTrap(bit 14)	Set to relocate the debug exception vector into dmseg at 0xff200200
ECR.DebugM(bit 3)	Indicates that a device is executing in debug mode.

### 6.4.2. Boot Mode: EJTAGBOOT vs NORMALBOOT

The EJTAGBOOT TAP instruction modifies the reset value of the ECR.ProbTrap, ECR.ProbEn, and ECR.EjtagBrk thereby changing device reset behavior. Subsequent soft resets will result in a debug exception after release from reset <sup>15</sup>. Any EJTAG TAP reset will clear the EJTAGBOOT indication as will sending a NORMALBOOT TAP instruction <sup>16 17</sup>.

A temporary EJTAG specification “hole” allowed taking a debug exception on an EJTAGBOOT indicated reset in place of a reset exception. (Cores designed to that specification may suffer the side effect of ErrorEPC not getting set on an EJTAGBOOT indicated reset.)

### 6.4.3. Basic Codescape Console Commands

This is a very quick tour of some basic Codescape Console debugger commands. They represent the minimum set of commands you should be familiar with to start making effective use of the debugger.

<sup>11</sup>An “sdbbp” instruction never completes as it always takes a debug exception.

<sup>12</sup>There are several data registers of varying sizes.

<sup>13</sup>The TAP instructions ALL and FASTDATA select multiple TAP data registers.

<sup>14</sup>ECR (EJTAG Control Register) is a shorthand for the EJTAG TAP data register named “CONTROL”.

<sup>15</sup>EJTAGBOOT and NORMALBOOT TAP instructions do not cause a reset, they modify/restore device reset behavior.

<sup>16</sup>Often the TAP reset signal is not brought out of a device and is driven internally by power-on-reset circuitry.

<sup>17</sup>A warm target reset should not cause a TAP reset but a power-on-reset may.

Full API documentation can be found in your Codescape Console installation under \\documentation\index.html.

```
>>> probe("DA-net 401")
Identifier DA-net 00401
Firmware 5.2.2.0
Mode autodetected
TCK Rate 20000kHz
>>> reset(normalboot) # reset target with boot mode set to NORMALBOOT.
>>> runstate()
status=running
>>> reset(ejtagboot) # reset target with boot mode set to EJTAGBOOT.
>>> runstate()
status=stopped pc=0xbfc00000
>>> dasm()
0xbfc00000 00431023 subu v0, v0, v1
0xbfc00004 0050102B sltu v0, v0, s0
0xbfc00008 1440FFFC bnez v0, 0x87fe7e58
0xbfc0000c 8FBF001C DS lw ra, 28(sp)
```

Get help on any function using the **help()** command, or get a brief help by entering the name of the command:

```
>>> probe
probe(identifier=None, ip address='', force disconnect=False, advanced options={})
    Connect to a probe, or displays information about the current probe.
>>> help(probe)
probe(identifier=None, ip_address='', force_disconnect=False, advanced_options={})

Connect to a probe, or displays information about the current probe.

The identifier should be of the following form

=====
DA Type          Identifier Format
=====
DA-net           "DA-net 1"
Local Simulator  "Simulator HTP221"
Remote Imperas   "RemoteImperas hostname:port"
Remote Simulator "RemoteSimulator hostname:port"
=====

If force disconnect is True, and the probe is currently in use, then the
other user will be forcibly disconnected. This should be used with
consideration for others.

If the probe cannot be located using DNS, or UDP broadcast, then it may
be necessary to specify an IP address using `ip address`.

`advanced options` is a dictionary of options that are passed to the
comms layer. It is not normally necessary to use these.
```

Read and write registers with the `regs()` command.

```
>>> regs()
zero 00000000 at      00000002 v0      8010c3e8 v1      00000000
a0   8010c418 a1      00000000 a2      00000000 a3      8011cbc8
t0   0000000a t1      676e6c62 t2      00000000 t3      ffffffff
t4   80080000 t5      00000004 t6      80008000 t7      00000001
s0   8002f3dc s1      80030000 s2      8002b7d0 s3      80030000
s4   00000000 s5      00000000 s6      ffffffff s7      00000000
t8   00000000 t9      00000000 k0      80021d58 k1      deadbeef
gp   801141c8 sp      8011dbc8 s8      87fffe48 ra      8010c288

hi   00002000 lo      00000000 depc    80100e1c pc      80100e1c
status 11000000 cause 00000000 epc     a0001000 badvaddr a001fd09
index 80000000 random 00000012 entrylo0 00000000 entrylo1 00000000
context 007ffff0 pagemask 00000000 wired  00000000 count  00083bed
entryhi 8007c000 compare 00000000 prid   0001974c errorepc ff20020c
config 80208483 config1 bee3519e config2 80001000 config3 00002c20
lladdr ffffffff watchlo 00000000 watchhi 80000000 debug  40118008
taglo  00000000 datalo  afb60070 pagegrain ffffffff tracecontrol ffffffff
>>> regs('pc')
0x80100e1c
>>> regs('pc', 0x80100e20) # Modify the "pc" (DEPC for this debugger.)
0x80100e20
```

Disassemble target memory or arbitrary byte sequences with `dasm()` and `dasm_bytes()`:

```
>>> dasm('pc')
0x80100e1c AC850008 sw      a1, 8(a0)
0x80100e20 24840010 addiu  a0, a0, 16
0x80100e24 1487FFFF bne    a0, a3, 0x80100e14
0x80100e28 AC85FFFF DS sw    a1, -4(a0)
>>> dasmbytes(0x80100e1c, '\xac\x85\x00\x08')
0x80100e1c AC850008 sw      a1, 8(a0)
```

Read and write a word of memory with the `word()` command (see also `byte`, `halfword`, `dump`):

```
>>> word('pc')
0x41606020
>>> word(0x80000000, count=4)
0x80000000 3c1b8033 401a4000 8f7b0000 001ad582 <...3@.e..{.....
>>> sum(word(0x80000000, count=4))
0x10BCB95B5
>>> word(0x80000000, count=4)[2]
0x8f7b0000
```

Control target execution state with `go()`, `halt()`, and `step()`, determine current execution status with `runstate()`:

```
>>> step()
status=single stepped pc=0x87fe7e64
0x87fe7e64 1440FFFC bnez    v0, 0x1014222355120
>>> go()
Running from 0x80062c14
>>> halt()
status=stopped pc=0x87fe7e5c
0x87fe7e5c 00431023 subu   v0, v0, v1
>>> runstate()
status=stopped pc=0x87fe7e5c
0x87fe7e5c 00431023 subu   v0, v0, v1
```

Software and hardware code breakpoints can be created, removed, enabled, and disabled with the `bkpt()` command:

```
>>> bkpt(sethw, regs('pc') + 8)
=====
Address      Enabled Type Data      HW Index
=====
0x87fe7e64 Enabled hw    0x00000000 0
=====
>>> go()
Running from 0x87fe7e5c
>>> runstate()
status=hw_break pc=0x87fe7e64
0x87fe7e64 1487FFFB bne a0, a3, 0x80100e14
```

### 6.4.4. Advanced Codescape Console Commands

After becoming familiar with the basic Codescape Console debugger commands it is a good idea to become familiar with the flexibility of the Python scripting language, and using some of the more advanced commands. These commands are included in Codescape Console, as with all commands help can be obtained with:

```
>>> help(tapaddress)
tapaddress(value, device=None)

Write and read the eJTAG address register.
```

### 6.4.5. Low level "scan" commands

There are many functions that perform JTAG scans to improve low level control and visibility. They perform many of the same functions that the debug adapter would normally perform automatically, but by implementing them in Python, steps can be performed slowly and more logging added to each intermediate stage.

*Note: To use these commands it is important to stop any asynchronous polling of target state and put the probe in scanonly mode. This allows undisturbed low level scan chain interaction with the target.*

These commands are built on top of Codescape Console's low level TAP instruction and data scan commands `tapi()` and `tapd()`.

The debug adapter can operate in one of three modes, *scanonly*, *autodetect*, and *table*.

Mode	Description
uncommitted	The debug adapter has just been reset and has not yet been assigned an operating mode.
autodetected	The debug adapter has auto detected the targets.
table	The debug adapter had a table loaded and did not auto detect its targets.
scanonly	The debug adapter is operating in scanonly mode, only JTAG commands are allowed.

The debug adapter initially goes into an *uncommitted* state when reset:

```
>>> reset(probe)
Identifier DA-net 00401
Firmware 5.3.0.0
Mode uncommitted
TCK Rate 20000kHz
```

The mode is then determined by the first command called after a reset(probe) call.

First Command	Mode
tap*/tapi/tapd/devtapi/devtapd, jtagchain, tcktest	scanonly
targetdata (when given a suitable table)	table
autodetect/regs/dump/word/dasm/asm or anything else requiring knowledge of the target layout	classic

*Note: Once initialised, the autodetect and table modes are the same, the only difference is the method used to discover the connected target. If not specified the rest of this document will refer to autodetect mode to mean either autodetect or table.*

Certain commands do not affect the operating mode, for example:

- logging
- config

A subsequent tapi or tapd command (or a command that uses these) will then put the probe into *scanonly* mode. Once in *scanonly* mode, certain commands will fail, for example:

```
>>> reset(probe)
Identifier DA-net 00401
Firmware 5.3.0.0
Mode uncommitted
TCK Rate 20000kHz
>>> jtagchain() # select scanonly, by performing a jtagchain immediately after reset(probe)
Bypass test found 1 tap
Determine IR lengths on scan chain and validating number of taps...
[5]
>>> probe()
Identifier DA-net 00401
Firmware 5.3.0.0
Mode scanonly
TCK Rate 20000kHz
>>> regs()
RuntimeError: read_console_config : CON: Command not available : command not allowed yet
```

The scan based commands, such as `jtagchain()`, `pcsamp()` can be run in autodetect or scanonly modes. When in autodetect or table mode the debug adapter will automatically stop any polling of the target whenever it receives a scan command. Polling can be restored by using the `autodetect()` followed by `runstate()`, and will only restore polling when `runstate()` is called. The debug adapter never performs polling when in scanonly mode.



In either mode, the scan commands all require knowledge of the TAP layout, this can either be set explicitly:

```
>>> reset(probe)
Identifier DA-net 00401
Firmware 5.3.0.0
Mode uncommitted
TCK Rate 20000kHz
>>> autodetect()
Identifier DA-net 00401
Firmware 5.3.0.0
Mode autodetected
TCK Rate 20000kHz
>>> pcsamp()
RuntimeError: Please use switch_target, configure_tap, or jtagchain before pcsamp.

>>> configuretap(0, [5])
>>> pcsamp()
0x33fe400420
PC New
ff200210 0
```

Or it can be determined automatically (this is the recommended approach):

```
>>> reset(probe)
Identifier DA-net 00401
Firmware 5.3.0.0
Mode uncommitted
TCK Rate 20000kHz
>>> autodetect()
Identifier DA-net 00401
Firmware 5.3.0.0
Mode autodetected
TCK Rate 20000kHz
>>> pcsamp()
RuntimeError: Please use switch_target, configure_tap, or jtagchain before pcsamp.

>>> jtagchain()
>>> pcsamp()
0x33fe400420
PC New
ff200210 0
```

In either case this changes only Codescape Console's TAP layout it does not change the debug adapter's view of the TAP layout.

The scan commands do not understand the c0v0 or core0 naming of devices, they require the TAP index to be specified

*Note: In future versions of Codescape Console the TAP layout will be able to automatically determine the TAP layout when the debug adapter is in autodetect mode, and the c0v0 names will be able to be used to specify the device but this has not yet been implemented.*

```
>>> reset(probe)
Identifier DA-net 00278
Firmware 5.3.0.0
Mode uncommitted
TCK Rate 20000kHz
>>> autodetect()
Identifier DA-net 00278
Firmware 5.3.0.0
Mode autodetected
TCK Rate 20000kHz
>>> listdevices()
[core0, core1]

>>> pcsamp(core1)
RuntimeError: Tap selection by core0 or c0v0 name is not yet supported.
Please use switch target, configure tap, or jtagchain before pcsamp.

>>> jtagchain()
>>> configuretap(1)
>>> pcsamp()
PC New
ff200210 0
```

The scan commands can be used to perform very low-level JTAG interaction, for example, sending the TAP instruction "CONTROL" (0x0a) to this TAP. In the return data line we see the correct 5-bit constant (00001) from the instruction register.

```
>>> tapi("5 0x0a")
[0x01]
```

We can now scan the selected TAP data register: a 32-bit ECR, the result shows the value which was read out BEFORE the value supplied is written.

```
>>> tapd("32 0x8004c000")
[0x4004c008]
```

The commands `configuretap()`, `devtapi()`, and `devtapd()` simplify interaction with a single device.

```
>>> configuretap(0) # OR switch_target('1000000')
>>> devtapi(5, 0x0a)
0x01
>>> devtapd(32, 0x8004c000)
0x0000c000
>>> devtap(0x0a, 32, 0x8004c000) # perform an ir scan and a dr scan in one command
0x0000c000
```

dmseg is a complex command that services dmseg requests. It uses devtapi and devtapi to supply/accept memory accesses using a supplied set of address and corresponding instruction/data values. This allows you to provide a simple custom exception handler code from the probe.

```
>>> configuretap(0)
>>> tapecr(0x8004d000) # Set ECR.EjtagBrk to signal a dint
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 2 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0
1 0
>>> hex(tapecr())
'0x4004c008L'
>>> dmseg(EnterDebug)
read of 0xff200200 00000000 nop
read of 0xff200204 0000000F sync
read of 0xff200208 1000FFFD b 0xff200200
read of 0xff20020c 00000000 nop
Second access seen to debug exception vector <done>.
```

The above functionality is also available in the `imgtec.console.enterdebug` command. For more information see `dmseg()` in the Codescape Console online help.

### 6.4.6. Diagnosing Cache Problems

First we need to get the debug adapter into auto-detect mode:

```
>>> autodetect()
Identifier DA-net 00401
Firmware 5.3.0.0
Mode autodetected
TCK Rate 20000kHz
```

The `cachedump()` command takes parameters for the cache type, and starting and ending byte index into the data array<sup>19</sup>. It will display all tag and line data for all ways in the specified range<sup>20</sup>.

```
>>> cachedump(instr, 0, 0)
Offset Set Way TagHi TagLo Word 0 Word 1 Word 2 Word 3 Word 4 Word 5 Word
6 Word 7
0000 000 0 000006a0 00010080 04110002 00000000 8002b900 03e0e021 8fe90000 0120e021
3c08bd00 35087000 .....!.....!<...5.p.
0000 000 1 00000000 07fec080 afb3004c afb20048 afb10044 afb00040 afbc0010 8f92000c
8f93000c 8f990084 ...L...H...D...@.....
0000 000 2 00000000 00000000 02402021 2442000c 0040f809 34059884 8e030000 8e02001c
8e050054 3c06000f .@ !$B...@..4.....T<...
0000 000 3 00000000 00000000 02002021 02a0f809 241e0001 24020001 1642ff64 8fbf003c
3c038720 2463000c .. !....$...$....B.d...<<.. $c..
```

The `invalidate()` command allows you to invalidate cache entries:

```
>>> invalidate(instr)
```

<sup>19</sup>The `cachedump` procedure will also accept `kseg0` or `kseg1` virtual addresses and strip them to data array byte offsets.

<sup>20</sup>The tags are partially decoded and used to highlight cacheline data based on line state invalid, valid, dirty, and locked.

## 6.5. Debugging a Soft Hang

The term “soft hang” is used to denote situations where a processor is executing instructions correctly but for some reason the target system is not making forward progress as intended. Infinite loops, waiting on events that never happen, and sleeping without a wake condition are examples of software hangs.

### 6.5.1. Using PC Sample

The (optional) PCSAMPLE feature was added to the EJTAG specification starting with version 3.00. If present and enabled it allows non-intrusive reading of a recently completed instruction address and notes whether this instruction had completed after the last PCSAMPLE read. Although the PCSAMPLE feature is not enabled in hardware out of reset<sup>21</sup>; the DA-net probe will enable PCSAMPLE on connection. Once enabled, it will remain enabled until manually disabled or the core is reset<sup>22 23 24</sup>.

```
>>> go()
Running from 0x87fe7e5c
status=running
>>> pcsamp()
PC      New
87fee4e0 1
>>> pcsamp()
PC      New
87fee4e4 1
>>> pcsamp()
PC      New
87fee4e4 0
```

`pcsamp()` can be very helpful in detecting long stalls without disrupting target state<sup>25</sup>. If the PCSAMPLE New bit is seen cleared it will indicate that no instructions have completed since the last read of the PCSAMPLE TAP data register.

This is different than reading the same pc twice in a row which can be seen in the two instruction loop below.

```
>>> asm('pc', "b 0x%08x" % regs('pc')) # code a branch-self-nop loop into memory.
0x87fe7e58 1000ffff      b      0x87fe7e58
>>> asm(None, 'nop')
0x87fe7e5c 00000000      nop
>>> go()
Running from 0x87fe7e58
status=running
>>> jtagchain()
Bypass test found 1 tap
Determine IR lengths on scan chain and validating number of taps...
[5]
>>> pcsamp()
PC      New
87fe7e58 1
>>> pcsamp()
PC      New
87fe7e58 1
```

PCSAMPLE, once enabled, remains enabled even while while in debug mode. For a halted device, recently completed instructions will likely have been executed from dmseg.

<sup>21</sup>Reading the PCSAMPLE register before the feature is enabled will return unpredictable results.

<sup>22</sup>The PCSAMPLE feature is not enabled after a NORMALBOOT indicated reset.

<sup>23</sup>Some implementations of PCSAMPLE include the possible capture of load/store addresses.

<sup>24</sup>Check core errata if you see unexpected behavior relating to PCSAMPLE.

<sup>25</sup>The PCSAMPLE feature does not rely on debug mode execution of instructions on the core.

```
>>> halt()
status=stopped pc=0x87fe7e58
0x87fe7e58 1000ffff    b        0x87fe7e58
>>> pcsamp()
PC        New
ff200218  1
```

Note that early implementations of the PC Sample feature captured the address of the instruction to complete AFTER the sample rate counter expired. This, combined with the maximum PC sample rate of once every 32 cycles hindered use of this feature for debugging hangs where instructions were not completing in the core pipeline. (In this case you are only able to read out the address of an instruction that completed within 32 cycles of the last instruction completed.) This behavior has been enhanced but still may not return the last instruction to complete if the core clock has been stopped, preventing availability of the “most current” PCSample.

### 6.5.2. Using DINT (Debug Interrupt)

Debug of software hangs with an EJTAG probe and compatible debugger is less intrusive than NMI (not supported in Codescape Console). CP0 Status information is not clobbered and you have free reign to view/modify system state and control execution including resuming execution at the point it was interrupted or any other location. Use the Codescape Console’s “halt” command to cause a DINT and suspend execution of your target software. (The CPU will take a debug exception and continue executing debug mode instructions supplied by the probe to accomplish any subsequent debugger commands.) At this point you can inspect the system state. If the problem is simple you may be able to use the debugger to modify system state to get past the hang condition without rebuilding and reloading your code.

A MIPS device has two sources of DINT. One is the ECR.EjtagBrk bit. Debuggers which support multi-core debug will likely use the ECR.EjtagBrk to manually halt individual devices.

```
>>> halt()
status=stopped pc=0x87fe7e5c
0x87fe7e5c 00431023    subu    v0, v0, v1
```

The other is the core interface signal SI\_DINT which is often driven by a cross-trigger matrix or debug group logic which monitors the EJTAG probe interface signal DINT as well as whether the device is in debug mode allowing automatic halting of a group of devices at the same time.

DA-net does not support asserting SI\_DINT.

## 6.6. Debugging a Hard Hang

Some system failures may result in the core no longer executing instructions. This is often the result of an infinite pipeline stall due to a required handshake not completing<sup>26</sup>. One fairly common cause of infinite stalls seen in support cases is an incomplete bus transaction in the system bus logic.

Incomplete bus transactions often lead to a stall on execution of a sync instruction, exhaustion of a buffering resource, waiting for load data which is needed but has not been returned by the system, or some other load/store completion barrier.

### 6.6.1. “Halt” Fails

Hard hangs are often accompanied by an inability of EJTAG debug tools to “halt” the processor and inspect target state. This is because most EJTAG debug capabilities rely on the target system being able to take a debug exception and correctly fetch and execute debug mode instructions supplied by

<sup>26</sup>Use the Codescape Console command **pcsamp()** to detect long stall conditions.

an EJTAG probe or a target debug monitor. If this basic level of functionality is unstable then debug operations will likely also be unstable. Often the easiest way to recover from this type of situation, if the system is unable to detect and recover on its own, is via a system reset.

Before resetting the target, however, see if you can determine the point at which the EJTAG tools stop making forward progress while attempting to halt the core as this can provide valuable clues as to the nature of the hang.

*Note: You may need to turn on logging using `logging()` or use the the low level scan commands to get this very low level target and tool state.*

### 6.6.2. Halt Fails: CPU not taking DINT

A good indication that a device is not taking the debug interrupt asserted by “halt” is shown in the ECR. If you see ECR.EjtagBrk set but ECR.DM cleared then there appears to be a DINT source active but the device is not in debug mode.

```
>>> halt()
status=running
>>> tapecr()
0x0000d000
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0
0 0
```

There are a limited set of conditions which can prevent a core from taking a debug exception.

#### Core does not have power and clock

There is synchronization logic between the EJTAG TAP TCK clock domain and a core's clock domain. If the core is not being clocked then information will not propagate between the clock domains including setting ECR.EjtagBrk.

In a CPS which implements a CPC, the powering and clocking of individual cores is controlled by a combination of static inputs, CPC state, coherent state, CPC commands, and whether or not an EJTAG probe has been detected since the last cold reset (CPC probe-mode.)

Most Malta/CoreFPGA5/6 programming files (bitfiles) are configured such that core0 is powered & clocked out of reset and all other cores are not powered<sup>27</sup>. To simplify debug, the CPC detects the presence of an EJTAG probe and enters “CPC probe-mode” which limits the lowest power state on a warm reset to “ClockOff” thereby maintaining a functional scan chain through all of the EJTAG TAPs.

#### Another VPE on this core is in debug mode

A core implementing the MT-ASE executes single-threaded while in debug mode. If one VPE on a core is in debug mode then another VPE on that same core will be unable to take a debug exception until the first VPE leaves debug mode. The DA-net implements independent VPE run control by not leaving a “halted” VPE in debug mode but instead, offlining all tc bound to that VPE and exiting debug mode to allow continued debug activity on either VPE.

### 6.6.3. Halt Fails: CPU never accesses dmseg

There are a few possible explanations for never seeing an access to dmseg even though a device's ECR.DM indicates that the device has entered debug mode.

#### Debug Exception Vector was not in dmseg

An EJTAG debug probe usually redirects the debug exception into dmseg by maintaining ECR.ProbEn and ECR.ProbTrap set.

<sup>27</sup>Power gating is not actually implemented in the FPGA but isolation logic gives the same net effect and results in a broken scan chain.

```
>>> tapecr()
0x0000c000
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0
```

If a debug exception occurs while ECR.ProbEn or ECR.ProbTrap are cleared then the debug exception vector is in target memory and will be handled by target memory and no access to the dmseg will be seen<sup>28 29</sup>.

Debug interrupts are blocked while in debug mode and do not cause debug mode reentry. If debug mode execution is not resulting in accesses to dmseg then there is little which an EJTAG probe can do to gain control of the target short of a reset.

Example: Use low level scan operations to write the value 0x80049000 to the ECR of the running device. This will request a debug interrupt (ECR.EjtagBrk=1) but not redirect the debug exception vector into dmseg (ECR.ProbTrap=0.)

```
>>> jtagchain()
Bypass test found 1 tap
Determine IR lengths on scan chain and validating number of taps...
[5]
>>> tapecr(0x80049000) # Signal a dint via ECR.EjtagBrk but clear ECR.ProbTrap.
0x0060c000
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0
0 0
```

Remember that EJTAG TAP register accesses are read-before-write. The ECR value 0x0060c000 above was read BEFORE the 0x80049000 was written. Perform the read again:

```
>>> tapecr()
0x40048008
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 2 0 0 0 0 0 0 1 0 0 1 0 0 0 0
1 0
```

At this point we can see that we are in debug mode but that no access to dmseg is pending (ECR.PrAcc != 1.) Attempts to “halt” with an EJTAG probe time out waiting for the target to fetch code from dmseg and a reset may be necessary for the EJTAG tools to regain control of the target system<sup>30</sup>.

```
>>> halt()
status=running
>>> runstate()
status=running
>>> tapecr()
0x0000c008
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
1 0
```

<sup>28</sup>If ECR.ProbEn is set then a target resident debug exception handler could access dmseg.

<sup>29</sup>The debug exception vector could be redirected back into dmseg via DCR.RDVec and DebugVectorAddr registers in drseg.

<sup>30</sup>If you can get the target resident debug exception handler code to leave debug mode then a reset will not be required.

Because the debugger had previously enabled the PCSAMPLE feature we can use `pcsamp()` to determine whether the target is still completing new instructions.

```
>>> pcsamp()
PC      New
0x80045D08 1
>>> pcsamp()
PC      New
0x80045C8C 1
```

In this case the target debug exception handler appears to have never left debug mode. Similar behaviour can be observed on attaching an EJTAG probe to a target which is already in debug mode.

#### **Debug exception acting as a completion barrier**

Some cores may take a debug exception and enter debug mode but not start fetching from the debug exception vector in `dmseg` until outstanding bus transactions are completed. If the target is unable to complete outstanding transactions the core may enter an infinite stall condition in debug mode without having ever accessed `dmseg`.



### 6.6.4. Halt Fails: CPU stops accessing dmseg

At other times a core may take a debug exception and start fetching and executing debug mode instructions provided by the probe via dmseg but hang during the execution of those instructions. If you are able to replicate the conditions just prior to the halt command failing then you can use debugger logging to record debugger/target interaction associated with the “halt” command” to help isolate the cause of the failure.

#### EJTAG Instruction Logging

The debug adapter can be configured to log all dmseg instructions executed. This can be helpful to diagnose difficult bus stalls.

```
>>> reset(ejtagboot)
>>> runstate()
status=stopped pc=0x87fe7e5c
>>> bkpt(sethw, 0x87fe7e60)
=====
Address   Enabled Type Data      HW Index
=====
0x87fe7e60 Enabled hw    0x00000000 0
=====
>>> go()
Running from 0x87fe7e5c
status=stopped pc=0x87fe7e60
0x87fe7e60 0050102b sltu    v0, v0, s0
>>> asm('pc', 'lb $k0, 0($k0)')
0x87fe7e60 835a0000 lb      k0, 0(k0)
>>> bkpt(clear, all)
=====
Address Enabled Type Data HW Index
=====
>>> regs('k0')
0x87fabfb0
>>> regs('k0', 0xa0100000)
0xa0100000
>>> config("Log Debug Instructions", 1)
1
>>> config("Verbose Logging", 1)
1
>>> logging(probe, on)
probe on
>>> go()
Running from 0x87fe7e60
68518.828:SoC X:Core 0 :<verbos>: resume           : resume - threads: 0x1
68518.828:SoC 0:Core 0 :<info>: run                : Run [thread 0]
68518.829:SoC 0:Core 0 :<verbos>: set single step       : [thread 0]
68518.829:SoC 0:Core 0 :<verbos>: read_cp0_register      :
68519.329:SoC 0:Core 0 :<error>: execute             : [thread 0] Timeout waiting for
PrACC
68519.329:SoC 0:Core 0 :<info>: print all ecrs        : TAP 0, ECR = 0x0000c000
68519.330:SoC X:Generic :<except>: dispatch cmd         : dbg::exception Timeout waiting
for PrACC
>>> pcsamp()
PC      New
ff200214 1
>>> pcsamp()
PC      New
ff200214 0
>>> tapecr()
0x0000c000
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0
```

To verify that the target was stalled we used the `pcsamp()` command to verify that the core was not making forward progress, in fact in this case the target has stalled in `dmseg`, but the ECR register shows that the core is not in debug mode (`Dm == 0`).

### Custom Debug Exception Handler

Sometimes the ability to supply a simple custom debug exception handler can be of great help in isolating debugger command failures.

```
>>> configuretap(0) # select c0v0 for devtapi and devtapd commands.
>>> tapecr() # Read ECR
0x4004c008
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 2 0 0 0 0 0 0 1 0 0 1 1 0 0 0
1 0

>>> tapecr(0x8004d000) # Set ECR.EjtagBrk to cause dint
0x4004c008
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 2 0 0 0 0 0 0 1 0 0 1 1 0 0 0
1 0

>>> tapecr() # Read ECR
0x4004c008
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 2 0 0 0 0 0 0 1 0 0 1 1 0 0 0
1 0

>>> dmseg(ReadDEPC)
read of 0xff200204 00000000 nop
read of 0xff200208 4001c000 mfc0 at, c0_depc
read of 0xff20020c 40027801 mfc0 v0, c0_ebase
read of 0xff200210 3c03ff20 lui v1, 0xff20
read of 0xff200214 ac610000 sw at, 0(v1)
write to 0xff200000: data accepted 0x87fe7e5c CP0 DEPC
read of 0xff200218 ac620004 sw v0, 4(v1)
write to 0xff200004: data accepted 0x80000000 CP0 EBase
read of 0xff20021c 34630200 ori v1, v1, 0x200
read of 0xff200220 00600008 jr v1
read of 0xff200224 00000000 nop
Second access seen to debug exception vector <done>.
CP0 DEPC 0x87fe7e5c
CP0 EBase 0x80000000
```

In this case it looks like the simple debug exception handler specified in the `ReadDEPC dmseg()` list runs to completion and shows that the instruction at `0x87fe7e5c` took the debug exception. Also note that this exception handler does not include any sync instructions. Try executing some of the other example debug exception handlers listed in the command `dmseg()`.

## Manually satisfying accesses to dmseg

If even the most simple of custom debug exception handlers do not complete, you can use `tapi/tapd` (or `devtapi/devtapd`) to manually interact with a core on a scan by scan basis.

```
>>> tapecr(0x8004d000)
0x4004c008
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 2 0 0 0 0 0 0 1 0 0 1 1 0 0 0
1 0
>>> tapecr()
0x4004c008
Rocc Psz Resv VPED Doze Halt PerRst PRnW PrAcc Resv PrRst ProbEn ProbTrap IsaOn EjtagBrk Resv
Dm Resv
0 2 0 0 0 0 0 0 1 0 0 1 1 0 0 0
1 0
>>> devtapi(5, 0x8)
0x00000001
>>> devtapd(32, 0xdeadbeef)
0xff200200
```

Note that these last two commands (the selection of the ADDRESS register and the `devtapd` to read/write the address register) could be more simply written using:

```
>>> tapaddress(0xdeadbeef)
0xff200200
```

### 6.6.5. Debug Through Reset

If your device does not implement PDtrace and if it is unable to take a debug exception and execute a debug handler then there is limited state which can be observed in the system. Recovery from this state often requires a reset. On taking a reset exception some processor state is updated but most remains unaffected and may be inspected after a reset to gain insight into the failure which required a reset to recover.

CP0 ErrorEPC will be loaded with the address of the instruction killed by the reset exception.

## 6.7. Multi-Core Coherent Processing Systems (CPS)

Some CPS components have a significant impact on system debug operations and deserve special attention.

### 6.7.1. Cluster Power Controller (CPC)

The CPC manages power-up, clocking, reset, and power-down for individual core power domains. The EJTAG probe interface signal RST\* is routed to the SI\_CPCReset.

### 6.7.2. CPC Reset

There are two types of CPC reset. A CPC cold-reset is caused by assertion of SI\_CPCReset with CM\_PwrOn\_n also asserted. CPC cold reset should only be generated during initial power-up of the CPS or as the result of a system power brown-out. If any core in a CPS is powered-down then power domain isolation will break the scan chain and nearly all EJTAG probe debug capabilities.

### 6.7.3. Probe-Mode

Before any communication with a CPS, an EJTAG probe should indicate its presence to the CPC. This is done by entering CPC probe-mode via five rising edge TCK with TMS=1<sup>31</sup>. CPC probe-mode will cause the CPC to power-up cores which were powered-down and prevent powering down any cores while in effect<sup>32</sup>. This is necessary to maintain a valid scan-chain through all power domains. Probe-mode triggers power-up of all domains and causes any power-down requests to be treated as clock-off requests ensuring a functional scan chain is maintained.

CPC probe-mode is cleared on a CPC cold-reset but survives a CPC warm-reset. For this reason, assertion of the EJTAG probe interface signal RST\* should always generate a CPC warm-reset.<sup>33 34</sup>

### 6.7.4. CPC DINT monitoring

The CPC monitors sources of DINT and will transition unlocked cores with an asserted DINT to a clocked state thereby allowing debug mode execution. After leaving debug mode a core will return to its previously programmed domain state. If debug tools are configured to halt devices out of reset, which would be in power-down or clock-off in the absence of debug tools, then it may be necessary to program the domain power state you would like them to return to on leaving debug mode.<sup>35 36</sup>

---

<sup>31</sup>The Navigator Console command “jtaginit force” preforms a soft TAP reset which also triggers probe mode.

<sup>32</sup>Cores that were already powered will see this as a TAP reset.

<sup>33</sup>If EJTAG probe RST\* assertion resulted in a CPC cold-reset then power domain isolation will break the scan chain.

<sup>34</sup>It is possible for software to disable CPC probe-mode allowing devices to be powered down breaking the scan chain.

<sup>35</sup>Some devices do not have a “previously programmed” power state out of CPC reset.

<sup>36</sup>Many early CoreFPGA5 bitfiles did not correctly propagate DINT from ECR.EjtagBrk into the CPC.

## 7. OCI debugging with a DBU Debug Monitor

### 7.1. Introduction

This chapter describes how to use the debug monitor and Codescape Console to debug a 64-bit multicore system with a Debug Unit (DBU). Warrior cores, such as the I6400, use the OCI debug system and make use of a dedicated Debug Unit (DBU) and debug monitor to pass debug commands and data to and from the core. Refer to the *MIPS OCI Cluster Debug Technical Reference Manual MD01077* for more information about the operation of the DBU and debug monitor.

### 7.2. Debug Unit (DBU)

Unlike previous MIPS multicore systems, OCI compliant devices have one tap per cluster of cores. Each tap connects to a DBU which then uses a register bus (RB) to communicate to the cores and for the cores to read from the DBU. The DBU has its own memory which is mapped to the debug address range dseg, starting at 0xFFFFFFFF200000. This memory is referred to as dmxseg.

### 7.3. Debug Monitor

The debug monitor is loaded into dmxseg prior to entering debug mode. This makes debugging more efficient as we can send commands to the monitor in fewer operations than the traditional method of hand feeding instructions using PrAcc.

Hand feeding instructions is still possible with global throttle turned on. See below.

#### 7.3.1. Global Throttle

The global throttle enable bit is found in the CONTROL JTAG register. When this is enabled, accesses to dmxseg are suspended until acknowledged by the probe clearing PrAcc. Unlike previous MIPS EJTAG systems, instructions are written to a location in dmxseg instead of a JTAG register.

The monitor runs with global throttle disabled which means that code executes normally on entry to debug mode. In certain situations global throttle is used in combination with the monitor to confirm a successful entry to debug mode or to step through monitor code. During normal use you should not need to modify global throttle.

#### 7.3.2. Debug Monitor States

The debug monitor implements a state machine to allow it to run continuously, which is controlled by the flags in the first word of the command buffer. These flags are called 'ready' and 'busy', they indicate the current monitor state:

Ready	Busy	State
0	0	Monitor is idle and waiting for a command.
1	0	Command set up, waiting for the monitor to begin executing it.
1	1	Monitor is executing the command.
0	1	Command has finished, monitor is waiting for the probe to acknowledge it.

#### 7.3.3. Key Components

Although Codescape Console hides most of the implementation details of the monitor, an overview of the key areas is helpful.

##### Debug Data

This structure is filled on entry to debug mode and written back to the real registers when the VP resumes. It contains:

- A 64-bit value 'scratch' used to temporarily store registers whilst saving others.

- The PC as it was when the VP entered debug. If exceptions take place in debug mode, DEPC itself may be modified but this is the value which will be restored.
- The debug entry level which shows how many times debug mode has been entered since the last resume. This will generally be one and will increment after a debug mode re-entry.
- Saved values of r1 to r31, r0 omitted. When monitor commands operate on GP registers they are actually using these values.
- Code scratch space where generated instructions are stored.

*Note: r1 is also saved in DESAVE and one or both of these values are valid depending on the situation. Commands will automatically return the valid copy.*

The debug data can be read directly at any point by using the `read_monitor_debug_data` command (code scratch is included but not shown in the print output).

```
[scan c0v0] >>> read monitor debug data()
      Scratch: 0x0
Debug Entry Level: 0x2
      PC: 0xfffffffffbfc00004
-Registers-
  r01 : 0x0
  <...>
  r31 : 0x0
```

*Note: A debug re-entry happens when an exception is taken whilst in debug mode. For example, hitting a breakpoint inside of `dmxseg` would cause a re-entry, as would trying to access invalid memory. Some exceptions trigger when the VP attempts to leave debug mode. These also cause a re-entry but the monitor makes this process transparent to Codescape Console.*

### Command Buffer

The command buffer is where the current command and the monitor's state can be found. To read the whole buffer use the `read_monitor_command_buffer` command, or to read just the details (contained in the first word) the command `read_monitor_command`.

```
[scan c0v0] >>> read monitor command buffer()
[0x43000a01, 0x2e0, 0x0, 0x1, 0xbfc00004, 0xffffffff, 0x0, 0x0]
[scan c0v0] >>> read monitor command()
ready busy size type command
0      1      3      000a 01
```

### Data Buffer

The data buffer is used for inputs or results that exceed the 128 bits of free space in the command buffer. This can be read with the `read_monitor_data_buffer` command.

```
[scan c0v0] >>> read monitor data buffer()
0xfffffffff201000 06400000 01000302 00000000 00000200 00000001 00000000
..@.....
0xfffffffff201018 1000fffe 00000000 00000000 00000000 00000000 00000000
.....
<...>
```

### Command Numbers

Each command type is assigned a number which is sent to the monitor and this number is shown in the error message if a command fails.

Number	Command type
1	Read
2	Write
3	Read immediate
4	Write immediate
5	Resume
6	Cache op
7	TLB probe
8	Freeze

*Note: An immediate read or write uses the command buffer for data storage and so is limited to 128 bits of input or output.*

### Response Codes

If a command is successfully sent to the monitor but fails because it is malformed or causes a re-entry, a non-zero response code is set. This code replaces the command number before the monitor enters the state in which it waits for the probe to acknowledge the command. The number and its meaning will be shown in the error message along with the value of the Debug register 'DExcCode' field.

Number	Meaning
0	Success
1	Command caused a debug re-entry
2	Unknown command
3	Unknown type

*Note: The type in 'Unknown type' refers to a combination of the memory type and size of the access, not the command type.*

## 7.4. Connecting

You can connect to the system in one of two ways. The first method is for use with a socket simulator, the second is for use with a probe in scan only mode connected to a real device.

*Note: The examples are for a DA-net probe but this can be substituted for SP55E.*

```
CodescapeConsole DBU
CodescapeConsole "DBU DA-net 438"
```

*Note: When connecting to a socket simulator the default location is localhost:44444. To change this use "DBU hostname:port".*

For either method the first step is to discover the cores and VPs present.

```
[tap 0 of 1] >>> dbuscandevices()
scan_c0v0 - mips
scan_c1v0 - mips
[scan_c0v0] >>>
```

dbuscandevices() builds a structure of high level objects to represent the VPs in the system. They provide the higher level methods for things such as reading memory blocks, reading registers and stopping the VP.

## 7.5. Using the Debug Monitor via high level commands

Codescape Console allows users to use high level commands such as `regs()`, `word()` and `bkpt()` without setting up the monitor themselves.

### 7.5.1. Debugging

You can use commands normally except that the first command will load the monitor and enter debug mode.

```
[scan_c0v0] >>> regs('pc')
Debug monitor identifier is incorrect (0xffffffff), monitor may invalid or not loaded.
0xffffffffbfc00004
```

The monitor is included with Codescape Console. After this initial delay the command will run normally.

*Note: Detecting whether the monitor is already loaded is done by using fixed identifier and version number placed at the start of `dmxseg`. This is simply a check that some form of monitor is present, it is not a validation of the data present. So in the case that the identifier was overwritten the monitor would be reloaded, despite the other contents being the same.*

Going to back to normal execution is done manually with the command `go()`.

```
[scan c0v0] >>> go()
Running from 0xffffffffbfc00004
status=running
```

Similarly, the `halt()` command can be used to enter debug mode.

```
[scan c0v0] >>> halt()
status=halted_by_probe pc=0xffffffffbfc00004
0xffffffffbfc00004 1000fffe b 0xffffffffbfc00000
```

Future commands will not need to load the monitor, only enter debug mode if required.

### 7.5.2. Exceptions

If a monitor command causes an exception the monitor's state will change normally but it will set a non-zero response code.

In the example below a breakpoint has been set on a function within the monitor, this triggers a re-entry to debug mode and hence the command fails. The monitor's state will be moved to idle as usual, but the last parameters are left in memory until the next command.

```
[scan c0v0] >>> bkpt(set, symbol(' Z17MakeCPInstructionjjjj'))
=====
Address          Enabled Type Data          HW Index
=====
0xfffffffffff200538 Enabled sw  0x3c020008 -1
=====
[scan_c0v0] >>> regs('debug')
DebugMonitorError: Command failed with response code 1 - Command triggered a re-entry to debug mode (DExcCode 9)

[scan c0v0] >>> from imgtec.console.dbu monitor import *
[scan c0v0] >>> read_monitor_command()
ready busy size type command
0 0 3 000a 01
```



```
[scan_c0v0] >>> regs('pc')
0xffffffffbfc00004
```

### 7.5.3. Incorrect states

In the case that the monitor is interrupted, by a reset for example, the flags may be left in a non-idle state. This will result in a warning when doing anything that triggers a monitor command. These flags will be reset automatically and the command will proceed as normal.

```
[scan_c0v0] >>> regs('pc')
Warning: DBU monitor was not in the Idle state (ready=1 busy=1)
```

### 7.5.4. Timeouts

A command may time out waiting for a response due to the absence of a monitor or the monitor being in an incorrect state.

```
[scan_c0v0] >>> regs('pc')
DebugMonitorError: Timed out waiting for command (0x3, read immediate) to complete.
```

*Note: If the command fails for a reason related to the register bus or dmxseg (as opposed to a non-responsive monitor), a `DbuDriverException` will be raised with details of the specific failure.*

To change these timeouts use the command `dbutimeouts()`.

```
[scan_s0c0v0] >>> dbutimeouts()
timeouts(dmxseg read=5, dmxseg write=5, rb valid=2)
[scan_s0c0v0] >>> dbutimeouts(rb valid=6)
timeouts(dmxseg_read=5, dmxseg_write=5, rb_valid=6)
```

## 7.6. Low Level Usage

Certain features of the monitor may not have corresponding high level console commands at time of writing, or you may wish to avoid the automatic handling of debug mode for some reason. For these purposes you can use the underlying monitor functions.

Follow the same steps as in the 'Connecting' section for high level usage, then import the monitor functions and finally load the monitor.

```
[scan_c0v0] >>> from imgtec.console.dbu_monitor import *
[scan_c0v0] >>> load_monitor()
```

### 7.6.1. Debug Mode

Next start debug mode by using `enterdebug`. This will detect the DBU style JTAG automatically and defaults to global throttle being off. If you need to make sure you entered debug mode correctly you can do the procedure shown below.

```
[scan_c0v0] >>> enterdebug(global throttle=True)
Numcores: 1
Setting Probeen and Probtrap
Reading vc control core 0 vc 0
Sending VC into debug mode
Core 0 VC 0 sent into debug mode
[scan_c0v0] >>> tapreg('control')
gt pracc rrb_reset dxerr rberr rb_buserr_occured dx_Size dx_fdcsize
1 1 0 2 0 0 7 0
[scan_c0v0] >>>
```

If debug mode was triggered correctly you should see global throttle in the CONTROL register set to 1 and PrAcc (pracc) set to 1. This means that the VP is waiting for the JTAG interface to acknowledge its request for (what should be) address 0xFFFFFFFF200200. This is the debug entry point into dmxseg. For verification you can step the initial part of the monitor to see what it is executing.

```
[scan_c0v0] >>> dbustep()
Target reset detected (roccmask: 0x00000001) continuing...
0xffffffff200200 4081f800 mtc0 at, DESAVE
```

After doing that you can manually disable global throttle and begin to run commands.

```
[scan_c0v0] >>> dbuglobalthrottle(False)
False
[scan_c0v0] >>> monitor read pc()
0xffffffffbfc00004
```

In this mode you are responsible for managing whether you are in debug mode or not, however you can still use the high level commands. For example runstate() will tell you whether you are in debug mode. Also any command such as regs(), which does an automatic stop, will know whether you are already in debug mode. From there you can use monitor\_resume to exit debug mode.

```
[scan_c0v0] >>> enterdebug(global_throttle=True)
<...>
[scan_c0v0] >>> runstate()
Debug monitor identifier is incorrect (0xffffffff), monitor may invalid or not loaded.
status=halted_by_probe pc=0x9c9c9c9c
[scan_c0v0] >>> regs('pc')
Debug monitor identifier is incorrect (0xffffffff), monitor may invalid or not loaded.
0xffffffffbfc00004
[scan_c0v0] >>> runstate()
status=halted by probe pc=0xffffffffbfc000044
[scan_c0v0] >>> monitor_resume()
[scan_c0v0] >>> runstate()
status=running
```

*Note: In certain situations runstate won't be able to read the current PC. In the example above, the device has no monitor loaded so the PC shown is simply a place holder as we know from the debug status register that we are not in debug mode but cannot read the PC at this time. It will not load the monitor automatically unlike other commands.*

## 7.6.2. Multiple VPs

By using the monitor\_freeze and monitor\_unfreeze you can have many VPs in debug mode at once. The monitor only has one context to save registers so 'freezing' a thread causes it to restore those values to its registers and then local throttle itself. This allows another VP to go into debug mode and not overwrite the previous VP's register state.

```
[tap 0 of 1] >>> dbuscandevices()
scan s0c0 - mips [scan core0]
  scan s0c0v0 - I6400-VPE0 [scan_c0v0]
  scan s0c0v1 - I6400-VPE1 [scan_c0v1]
[scan_s0c0v0] >>> from imgtec.console.dbu_monitor import *
[scan_s0c0v0] >>> regs('at', 0x1111)
Debug monitor identifier is incorrect (0xffffffff), monitor may invalid or not loaded.
0x00001111
[scan_s0c0v0] >>> monitor_freeze(0, 0)
[scan_s0c0v0] >>> device(listdevices()[1])
scan s0c0v1 - I6400-VPE1 [scan_c0v1]
[scan_s0c0v1] >>> regs('at', 0x2222)
0x00002222
[scan_s0c0v1] >>> monitor_freeze(0, 1)
[scan_s0c0v1] >>> device(listdevices()[0])
scan_s0c0v0 - I6400-VPE0 [scan_c0v0]
[scan_s0c0v0] >>> monitor_unfreeze(0, 0)
```

```
[scan_s0c0v0] >>> regs('at')
0x00001111
[scan_s0c0v0] >>> monitor_freeze(0, 0)
[scan_s0c0v0] >>> device(listdevices()[1])
scan_s0c0v1 - I6400-VPE1 [scan_c0v1]
[scan_s0c0v1] >>> monitor_unfreeze(0, 1)
[scan_s0c0v1] >>> regs('at')
0x00002222
```

The example above shows the use of the commands. Freezing the first VP and changing the value of 'at' on VP 2 without effecting the first VP.

### 7.6.3. Monitor Commands

Each monitor command has a corresponding console function (these must be imported specifically).

monitor_read_memory	monitor_write_memory
monitor_read_cp0_register	monitor_write_cp0_register
monitor_read_cp1_register	monitor_write_cp1_register
monitor_read_cplc_register	monitor_write_cplc_register
monitor_read_fpu_double_register	monitor_write_fpu_double_register
monitor_read_fpu_single_register	monitor_write_fpu_single_register
monitor_read_guest_cp0_register	monitor_write_guest_cp0_register
monitor_read_msa_control_register	monitor_write_msa_control_register
monitor_read_msa_register	monitor_write_msa_register
monitor_read_pc	monitor_write_pc
monitor_read_gp_register	monitor_write_gp_register
monitor_read_tlb	monitor_write_tlb
monitor_tlb_probe	monitor_cache_op
monitor_resume	monitor_freeze
monitor_unfreeze	

There are also functions to help set and reset the various flags and buffers.

write_monitor_command	read_monitor_command
acknowledge_monitor_command	reset_monitor_command
reset_monitor_changes	check_monitor_version
read_monitor_debug_data	read_monitor_data_buffer
read_monitor_context_register	write_monitor_context_register
read_monitor_command_buffer	

To get help on any of these functions use the built in help command with the function's name.

```
[tap 0 of 1] >>> from imgtec.console.dbu monitor import *
[tap 0 of 1] >>> help(monitor_read_pc)
Help on function monitor_read_pc in module imgtec.console.dbu.monitor:

monitor_read_pc(device=None)
    Read the current PC.
    Note that all DEPC operations are redirected to the PC in the saved context.
    As DEPC may be modified further by exceptions in debug mode.
```

For a command such as read, the type parameter is an MDI resource number. This defaults to 25 which means virtual memory so monitor\_read is the basic read memory command but could be used with a different memory type. However each other type has a command that will do that for you, for example monitor\_read\_msa\_register for MSA registers.

At this level no register name translation is done. So trying to read 'config1' will not work. You need to give the index or bank and select for that register. Again, you can always mix in the high level commands if needed.

```
[scan_c0v0] >>> monitor_read_gp_register(1)
[0x0]
[scan_c0v0] >>> regs('config')
0x80002802
```

```
[scan_c0v0] >>> monitor_read_cp0_register(16, 0, 1)
[0x80002802]
```

## 8. Advanced Debug Adapter settings

There are various settings to control the debug adapter's behavior. They are mainly used to increase logging of the debug adapter's actions to help diagnose target problems (usually at the expense of performance) or to help deal with unusual targets. These settings can be configured in Codescape on the Tools > Configure Probe... menu.

They can also be configured in a Python script, standard Python interpreter or Codescape Console using the following commands:

### Python Script or Interpreter:

```
>>> probe.GetDASettingList() # lists all available settings
>>> probe.GetDASettingValue("Fast Writes") # read a setting
>>> probe.SetDASettingValue("Fast Writes", 1) # enable a setting
```

### Codescape Console:

```
>>> config() # lists all available settings
>>> config("Fast Writes") # read a setting
>>> config("Fast Writes",1) # enable a setting (and read it again)
```

### 8.1. Global settings

These settings affect all cores and can be set before the target has been auto-detected or setup.

Some commands apply only to a specific probe type as indicated.

Name	Type	Description	Default
<i>DA-net only</i> JTAG Clock	DANetJtagClocks	Selects JTAG clock frequency, this must be one of: 0 = 20MHz 1 = 10MHz 2 = 5MHz 3 = 2.5MHz 4 = 1.25MHz 5 = 625KHz 6 = 312KHz 7 = 156KHz.	20MHz
<i>SysProbe only</i> JTAG Clock	int	Selects JTAG clock frequency in Hertz.	31250000
Halt After Reset	bool	When True on a HardReset an EJTAG boot is performed. This stops the core from running from the Boot Exception Vector and goes straight into debug mode. Codescape Debugger also controls this option through the Halt After Reset option in Target Debug Options.	false
<i>SysProbe only</i> Log Level	int	Controls the level at which debug messages get sent to the main log file (info log) and live logging.	0

Name	Type	Description	Default
<i>DA-net only</i> Verbose Logging	bool	Enables verbose logging, enabling this gives a small reduction in performance	false
<i>SysProbe only</i> Reset Duration	int	The time in ms that the nRESETOUT signal is assert on hard reset	500
<i>SysProbe only</i> Post Reset Delay	int	Time in ms to wait after a hard reset to allow bootrom to run before attempting any access from the probe	0
Reset on Connect	bool	Issue Hard Reset on probe connection.	false
<i>SysProbe only</i> Reset Tap Too	bool	After a hard reset CPC systems need a TAP reset to get the CPC into probe mode, so it'll power up all cores.	true
<i>SysProbe only</i> Sampling Duration	int	The number of ms between statistical profiling samples. If zero no statistical profiling is performed.	0
<i>SysProbe only</i> Sampling SoC Num	int	The soc index on which statistical profiling samples should be collected.	0
<i>SysProbe only</i> Sampling Core Num	int	The core index on which statistical profiling samples should be collected.	0
<i>SysProbe only</i> Sampling Threads	int	A mask of the threads on which statistical profiling samples should be collected. For example 0b11 indicates threads 0 and 1.	0
<i>SysProbe only</i> APB Timeout	int	Timeout applied to debug transactions on devices which implement an APB (parallel) debug bus.	100
<i>SysProbe only</i> Assert DINT	bool	Override for the DINT Signal	false
Assert nHardReset	bool	Override for the nRESETOUT signal	true
Assert nTRST	bool	Override for the nTRST Signal	true
<i>DA-net only</i> Assert nTRST during tap reset	bool	When set on a tap reset nTRST in assert then the JTAG state machine is walked to the reset state then nTRST is released, when cleared only a synchronous tap reset is performed (walking state machine to reset state).	true
JTAG Logging	bool	Enables logging of all JTAG scans, causes significant performance degradation and Codescape Debugger will probably time out. This option should only be used for debugging low level JTAG scan issues in Codescape Console.	false
Polling	bool	Disable / Enable ALL background polling of All targets	true

Name	Type	Description	Default
Timeout Scale	int	Controls the scaling factor for timeouts, increasing this value increased the probes timeout (at the risk of lack of responsiveness on broken systems) usually only needed by very slow targets running on emulation platforms.	50

## 8.2. MIPS

These settings apply to all MIPS cores in a target system.

Name	Type	Description	Default
PC Sample	bool	Enable PC Sampling on connection	true
Allow FixedMap Accesses	bool	With a Fixed Map MMU all mapped accesses proceed with a simple address translation, thus when enabled its easy to lock up targets using this MMU if HSPs have not been set correctly	false
Allow KUSEG Accesses	bool	When Status ERL+EXL =1, USEG gets a 1:1 mapping between virt and phys space (ignoring MMU) if set probe accesses are allowed to proceed in USEG in this state.	false
Allow Mapped Accesses	bool	Allows access to mapped regions (eg useg, kseg2/3) note access may still fail if address not mapped in MMU	true
Disable MMU Checking	bool	Prevent any checking of the MMU to see if mapped accesses will work, risky to set will cause exceptions in debug mode if access not mapped	false
Fast Monitor Address	int	Address to which fast transfer monitor is loaded to before it gets locked into the cache, this must be a KSEG0 address	0x80000000
Use Current ASID	bool	Uses the current ASID in entryhi as this is most likely the current running process, for the ASID of the access, ignoring the value from Codescape	true
Use ISPRAM	bool	Enables debugger handling of ISPRAM, note ISPRAM setup is cached on setting Use ISPRAM = 1, if ISPRAM setup is changed this setting must be disabled and re-enabled	false
DA-net only Post Reset Delay	int	Time in ms to wait after a hard reset to allow bootrom to run before attempting any access from the probe	0
Reset ACK Timeout	int	The time in ms to wait for while acknowledging Reset (waiting for Rocc)	500
DA-net only Reset Duration	int	The time in ms that the nRESETOUT signal is assert on hard reset	500
DA-net only CPC Probe Mode	bool	After a hard reset CPC systems need a TAP reset to get the CPC into probe mode, so it'll power up all cores	true
Disable Ints on HW Single Step	bool	Disables interrupts on HardwareSingle Step (So you don't end up stepping into interrupt handling code eg a timer interrupt).	false
Disable trace on halt	bool	When set the probe disable trace data collection on an unexpected halt (Breakpoint/SingleStep etc).	true



Name	Type	Description	Default
DA-net only EJTAG Boot All	bool	When False, a reset with 'Halt after Reset' selected only applies the EJTAG boot indication/instruction to the first tap (ie c0v0). When True, all taps receive an EJTAG boot instruction.	true
Enter Debug Timeout	int	The time in ms to wait while trying to get the core into debug mode	100
DA-net only Guest TLB	bool	Make TLB commands operate on the guest TLB.	false
Max FDC Channels	int	Controls the number of channels (from 0) which get mapped to DA virtual channels	0
Print ECR on Timeout	bool	Prints the ECR of all TAPs in the system including potentially none MIPS taps, hence this could be disturbing for non MIPS taps.	true
Stop Count in DM	bool	Stops the count register from being incremented when the CPU is in debug mode.	true
Using BEV overlay	bool	When this option transitions from 0->1 the probe will re-cache the BEV overlay settings, If user code reconfigures the BEV overlay at runtime then this option needs toggling so the probe can re-cache the new settings.	false
Trace Fast	bool	Experimental option not for users.	false

### 8.3. Meta

These settings apply per core.

Name	Type	Description	Default
Allow Intrusive Debug	bool	When set the DA will perform an intrusive poll of the target whilst running, this will have a slight impact on any performance measurements which need to be cycle accurate, But gives better detection of a thread stopping or starting outside of the DAs control.	false
Core Reg Interlock	bool	Take Lock2 when accessing TXXUXRXQ as part of sharing protocol (enable above) set both of these if sharing	false
Core Reg Negotiate	bool	Use the soft locking / negotiation protocol when sharing the register port with another user (eg another thread or a host via Slave port).	false
DCL High Precision Mode	bool	Takes LOCK2 while DCL script is running, can help reduce jitter in performance measurements	false
Debug Route	MetaDebugRoute		auto
Force Availability	bool	When set the DA will always set the force availability bit in the JTAG control register	true

Name	Type	Description	Default
MCM Port Locking	bool	Use the Soft locking protocol to share the MCM port with another user (eg another thread or host via slave port)	true
MDBG Diagnostics Mode	bool	Puts the Meta Debug Port in to diagnostics mode, to allow postmortem debug of a locked up system.	false
Minim Translations	bool	Disables the DA from performing minim translations on minim code addresses on memory type 0.	true
Per Thread Channels	bool	Each Meta thread gets it own set of DA channels	true
Polling in Diagnostics Mode	bool	When in debug port diagnostic mode this disables polling of the ready bit during debug transactions, clear it if the core is very badly locked up, you may get some state out if you are lucky	true
TBI Stack Unwinding	bool	When Halt interrupts are enabled and a target stops at a breakpoint (or due to single-step) the state shown to the user by the DA isn't the actual state of the target. When the target stops at the breakpoint a halt interrupt fires and we jump to TBIs halt interrupt handler which see the halt as a breakpoint, so turns off halt interrupts and executes another switch instruction (breakpoint), this happens in a function called TBIUnexpectXXX(). The DA then unwinds the stack to show the user the original halt state. When this setting is cleared you get to see the true state (ie stopped at a switch in TBIUexpectXXX).	

## Appendix A. Hardware Definition Reference Documentation

This appendix describes the syntax of elements in XML hardware definition files.

### A.1. Hardware Definition XML Elements

#### A.1.1. Document

The root element of the Hardware Definition XML data. All objects must have this element as the outermost enclosing object.

```
<iocconfig>
  <Board N="Board1"/>
  <p N="Processor1"/>
  <pl N="ProcessorLink1">
    <src></src>
  </pl>
  <SoC N="SoC1"/>
  <SoCLink N="SoCLink1">
    <src></src>
  </SoCLink>
  <CoreID N="CoreID1">
    <CoreIDValue>0x0000</CoreIDValue>
    <src></src>
  </CoreID>
</iocconfig>
```

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
Board	<Board N="name">...</Board>	Zero or more of this type
Processor	<p N="name">...</p>	Zero or more of this type
ProcessorLink	<pl N="name">...</pl>	Zero or more of this type
SoC	<SoC N="name">...</SoC>	Zero or more of this type
SoCLink	<SoCLink N="name">...</SoCLink>	Zero or more of this type
CoreID	<CoreID N="name">...</CoreID>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

#### A.1.2. Board

Defines a Board.

```
<Board N="Board">
  <Taps>0x00000000</Taps>
  <SoC N="SoC1"/>
  <settings N="Settings1"/>
  <setting N="Setting1"/>
</Board>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<Board N="object name"> ...properties and child objects... </Board>	The name of the item.

The following properties are defined for this object:

Property	XML Tag(s) and Syntax	Description
Number of Taps	<Taps> ... </Taps>	Number of test access ports. A positive integer.

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
<u>SoC</u>	<SoC N="name">...</SoC>	Zero or more of this type
<u>SoCLink</u>	<SoCLink N="name">...</SoCLink>	Zero or more of this type
<u>Settings</u>	<settings N="name">...</settings>	Zero or more of this type
<u>Setting</u>	<setting N="name">...</setting>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

### A.1.3. SoC

Defines a System on Chip. All the properties except Name are relevant to META processors only.

```
<SoC N="SoC">
  <JTagPosition>0x00000000</JTagPosition>
  <IRLength>0x00000000</IRLength>
  <JtagID>0x00000000</JtagID>
  <J_IMG_ATTEN>0x00000000</J_IMG_ATTEN>
  <TapType>0x00000000</TapType>
  <J_IMG_STATUS>0x00000000</J_IMG_STATUS>
  <J_IMG_CONTROL>0x00000000</J_IMG_CONTROL>
  <CoreInfo N="CoreInfo1"/>
  <settings N="Settings1"/>
  <setting N="Setting1"/>
</SoC>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<SoC N="object name"> ...properties and child objects... </SoC>	The name of the item.

The following properties are defined for this object which must, if present, be in the order shown:

Property	XML Tag(s) and Syntax	Description
Position of this item on the JTAG scan chain.	<JTagPosition> ... </JTagPosition>	Position of this item on the JTAG scan chain.
The length of the TAP Instruction Register in bits.	<IRLength> ... </IRLength>	The length of the TAP Instruction Register in bits.
JTAG ID in hexadecimal.	<JtagID> ... </JtagID>	JTAG ID in hexadecimal.
JTAG Attention Instruction in hexadecimal.	<J_IMG_ATTEN> ... </J_IMG_ATTEN>	JTAG Attention Instruction in hexadecimal.
Tap Type in hexadecimal.	<TapType> ... </TapType>	Tap Type in hexadecimal.
JTAG Status Instruction in hexadecimal.	<J_IMG_STATUS> ... </J_IMG_STATUS>	JTAG Status Instruction in hexadecimal.
JTAG Control Instruction in hexadecimal.	<J_IMG_CONTROL> ... </J_IMG_CONTROL>	JTAG Control Instruction in hexadecimal.

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
<u>CoreInfo</u>	<CoreInfo N="name">...</CoreInfo>	Zero or more of this type
<u>Settings</u>	<settings N="name">...</settings>	Zero or more of this type
<u>Setting</u>	<setting N="name">...</setting>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

#### A.1.4. CoreInfo

Defines a Core.

```
<CoreInfo N="CoreInfo">
  <DAConfiguration N="DAConfiguration1">
    <src></src>
  </DAConfiguration>
  <p N="Processor1"/>
  <mt N="MemoryType1">
    <mtv>0x00</mtv>
  </mt>
  <settings N="Settings1"/>
  <setting N="Setting1"/>
</CoreInfo>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<CoreInfo N="object name"> ...properties and child objects... </CoreInfo>	The name of the item.

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
<u>DAConfiguration</u>	<DAConfiguration N="name">...</DAConfiguration>	Zero or more of this type
<u>Processor</u>	<p N="name">...</p>	Zero or more of this type
<u>ProcessorLink</u>	<pl N="name">...</pl>	Zero or more of this type
<u>MemoryType</u>	<mt N="name">...</mt>	Zero or more of this type
<u>Settings</u>	<settings N="name">...</settings>	Zero or more of this type
<u>Setting</u>	<setting N="name">...</setting>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

### A.1.5. DAConfiguration

Defines a DA Configuration.

```
<DAConfiguration N="DAConfiguration">
  <src></src>
</DAConfiguration>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<DAConfiguration N="object name"> ...properties and child objects... </DAConfiguration>	The name of the item.

The following properties are defined for this object:

Property	XML Tag(s) and Syntax	Description
Source	<src> ... </src>	The path to a file for the xml configuration file for this object. This can be an absolute path, or a path relative to the main prconfig.xml file.

### A.1.6. Processor

Defines a Processor.

```
<p N="Processor">
  <mt N="MemoryType1">
    <mtv>0x00</mtv>
  </mt>
  <m N="Module1"/>
  <settings N="Settings1"/>
  <d>This is a description of Processor</d>
</p>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<p N="object name"> ...properties and child objects... </p>	The name of the item.

The following properties are defined for this object:

Property	XML Tag(s) and Syntax	Description
Description	<d> ... </d>	A description of the item which will appear as a tool tip when the mouse passes over the name of the item in the Peripheral Region in Codescape.

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
<u>MemoryType</u>	<mt N="name">...</mt>	Zero or more of this type
<u>Module</u>	<m N="name">...</m>	Zero or more of this type
<u>Settings</u>	<settings N="name">...</settings>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

### A.1.7. MemoryType

Defines a class of memory storage.

The Memory Types are used to identify the Codescape memory types that should be used for memory spaces in a processor. There may be a single memory space for code, data, and memory mapped registers, or on some platforms separate memory spaces.

For example, the Enigma RPU has separate memory spaces for code, data and memory mapped registers, whilst MIPS cores use a single address space. The value of a Memory Type is an internal constant used between the debugger and the probe, to identify the memory space, so users do not normally need to change these settings. The Memory Type value of 0, usually with the Memory Type name of "Ram" is appropriate for normally addressable memory. A Memory Type may be implicitly dynamic, meaning that Codescape knows the size and accessibility of address ranges within the address space; or it may be explicitly static using the <static/> tag, which means that the memory type element should contain Memory Block elements describing those address ranges that are readable, writable and cacheable by the debugger.

A Memory Block may also configure the access size using the <sb/>, <sw/>, <sd/>, or <sq/> tags (for 8-, 16-, 32-, and 64-bit access sizes respectively), but in practice the probe ignores these accesses and performs the best access size for the comms and target type. In general the <sd/> should be used.

```
<mt N="MemoryType">
  <mtv>0x00</mtv>
  <dynamic/>
  <mb N="MemoryBlock1">
    <s>0x00000000</s>
    <en>0xffffffff</en>
  </mb>
  <d>This is a description of MemoryType</d>
</mt>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<code>&lt;mt N="object name"&gt; ...properties and child objects... &lt;/mt&gt;</code>	The name of the Memory Type. This is used in Registers to identify the Memory Type for the Register.

The following properties are defined for this object which must, if present, be in the order shown:

Property	XML Tag(s) and Syntax	Description
Memory Type Value	<code>&lt;mtv&gt; ... &lt;/mtv&gt;</code>	A hex number read by Codescape to identify the Memory Type. This value should not be altered without consultation with Codescape Technical Support
Memory Settings	<code>&lt;dynamic/&gt; &lt;static/&gt;</code>	Specifies whether the memory is static or dynamic. Only one of the XML items may be present.
Description	<code>&lt;d&gt; ... &lt;/d&gt;</code>	A description of the item which will appear as a tool tip when the mouse passes over the name of the item in the Peripheral Region in Codescape.

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
<u>MemoryBlock</u>	<code>&lt;mb N="name"&gt;...&lt;/mb&gt;</code>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

### A.1.8. MemoryBlock

Defines a block of memory.

```
<mb N="MemoryBlock">
  <s>0x00000000</s>
  <en>0xffffffff</en>
  <sl/>
  <rw/>
  <timings>-1,-1,-1,-1,-1,-1,-1,-1,-1,-1</timings>
  <shared N="SharedMemory1">
    <s>0x00000000</s>
  </shared>
  <d>This is a description of MemoryBlock</d>
</mb>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<code>&lt;mb N="object name"&gt; ...properties and child objects... &lt;/mb&gt;</code>	The name of the item.

The following properties are defined for this object which must, if present, be in the order shown:



Property	XML Tag(s) and Syntax	Description
Start Address	<s> ... </s>	The start address in the memory area of this block in hex which must align on a boundary according to the size of an addressable element in this memory block.
End Address	<en> ... </en>	The end address within the memory area of this block in hex such that the end address + 1 aligns on a boundary according to the size of an addressable element in this memory block.
Element Size	<sb/> <sq/> <sd/> <sw/> <sl/>	The size in bytes of a individual memory element. Only one of the XML items may be present.
Access	<wo/> <ro/> <rw/>	Describes how the memory item can be accessed. Only one of the XML items may be present.
Cacheable	<cacheable/>	Indicates if the debugger (Codescape or DAScript) can cache this memory block. All memory should be regarded as cacheable unless it describes registers or ports, in which case the XML item is present.
Access Timings	<timings> ... </timings>	Timings for defined accessibility of this memory block: a list of 10 integers separated by commas.
Description	<d> ... </d>	A description of the item which will appear as a tool tip when the mouse passes over the name of the item in the Peripheral Region in Codescape.

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
<u>SharedMemory</u>	<shared N="name">...</shared>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

### A.1.9. SharedMemory

Defines an area of shared memory.

```
<shared N="SharedMemory">
  <s>0x00000000</s>
  <d>This is a description of SharedMemory</d>
</shared>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<shared N="object name"> ...properties and child objects... </shared>	The name of the item.

The following properties are defined for this object which must, if present, be in the order shown:

Property	XML Tag(s) and Syntax	Description
Start Address	<s> ... </s>	The start address in the memory area of this block in hex which must align on a boundary according to the size of an addressable element in this memory block.
Description	<d> ... </d>	A description of the item which will appear as a tool tip when the mouse passes over the name of the item in the Peripheral Region in Codescape.

### A.1.10. Module

Defines a Module.

A Module is used for associating information about one or more devices, for example, a Bus State Controller (BSC). Modules can contain:

```
<m N="Module">
  <m N="Module1"/>
  <r N="Register1"/>
  <d>This is a description of Module</d>
</m>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<m N="object name"> ...properties and child objects... </m>	The name of the item.

The following properties are defined for this object:

Property	XML Tag(s) and Syntax	Description
Description	<d> ... </d>	A description of the item which will appear as a tool tip when the mouse passes over the name of the item in the Peripheral Region in Codescape.

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
<u>Module</u>	<m N="name">...</m>	Zero or more of this type
<u>Register</u>	<r N="name">...</r>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

### A.1.11. Register

Defines a processor, co-processor or memory mapped register.

```
<r N="Register">
  <mtn></mtn>
  <s>0x0</s>
  <sl/>
  <rw/>
  <rm>0xffffffffffffffff</rm>
  <wam>0xffffffffffffffff</wam>
  <wom>0x00000000</wom>
  <H/>
  <timings>-1,-1,-1,-1,-1,-1,-1,-1,-1,-1</timings>
  <f N="BitField1"/>
  <d>This is a description of Register</d>
</r>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<r N="object name"> ...properties and child objects... </r>	The name of the register that is displayed in the Peripheral Region in Codescape.

The following properties are defined for this object which must, if present, be in the order shown:

Property	XML Tag(s) and Syntax	Description
Memory Type	<mtn> ... </mtn>	The memory type (class) to which this register belongs.
Start Address	<s> ... </s>	The start address within the memory type of this register.
Size	<sb/> <sq/> <sd/> <sw/> <sl/>	The size of the register. Only one of the XML items may be present.
Access	<wo/> <ro/> <rw/> <woc/> <roc/>	Describes how the register can be accessed. Only one of the XML items may be present.
Read AND Mask	<rm> ... </rm>	Specifies a mask in hex to bitwise AND against the register value before the value is displayed. (Default: 0xFFFFFFFF)
Write AND Mask	<wam> ... </wam>	Specifies a mask in hex to bitwise AND against the register value before the value is written. (Default: 0xFFFFFFFF)
Write OR Mask	<wom> ... </wom>	Specifies a mask in hex to bitwise OR against the register value before the value is written. (Default: 0)
Radix	<H/> <B/> <D/> <O/>	Specifies the format of the item. Only one of the XML items may be present.
Access Timings	<timings> ... </timings>	Timings for defined accessibility of this memory mapped register: a list of 10 integers separated by commas.
Description	<d> ... </d>	A description of the item which will appear as a tool tip when the mouse passes over the name of the item in the Peripheral Region in Codescape.

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
<u>BitField</u>	<f N="name">...</f>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

### A.1.12. BitField

Defines a Bit Field.

A Bitfield is a specific number of bits in the register that can represent several different values depending on how the bits are set. A Bitfield can be one specific bit from a register. This bit can be combined with Bitfield Values to represent a boolean value.

An AND mask and shift is applied to extract the relevant bits.

```
<f N="BitField">
  <dv>0x00000000</dv>
  <fm>0x00000000</fm>
  <H/>
  <v N="BitFieldValue1"/>
  <d>This is a description of BitField</d>
</f>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<f N="object name"> ...properties and child objects... </f>	The name of the Bitfield that is displayed in the Peripheral Region in Codescape.

The following properties are defined for this object which must, if present, be in the order shown:

Property	XML Tag(s) and Syntax	Description
Default	<dv> ... </dv>	Default bit field value in hex.
AND Mask	<fm> ... </fm>	Mask used to extract the required bits of the field.
Shift	<sh> ... </sh>	The shift to be applied to the bits after masking. Positive for a left shift, negative for a right shift.
Radix	<H/> <B/> <D/> <O/>	Specifies the format of the item. Only one of the XML items may be present.
Description	<d> ... </d>	A description of the item which will appear as a tool tip when the mouse passes over the name of the item in the Peripheral Region in Codescape.

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
<u>BitFieldValue</u>	<v N="name">...</v>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

### A.1.13. BitFieldValue

Defines a Bit Field Value.

```
<v N="BitFieldValue">
  <x>0x0000000000000000</x>
  <d>This is a description of BitFieldValue</d>
</v>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<v N="object name"> ...properties and child objects... </v>	The name of the item.

The following properties are defined for this object which must, if present, be in the order shown:

Property	XML Tag(s) and Syntax	Description
Value	<x> ... </x>	A hex value whose width in bits is not greater than the number of bits in the BitField mask to which this value belongs.
Description	<d> ... </d>	A description of the item which will appear as a tool tip when the mouse passes over the name of the item in the Peripheral Region in Codescape.

### A.1.14. Settings

Defines a collection of Setting or Settings objects.

```
<settings N="Settings">
  <setting N="Setting1"/>
  <settings N="Settings1"/>
</settings>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<settings N="object name"> ...properties and child objects... </settings>	The name of the item.

It may contain child objects as follows:

Child Name	XML Tag(s) and Syntax	Quantity Allowed
<u>Setting</u>	<setting N="name">...</setting>	Zero or more of this type
<u>Settings</u>	<settings N="name">...</settings>	Zero or more of this type

Children, if any, should be grouped according to type and appear in the order shown in the above list and the XML fragment above.

### A.1.15. Setting

Defines a single Setting.

These are single-value string items not directly related to the physical configuration of targets.

```
<setting N="Setting"/>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<code>&lt;setting N="object name"&gt; ...properties and child objects... &lt;/setting&gt;</code>	The name of the item.

The following properties are defined for this object:

Property	XML Tag(s) and Syntax	Description
Value	<code>&lt;setting&gt; ... &lt;/setting&gt;</code>	The value of this setting.

### A.1.16. ProcessorLink

Defines a Processor Link.

```
<pl N="ProcessorLink">
  <src></src>
</pl>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<code>&lt;pl N="object name"&gt; ...properties and child objects... &lt;/pl&gt;</code>	The name of the item.

The following properties are defined for this object:

Property	XML Tag(s) and Syntax	Description
Source	<code>&lt;src&gt; ... &lt;/src&gt;</code>	The path to a file for the xml configuration file for this object. This can be an absolute path, or a path relative to the main prconfig.xml file.

### A.1.17. SoCLink

Defines a SoC Link.

```
<SoCLink N="SoCLink">
  <src></src>
</SoCLink>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<code>&lt;SoCLink N="object name"&gt; ...properties and child objects... &lt;/SoCLink&gt;</code>	The name of the item.

The following properties are defined for this object:

Property	XML Tag(s) and Syntax	Description
Source	<code>&lt;src&gt; ... &lt;/src&gt;</code>	The path to a file for the xml configuration file for this object. This can be an absolute path, or a path relative to the main prconfig.xml file.

### A.1.18. CoreID

Defines a Core ID.

```
<CoreID N="CoreID">
  <CoreIDValue>0x0000</CoreIDValue>
  <src></src>
</CoreID>
```

The following attributes are defined for this object:

Attributes	XML Tag(s) and Syntax	Description
Name	<code>&lt;CoreID N="object name"&gt; ...properties and child objects... &lt;/CoreID&gt;</code>	The name of the item.

The following properties are defined for this object which must, if present, be in the order shown:

Property	XML Tag(s) and Syntax	Description
Core ID Value	<code>&lt;CoreIDValue&gt; ... &lt;/CoreIDValue&gt;</code>	The cores' id as an up to 4 digit hex integer.
Source	<code>&lt;src&gt; ... &lt;/src&gt;</code>	The path to a file for the xml configuration file for this object. This can be an absolute path, or a path relative to the main prconfig.xml file.

## A.2. The Document Type Definition

The DTD for the xml format is not validated at load time, this is so that the file format can be extended without breaking backwards compatibility. However for reference this is the DTD that would be used:

```

<!ELEMENT ioconfig (Board, p, pl, SoC, SoCLink, CoreID)*>

<!ELEMENT Board ((Taps?), (SoC | SoCLink)+, (settings, setting)*)>
  <!ATTLIST Board N CDATA #REQUIRED>
  <!ELEMENT Taps (#PCDATA)>

<!ELEMENT SoC ((JTagPosition?, IRLength?, JtagID?, J_IMG_ATTEN?, TapType?, J_IMG_STATUS?,
J_IMG_CONTROL?), (CoreInfo)+, (settings, setting)*)>
  <!ATTLIST SoC N CDATA #REQUIRED>
  <!ELEMENT JTagPosition (#PCDATA)>
  <!ELEMENT IRLength (#PCDATA)>
  <!ELEMENT JtagID (#PCDATA)>
  <!ELEMENT J_IMG_ATTEN (#PCDATA)>
  <!ELEMENT TapType (#PCDATA)>
  <!ELEMENT J_IMG_STATUS (#PCDATA)>
  <!ELEMENT J_IMG_CONTROL (#PCDATA)>

<!ELEMENT CoreInfo (p | pl)?, (DAConfiguration)?, (mt, settings, setting)*>
  <!ATTLIST CoreInfo N CDATA #REQUIRED>

<!ELEMENT DAConfiguration ((src))>
  <!ATTLIST DAConfiguration N CDATA #REQUIRED>
  <!ELEMENT src (#PCDATA)>

<!ELEMENT p ((d?), (mt, m, settings)*)>
  <!ATTLIST p N CDATA #REQUIRED>
  <!ELEMENT d (#PCDATA)>

<!ELEMENT mt ((mtv, (dynamic|static)?, d?), (mb)*)>
  <!ATTLIST mt N CDATA #REQUIRED>
  <!ELEMENT mtv (#PCDATA)>
  <!ELEMENT dynamic EMPTY>
  <!ELEMENT static EMPTY>

<!ELEMENT mb ((s, en, (sb|sq|sd|sw|sl)?, (wo|ro|rw)?, cacheable?, timings?, d?), (shared)*)>
  <!ATTLIST mb N CDATA #REQUIRED>
  <!ELEMENT s (#PCDATA)>
  <!ELEMENT en (#PCDATA)>
  <!ELEMENT sb EMPTY>
  <!ELEMENT sq EMPTY>
  <!ELEMENT sd EMPTY>
  <!ELEMENT sw EMPTY>
  <!ELEMENT sl EMPTY>
  <!ELEMENT wo EMPTY>
  <!ELEMENT ro EMPTY>
  <!ELEMENT rw EMPTY>
  <!ELEMENT cacheable (#PCDATA)>
  <!ELEMENT timings (#PCDATA)>

<!ELEMENT shared ((s, d?))>
  <!ATTLIST shared N CDATA #REQUIRED>

<!ELEMENT m ((d?), (m, r)*)>
  <!ATTLIST m N CDATA #REQUIRED>

<!ELEMENT r ((mtn?, s?, (sb|sq|sd|sw|sl)?, (wo|ro|rw|woc|roc)?, rm?, wam?, wom?, (H|B|D|O)?,
timings?, d?), (f)*)>
  <!ATTLIST r N CDATA #REQUIRED>
  <!ELEMENT mtn (#PCDATA)>
  <!ELEMENT woc EMPTY>
  <!ELEMENT roc EMPTY>
  <!ELEMENT rm (#PCDATA)>
  <!ELEMENT wam (#PCDATA)>
  <!ELEMENT wom (#PCDATA)>
  <!ELEMENT H EMPTY>
  <!ELEMENT B EMPTY>
  <!ELEMENT D EMPTY>
  <!ELEMENT O EMPTY>

<!ELEMENT f ((dv?, fm?, sh?, (H|B|D|O)?, d?), (v)*)>
  <!ATTLIST f N CDATA #REQUIRED>
  <!ELEMENT dv (#PCDATA)>
  <!ELEMENT fm (#PCDATA)>
  <!ELEMENT sh (#PCDATA)>

<!ELEMENT v ((x?, d?))>

```



```

<!ATTLIST v N CDATA #REQUIRED>
<!ELEMENT x (#PCDATA)>

<!ELEMENT settings (setting, settings)*>
  <!ATTLIST settings N CDATA #REQUIRED>

<!ELEMENT setting ((setting?))>
  <!ATTLIST setting N CDATA #REQUIRED>
  <!ELEMENT setting (#PCDATA)>

<!ELEMENT pl ((src))>
  <!ATTLIST pl N CDATA #REQUIRED>

<!ELEMENT SoCLink ((src))>
  <!ATTLIST SoCLink N CDATA #REQUIRED>

<!ELEMENT CoreID ((CoreIDValue, src))>
  <!ATTLIST CoreID N CDATA #REQUIRED>
  <!ELEMENT CoreIDValue (#PCDATA)>
    
```